STUDIES RELATED TO THE PROCESS OF PROGRAM DEVELOPMENT

MORGAN HOWARD WILLIAMS

Submission for the degree of Doctor of Science
of
Rhodes University

Edinburgh
January 1993

ABSTRACT

The submitted work consists of a collection of publications arising from research carried out at Rhodes University (1970-1980) and at Heriot-Watt University (1980-1992). The theme of this research is the process of program development, i.e. the process of creating a computer program to solve some particular problem. The papers presented cover a number of different topics which relate to this process, viz.

(a) Programming methodology - aspects of structured programming.
(b) Properties of programming languages.
(c) Formal specification of programming languages.
(d) Compiler techniques.
(e) Declarative programming languages.
(f) Program development aids.
(g) Automatic program generation.
(h) Databases.
(i) Algorithms and applications.

ACKNOWLEDGEMENTS

I am deeply indebted to all those who have contributed in any way, directly or indirectly, in producing the collection of works in this submission.

Particular thanks are due to those colleagues and students who have collaborated in the research projects and have been co-authors to the publications. But thanks must also extend to those many unnamed individuals who participated indirectly in the research, from students who acted as guinea pigs for software developed to secretaries who typed the manuscripts for publication.

However, the greatest thanks must go to my family and friends - to my parents without whom this would not have been possible, to Jean who has stoically borne with me through this research, and to Christopher and Michael who make it all seem worthwhile.

1. INTRODUCTION

The process of creating a program which when executed by a computer, will satisfactorily solve some particular problem, is one of the fundamental activities of Computer Science. It is not an easy one and it is very prone to error. As a consequence much of Computer Science is concerned with improving this process.

One of the first and most significant aids to the programmer to be developed is that of programming languages. The progress from having to express programs directly in the machine language of the machine concerned to the use of symbolic forms of these languages, referred to as assembly languages, to the introduction of problem-oriented languages such as Fortran, took place in little more than a decade. This development had a profound effect on the rate of production of programs, the quality of the programs produced, and the maintainability of these programs.

Since the implementation of the first high-level language in the mid-fifties, a large number of such languages has been developed. But although these languages have been steadily improving in various respects, there has not been any dramatic breakthrough in programming language development since then.

Instead effort has been channelled in other directions resulting in three significant developments which affect the programmer. These are:

(1) The creation of a set of software tools which assist the programmer in the program development process. These tools include operating systems (which affect the way in which the user interfaces with the computer), different types of editors (from screen-based to syntax-directed editors) and a variety of debugging tools.

(2) The introduction of programming methodologies (such as structured programming) to systematize the program development process.

(3) The appearance of packaged programs which are capable of solving a whole range of problems of a particular type by setting up the appropriate parameters; this has led in turn to fourth-generation languages.

This study is concerned with various aspects of the program development process, from programming methodology through languages and tools to the program produced.

2. STRUCTURED PROGRAMMING

Paper [1] was the first attempt to isolate the types of constructs in a flow diagram representation of a program which render it "unstructured". Although it only appeared in print after the paper by McCabe on the same subject, this was due to the long delay time for publication in the Computer Journal as the manuscript was submitted over a year before that of McCabe.

The aim of paper [2] was to present a method for converting unstructured flow diagrams into structured ones. The intention behind this was not primarily to produce a tool which programmers might use for improving unstructured programs but as a means of showing that any algorithm can be expressed in a reasonable structured form.

[3] deals with two problems in connection with the decomposition of flow diagrams. The first is concerned with different methods of pre-processing flow diagrams in preparation for the identification of unstructured subflowcharts. The second relates to the concept of dominance and the algorithms which Prather and Giulieri present for determining subflowcharts.

By 1983 terminology had become a serious problem in this particular area of Computer Science, with different names being

used by different authors to refer to the same object or idea and the same name being used to refer to different objects and ideas. [4] attempts to clarify some of the issues in relation to structuredness and reducibility.

The idea of automatic restructuring can be applied to practical programming languages such as Pascal as a pre-processing phase to restructure programs to remove goto statements as described in [5]. The reason for this will be described further in the next section.

3. PROPERTIES OF LANGUAGES

Since programming languages are the main vehicles by which the programmer communicates his ideas to the computer, it is essential that they be designed as carefully as possible. Nevertheless existing programming languages have many problems associated with them.

One example is an anomaly which arises in arrays with SYNCHRONIZED clauses in Cobol which, if the compiler writer is not aware of it, can lead to the generation of incorrect code under certain circumstances. Paper [6] outlines the problem and gives an algorithm for detecting the situation and correcting for it.

The basic tenet of structured programming is that any algorithm describing a computation process can be translated into a structured program. Some strict adherents of structured programming therefore see no necessity for having a GOTO statement in any programming language which provides suitable structured control structures. On the other hand, others point out that there are many instances where a GOTO statement will improve the efficiency or clarity of a program, and though unnecessary and careless usage of the GOTO statement can be dangerous and can lead to unreadable programs, careful use of the same can be a definite advantage. As a compromise, [7] investigates the possibility of using a generalised form of EXIT statement as an alternative to the GOTO statement.

The next three papers are concerned with the development of a programming language, BPL, which combines the best features of BASIC and Pascal together with a few novel features into a single integrated language which is well-suited for teaching purposes. Papers [8] and [9] describe the principles on which this language is designed while [10] gives a complete account of its syntax and an informal account of its semantics. The language was implemented in Pascal on an ICL 1900 computer. Unfortunately, the work was only completed after I had left Rhodes University and has not been exploited further.

Although almost all program development thus far has been based on imperative programming languages, during the past decade there has been growing interest in the class of programming languages referred to as declarative languages (encompassing both logic programming and functional programming). The most notable language in this class is the programming language Prolog.

In the early phases of the Japanese Fifth Generation Computer Programme, one of the aims was to produce computers designed to execute Prolog as a base language. If such machines were developed and were to become popular, this would create problems with existing software written in imperative programming languages. As a result the problem of translating programs in an imperative language such as Pascal into Prolog programs was studied. The first part of this process has been described in paper [5], mentioned in the previous section. A more complete description of the process is given in [11] and [12], although even here some problems (such as pointers) have not been dealt with.

The work highlighted one of the significant differences between Prolog and imperative programming languages - namely, the handling of arrays. In imperative programming languages access to values stored in an array is handled very efficiently, in Prolog it is handled very inefficiently through the use of lists. [13] describes work done on implementing array handling directly in Prolog and the problems surrounding this.

4. FORMAL SPECIFICATION OF LANGUAGES

In designing a language, a language designer needs to be able to define every aspect of the language to ensure that no unexpected problems arise in implementation and to communicate his ideas in a clear and unambiguous way to implementors and users alike. For this purpose formal notations have been developed for specifying the syntax, static semantics and dynamic semantics of programming languages. [14] presents one such formal notation based on BNF which can be used to specify the syntax and static semantics of a language (i.e. those aspects of a language which are checked by the front end of a compiler) in a manner which is easy for the implementor to interpret. [15] demonstrates how the notation can be used to specify the complete syntax of BASIC and Algol60, including the nontrivial problem of type checking parameters in Algol60. A comparative study of several different formal notations is given in [16] while [17] reports on the benefit obtained from using this notation in compiler implementation.

[18] considers the problem of constructing two-level grammar specifications and the notion of structuring such specifications.

Many different notations have been put forward for specifying the syntax of programming languages; yet another one is presented in [19] and its advantages are illustrated.

5. COMPILER TECHNIQUES

To some extent the user's view of a programming language will be coloured by the way in which a compiler writer has implemented a compiler for the language. It is certainly true that a good compiler can help to make a poor language tolerable whereas a poor compiler can deter users from using a good programming language. This section concentrates, therefore, on compiler techniques.

One well-known technique for parsing programming languages, the operator precedence parsing method, suffers from the problem that

unless ad hoc routines are added to the parser, it is capable of accepting strings which are not valid strings of the language. In an attempt to overcome this problem Henderson and Levy proposed the extended operator precedence method. However, this latter method is not quite as simple as they describe, as demonstrated in [20] and [21]. [22] presents an algorithm which can be employed to test whether or not the extended operator precedence technique is applicable for any particular grammar.

Another set of compiler techniques is concerned with optimising the code generated by a compiler. One particular form of optimisation applies to computer systems in which some symbolic opcodes may be translated as either long form or short form instructions in machine code depending on the location of the target. An algorithm for performing long/short address optimisation is described in [23].

The programming language BASIC lends itself to interpretation and the first translators for this language were all interpreters. [24] reports on one of the first compilers developed for the language and provides some details of the approaches used in its implementation. This particular system was used by a number of universities both in South Africa and in the United Kingdom.

At the meta-level there has been considerable interest in compiler-compilers in the past. [25] describes a transportable code generator generator system which was implemented as part of a compiler-compiler project.

6. PROLOG AND DATABASES

As mentioned previously there has been considerable interest in the programming language Prolog which has proved to be suitable for a variety of applications. In particular, it has proved to be well-suited to implementing database applications. [26] describes how it was used to implement the database query language Query-by-Example whereas [27] outlines a generalised approach to security and integrity constraints in QBE which was implemented in Prolog.

[28] discusses its suitability for handling database query languages in general.

One of the advantages of Prolog in this domain is that it offers the possibility of easily extending database applications to provide a degree of intelligence within the applications. In this respect [29] discusses ways in which QBE can be made more co-operative (a step towards more "intelligent" query languages). [30] describes another application, namely that of ensuring that with changing staff and changing circumstances the authorisations allocated to individual staff do not lead to situations in which a user could misuse the system to perform incompatible duties and possibly commit fraud or embezzlement. [31] considers the problem of automatic reorganisation of relations in a simple relational database, and once again is based on a rule-based system implemented in Prolog.

While the research into using Prolog for database applications showed clearly its suitability for this purpose and the added advantages which could accrue from using it to realise intelligent extensions to such applications, nevertheless one major problem exists with this approach - namely, that of efficiency and performance. Prolog language systems (compilers or interpreters) do not provide sufficient performance in the generated code to compete against purpose-built database systems. This led naturally to research into improving the performance of Prolog systems for database applications. [32] describes some initial ideas on the design of a Prolog database machine. At this stage preliminary results on performance for the loading of clauses looked promising. This work was continued a step further to the implementation of a general Prolog database engine capable of supporting a deductive database system (see next section), an overview of which is given in [33]. This work was further extended to enable concurrent logic programming languages (e.g. Concurrent Prolog or GHC) to interact with the Prolog system and store clauses in a common database (see [34]). To evaluate this system a set of benchmarks was developed which are geared specifically to the use of Prolog for database applications [35,36].

To further improve performance the use of special purpose hardware has been considered and experimented with. Initial ideas on the design of a hardware engine to accelerate clause retrieval from secondary storage are described in [37] while [38] explores different aspects of this design such as the bus/disk architecture. As a result of this study a hardware engine was designed and implemented, and used in conjunction with the Prolog database system developed in software. The hardware engine is described in [39], [40] and [41]. Unfortunately, although the hardware performed according to specification, the operating system (UNIX) proved to be a bottleneck which prevented the full benefit of the hardware from being realised. In practice, instead of obtaining performance improvements of the order of two orders of magnitude (which the hardware provided), improvements were limited to a factor of two or three - due entirely to the operating system. A comparison of the performance of the hardware against various software techniques is described in [42].

On a slightly different tack, an investigation of CAD applications was conducted. To test the suitability of Prolog for CAD applications, a comparative study was performed with Pascal as a control. The result is reported in [43].

7. DEDUCTIVE DATABASES

Research into the use of Prolog for database applications leads naturally to research into databases themselves, and, in particular, into deductive databases. Some issues in the development of deductive databases are described in [44] while a survey of the different approaches to dealing with indefinite and negative data in deductive databases is given in [45].

One particular research interest has been the handling of incomplete information in such systems. Preliminary work on the incorporation of notions of incomplete information into relational database systems (via the language QBE) is described in [46]. This work has been extended to deductive database systems, and a

model for handling incomplete information in such systems is described in [47]. More detailed algorithms for handling incomplete information in a deductive database are given in [48] while an evaluation of two different approaches (set-oriented and tuple-oriented) is given in [49]. Since then this work has been further extended to combine the handling of time with that of incomplete information, as described in [50]. The design of the CLARE clause retrieval engine (described in the previous section) has been extended to include set handling in [51], as a first step to handling incomplete information.

Another avenue of research has been the handling of integrity constraints in deductive databases. A new method for checking integrity constraints in deductive databases is put forward in [52], and an evaluation of the performance of different methods is given in [54]. One problem with current integrity checking methods is that they are restricted to closed first-order formulae and are only applicable to definite deductive databases. [53] considers how this restriction can be relaxed and methods generalised appropriately. A slightly different use of integrity constraints to control the handling of recursion is described in [55].

These ideas have been incorporated into a deductive database system which has been implemented in Prolog. The query language for this system is described in [56]. The system has been used to provide a knowledge management system for several different expert systems. This enables different expert systems with different knowledge representation formats to store knowledge in a common format so that it can be shared between them. This is described in [57] and [58].

One recent focus of this research has been on adapting deductive databases for parallel computer systems. One of the problems with parallel database systems is that of data placement, i.e. deciding how to split relations (or predicates) and which processors to allocate them to. One method for doing this which is suited to shared-nothing main memory parallel database systems is described in [59]. Another type of parallel system is the PDBM

system, for which the problem of cache management is discussed in [60].

Another recent focus of this research has been aimed at finding a clean and consistent way of combining the deductive approach with the object-oriented approach to produce a sound theoretical model for deductive object-oriented database systems (DOODs). To this end a logical model has been developed for the object-oriented database approach which can be combined with the deductive database model to produce a single model for deductive object-oriented database systems. The first reports on this research are given in [61] and [93]. Initial reactions to this work have been encouraging and further papers have been submitted giving more details of it.

8.PROGRAM DEVELOPMENT AIDS

Operating systems have been developed for several different reasons, the most important of which must undoubtedly be to assist the programmer. On the subject of operating systems, [62] reports on a network which was developed to connect an ICL 1904S, an Interdata 6/16 and a number of Motorola 6809 microcomputers. [63] addresses one aspect of the problem of security in operating systems which is often ignored but which stands in the way of total security, and [64] describes a single-stepping debugging aid which was developed for the SWTBUG monitor program for 6800 microcomputers.

Simulator programs are useful both for software development and for teaching about machines and system software. In [65] a generalised machine simulator is described which can be set up to simulate the operation of a wide range of machines while [66] discusses a simulator which simulates the operation of an assembler. Both programs have been used for teaching, one in a course on computer architecture, the other in a course on systems programming. [67] describes a machine cycle simulator for the 68000 microprocessor.

9. AUTOMATIC PROGRAM GENERATION

One idea which has attracted a number of researchers is the ultimate goal of programmerless systems. Obviously this is a long term goal which may never be accomplished completely - nevertheless small steps in this direction may be achieved by organising the knowledge associated with particular application areas. [68] describes an approach which was followed in one project aimed at producing a generalised system. [69] gives some detail on a specific system developed to generate payroll programs. The system, which was geared towards the South African situation in many respects, was implemented in Pascal on an ICL 1900 system. Unfortunately, the system ran into difficulties with large applications owing to space limitations imposed by the Pascal system which could not be overcome at the time.

In a separate project of a rather different nature, a system was developed to assist researchers who wish to conduct a survey. The survey analyst interacts with the system at an early stage and the system generates a questionnaire for use in the survey. Once the data have been collected, the system generates a tailor-made verifier program which can be used when the data are entered to check that the value in each field conforms to the format and lies within the range expected for that field. Finally the system generates a set of SPSS parameters to analyse the data. The system is described in [70].

10. KNOWLEDGE BASED SYSTEM APPLICATIONS

One of the main application areas for Prolog is knowledge based systems applications. This section describes three such applications - one in the area of chemical analysis, one in manufacturing engineering and the third in health care.

A spectrofluorimeter is a device which is used to analyse the chemical composition of a given sample. However, the interpretation of a fluorogram is not always easy, particularly if the sample consists of a mixture of a number of different substances.

[71], [72] and [73] are concerned with a system developed to control a spectrofluorimeter and used to assist in derivative and synchronous scanning fluorescence spectroscopy.

The ideas developed for spectrofluorimetry were extended to handle the more complex case of high performance liquid chromatography (HPLC). In HPLC one pumps a solution of some sample (control or unknown) through a column filled with some packing material. This causes a separation of different components of the sample by retarding the flow of different substances in the mix. As the solution emerges from the column, it is analysed by an array of photodiodes, which provide a similar form of analysis to spectrofluorimetry. Thus by varying the solvent used, the width and length of the column, the nature of the packing material, and the pressure provided by the pump, one can obtain different results.

The problem in analysis by HPLC is first to decide on which chemicals are thought to be present in a sample, and then to confirm this by choosing parameter values which will optimise the separation of components. An expert system has been designed and implemented to perform these functions, and is described in [74], [75] and [76]. These ideas are extended to reversed-phase HPLC in [77] and [78], and the techniques for assessing peak homogeneity are described in [79] and [80]. Biomedical applications of the ideas are described in [81].

The second application lies in the field of manufacturing engineering and is concerned with process planning. This is an activity which takes place between design and manufacture, and is concerned with deciding how best to manufacture the subject of the design. The process is much simpler than that of design but nevertheless requires specialised expertise. In Ferranti (our industrial collaborator) parts are generally machined from blocks of metal. Thus, given a design, one must determine the size of the initial block of metal and the set of machining operations required to produce the final object - taking account of different tool sizes, set-ups, etc. The problem is ideally suited to a knowledge based

system approach, and [82] and [83] describe some of the work done on this application.

The third application lies in the field of medicine, and is concerned with the use of information technology to facilitate the practice of medicine at a distance - a concept known as telemedicine. In particular, we were concerned with monitoring the state of the foetus in pregnant women in their home environment to avoid unnecessary and extended hospitalisation of women deemed to be "at risk" and about whom there is the slightest concern that they might lose the foetus. The system involved the use of sensors, communication, databases and expert systems. We were involved in the development of the latter three components. The work is described in [84]. Subsequently the work on telemedicine has been extended to consider the wider problem of providing advice and support to different types of individuals interested in mounting telemedicine services. This takes the form of a framework for telemedicine services [85].

Another focus of the work done in the medical area has been on medical databases. [86] describes work done on the development of a multimedia medical interface to connect to an Extended SQL interface to a parallel database system, while [87] and [88] describe work done on connecting together heterogeneous medical databases.

11. OTHER APPLICATIONS

Finally the one problem which has captured the attention of more Computer Scientists than any other single problem must surely be that of colouring graphs or maps. One particular aspect of this problem to which Computer Scientists have made a contribution is in developing efficient algorithms for colouring graphs with as few colours as possible (though obviously with no guarantee of optimality). In view of its relevance to scheduling or timetabling problems, this is included as the final topic here.

One method proposed by Lipton and Miller can colour any planar map with at most five colours in O(n log n) time. The proof of this method as given by Lipton and Miller makes certain simple assumptions which result in a minimum batch size which is unnecessarily low. [89] presents an improved estimate for the lower bound of this quantity. A better method which will colour a planar map with at most five colours in O(n) time is described in [90].

In studying the problem of colouring planar maps, it was found that not all the map configurations expected from the formula are physically realisable. This together with some properties of planar graphs is reported in [91].

Numerous methods are now available for colouring graphs. In order to assist implementors wishing to use these algorithms, a comparative study of the performance of ten of these algorithms is reported in [92].

LIST OF PUBLICATIONS

1. M.H.WILLIAMS, Generating structured flow diagrams: the nature of unstructuredness, The Computer Journal, vol. 20, no. 1, 45-50 (1977).

2. M.H.WILLIAMS and H.L.OSSHER, Conversion of unstructured flow diagrams to structured form, The Computer Journal, vol. 21, 161-167 (1978).

3. M.H.WILLIAMS, A comment on the decomposition of flowchart schemata, The Computer Journal, vol. 25, no. 3, 393-396 (1982).

4. M.H.WILLIAMS, Flowchart schemata and the problem of nomenclature, The Computer Journal, vol. 26, no. 3, 270-276 (1983).

5. M.H.WILLIAMS and G.CHEN, Restructuring Pascal programs containing goto statements, The Computer Journal, vol. 28, no. 2, 134-137 (1985).

6. M.H.WILLIAMS and A.R.BULMER, Array handling in Cobol compilers, Software Practice and Experience, vol. 7, 469-474 (1977).

7. M.H.WILLIAMS, Is an exit statement sufficient?, Information Processing Letters, vol. 17, 47-51 (1983).

8. M.H.WILLIAMS, Yet another conversational programming language - BPL, Proceedings of the ACM 78 conference, ACM, New York, 516-521 (1978).

9. M.H.WILLIAMS, Design principles of the language BPL, Quaestiones Informaticae, vol. 1, no. 2, 23-25 (1979).

10. M.H.WILLIAMS, The programming language BPL, The Computer Journal, vol. 25, no. 3, 289-306 (1982).

11. M.H.WILLIAMS and G.CHEN, Translating Pascal for execution on a Prolog-based system, The Computer Journal, vol. 29, no. 3, 246-250 (1986).

12. G.CHEN and M.H.WILLIAMS, Executing Pascal programs on a Prolog architecture, Information and Software Technology, vol. 29, no. 6, 285-290 (1987).

13. G.CHEN and M.H.WILLIAMS, The value of an array facility in Prolog, Information Processing Letters, vol. 23, no. 5, 247-251 (1986).

14. M.H.WILLIAMS, A formal notation for specifying static semantic rules, Computer Languages, vol. 5, no. 1, 37-55 (1980).

15. M.H.WILLIAMS, Static semantic features of Algol60 and Basic, The Computer Journal, vol. 21, no. 3, 234-242 (1978).

16. M.H.WILLIAMS, Methods for specifying static semantics, Computer Languages, vol. 6, no. 1, 1-17 (1981).

17. M.H.WILLIAMS and A.R.BULMER, The use of a formal notation for static semantics in compiler design, Software Practice and Experience,vol. 8, 579-584 (1978).

18. M.H.WILLIAMS, Structuring two-level grammar specifications, The Computer Journal, vol. 28, no. 3, 250-256 (1985).

19. M.H.WILLIAMS, A flexible notation for syntactic definitions, ACM Transactions on Programming Languages and Systems, vol. 4, no. 1, 113-119 (1982).

20. M.H.WILLIAMS, Complete operator precedence conditions, Information Processing Letters, vol. 6, no. 2, 60-62 (1977).

21. M.H.WILLIAMS, Conditions for extended operator precedence parsing, The Computer Journal, vol. 22, no. 2, 164-168 (1979).

22. M.H.WILLIAMS, A systematic test for extended operator precedence, Information Processing Letters, vol. 13, nos. 4-5, 187-190 (1981).

23. M.H.WILLIAMS, Long/short address optimisation in assemblers, Software Practice and Experience, vol. 9, no. 3, 227-235 (1979).

24. M.H.WILLIAMS and J.M.MULLINS, A Basic compiler for undergraduate programs, Systems/Stelsels, vol. 4, no. 2, 8-16 (1974).

25. M.H.WILLIAMS and A.R.BULMER, A transportable code generator system, Information Processing Letters, vol. 9, no. 3, 122-125 (1979).

26. J.C.NEVES, S.O.ANDERSON and M.H.WILLIAMS, A Prolog implementation of Query-by-Example, Proceedings of the Seventh International Computing Symposium, ed. H.J.Schneider, B.G.Teubner, Stuttgart, 318-332 (1983).

27. M.H.WILLIAMS, J.C.NEVES and S.O.ANDERSON, Security and integrity in logic data bases using Query-by-Example, Proceedings of Logic Programming Workshop '83, ed. L.M.Pereira, Universidade Nova de Lisboa, Lisbon, 303-340 (1983).

28. F.MOUTA, M.H.WILLIAMS and J.C.F.M.NEVES, Implementing query languages in Prolog, in 'Prolog and Databases: Implementations and New Directions', Ed. P.M.D.Gray and R.J.Lucas, Ellis Horwood, Chichester, 13-21 (1988).

29. J.C.F.M.NEVES and M.H.WILLIAMS, Towards a co-operative data base management system, Proceedings of Logic Programming Workshop '83, Ed. L.M.Pereira, Universidade Nova de Lisboa, Lisbon, 341-370 (1983).

30. M.H.WILLIAMS and J.RIDDALL, The segregation of incompatible duties, The Computer Journal, vol. 29, no. 1, 9-11 (1986).

31. M.H.WILLIAMS, I.M.PATTISON and J.C.F.M.NEVES, Reorganization in a simple database system, Software Practice and Experience, vol. 16, no. 8, 719-729 (1986).

32. M.H.WILLIAMS, The design of a Prolog database machine, Proceedings of the First Workshop of the Special Interest Group in Knowledge Manipulation Engines, Alvey Directorate, L1-5 (1987).

33. M.H.WILLIAMS, G.CHEN, D.FERBRACHE, P.MASSEY, S.SALVINI, H.TAYLOR and K.F.WONG, Prolog and deductive databases, Knowledge-Based Systems, vol. 1, no. 3, 188-192 (1988).

34. H.TAYLOR and M.H.WILLIAMS, Applying coupled resolution engines to knowledge bases, Proceedings of the UK IT 1990 Conference, IEE/BCS, Southampton, 379-386 (1990).

35. M.H.WILLIAMS, P.A.MASSEY and J.A.CRAMMOND, Benchmarks for Prolog from a database viewpoint, Proceedings of the Second Workshop of the Special Interest Group for Knowledge Manipulation Engines, Alvey Directorate, C1-8 (1987).

36. M.H.WILLIAMS, P.A.MASSEY and J.A.CRAMMOND, Benchmarking Prolog for database applications, in 'Prolog and Databases: Implementations and New Directions', Ed. P.M.D.Gray and R.J.Lucas, Ellis Horwood, Chichester, 161-187 (1988).

37. K.F.WONG and M.H.WILLIAMS, Limiting a resolution set in a large Prolog database using special hardware, Proceedings of the ACM SIGSMALL/PC Symposium on Personal and Small Computers, Cannes, France, 133-145 (1988).

38. K.F.WONG and M.H.WILLIAMS, Design considerations for a Prolog engine, Proceedings of the Third International Conference on Data and Knowledge Bases, Jerusalem, Israel, 111-120 (1988).

39. K.F.WONG and M.H.WILLIAMS, A type driven hardware engine for Prolog clause retrieval over a large knowledge base, Proceedings of the Sixteenth International Symposium on Computer Architecture, Jerusalem, 211-222 (1989).

40. K.F.WONG and M.H.WILLIAMS, CLARE - a clause retrieval engine for large Prolog databases, Proceedings of the IEE Colloquium on VLSI and Architecture for Symbolic Processing, IEE Digest No. 1989/37, 3/1-3/3 (1989).

41. K.F.WONG and M.H.WILLIAMS, CLARE - A Prolog database machine, in Proceedings of 1990 Sigsmall/PC Symposium on Small Systems, Arlington, Virginia, USA, 29-30 March 1990, 172-177 (1990).

42. S.ZHOU and M.H.WILLIAMS, Assessment of two-level codeword indexing applied to Prolog databases, Information and Software Technology, vol. 33, 157-162 (1991).

43. J.CAMACHO GONZALEZ,M.H.WILLIAMS and I.E.AITCHISON, Evaluation of the effectiveness of Prolog for a CAD application, IEEE Computer Graphics and Applications, vol. 4, no. 3, 67-75 (1984).

44. M.H.WILLIAMS, Towards deductive database systems, Proceedings of the Third Portuguese Conference on Artificial Intelligence (EPIA), University of Minho, Braga, Portugal, 365-374 (1987).

45. P.CHISHOLM, G.CHEN, D.FERBRACHE, P.THANISCH and M.H.WILLIAMS, Coping with indefinite and negative data in deductive databases: a survey, Data and Knowledge Engineering, vol. 2, 259-284 (1987).

46. M.H.WILLIAMS and K.A.NICHOLSON, An approach to handling incomplete information in databases, The Computer Journal, vol. 31, no. 2, 133-140 (1988).

47. M.H.WILLIAMS and Q.KONG, Incomplete information in a deductive database, Data and Knowledge Engineering, vol. 3,197-220 (1988).

48. M.H.WILLIAMS, Q.KONG and G.CHEN, Handling incomplete information in a logic database, Proceedings of the UK Information Technology '88 Conference, IEE/BCS, Swansea, 224-227 (1988).

49. Q.KONG and M.H.WILLIAMS, Evaluating different strategies for handling incomplete information in a logic database, in 'Logic Programming Expanding the Horizons', Eds. T.Dodd, R.Owens and S.Torrance, Intellect Books, Oxford, 124-146 (1990).

50. M.H.WILLIAMS and Q.KONG, Time and incompleteness in a deductive database, Proceedings of the Third International Conference on Information Processing and the Management of Uncertainty in Knowledge Based Systems, Eds. B.Bouchon-Meunier, R.R.Yager and L.A.Zadeh, Springer Verlag Lecture Notes in Computer Science, No. 521, 443-455 (1991).

51. K.F.WONG and M.H.WILLIAMS, A Superimposed Codeword indexing scheme for handling sets in Prolog databases, in Proceedings of International Symposium on Database Systems for Advanced Applications, Tokyo, Japan, 468-476, 1991.

52. S.K.DAS and M.H.WILLIAMS, A path finding method for constraint checking in deductive databases, Data and Knowledge Engineering, vol 4, 223-244 (1989).

53. S.K.DAS and M.H.WILLIAMS, Extending the integrity maintenance capability in deductive databases, Logic Programming New Frontiers, Ed. D.R.Brough, Intellect Books, Oxford, 75-111 (1992).

54. S.K.DAS and M.H.WILLIAMS, Integrity checking methods in deductive databases: a comparative evaluation, Proceedings of the Seventh British National Conference on Databases, Cambridge University Press, 85-116 (1989).

55. K.WICOVSKY, P.F.THANISCH and M.H.WILLIAMS, Improved recursion handling through integrity constraints, The Computer Journal, vol. 34, 282-285 (1991).

56. K.G.WAUGH, M.H.WILLIAMS, Q.KONG, S.SALVINI and G.CHEN, Designing SQUIRREL: an extended SQL for a deductive database system, The Computer Journal, vol. 33, 535-546 (1990).

57. S.SALVINI and M.H.WILLIAMS, Knowledge management for expert systems, IEE Digest No. 1990/077, 3/1-3/3 (1990).

58. S.SALVINI and M.H.WILLIAMS, Central knowledge management for expert systems, Mathematical and Computer Modelling, vol. 16, nos. 6/7, 137-144 (1992).

59. M.B.IBANEZ-ESPIGA and M.H.WILLIAMS, Data placement strategy for a parallel database system, Proceedings of the International Conference on Database and Expert Systems Applications '92, Eds. A.M.Tjoa and I.Ramos, Springer-Verlag,48-54 (1992).

60. S.ZHOU and M.H.WILLIAMS,Cache management for retrieved solutions in a Prolog database system,to appear in Proceedings of 14th Colloquium on Information Retrieval, Lancaster University, April 1992, (1993).

61. M.H.WILLIAMS, N.W.PATON, A.ABDELMOTY, A.A.A.FERNANDES and M.L.BARJA, Applying advanced database techniques to geographic information systems, Proceedings of JFIT Conference 93, to appear in March 1993.

62. M.H.WILLIAMS, A.G.SARTORI-ANGUS and G.P.STROMBERG, Alternative approach to a microprocessor

teaching laboratory, Microprocessors and Microsystems, vol. 4, no. 7, 219-222 (1980).

63. M.H.WILLIAMS, The problem of absolute privacy, Information Processing Letters, vol. 17, 169-171 (1983).

64. M.H.WILLIAMS, A useful trace facility for microcomputers, Microprocessors and Microsystems, vol. 5, no. 3, 99-102 (1981).

65. M.H.WILLIAMS and H.L.OSSHER, A computer model for instructional purposes, The Computer Journal, vol. 18, no. 4, 333-341 (1975).

66. M.H.WILLIAMS, G.R.POTE and J.P.BROOKS, A simulator for teaching the internal workings of an assembler, Computers in Education, vol. 7, 55-63 (1983).

67. M.H.WILLIAMS and A.J.STEWART, Machine cycle simulator for a microprocessor, Microprocessors and Microsystems, vol. 6, no. 3, 131-134 (1982).

68.M.H.WILLIAMS, A question answering system for automatic program synthesis, SIGPLAN Notices, vol. 11, 63-68 (1976).

69. M.H.WILLIAMS and E.V.C.FIELDING, Automatic generation of payroll programs, The Computer Journal, vol. 21, 378-380 (1978).

70. M.H.WILLIAMS, A system to assist survey analysts, Software Practice and Experience, vol. 12, 487-491 (1982).

71. K.T.MILNE, M.H.WILLIAMS, B.J.CLARK and A.F.FELL, An expert system for the analysis of fluorescent substances, Proceedings of the Sixth International Workshop on Expert Systems and their Applications, Agence de L'Informatique, Avignon, vol. 2, 1203-1217 (1986).

72. K.T.MILNE, M.H.WILLIAMS, B.J.CLARK and A.F.FELL, Expert systems in luminescence analysis, Analytical Proceedings, vol. 23, 157-160 (1986).

73. B.J.CLARK, A.F.FELL, K.T.MILNE, D.M.G.PATTIE and M.H.WILLIAMS, Pharmaceutical applications of variable-angle synchronous scanning fluorescence spectroscopy, Analytica Chimica Acta, vol. 170, 35-44 (1985).

74. T.P.BRIDGE, M.H.WILLIAMS and A.F.FELL, Analytical chemistry and expert systems, Chemistry in Britain, vol. 23, no. 11, 1085-1088 (1987).

75. T.P.BRIDGE, M.H.WILLIAMS, G.G.R.SEATON and A.F.FELL, An expert system for eluent optimization in liquid chromatography with photodiode array detection, Chromatographia, vol. 24, 691-695 (1987).

76. T.P.BRIDGE, M.H.WILLIAMS and A.F.FELL, Expert systems design in high performance liquid chromatography, Analytical Proceedings, vol. 25, 43-44 (1988).

77. A.F.FELL, T.P.BRIDGE and M.H.WILLIAMS, Design and application of an expert system for eluent optimization in reversed-phase high performance liquid chromatography, J. Pharm. Biomed. Anal., vol. 6, 555-564 (1988).

78. T.P.BRIDGE, M.H.WILLIAMS and A.F.FELL, An expert system approach to eluent optimization in RPLC with multichannel detection, Journal of Liquid Chromatography, vol. 12, no. 1, 23-34 (1989).

79. T.P.BRIDGE, M.H.WILLIAMS and A.F.FELL, Comparative evaluation of methods for assessing peak homogeneity in liquid isocratic reverse-phase chromatography, Anal. Chimica Acta, vol. 223, 175-182 (1989).

80. T.P.BRIDGE, M.H.WILLIAMS and A.F.FELL, Artificial intelligence techniques for peak homogeneity validation in liquid chromatography, Journal of Chromatography, vol. 465, 59-67 (1989).

81. B.J.CLARK, A.F.FELL and M.H.WILLIAMS, Biomedical applications of expert systems in analytical spectroscopy, Advances in Multidimensional Luminescence, vol 1, Eds. I.M.Warner and L.B.McGown, JAI Press Inc., Connecticut, 55-72 (1991).

82. J.L.MURRAY and M.H.WILLIAMS, Knowledge based systems in process planning and assembly design, in 'Geometric Reasoning', Ed. J.Woodwark, Clarendon Press, Oxford, 217-236 (1989).

83. D.WILLIS, I.A.DONALDSON, A.D.RAMAGE, J.L.MURRAY and M.H.WILLIAMS, A knowledge-based system for process planning based on a solid modeller, Computer Aided Engineering Journal, vol. 6, no. 1, 21-26 (1989).

84. J.H.DRIPPS, S.SALVINI, M.H.WILLIAMS, W.A.FULTON, K.BODDY and G.A.VENTERS, Telemedicine in obstetrics, Proceedings of Medical Informatics Europe '90, Glasgow, 501-505 (1990).

85. M.H.WILLIAMS, B.MAHR, G.VENTERS and D.LUTZEBACK, A framework for telemedicine services, Proceedings of Medical Informatics Europe 93, to appear in April 1993.

86. M.H.WILLIAMS and G.MUIRHEAD, Database query interface for medical information systems, Proceedings of IDS 92, Springer Verlag, to appear 1993.

87. M.H.WILLIAMS and J.HU, Sharing patient data among heterogeneous medical databases, Proceedings of Medical Informatics Europe, to appear in April 1993.

88. M.H.WILLIAMS and J.HU, Communication between heterogeneous medical data bases, Journal of Information Science and Technology, vol.2, no. 1, 12-30 (1992).
89. M.H.WILLIAMS, Batch sizes for the batching method of colouring planar maps, Information Processing Letters, vol. 11, nos. 4-5, 186-189 (1980).

90. M.H.WILLIAMS, A linear algorithm for colouring planar graphs with five colours, The Computer Journal, vol. 28, no. 1, 78-81 (1985).

91. M.H.WILLIAMS, Cubic map configurations, Information Processing Letters, vol. 11, nos. 4-5, 180-185 (1980).

92. M.H.WILLIAMS and K.T.MILNE, The performance of algorithms for colouring planar graphs, The Computer Journal, vol. 27, no. 2, 165-170 (1984).

93. A.A.A.FERNANDES, N.W.PATON, M.H.WILLIAMS and A.BOWLES, Approaches to deductive object-oriented databases, to appear in Information Software and Technology (1993).

1. M.H.WILLIAMS, Generating structured flow diagrams: the nature of unstructuredness, The Computer Journal, vol. 20, no. 1, 45-50 (1977).

Generating structured flow diagrams: the nature of unstructuredness

M. H. Williams

Department of Computer Science, Rhodes University, Grahamstown 6140, South Africa

Various methods have been put forward by different authors for converting an 'unstructured flow diagram' into an equivalent 'structured' one. This paper examines the basic flow diagram sub-structures, the presence of which in a flow diagram causes the flow diagram to be unstructured, and proves that these are the only structures which lead to unstructuredness.

(Received March 1975)

1. Introduction

In the last decade there has been a growing interest in flow diagrams and program schemata, and, in particular, in the problem of determining the equivalence of flow diagrams (Ershov, 1972). Böhm and Jacopini (1966) introduced the idea of using a small set of base diagrams and showed how more general flow diagrams could be decomposed into these base diagrams. However, since it is not possible to decompose all flow diagrams into 'a finite number of given base diagrams', they proposed two normalisation methods for converting diagrams which could not be decomposed into a finite number of base diagrams, into equivalent diagrams which can be decomposed thus. This idea, viewed from the point of view of synthesis instead of analysis, namely that flow diagrams should be constructed initially from a small set of three base diagrams, has been termed structured programming and considerable success has been ascribed to this technique (Yourdon, 1974; Baker and Mills, 1973; Miller and Lindamood, 1973).

Since the terminology associated with structured programming appears to vary somewhat, for the purposes of this paper the term *structured flow diagram* will be used to refer to a flow diagram that can be decomposed completely in terms of the three base diagrams shown in **Fig. 1.** The selection mechanism generally used (Fig. 1(*b*)) is a two-way branch; however, one can extend this to an *n*-way conditional branch (Wulf, 1972) if desired. Contrariwise, an *unstructured flow diagram* is one that cannot be decomposed completely in these terms.

The notion of a method for converting an unstructured flow diagram into a structured one has attracted some interest since Böhm and Jacopini's original paper, and several methods for doing this have been put forward (Cooper, 1967; Ashcroft and Manna, 1972; Wulf, 1972). In each case the method was illustrated by one or two examples.

This paper examines the general nature of 'unstructuredness', i.e. those structures within flow diagrams which cannot be directly resolved in terms of the three base diagrams.

2. The basic causes of unstructuredness

There are five basic structures in flow diagrams which lead to nondecomposability in terms of the three base diagrams. These may be classified as:

(*a*) abnormal selection path

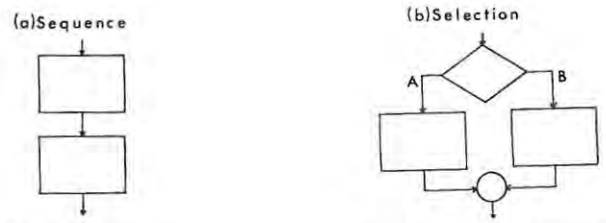
(*b*) loop with multiple exit points

(*c*) loop with multiple entry points

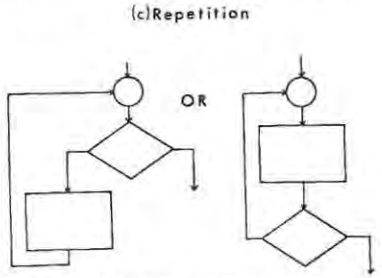
(*d*) overlapping loops, and

(*e*) parallel loops.

These five basic constructs are illustrated in **Fig. 2.** Cases (*d*) and (*e*) are very similar and could be combined into a single category; however, it is convenient to treat them as separate cases. If a flow diagram is unstructured, it must contain at least one (or possibly a combination) of these basic structures. In the next section it is proved that these are the only structures which cannot be decomposed in terms of the three basic diagrams in Fig. 1.



(*a*) **Simple sequence** (*b*) **Selection (if-then-else)**



(*c*) **Repetition (do-while)**

Fig. 1 The three basic structures of structured programming

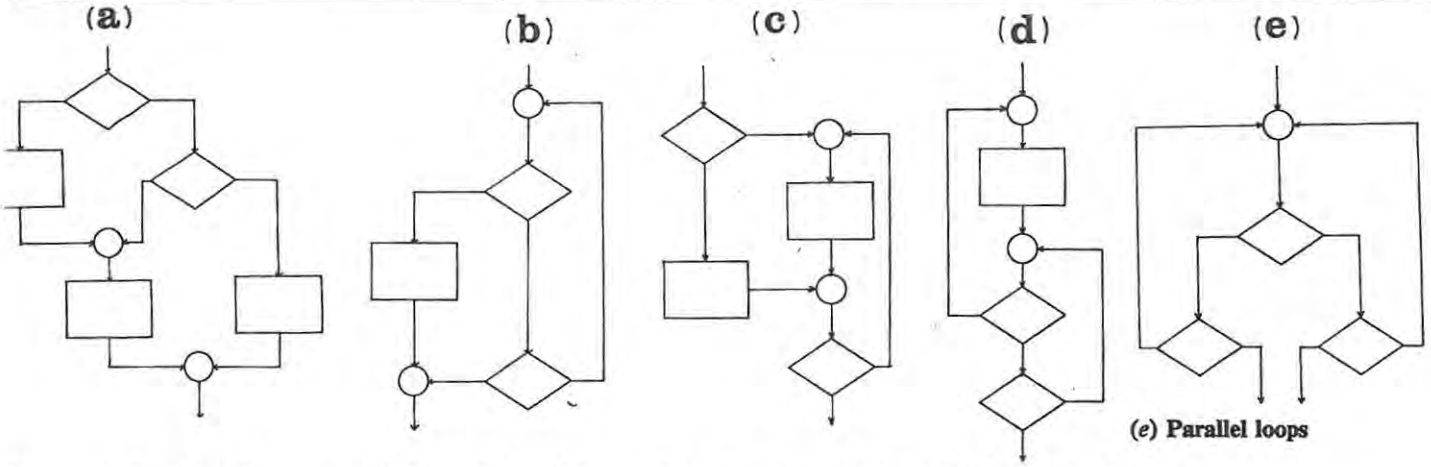
Consider some of the examples used by other authors and how these all reduce to one or more of these five structures:

1. **Fig. 3**(*a*) shows an example used by Wulf in which the line marked *x* is an abnormal exit from the selection path of *B* and the line *y* an abnormal exit from the selection path of *A*, while both selection paths of *C* have abnormal entries.
2. Fig. 3(*b*) contains an example given by Yourdon in which all four selection structures have abnormal entries or exits.
3. **Fig. 4**(*a*) demonstrates an example from Ashcroft and Manna which can be reduced to the form given in Fig. 4(*b*), a loop with multiple exit points.
4. **Fig. 5**(*a*) is an example from the paper by Böhm and Jacopini (Ω_5). This is rearranged without alteration in Fig. 5(*b*) to show clearly that it is simply a loop with multiple exit points.
5. **Fig. 6**(*a*) displays an example taken from Mullins *et al.* (1974). Fig. 6(*b*) contains a similar example taken from the paper by Böhm and Jacopini (their Type II). Both examples correspond to the parallel loops class of structure.
6. **Fig. 7** is an example used by Yourdon which illustrates overlapping loops.

3. Proof of existence of five unstructured forms

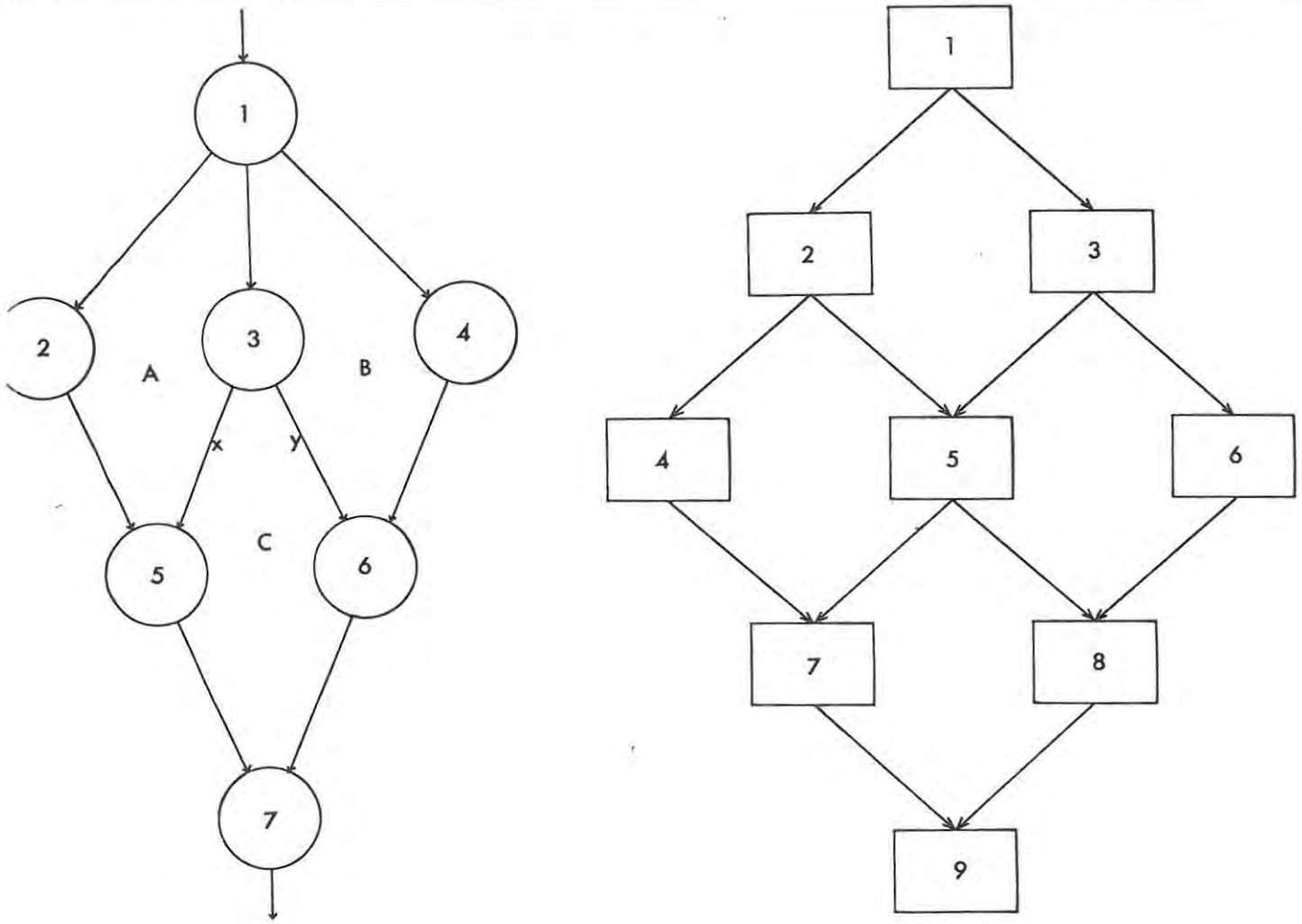
Consider any flow diagram consisting of a collection of boxes and lines. Assume that there are five different types of boxes

RHODES UNIVERSITY



Abnormal selection path | (*b*) Loop with multiple exit points | (*c*) Loop with multiple entry points | (*d*) Overlapping loops

. 2 The five basic structures which cause unstructured flow diagrams



ig. 3 Examples of abnormal selection paths given by: (*a*) Wulf (1972) (*b*) Yourdon (1974)

ee Fig. 8), viz.

) decision box

) process box

) junction box

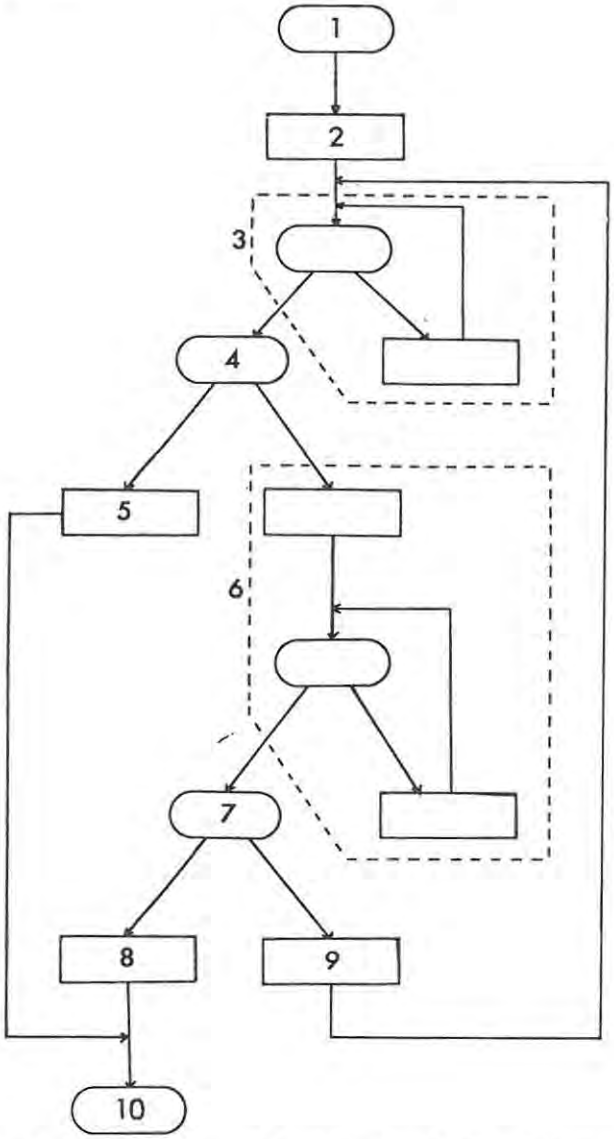
) START box, and

) FINISH box

nd that the flow diagram has only one entry point, the START ox, and one exit point, the FINISH box.

Define a *path* as a sequence of successive directed lines through the diagram. The *length* of the path is the number of lines in the sequence. For each line (or box) in the diagram, there are a set of paths through the diagram from the START box to the line (or box) in question. These will be called *origin paths*. The *level* of a line or box is the length of its shortest origin path (the shorter the length, the lower the level).

A line is said to lie within a *loop* if there exist for that line two origin paths, A with path length n and B with path length $m\,(>n)$, such that the first n lines of the sequence B are identical to those of the sequence A.



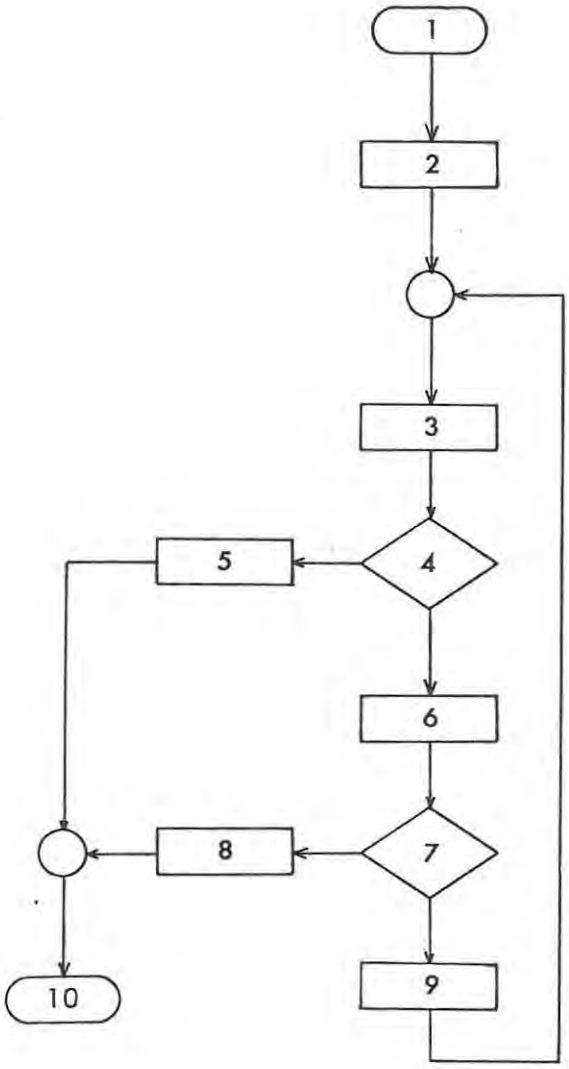


Fig. 4 (*a*) **Example used by Ashcroft and Manna (1972)**

(*b*) **The same example after reduction and rearrangement—an obvious example of a loop with multiple exit points**

The two paths of the selection process shown in Fig. 1(*b*) (labelled *A* and *B* in the diagram) are referred to as *selection paths*.

The process of *reducing* a flow diagram consists of replacing each occurrence of a base diagram (i.e. one of the three shown in Fig. 1) within the flow diagram by a single process box (Wulf, 1972). This is repeated until no further replacement is possible.

A flow diagram which reduces to a single process box (together with one START box and one FINISH box) is a structured flow diagram.

A further process, called *compression*, involves reducing a flow diagram and then, if any loops remain (i.e. loops which are not of the form given in Fig. 1(*c*)), replacing them by single boxes with the appropriate entry and exit lines (as shown in **Fig. 9**).

Theorem 1:
A flow diagram which contains any of the five structures in Fig. 2 is unstructured.

Proof:
By inspection it is obvious that a flow diagram which contains any of these five structures cannot be reduced to a single process box.

Theorem 2:
Given transformations to convert each of these five structures to structured form, then any flow diagram can be transformed to a structured flow diagram.

Proof:
Consider any line connecting two boxes in a *reduced* unstructured flow diagram. It can either lie within at least one loop, or it can lie outside any loops.

Case I. Outside any loops
If the line lies outside any loops, consider the set of origin paths for this line in the *compressed* flow diagram. There are two possibilities: either there is a single origin path for this line in the compressed flow diagram, or there is more than one such path.

1. *Single origin path*
This corresponds to a normal structured flow diagram path. For example, see **Fig. 10**(*a*).

2. *More than one origin path*
Since there is more than one origin path in the compressed diagram, there must be either:

(*a*) at least one decision box for which both selection paths lead

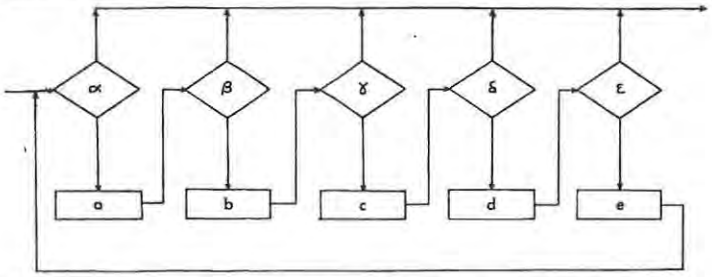
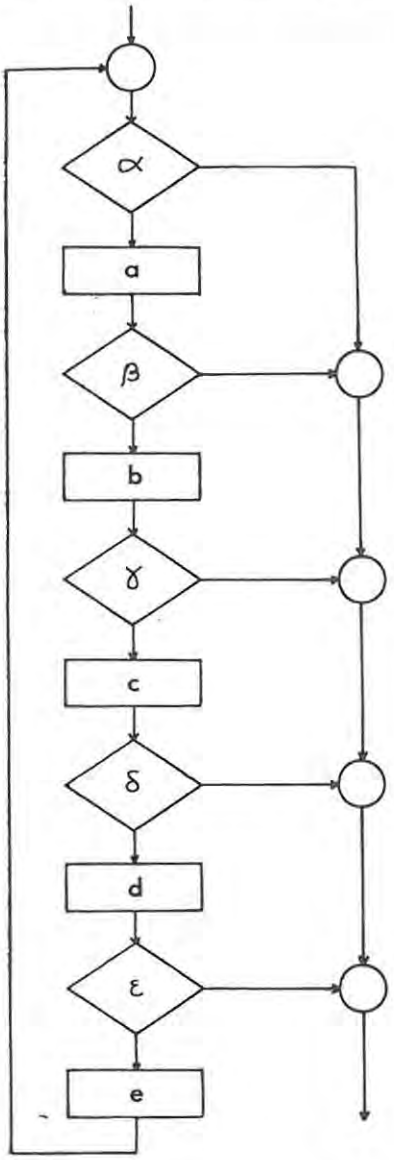


Fig. 5 (*a*) **Example (Ω_5) from paper by Böhm and Jacopini (1966)**



(*b*) **The same flow diagram, rearranged slightly, is obviously a loop with five exit points**

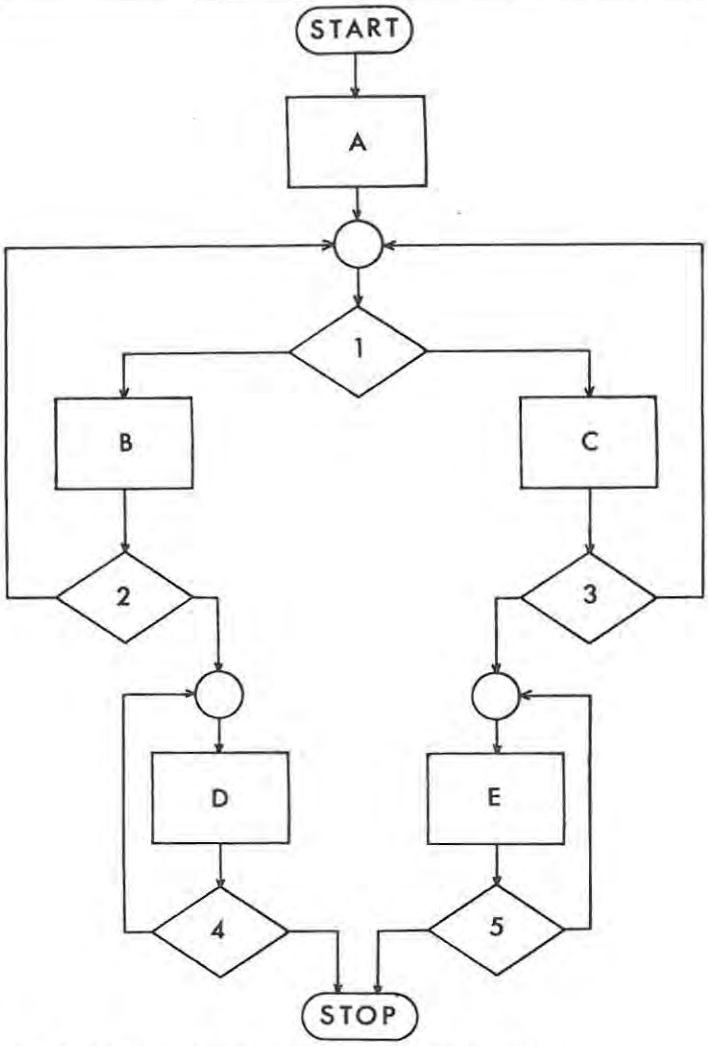
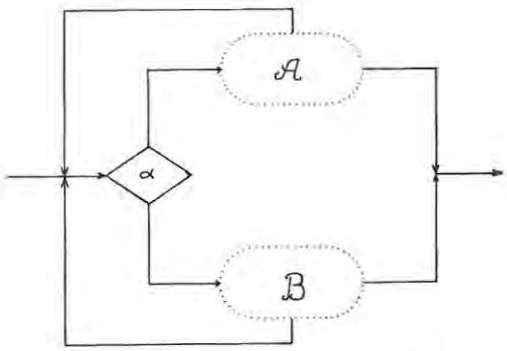


Fig. 6 (*a*) **Example taken from Mullins et al. (1974)**



(*b*) **Similar example (Type II) from Böhm and Jacopini (1966)**

to the line under consideration, or

(*b*) at least one loop-replacement box for which at least two of the exit paths lead to the line under consideration.

In the first case, since the two selection paths of the decision box lead to the same line, the two paths must join at a junction box, forming a closed circuit. However, since it has not been replaced by a single box in the reduction process, at least one of the selection paths must include some peculiarity. This could either be a branch out of the selection path which does not return (see Fig. 10(*b*)) or a branch into the selection path from outside.

In the case of a branch into the selection path from outside (labelled X in Fig. 10(*c*)), there must exist a second decision box (2) at a lower level such that the first decision box (1) lies on one of its selection paths and the line labelled X lies on the other. The result, shown in Fig. 10(*c*), is identical to the structure shown in Fig. 10(*b*).

The remaining possibility is that the line under consideration lies on two of the exit paths emanating from a loop-replacement box. If this is the case, one can replace the loop-replacement box by the original loop. Then there must exist some decision box within the loop such that the two exit paths concerned lie on its two selection paths. Hence a decision box has been found for which the two selection paths both lead to the line

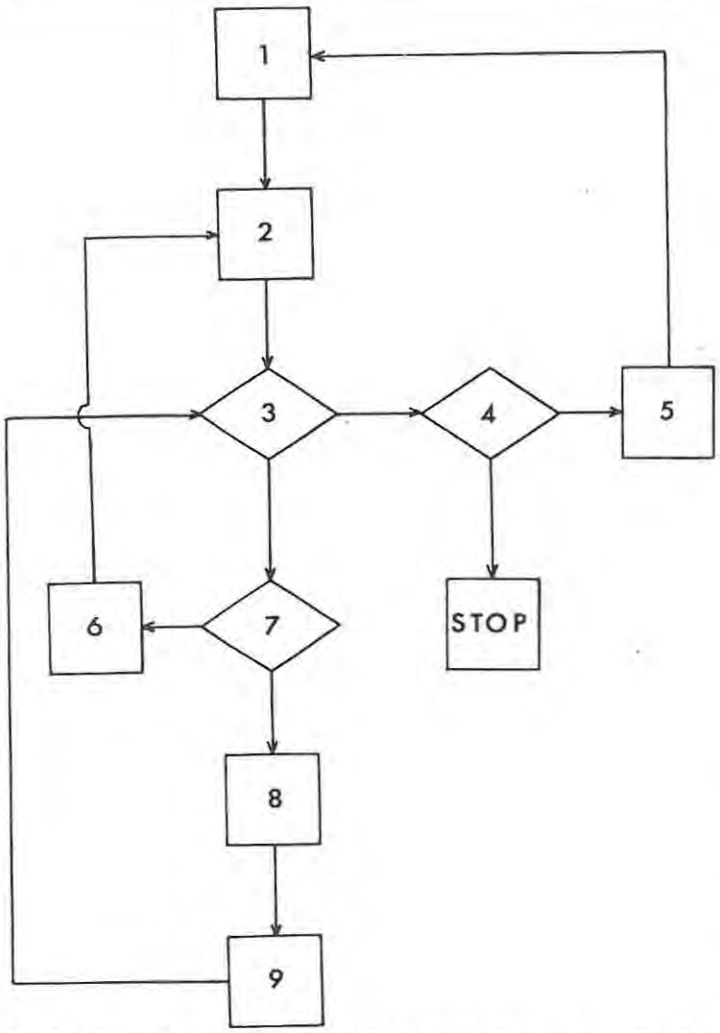


Fig. 7 Example from Yourdon (1974) which illustrates overlapping loops

under consideration, and the argument of the previous two paragraphs applies.

Thus if there is more than one origin path to a line (which is outside any loop) in the compressed diagram, this is always due to an abnormal selection path as shown in Fig. 2(*a*). Hence if a transformation exists which converts the structure in Fig. 2(*a*) to structured form, then any portion of a flow diagram which is outside any loop can be converted to structured form by such a transformation. The simplest transformation for this structure is to duplicate boxes. Alternatively one may introduce boolean variables (Wulf).

Case II. Within a loop

A structured loop (Fig. 1(*c*)) should consist of one process box, one junction box and one two-way decision box, arranged in a closed circuit with one entry point and one exit point. Any loop which has not been removed in the reduction process, does not have this form. This can only be due to one or more of the following reasons:

(*a*) the loop has more than one entry point

(*b*) the loop has more than one exit point

(*c*) the loop contains a structure with an abnormal selection path of the type dealt with in Case I, or

(*d*) the loop contains an inner loop with one of these four structures,

for if the loop has only one entry point and one exit point and does not contain any structure with an abnormal selection path or any unreducible loop structures, then the contents of the

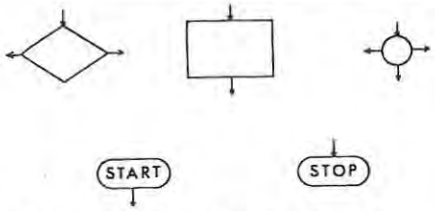


Fig. 8 The five different types of flowchart boxes

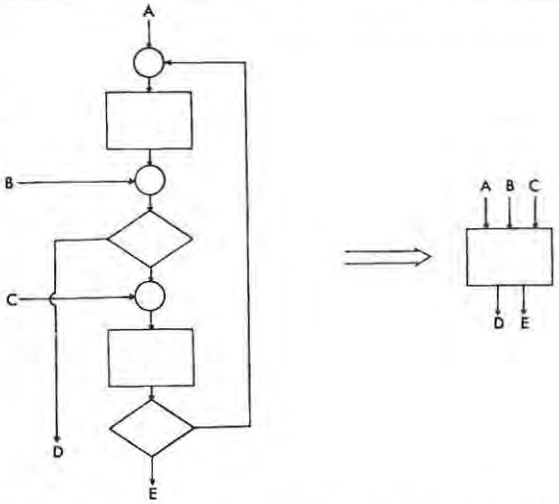


Fig. 9 The process of compression replaces all loops by single boxes. This illustrates how a loop with three entry points and two exit points is replaced by a single box

loop excluding the junction box and the decision box leading to the exit point must be reducible to a single process box.

Case (*c*) has already been dealt with and the method for normalising this to structured form has been mentioned. Thus if one considers the innermost unreducible loop, only cases (*a*) and (*b*) remain.

1. *More than one entry point*

Since a loop should have only one entry point, choose one of the entry points as the main entry point.* Now compress the diagram so that the loop is replaced by a single box (as in Fig. 9). For each entry point besides the main one, there are two possibilities:

(*a*) There exists a decision box at a lower level than the main entry point such that the aforementioned entry point lies on one of its selection paths and the main entry point lies on the other (see **Fig. 11**(*a*)). This corresponds to the category of 'loops with multiple entry points' (Fig. 2(*c*)).

(*b*) There exists a decision box at a higher level having the loop-replacement box on one of its origin paths and the entry point in question on one of its selection paths (see Fig. 11(*b*)). This corresponds to the case of overlapping loops (Fig. 2(*d*)).

By applying transformations for Figure 2(*c*) and (*d*) to each entry point besides the main one, the loop will be replaced by one with a single entry point.

2. *More than one exit point*

As for the case with more than one entry point, choose one exit point as the main exit point and compress the diagram, replacing the loop by a single box. In this case, for each exit point besides the main one, one can distinguish between three possibilities:

*In general this process of choosing the main entry point may have to be defined more rigorously. For the purpose of this paper, however, this process will be assumed to be defined.

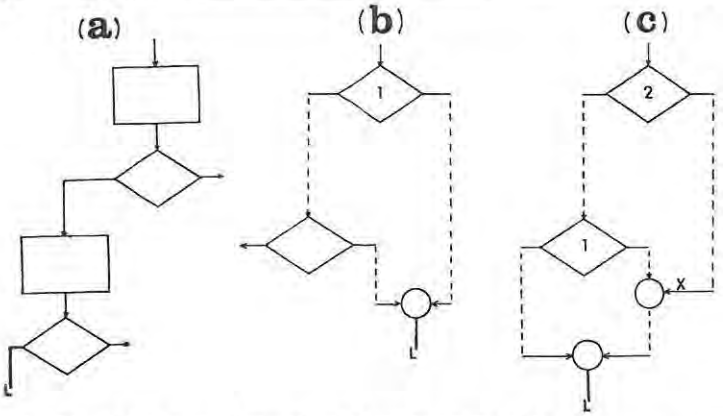


Fig. 10 (*a*) **A normal structured flow diagram path**
(*b*) **A line having two origin paths in a compressed flow diagram, caused by a branch out of the selection path of box 1**
(*c*) **A line having two origin paths in a compressed flow diagram, caused by a branch into the selection path of box 1. In each case the line marked *L* is the one under consideration**

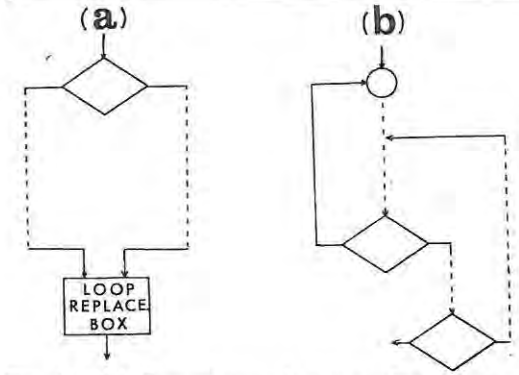


Fig. 11 (*a*) **Case when main entry point and secondary entry point lie on opposite selection paths of a decision box at a lower level**
(*b*) **Case when exit point leads to a decision box at a higher level, which in turn has a secondary entry point on one of its selection paths**

(*a*) There exists a path from the exit point under consideration to one of the entry points other than the main one. This is identical to condition (*b*) of 1. above, viz. an example of overlapping loops (Fig. 2(*d*)).

(*b*) There exists a path from the exit point under consideration to the main entry point. If the path does not pass through any boxes other than junction boxes at a lower level than the main entry point, this corresponds to the case of parallel loops (Fig. 2(*e*)). Otherwise this corresponds to the case of overlapping loops (Fig. 2(*d*)).

(*c*) No path exists from the exit point in question to any of the entry points. This corresponds to the case of a loop with multiple exit points (Fig. 2(*b*)).

By applying transformations for Figures 2(*b*), 2(*d*) and 2(*e*) to each exit point besides the main one, the loop will be replaced by one with a single exit point.

In the case of the main exit point, there are also three possibilities.

(*a*) A path exists from the main exit point to the main entry point. This corresponds to a normal structured flow diagram nested loop.

(*b*) No path exists from the main exit point to any of the entry points. This corresponds to a normal structured flow diagram without an enclosing loop.

(*c*) A path exists from the main exit point to one of the entry points other than the main one. Such a path may or may not pass through a decision box. If the path does pass through at least one decision box, as shown in Fig. 11(*b*), then the resulting structure is that of overlapping loops (see Fig. 2(*d*)). This case has already been taken into account. However, if the path does not pass through any decision boxes, then this cannot be the main exit point as both paths lead directly back into the loop.

Hence, if transformations exist to convert each of the five structures in Fig. 2 to structured form, then any loop with an arbitrary number of entry and exit points can be converted to one with only one entry point and one exit point. If the boxes within the loop can still not be reduced to a single process box then the loop must either contain an inner unstructured loop or a structure of the type shown in Fig. 2(*a*). In each case by repeated application of the transformations any loop can be converted to structured form.

Thus any flow diagram can be converted to structured form by transformations on these five structures.

Hence the only possible structures which give rise to unstructured flow diagrams are those shown in Fig. 2.

4. Conclusions

There are five basic structures which lead to unstructured flow diagrams, viz.

(*a*) abnormal selection path

(*b*) loop with multiple exit points

(*c*) loop with multiple entry points

(*d*) overlapping loops, and

(*e*) parallel loops.

Examples of unstructured flow diagrams considered by different workers are shown to contain one or more of these basic structures. Thus any method for automatically converting unstructured flow diagrams into equivalent structured forms should be tested on each of these five basic structures.

Acknowledgements

This work was done while on study leave at Cambridge University Computer Laboratory on a British Council Scholarship. The author is grateful to Dr R. M. Needham of the Computer Laboratory for reading and commenting on the paper.

References

ASHCROFT, E. and MANNA, Z. (1972). The Translation of 'GOTO' Programs to 'WHILE' Programs, in *Proceedings of IFIP Congress* 71, North-Holland Publishing Co., pp. 250-255.

BAKER, F. T. and MILLS, H. D. (1973). Chief Programming Teams, *Datamation*, Vol. 19, No. 12, pp. 58-61.

BÖHM, C. and JACOPINI, G. (1966). Flow Diagrams, Turing Machines and Languages with only Two Formation Rules, *CACM*, Vol. 9, No. 5, pp. 366-371.

COOPER, D. C. (1967). Böhm and Jacopini's Reduction of Flow Charts, *CACM*, Vol. 10, No. 8, pp. 463-473.

ERSHOV, A. P. (1972). Theory of Program Schemata, in *Proceedings of IFIP Congress* 71, North-Holland Publishing Co., pp. 28-45.

MILLER, E. F. and LINDAMOOD, G. E. (1973). Structured Programming: Top-down Approach, *Datamation*, Vol. 19, No. 12, pp. 55-57.

MULLINS, J. M., RADUE, J. E., and WEBB, G. L. (1974). A Tutorial Note on Functional Programming, *Systems/Stelsels*, Vol. 4, No. 8, pp. 13-14.

WULF, W. A. (1972). Programming without the GOTO, in *Proceedings of IFIP Congress* 71, North-Holland Publishing Co., pp. 408-413.

YOURDON, E. (1974). A Brief Look at Structured Programming, *Systems/Stelsels*, Vol. 4, No. 5, pp. 7-12.

2. M.H.WILLIAMS and H.L.OSSHER, Conversion of unstructured flow diagrams to structured form, The Computer Journal, vol. 21, 161-167 (1978).

Conversion of unstructured flow diagrams to structured form

M. H. Williams and H. L. Ossher

Department of Computer Science, Rhodes University, Grahamstown 6140, South Africa

Various workers have suggested methods for converting unstructured flow diagrams into equivalent structured ones. Some of these are compared here and a general method for performing such a conversion is discussed.

(Received November 1975, Revised October 1976)

1. Introduction

Because of the desirable properties of structured flow diagrams, the process of converting an unstructured flow diagram to an equivalent structured one has attracted the attention of several workers.

The three basic constituents of structured flow diagrams, known as *base diagrams*, are simple sequence, selection (**if—then—else**) and the **while** loop. Base diagrams can be nested to form complex flow diagrams. A flow diagram which consists entirely of a nest of base diagrams will here be referred to as *structured*, whereas one which does not will be referred to as *unstructured*. (Three other constructs are sometimes also considered basic elements of structured flow diagrams, viz. the **until** loop, the **case** construct and the **loop—while—repeat** construct.)

The intention behind designing a method to structure flow diagrams automatically is not so much to use it for improving unstructured flow diagrams *per se*, as to prove that *any* algorithm can be expressed in a reasonable structured form. However, since some workers involved in automatic programming (Cheatham and Wegbreit, 1972) are looking at the problem of rearranging a program so as to improve it, the techniques for structuring flow diagrams may not go amiss.

In a previous paper (Williams, 1976) unstructuredness in flow diagrams was shown to be due to the presence of one or more of the following:

(*a*) abnormal selection path

(*b*) loop with multiple exit points

(*c*) loop with multiple entry points

(*d*) overlapping loops

(*e*) parallel loops.

These forms are illustrated in **Fig. 1.** It is necessary and sufficient for a general structuring method to be able to eliminate them from any flow diagram.

Several workers have put forward methods for converting unstructured flow diagrams to structured form (Böhm and Jacopini, 1966; Ashcroft and Manna, 1972; Wulf, 1972). A method is described below which is both general and sufficiently well-defined to be performed automatically by computer. It has certain advantages over the other methods mentioned, which should become apparent in the final section where the various methods are compared.

2. Overview of method

An unstructured flow diagram is converted to structured form by systematic structuring of the first three of the forms shown in **Fig. 1,** viz:

(*a*) abnormal selection paths

(*b*) loops with multiple exit points

(*c*) loops with multiple entry points.

The general forms of multiple exit loops and multiple entry loops are shown in **Fig. 2.** In each case, H is the *head* of the loop, R its *range* and K the *back-branch* leading from the end of the range to the head. $x_1, x_2, \ldots, x_m$ $(m \geqslant 2)$ are exits leading to points $X_1, X_2, \ldots, X_m$ outside the loop. Similarly, $e_0, e_1, \ldots, e_n$ $(n \geqslant 1)$ are entries leading from points

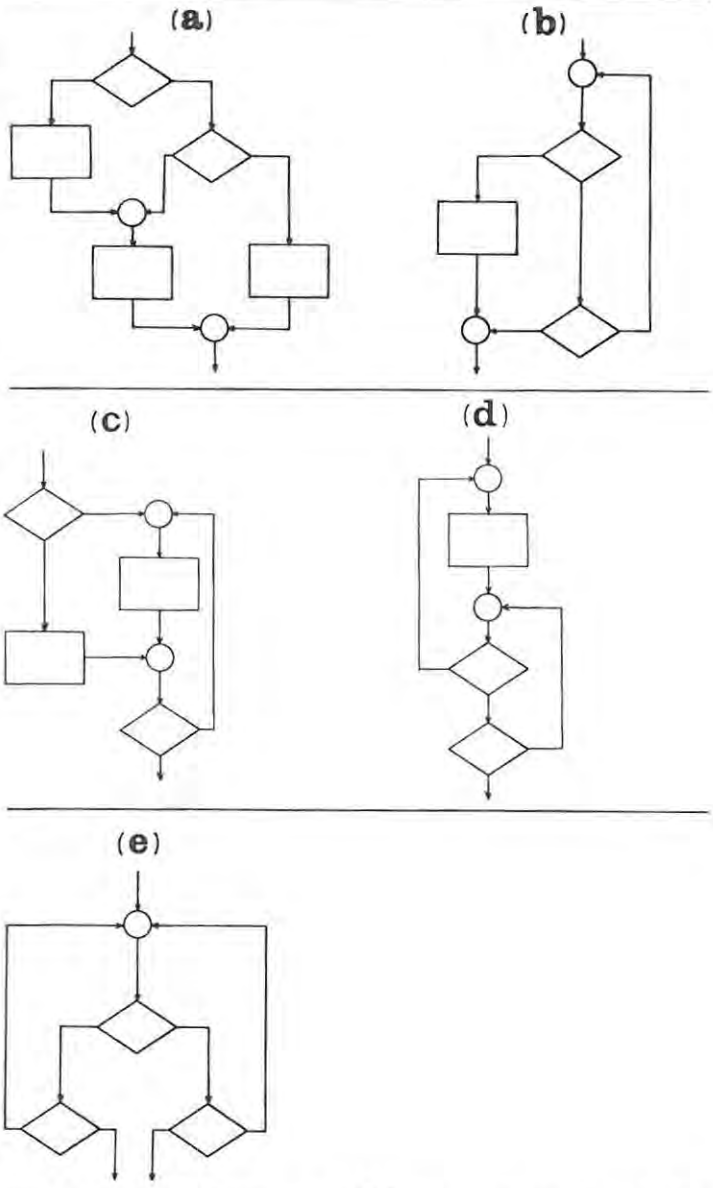


Fig. 1 The five basic structures which cause unstructuredness in flow diagrams
(*a*) Abnormal selection path
(*b*) Loop with multiple exit points
(*c*) Loop with multiple entry points
(*d*) Overlapping loops
(*e*) Parallel loops

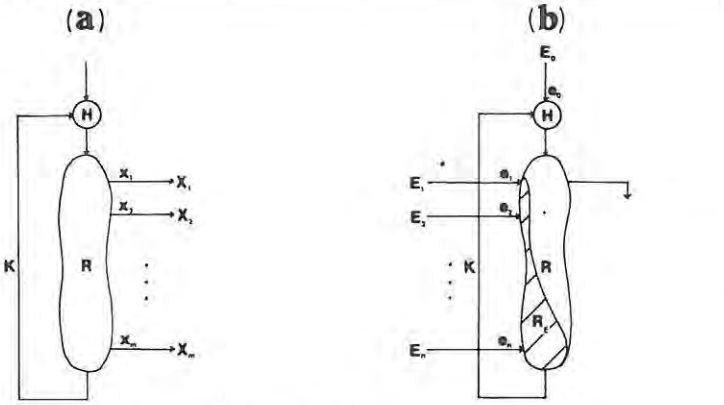


Fig. 2 The general forms of loops
(*a*) Loop with multiple exit points
(*b*) Loop with multiple entry points

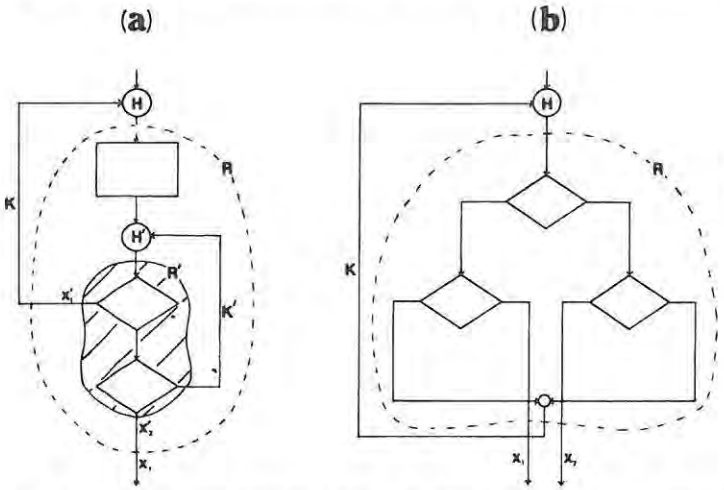


Fig. 3 (*a*) The overlapping loops in Fig. 1(*d*) drawn as nested multiple exit loops
(*b*) The single multiple exit loop corresponding to the parallel loops in Fig. 1(*e*)

$E_0, E_1, \ldots, E_n$ outside the loop to points within it. All entries except e_0 are termed *abnormal*, and R_E consists of every box in R which can be reached from an abnormal entry without branching back to H.

The method owes its generality to the fact that overlapping loops can be interpreted as nested multiple exit loops, and parallel loops can easily be converted to single multiple exit loops. This is illustrated by **Figs. 3**(*a*) and **3**(*b*), which show the overlapping and parallel loops in Figs. 1(*d*) and 1(*e*), respectively, redrawn as multiple exit loops with heads, ranges, back-branches and exits clearly shown. The actual boxes of the overlapping loops remain unchanged, and the only change to the parallel loops is that both back-branches have been connected to an additional circle created at the end of the range, which in turn has been connected to the loop head by a single back-branch. This treatment of overlapping and parallel loops is certainly artificial, but has the advantage that it allows loops to be considered individually and virtually independently.

In brief outline, the method is as follows:

(*a*) replace all 'STOP' boxes by a single one, so the flow diagram will have only one exit point

(*b*) insert a circle at each junction point so that circles are the only flow diagram elements having more than one arrow leading to them

(*c*) identify each loop, determining its head, range, exits and entries

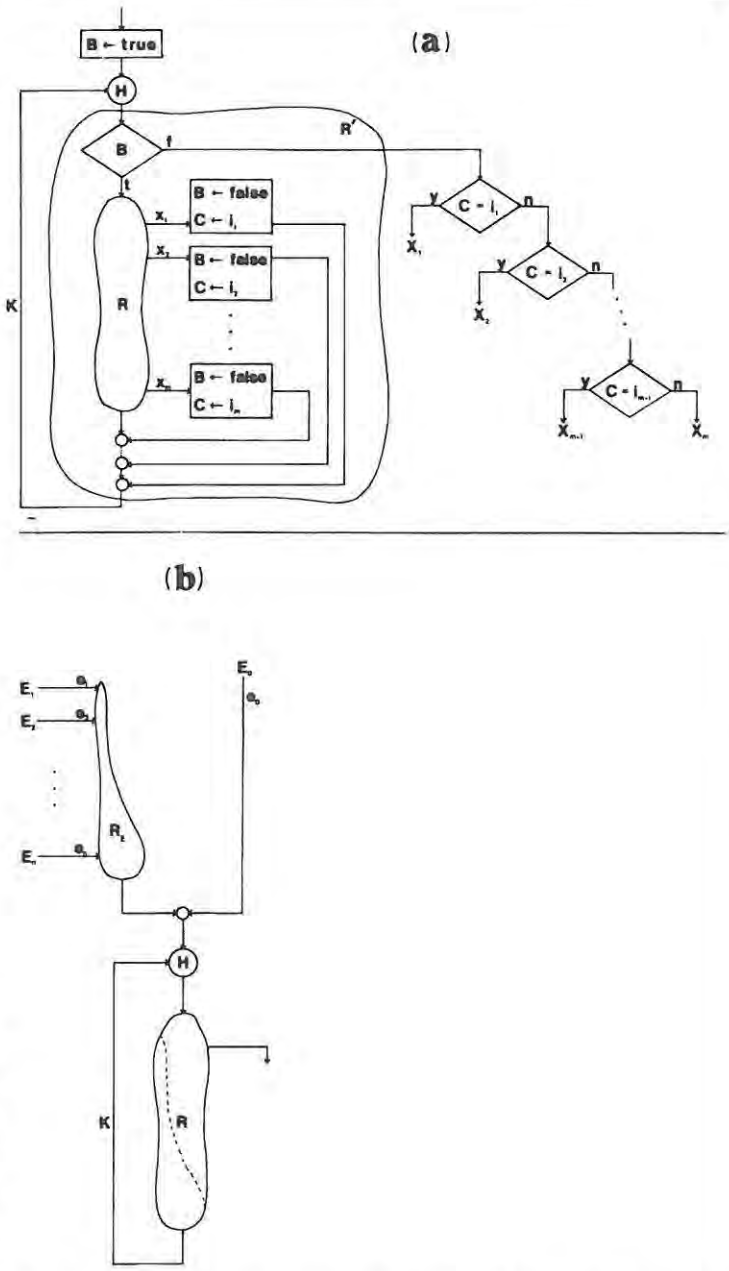


Fig. 4 (*a*) The single exit while loop formed by the removal of multiple exits from the loop shown in Fig. 2(*a*)
(*b*) The structured while loop formed by the removal of multiple entries from the loop shown in Fig. 2(*b*)

(*d*) convert each unstructured loop to structured form by eliminating multiple exits and entries

(*e*) eliminate abnormal selection paths using the well-known method of duplication or node-splitting

(*f*) simplify the resultant structured flow diagram if necessary.

The first two steps are merely preliminary ones, and require no further explanation; nor does node-splitting, which is the accepted technique for eliminating abnormal selection paths. The other steps are considered in detail below. They have been split into several substeps for ease of description, some of which may be combined in practice.

3. Identification of loops

To facilitate the systematic conversion of loops to structured form, it is necessary to identify the heads, ranges, exits and entries of each loop in the flow diagram. Although this can often be done very easily by inspection, the algorithm for

performing it automatically is by no means trivial. The following is an outline of it:

(*a*) find the heads of all loops and convert parallel loops to single multiple exit loops. A procedure for doing this is given in **Appendix 1.** It is based on the fact that a loop head is a circle having a path leading from it which eventually returns to it, without passing through the head of an outer loop

(*b*) assign level numbers to all loop heads in such a way that heads of inner loops have higher level numbers than heads of outer loops. A procedure for doing this is given in Appendix 1

(*c*) determine the range of each loop and mark every box with an indication of the innermost loop to which it belongs. This can be done using a procedure given in Appendix 1

(*d*) identify all exits from and entries to each loop. An exit from a loop is any branch leading from a box within the loop to one outside it. Similarly, an entry to a loop is any branch leading from a box outside the loop to one within it. All loop exits and entries can therefore be identified by examining each branch in the flow diagram once.

4. Structuring of loops

Once all loops have been identified, the following algorithm can be used to structure them:

(*a*) select one of the most deeply nested loops with abnormal exits and/or abnormal entries. All multiple exits from a loop are abnormal, and a single exit which is not at the top of the range is considered abnormal as well, because, in Section 1, the **while** loop was assumed to be the only form of loop allowed in structured flow diagrams

(*b*) if the loop has abnormal exits, then, ignoring possible abnormal entries, it has the form shown in Fig. 2(*a*) (except that it might have only one exit, which is not at the top of the range). Choose a unique Boolean variable B and distinct integers $i_1, i_2, \ldots, i_m$ (one for each exit), and convert the loop to the single exit **while** loop shown in **Fig. 4**(*a*). A box setting B **true** must be inserted before each entry (normal or abnormal) to the loop. C is a case variable which may be used for all loops

(*c*) if the loop has multiple entries, then, since any possible abnormal exits have already been eliminated, it has precisely the form shown in Fig. 2(*b*), with the exit at the top of the range. (If step (*b*) is applied to the loop, each E_i will be a box setting the unique Boolean variable **true,** and the range R will be the new range R' shown in Fig. 4(*a*).) Convert it to the structured **while** loop shown in Fig. 4(*b*) by duplicating all those boxes in R which can be reached from the abnormal entries (i.e. R_E)

(*d*) if any unstructured loops remain, repeat this entire process.

5. Simplification

Although structured, loops of the form shown in Fig. 4(*a*) are often very clumsy because of the decision boxes testing the case variable at their exit points. These will have been converted to structured selections by node-splitting, and so will have the form shown in **Fig. 5**(*a*). If it makes the loop any simpler, each selection path may be incorporated at the appropriate point in the loop's range, as shown in Fig. 5(*b*). The case variable is no longer required.

This process will not always produce simpler loops, and so should be applied with discretion. Note also that it cannot be applied at all until the flow diagram has been completely structured.

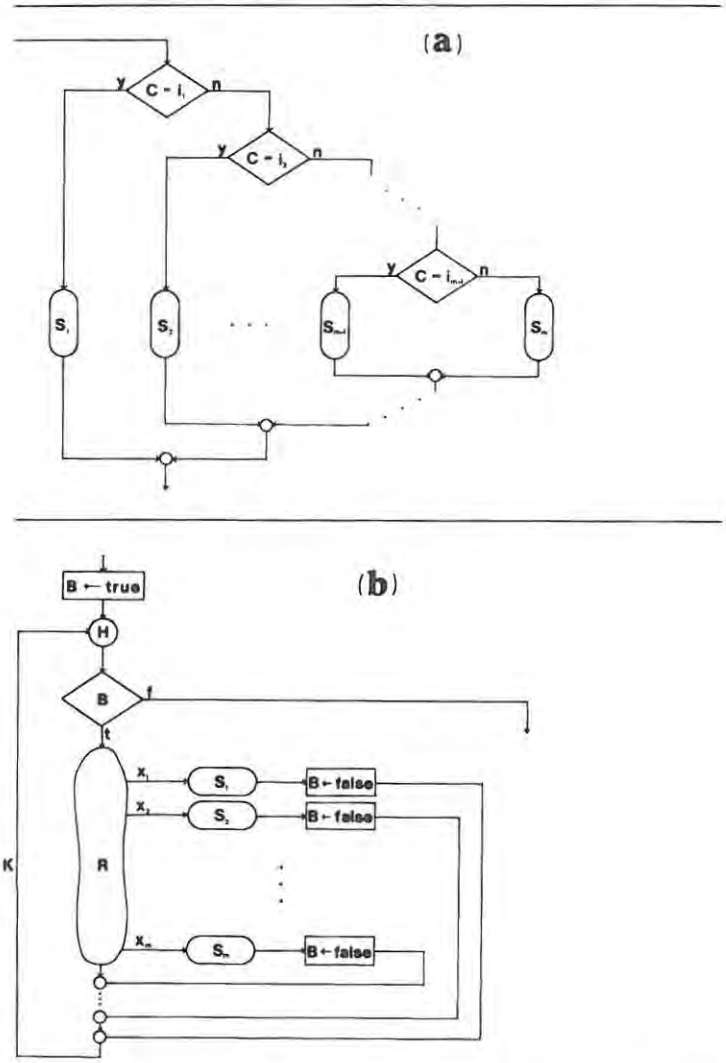


Fig. 5 (*a*) **The selection elements testing the case variable on exit from a loop of the form shown in Fig. 4**(*a*)
(*b*) **The loop with the selection paths incorporated in its range**

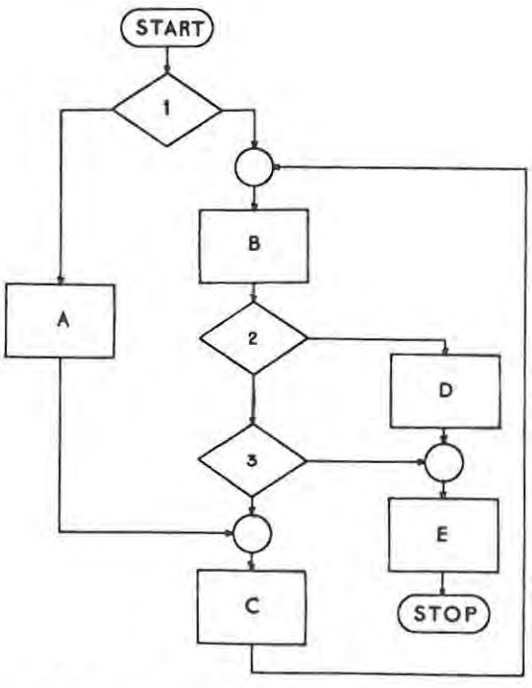
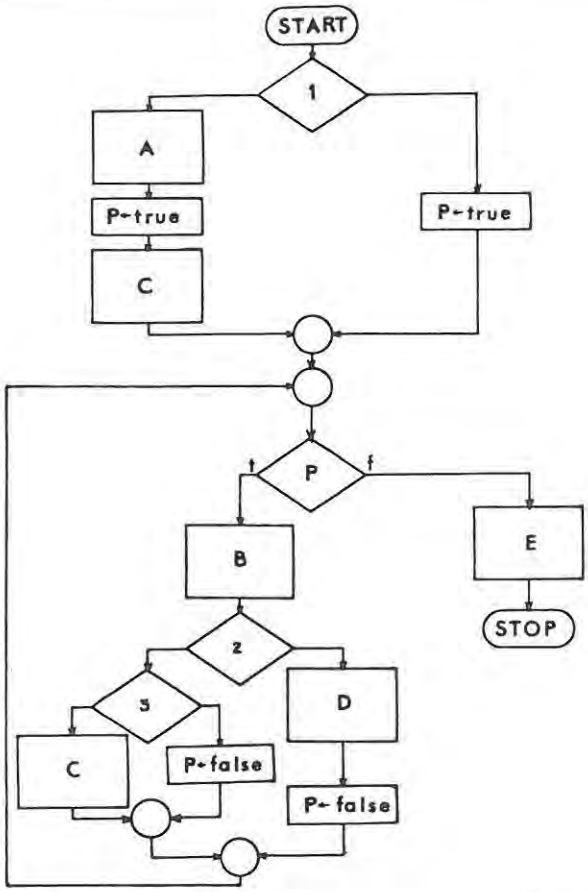
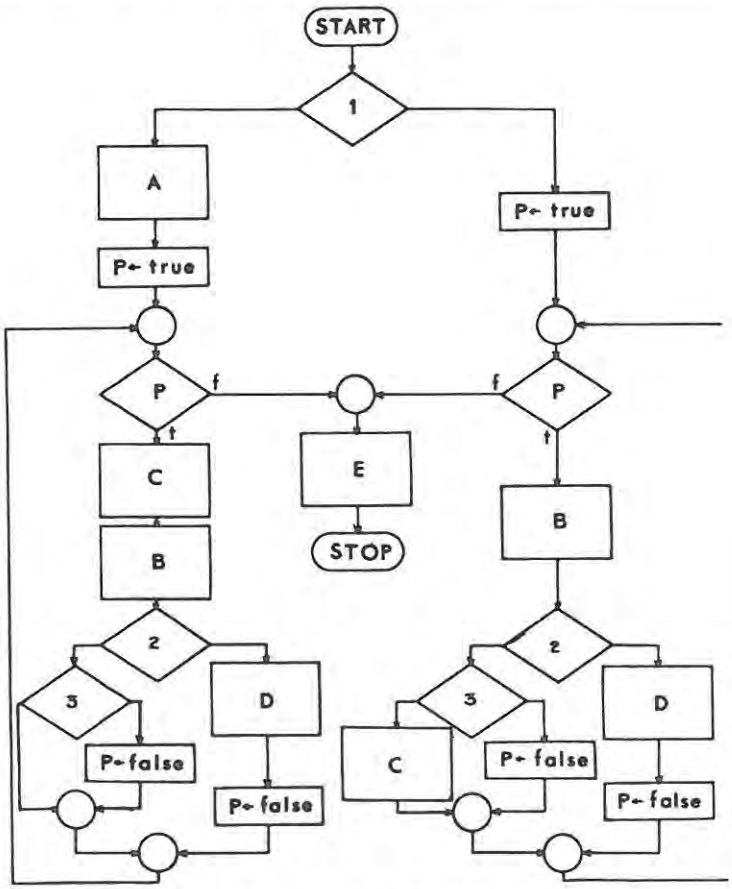


Fig. 6 (*a*) **Flow diagram containing loop with multiple exit and multiple entry points**



(*b*) Equivalent structured form of Fig. 6(*a*), obtained using the current method



(*c*) Equivalent structured form of Fig. 6(*a*), obtained using Ashcroft and Manna's method

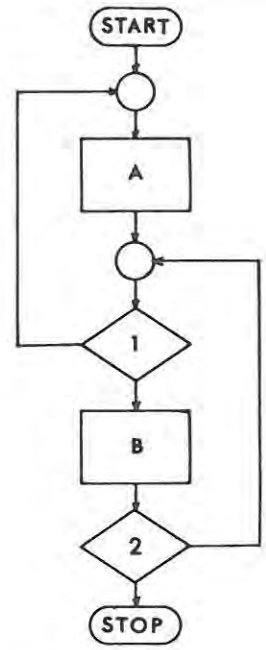
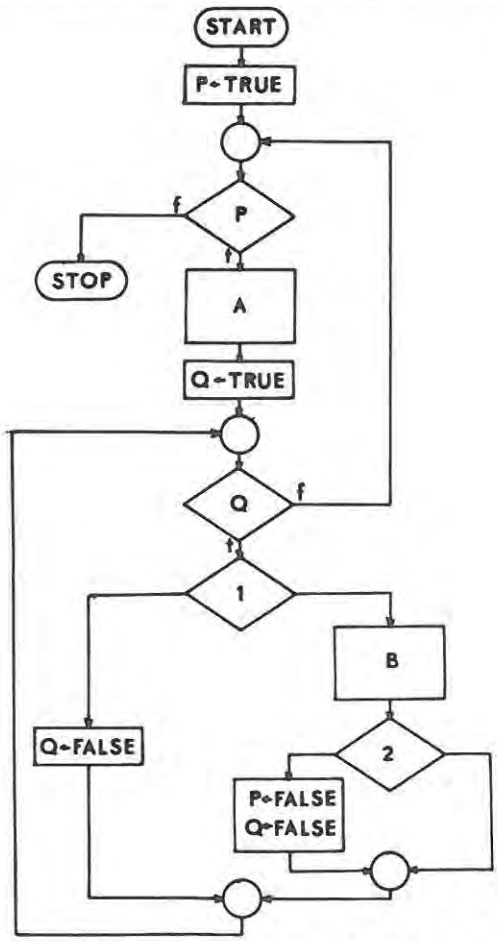


Fig. 7 (*a*) Flow diagram containing overlapping loops



(*b*) Equivalent structured form of Fig. 7(*a*), obtained using the current method

6. Comparison of methods

The method described above and the various other methods mentioned in Section 1 will now be compared by considering their effects on the four forms of unstructured loop shown in Figs. 1(*b*)-(*e*). All methods use node-splitting to handle abnormal selection paths (Fig. 1(*a*)), which are therefore of little interest.

A loop with both multiple exits and multiple entries is shown in **Fig. 6**(*a*), and the result of applying the method described above to it, in Fig. 6(*b*). Ashcroft and Manna's method (1972) can also handle this loop, converting it to the double loop drawn in Fig. 6(*c*).

The other methods do not cater for multiple entry loops at all and so are not completely general. They do handle multiple exit loops, however. Böhm and Jacopini's method (1966) has much the same effect on them as the current method, whereas the effect of Wulf's method is similar to that of the current method without the simplification step.

Fig. 7(*a*) is an example of overlapping loops. The current method converts it to the nested loops shown in Fig. 7(*b*). Ashcroft and Manna's method can also handle this case, though in a slightly different way, but the other methods do not seem able to do so.

Finally, consider the parallel loops drawn in **Fig. 8**(*a*) (Mullins *et al.*, 1974). The equivalent structured form produced by the current method is shown in Fig. 8(*b*). Both Ashcroft and Manna's and Böhm and Jacopini's methods produce similar results. Wulf's method, on the other hand, converts the parallel loops to the **while** loop shown in Fig. 8(*c*), which is very similar to the flow diagram produced by the current method if no simplification is performed.

7. Conclusion

Various methods for converting unstructured flow diagrams to equivalent structured ones have been compared in the light of the five structures which cause unstructuredness and a general conversion method, capable of being performed automatically, has been outlined. A computer program which restructures flow diagrams, based on this technique, has been written and has been applied successfully to a number of simple examples.

Of course, mechanical restructuring of this type does not

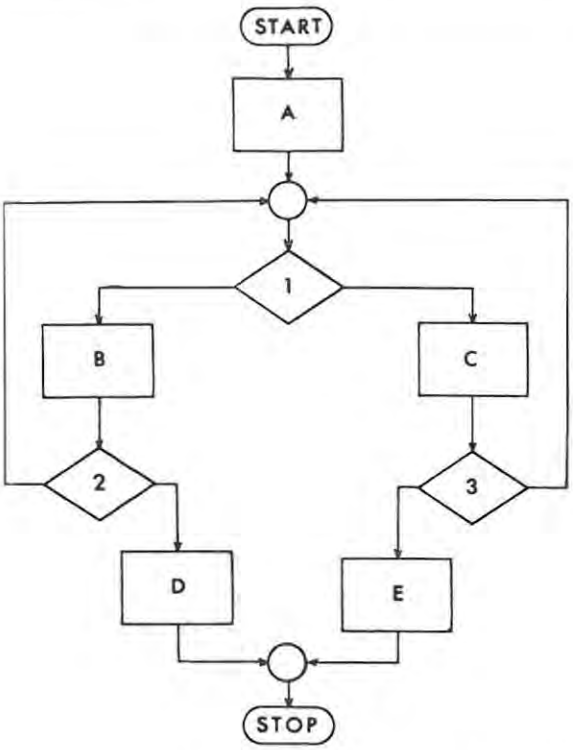
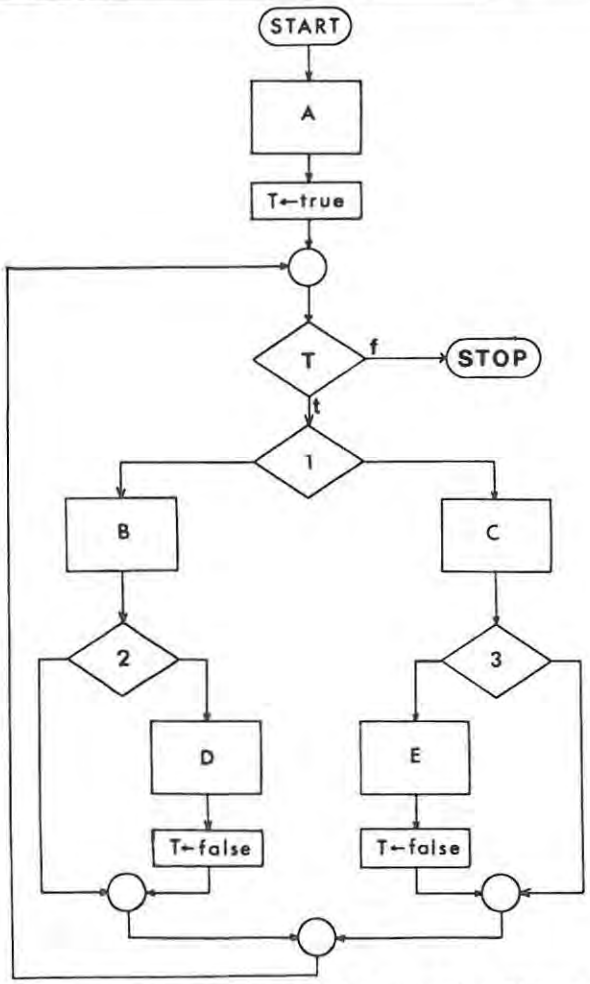
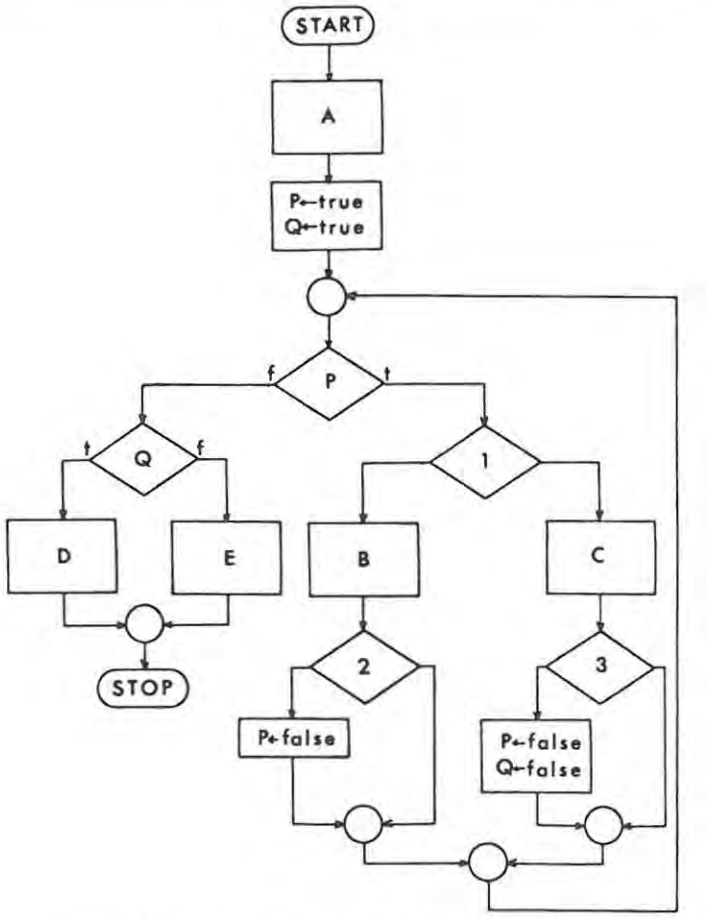


Fig. 8 (*a*) **Flow diagram containing parallel loops (Mullins *et al*)**



(*b*) **Equivalent structured form of Fig. 8**(*a*), **obtained using the current method**



(*c*) **Equivalent structured form of Fig. 8**(*a*), **obtained using Wulf's method**

always produce the most efficient or most pleasing solution, since it does not take account of information contained within the flow diagram boxes to reduce the number of Boolean variables required and simplify the equivalent structured flow diagram. This is a much more difficult problem, which in general has not been solved.

Acknowledgements
One of the authors, Mr H. L. Ossher, would like to thank the South African CSIR for a grant to enable this work to be completed.

Appendix 1

The procedures used to identify loops are given below in a semi-formal notation. They operate on flow diagrams represented by linked lists of boxes. Each box has a number of fields, including a CLASS field (specifying whether the box is a process box, decision box, circle or terminal box), and a LOOP field (to hold a pointer to the innermost loop to which the box belongs). Each circle has four additional fields, viz. a TYPE field (specifying the type of the circle, e.g. *loop head* or *selection circle*), a LEVEL field (to contain the level number assigned to the circle), a FLAG field and a PROCESSED field (used by the procedure MARK to indicate whether or not the circle has already been processed).

Three functions, NEXT, RIGHT and LEFT, are used by the procedures for following paths within flow diagrams. NEXT (BOX), where BOX is a process box or a circle, is the box to which the arrow leaving BOX points; RIGHT(BOX) and LEFT(BOX), where BOX is a two-way decision, are the boxes to which the rightmost and leftmost arrows leaving BOX point, respectively.

The procedure FINDHEADS, given below, does three things:

(*a*) classifies circles as either *loop heads* or *selection circles*, and marks them accordingly (by setting their TYPE fields)

(*b*) converts parallel loops to single multiple exit loops; and

(*c*) duplicates circles where necessary so that each circle will have precisely two arrows pointing to it.

It should be called with the actual parameter specifying the first box of the flow diagram.

```
FINDHEADS (BOX)=
  if BOX is a process box then
                        FINDHEADS (NEXT (BOX)) else
  if BOX is a decision box then
                        FINDHEADS (RIGHT (BOX));
                        FINDHEADS (LEFT (BOX)) else*

  if BOX is a circle then
   if BOX is unmarked then
    mark BOX "VISITED ONCE";
    FINDHEADS (NEXT BOX));
    if BOX is still marked "VISITED ONCE"
     then mark BOX "PROVISIONAL SELECTION
          CIRCLE"
     else mark BOX "LOOP HEAD" fi else
   if BOX is marked "VISITED ONCE" then
    mark BOX "PROVISIONAL LOOP HEAD" else
   if BOX is marked "PROVISIONAL SELECTION
          CIRCLE" then
    mark BOX "SELECTION CIRCLE" else
   if BOX is marked "PROVISIONAL LOOP HEAD" then
    CONVERT PARALLEL LOOP else
   if BOX is marked "SELECTION CIRCLE" then
    DUPLICATE SELECTION CIRCLE else
   if BOX is marked "LOOP HEAD" then
    CREATE SELECTION CIRCLE fi fi fi fi fi fi fi fi fi
```

*These may be performed in the reverse order, i.e.
. . . FINDHEADS (LEFT (BOX));
FINDHEADS (RIGHT (BOX)) . . .
in which case the order must be reversed at the relevant points in all the other procedures as well. The effect produced, though equivalent, may be slightly different.

CONVERT PARALLEL LOOP is called when a circle is encountered which has two back-branches leading to it (i.e. which is the head of parallel loops). It converts the parallel loops to a single multiple exit loop as described in Section 2. DUPLICATE SELECTION CIRCLE or CREATE SELECTION CIRCLE is invoked when a circle is encountered which has too many arrows (other than back-branches) leading to it, to create a new selection circle and thereby reduce the number of arrows leading to the original circle.

The procedure ASSIGNLEVELS, given below, assigns level numbers to all circles in such a way that if a path (not including the back-branch of a loop) leads from one circle to another, then the first will have a lower level number than the second. As a result, in the case of nested loops, the head of the inner loop will always have a higher level number than the head of the outer loop. The level numbers assigned to loop heads can thus be used to determine the way in which loops are nested. Those assigned to selection circles, on the other hand, help to perform node-splitting automatically.

When ASSIGNLEVELS is called, the first actual parameter should specify the first box of the flow diagram and the second the level number to be given to the first circle encountered.

```
ASSIGNLEVELS (BOX, LEV) =
 if BOX is a process box then
                ASSIGNLEVELS (NEXT(BOX), LEV) else
 if BOX is a decision box then
                ASSIGNLEVELS (RIGHT(BOX), LEV);
                ASSIGNLEVELS (LEFT(BOX), LEV) else
 if BOX is a circle then
  if BOX is an unflagged loop head then
   BOX[LEVEL]←LEV;
   flag BOX;
   ASSIGNLEVELS (NEXT (BOX), LEV+1) else
  if BOX is a flagged loop head then unflag BOX else
 if BOX is an unflagged selection circle then
  BOX[LEVEL]←LEV;
  flag BOX else
 if BOX is a flagged selection circle then
  BOX[LEVEL]←MAX(BOX[LEVEL], LEV);
  unflag BOX;
  ASSIGNLEVELS (NEXT(BOX), BOX[LEVEL]+1)
                                  fi fi fi fi fi fi fi
```

The procedure MARKRANGES, given below, marks the ranges of all loops by placing, in the LOOP field of each box, a pointer to the head of the innermost loop to which the box belongs. It invokes the procedure MARK, also given, which marks the range of a single loop, and the function POINTER-TO, which returns a pointer to the box specified as its parameter. It should be called with the actual parameter specifying the first box of the flow diagram.

```
MARKRANGES (BOX) =
 if BOX is a process box then
                MARKRANGES (NEXT(BOX)) else
 if BOX is a decision box then
                MARKRANGES (RIGHT(BOX));
                MARKRANGES (LEFT(BOX)) else
 if BOX is a circle then
  if BOX is an unflagged loop head then
   flag BOX;
```

```
    MARK(NEXT(BOX), POINTERTO(BOX));
    MARKRANGES(NEXT(BOX)) else
   if BOX is an unflagged selection circle then
    flag BOX;
    MARKRANGES (NEXT(BOX)) else
   if BOX is flagged then unflag BOX fi fi fi fi fi fi

MARK(BOX,HPTR) =
 if BOX is a process box then
  MARK (NEXT(BOX), HPTR);
  if NEXT(BOX)[LOOP] = HPTR then
        BOX[LOOP] ← HPTR fi else
 if BOX is a decision box then
  MARK (RIGHT(BOX), HPTR);
  MARK (LEFT(BOX), HPTR);
  if RIGHT(BOX)[LOOP] = HPTR ∨ LEFT
        (BOX)[LOOP] = HPTR then
   BOX[LOOP]←HPTR fi else
 if BOX is a circle then
  if HPTR=POINTERTO (BOX) then
        BOX[LOOP] ←HPTR else
  if BOX is unflagged ∧ BOX[PROCESSED]≠HPTR then
   BOX[PROCESSED]←HPTR;
   MARK (NEXT(BOX), HPTR);
   if NEXT(BOX)[LOOP]=HPTR then
         BOX[LOOP] ←HPTR fi fi fi fi fi fi
```


References

ASHCROFT, E. and MANNA, Z. (1972). The Translation of 'GOTO' Programs to 'WHILE' Programs, in *Proceedings of IFIP Congress 71*, North-Holland Publishing Co., pp. 250-255.

BÖHM, C. and JACOPINI, G. (1966). Flow Diagrams, Turing Machines and Languages with only Two Transformation Rules, *CACM*, Vol. 9, No. 5, pp. 366-371.

CHEATHAM, T. E. and WEGBREIT, B. (1972). A Laboratory for the Study of Automatic Programming, *AFIPS Conf. Proc.*, 40, pp. 11-21.

MULLINS, J. M., RADUE, J. E., and WEBB, G. L. (1974). A Tutorial Note on Functional Programming, *Systems/Stelsels*, Vol. 4, No. 8, pp. 13-14.

WILLIAMS, M. H. (1976). Generating Structured Flow Diagrams: The Nature of Unstructuredness, *The Computer Journal*, Vol. 20, No. 1, pp. 45-50.

WULF, W. A. (1972). Programming without the GOTO, in *Proceedings of IFIP Congress 71*, North-Holland Publishing Co., pp. 408-413.

Book reviews

Continued from page 160

is the 'best' book since subjectivity and experience of other methods will bias not only myself but those who are already committed to a particular approach to programming activity. And as the author points out he does not claim infallibility and he recognises the validity of other approaches.

Tausworthe writes easily and it does not take long to become absorbed in what he writes. There are plenty of illustrations and examples to explain further what he wants to put across. Only one note jarred and that is in the earlier sections: on more than one occasion he apologises for moving in a particular direction. Once he gets into his stride, and assumes the reader has decided to go along with him, he drops this mildly irritating posture.

For the 'other camp', that of students and research workers, the book will be less satisfying. Anyone who simply wants a reasonable methodology to apply in programming would be content to read, mark, learn and inwardly digest. Anyone who wants Tausworthe to justify and prove what he writes will not be happy and neither will anyone wanting the state-of-the-art briefing. They should read Yeh's book instead. I fear that some will dismiss Tausworthe also as 'too pragmatic'.

D. R. A. COAN (Manchester)

Simulation of Systems, Proceedings of the 8th AICA Congress, Edited by L. Dekker, 1977; 1122 pages. (*North-Holland*, $91.95)

The proceedings of the 8th AICA Congress takes the now traditional form of preprints together with some short discussion notes and (very valuable) corrections. Major sections of the proceedings cover: System modelling; simulation tools; simulation of specific system, together with examples of current research in the Netherlands presented by demonstration.

Of the theoretical discussions, the paper by Spriet and Vansteenkiste developing Walche functions is likely to be seminal; these functions, the orthogonal binary functions well suited to digital realisation, have proved their worth in several fields now. It is interesting to note that they originate from a highly 'academic' and unpractical thesis in mathematics of the mid-thirties that had to wait for the full development of the digital computer to see practical application. The adaption to the Laplace transform (for linear systems) of the Fast Fourier transform (FFT) is neatly described in a paper by Heinz Waller.

Section Two, simulation tools, has some useful discussion of hybrid computers which will be a source of useful ideas and an attack on defining simulation languages for a range of problems either lying outside conventional engineering or (rather ambitious) in the domain of 'universal' languages. Two other 'in principle' papers are worth mentioning here: the Polish paper (A. Szymański) giving a practical algorithm for a time optimal (Pontryagin) problem with random disturbances and an EAI practical realisation (by Buli and Janác) of an integral equation solution method by analogue computer.

The final section on specific topics is too wide to review in detail but represents a valuable bringing together of practical realisation. The foreword contains Dekker's thoughts on simulation. In the old days we spoke of analogue (or if American, of analog) computers where now we speak of simulating systems. The virtues and vices are still there; limitations in time and equipment require simplifications of the model. These simplifications would be desirable in their own right almost certainly, as a necessity if the experimentation on the model is to be usefully carried out. The art of simulation is no less than the art of analogue programming, for all the availability of 'universal programming languages' and 'hybrid systems with automatic patching'.

The end papers include a short index as well as the panel discussions first on the independence of simulation as a separate discipline, and then on the problem of the credibility of simulation (although the Club of Rome simulations do not have reference here, they are a prime example of the problem of convincing the outsider of credibility).

The proceedings should be available to every technical university, etc.

J. LEWINS (London)

3. M.H.WILLIAMS, A comment on the decomposition of flowchart schemata, The Computer Journal, vol. 25, no. 3, 393-396 (1982).

A Comment on the Decomposition of Flowchart Schemata

M. H. Williams
Computer Science Department, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ, UK

A recent paper by Prather and Giulieri raises two issues. The first concerns the pre-processing of flowcharts before attempting to isolate unstructured subflowcharts, and a modification to the principle of collapsibility defined by Hecht is introduced as an aid in this regard. The second concerns the concept of dominance and the algorithms which Prather and Giulieri present for determining subflowcharts.

DECOMPOSITION OF FLOWCHARTS

The problem of restructuring an unstructured flowchart to produce an equivalent structured diagram has been a subject which has given rise to a number of articles over the past decade (e.g. Refs 1–4). Essentially any solution to this problem consists of two parts: (a) recognition of the basic unstructured units which require to be transformed, and (b) replacement of these units with equivalent structured forms.

In a recent paper Prather and Giulieri take a new look at the first part of this process.[5] They define a subflowchart to be a set S of flowchart boxes which are connected together in such a way that all flowlines into the set S lead directly to some box i ($i \in S$) while all flowlines leaving the set S lead directly to some box j ($j \notin S$). The procedure which they describe for identifying subflowcharts and classifying the nodes of the flowchart makes no distinction between structured and unstructured subflowcharts.

However, a structured subflowchart[6] is easy to identify and can be removed from the flowchart and replaced by a single box. This procedure can be repeated until no further structured subflowcharts remain. The resulting flowchart can be restructured and then the original structured subflowcharts can be reinstated in the reverse order to their removal.

For this purpose suppose that each process box in the flowchart is written in the form

$$1:(p_1, \ldots p_n)x(s_1)$$

and each decision box is written as

$$1:(p_1, \ldots p_n)X(s_1, s_2)$$

where 1 denotes the number or label of the statement, $p_1, \ldots, p_n$ denote the predecessors of statement 1 (i.e. $p_1 \to 1, \ldots, p_n \to 1$), s_1, s_2 denote the successors of statement 1 (i.e. $1 \to s_1, 1 \to s_2$). In this case the removal of structured subflowcharts can be described as follows.

For any node i which is a process box $i:(\alpha)a(j)$, if

$j:(i)B(i, x)$ then replace i, j by $i:(\alpha)a'(x)$
$j:(i)B(x, i)$ then replace i, j by $i:(\alpha)a'(x)$
$j:(i)b(k)$ then replace i, j by $i:(\alpha)a'(k)$.

For any node i which is a decision box $i:(\alpha)A(x, y)$ if

$x:(i)b(i)$ then replace i, x by $i:(\alpha)b'(y)$
$y:(i)b(i)$ then replace i, y by $i:(\alpha)b'(x)$
$x:(i)b(j)$ and $y:(i)c(j)$ then replace i, x, y by $i:(\alpha)b'(j)$.

The effect of this on the flowchart considered by Prather and Giulieri is demonstrated in Fig. 1.

An alternative approach to simplifying flowcharts is a variation on the idea of collapsibility.[7] The principle here is to remove any process box which is linked directly to another box (process, decision, entry point or exit point) without any incoming flowline separating them. Any structured conditions or loops or any $n + \frac{1}{2}$ loops (the simplest of unstructured subflowcharts) may also be removed at the same time. The transformations applied are as follows:

(1) $1:(0)a(i) \to$ (initial node is process box) remove node 1
(2) $n:(i)a(n+1) \to$ (final node is process box) remove node n

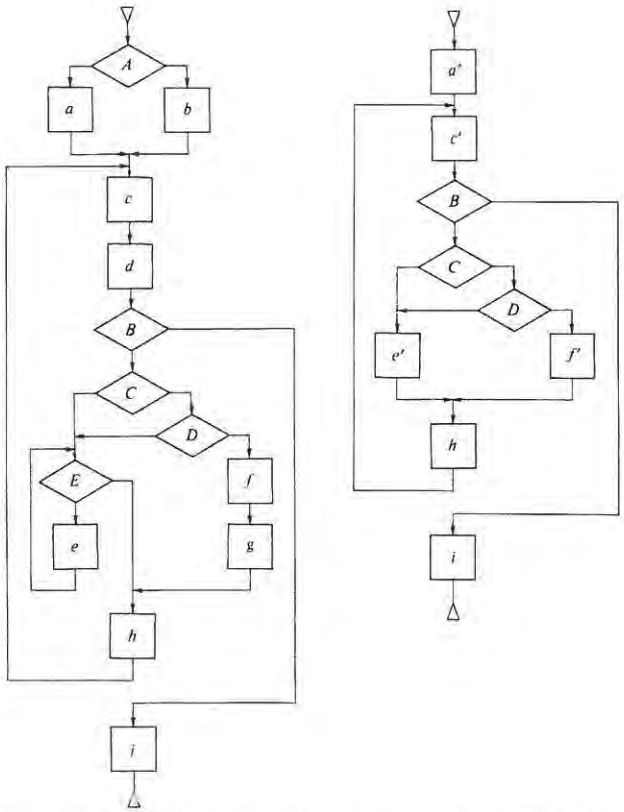


Figure 1. The original flowchart considered by Prather and Giulieri and the equivalent flowchart after removal of structured subflowcharts.

CCC–0010–4620/82/0025–0393 $02.00

© Heyden & Son Ltd, 1982

(3) $i:(\alpha)a(j)$ and $j:(i)b(k) \rightarrow$ replace i, j by $i:(\alpha)a(k)$
(4) $i:(\alpha)a(j)$ and $j:(i)A(m, n) \rightarrow$ replace i, j by $i:(\alpha)A(m, n)$
(5) $i:(\alpha)A(j, x)$ and $j:(i)a(k) \rightarrow$ replace i, j by $i:(\alpha)A(k, x)$
(6) $i:(\alpha)A(x, j)$ and $j:(i)a(k) \rightarrow$ replace i, j by $i:(\alpha)A(x, k)$
(7) $i:(\alpha)A(i, x)$ or $i:(\alpha)A(x, i) \rightarrow$ (loop) replace i by $i:(\alpha)a(x)$
(8) $i:(\alpha)A(j, j), j \neq i \rightarrow$ (condition) replace i by $i:(\alpha)a(j)$

Note that this process differs from the notion of collapsibility in two respects: first, two decision boxes which are connected without any incoming flowline separating them are not removed, and second, structured conditions are removed. This process, which might be termed *structured collapsibility*, reduces a flowchart to its minimum size (all structured components having been collapsed) without affecting the structure of any unstructured subflowcharts (with the exception of $n + \frac{1}{2}$ loops). In this case the initial flowchart illustrated in Fig. 1 will be reduced to the form shown in Fig. 2.

Furthermore this approach has advantages over the flowgraph approach suggested by Cowell, Gillies and Kaposi[8] since the latter does not retain a one-to-one relationship with the original flowchart and in certain cases both unstructured and structured flowcharts map into the same flowgraph.

The use of either of these algorithms in conjunction with the preprocessing phase outlined by Prather and Giulieri will speed up the latter process.

ISOLATION OF UNSTRUCTURED SUBFLOWCHARTS

The mechanism suggested by Prather and Giulieri for isolating unstructured subflowcharts is based on what they refer to as dominance. However, their definition of dominance (i is dominated by j if every path from i to $n + 1$ includes j) is the reverse of the normal definition (i is dominated by j if every path from 0 to i includes j) as given by Refs 7 and 9 and others. Thus one might introduce the term *reverse dominance* to describe the attribute which they use, viz. a labelled statement or flowchart box j is said to be the *reverse dominator* of statement or box i if every path from i to $n + 1$ passes through j. Likewise the dominance tree given in their Fig. 3 is actually the *reverse dominance tree* while the true dominance tree for their example is given in Fig. 3 of this paper.

Using this new nomenclature their method can be described as follows. First the reverse dominance tree for the flowchart concerned is constructed. From this tree the set of reverse intervals is determined. Then each reverse interval in turn is tested to see whether or not it is a subflowchart until the set of subflowcharts has been isolated. This latter process involves: (a) Determining the set of flowchart boxes I defined by each reverse interval. This process is effectively a repetition of the process involved in constructing the reverse dominance tree. (b) Determining the set of paths which lead from boxes which are not members of the set I to those which are.

It might be possible to derive all the information one requires from the two trees (dominance tree and reverse dominance tree). An alternative approach is as follows:

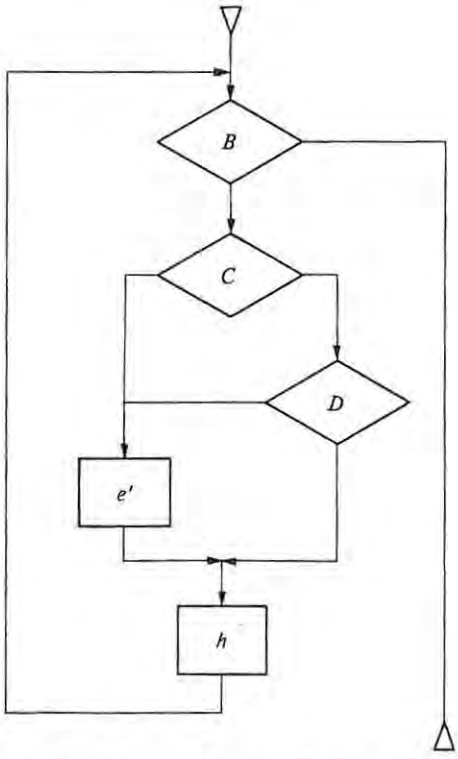


Figure 2. The flowchart considered by Prather and Giulieri after application of structured collapsibility transformations.

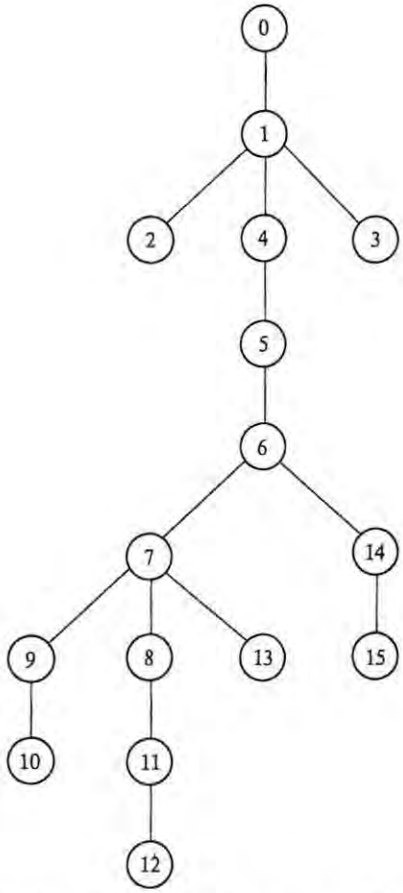


Figure 3. Dominance tree for the flowchart used by Prather and Giulieri.

© Heyden & Son Ltd, 1982

Algorithm for determining unstructured subflowcharts

1. **for** each decision box i **do**
   begin
   create tree with i as an unmarked node;
   while there is an unmarked leaf L **do**
   begin
   change L to a branch node with one or two leaves $l_i (i = 1 \text{ or } 2)$ corresponding to the successors of L;
   for each of these new leaves l_i **do**
   if l_i is the final node **then** mark it
   else if the flowchart box represented by l_i already occurs in some other node on the path between l_i and the root.
   then begin
   trace back up the tree from l_i to the first node D which is the root of two subtrees;
   remove the subtree from l_i to D (excluding D);
   create a set s consisting of all the nodes removed from the subtree together with any sets associated with the pruned nodes;
   associate this set s with node D
   end
   end;
   if there is only one path left **then** discard it
   else while all leaves are identical
   do remove the set of leaf nodes (including any associated sets);
   let S_i = set of remaining nodes together with any associated sets of nodes
   end

2. **for** each decision box i **do**
   begin
   let $T_i = \phi$;
   for each element $x \in S_i$ **do**
   if predecessors $(x) \not\subset S_i$ **then** add x to T_i;
   if T_i contains only one element **then**
   the corresponding set S_i is a subflowchart
   end

Figure 4 illustrates this procedure for box C of the reduced flowchart given in Fig. 1. The result for the flowchart as a whole is given in Table 1. This shows that $\{C, D, e', f'\}$ and $\{c', B, C, D, e', f', h\}$ are the only subflowcharts present in this flowchart (besides the flowchart as a whole).

Finally, a further confusion arises from Prather and Giulieri's use of the term 'irreducible'. The notion of irreducibility of flowcharts was first introduced by Tarjan[10] and has subsequently been referred to by various authors, including Aho, Hopcroft and Ullman[9] and Hecht.[7] Following their definitions, an *elementary irreducible flowchart* is a flowchart with more than one

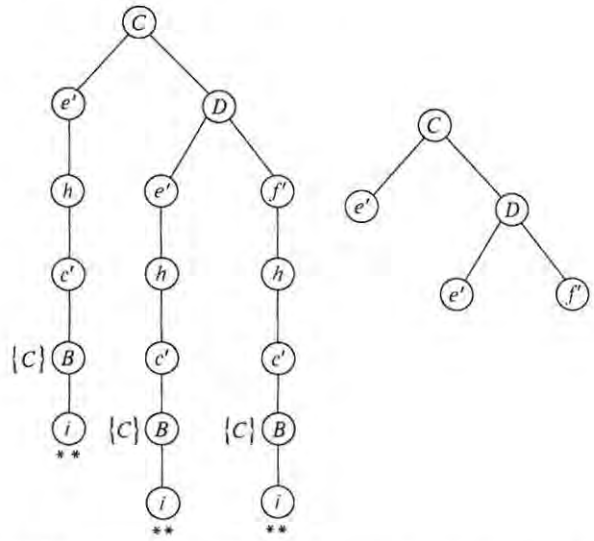


Figure 4. Illustration of alternative method: tree constructed from node C (marked nodes indicated with double asterisks); same tree with common leaves removed.

node in which no pair of nodes X, Y can be found such that the only paths leading to X arise from Y. The simplest example of an elementary irreducible flowchart is

1:(0)A(2, 3)
2:(1, 3)a(3)
3:(1, 2)b(2)

An *irreducible flowchart* is any flowchart which, if reduced by intervals, will produce an elementary irreducible flowchart.

On the other hand, Prather and Giulieri are concerned only with subflowcharts which have a single entry point and a single exit point, and their use of the term 'irreducible' refers to the fact that a subflowchart cannot be subdivided into smaller subflowcharts each with a single entry point and a single exit point. Thus their notion of irreducibility (sequential or maximal) is a superset of that defined by Tarjan and includes standard structured and unstructured subflowcharts.

Table 1. Alternative method for determining subflowcharts using reduced flowchart from Fig. 1

Decision node	S_i	T_i	Subflowchart
B	B, C, D, e', f', h, c'	c'	$\{B, C, D, e', f', h, c'\}$
C	C, D, e', f'	C	$\{C, D, e', f'\}$
D	D, e', f'	D, e'	—

© Heyden & Son Ltd, 1982

REFERENCES

1. E. Ashcroft and Z. Manna, The translation of 'GOTO' programs to 'WHILE' programs, in *Proceedings of IFIP Congress 71*, pp. 250–255. North-Holland, Amsterdam (1972).
2. W. A. Wulf, Programming without the GOTO, in *Proceedings of IFIP Congress 71*, pp. 408–413. North-Holland, Amsterdam (1972).
3. M. H. Williams and H. L. Ossher, Conversion of unstructured flow diagrams to structured form. *The Computer Journal* **21** (No. 2), 161–167 (1978).
4. G. Urschler, Automatic structuring of programs. *IBM Journal of Research and Development* **19** (No. 3), 181–194 (1975).
5. R. E. Prather and S. G. Giulieri, Decomposition of flowchart schemata. *The Computer Journal* **24** (No. 3), 258–262 (1981).
6. M. H. Williams, Generating structured flow diagrams: the nature of unstructuredness. *The Computer Journal* **20** (No. 1), 45–50 (1976).
7. M. S. Hecht, *Flow Analysis of Computer Programs*. North-Holland, New York (1977).
8. D. F. Cowell, D. F. Gillies and A. A. Kaposi, Synthesis and structural analysis of abstract programs. *The Computer Journal* **23** (No. 3), 243–247 (1980).
9. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts (1975).
10. R. E. Tarjan, Testing flowgraph reducibility, in *Proceedings of the 5th Annual ACM Symposium on Theory of Computing*, pp. 96–107 (1973).

Revised version received January 1982

© Heyden & Son Ltd, 1982

4. M.H.WILLIAMS, Flowchart schemata and the problem of nomenclature, The Computer Journal, vol. 26, no. 3, 270-276 (1983).

Flowchart Schemata and the Problem of Nomenclature

M. H. Williams
Computer Science Department, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ, UK

The problem of nomenclature (different names being used to refer to the same object or idea, and the same name being used to refer to different objects and ideas) plagues many areas of Computer Science. The theory of flowchart schemata is no exception to this. This paper attempts to clarify some of the issues concerned with reducibility and structuredness in flowchart schemata.

1. INTRODUCTION

The theory of *flowchart schemata*[1,2] (or program schemata[3] or flow chart schemas[4] or program schemas[5]) is concerned with the study of *flowcharts*[1] (or flow charts[4] or charts[4] or flow diagrams[6]). A flowchart is a collection of interconnected *nodes*[5] (or boxes[1,6] or primitives[4]). Some authors distinguish between four separate types of node:

(a) an operation node or process box[2,6] or action[7]
(b) a decision node[5] or decision box[2,6] or predicate[7]
(c) an entry node[7] or START box[6] or START node[5]
(d) an exit node[7] or FINISH box[6] or HALT node[5]

others use only the first two types but choose one particular node as the entry node and one as the exit node of the flowchart; others, again, made no distinction at all between different types of node.

This brief introductory paragraph is evidence of the way in which different names are used to refer to the same thing. Even worse is the problem of the same name being used by different authors to denote different objects or ideas.

Within the theory of flowchart schemata the term 'reducibility' is a particularly overworked term and is (in some cases) closely related to 'collapsibility'. 'Structuredness' is another closely related concept. Since some confusion has arisen out of the differences in interpretation given by different authors, this paper attempts to clarify some of the issues surrounding these terms.

2. REDUCIBILITY AND COLLAPSIBILITY

2.1. Reducibility (1)

The concept of reducibility as defined by Kosaraju, based on work by Knuth and Floyd,[8] is a form of equivalence relationship. One flowchart G is said to be strongly reducible to another flowchart $H (G \leq_S H)$ if they are equivalent to within certain simple restructuring rules, such as node splitting. More formally, G is *strongly reducible* to H if and only if every node of H is a node of G and for every interpretation (execution) and every input the computational histories are identical for the two flowcharts. Similarly G is *weakly reducible* to H (written $G \leq_W H$) if and only if every node of H is a node of G and for every interpretation and every input the outputs are equal (or both do not terminate) for both flowcharts.

To illustrate this, consider the portions of flowcharts shown in Fig. 1. The portion in Fig. 1(b) is both strongly and weakly reducible to that in Fig. 1(a). The same is true of the portions shown in Fig. 1(d) and Fig. 1(c). On the other hand the portion in Fig. 1(f) is weakly reducible to that in Fig. 1(e) but not strongly reducible to it.

Similarly, one class of flowcharts C_1 is reducible (strongly or weakly) to another class C_2 if and only if every flowchart of C_1 is reducible to a flowchart of C_2 (written $C_1 \leq_S C_2$ or $C_1 \leq_W C_2$). If C_1 is reducible to C_2 but C_2 is not reducible to C_1 this is written as $C_1 <_S C_2$ or $C_1 <_W C_2$. If $C_1 \leq_W C_2$ and $C_2 \leq_W C_1$ then $C_1 \equiv_W C_2$.

Kosaraju then defined certain classes of flowcharts which he studied in detail. Four interesting classes are:

(1) The class of D-charts which is defined as:
  (a) Any process box (action) is a D-chart.
  (b) If G_1 and G_2 are D-charts and p is a predicate then SEQUENCE(G_1, G_2), IF p THEN G_1 ELSE G_2 and WHILE q DO G_1 (where $q \in \{p, \neg p\}$) are D-charts

  In other words a D-chart is a flowchart composed only of simple sequences, simple (2-way) selection and while-loops.

(2) The class of WR-charts (WR ≡ While and Repeat) which is defined as:
  (a) Any process box is a WR-chart.
  (b) If G_1 and G_2 are WR-charts and p is a predicate then SEQUENCE(G_1, G_2), IF p THEN G_1 ELSE G_2, WHILE q DO G_1 and REPEAT G_1 UNTIL q (where $q \in \{p, \neg p\}$) are WR-charts.

  In other words a WR-chart is a flowchart composed only of simple sequences, simple selection, while-loops and repeat-loops.

(3) The class of TD_1-charts (TD ≡ Top-Down) which is defined as:
  (a) Any process box is a TD_1-chart.
  (b) Any decision box is an IP_1-chart.
  (c) Any flowchart constructed from any number of TD_1-charts and at most one IP_1-chart is a TD_1-chart if it contains a single exit path and an IP_1-chart if it contains two exit paths. The class of TD_1-charts thus includes while-loops and $n + \frac{1}{2}$ loops.[9]

CCC-0010-4620/83/0026-0270$03.50

© Wiley Heyden Ltd, 1983

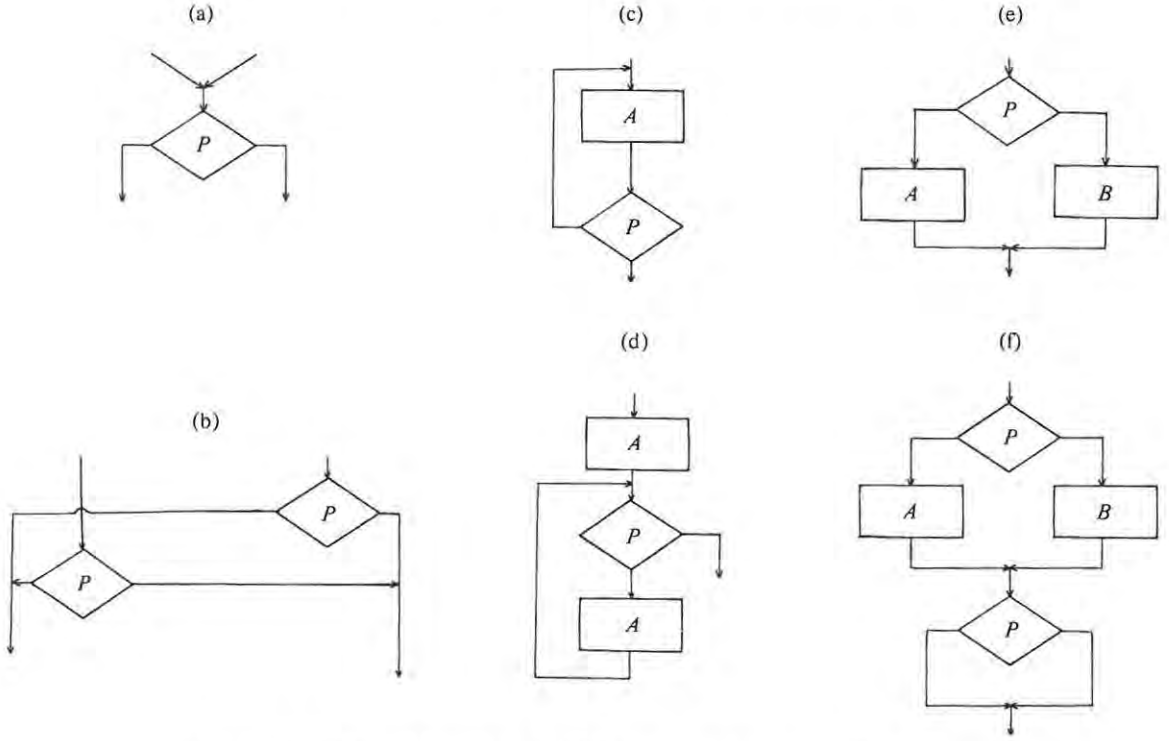


Figure 1. Portions of flowcharts which illustrate strong and weak reducibility.

(4) The class of BJ_n-charts (BJ ≡ Böhm and Jacopini[10]) which is defined as:
   (a) Any process box is a BJ_n-chart.
   (b) If G_1 and G_2 are BJ_n-charts and p is a predicate then SEQUENCE(G_1, G_2) and IF p THEN G_1 ELSE G_2 are BJ_n-charts.
   (c) For $i \leq n$ if $G_1, G_2, \ldots G_i$ are BJ_n-charts and $p_1, p_2, \ldots p_i$ are predicates then the i-exit loop shown in Fig. 2 is a BJ_n-chart.

   Thus, the class of BJ_n-charts includes any loops of the form shown in Fig. 2 with up to i exits.

Kosaraju proved various properties of these classes such as:

(a) The class of BJ_1-charts is equivalent to the class of D-charts.
(b) Any flowchart is reducible to a D-chart provided that it does not contain a reachable loop with more than one exit.

and so on.

2.2. Reducibility (2)

Cowell, Gillies and Kaposi[11] introduced a notion of reducibility which is based on flowgraphs rather than flowcharts. Following their definition a flowgraph is a graphical representation in which operation nodes are ignored. Each node in a flowgraph corresponds to either the exit node or a decision node in the original flowchart. Thus a flowgraph is a representation of the underlying structure of a flowchart. If one flowchart G is strongly reducible (by the definition in the previous section) to another flowchart H and they both have the same number of decision nodes then G and H both have the same flowgraph.

If the class of flowgraphs corresponding to some class of flowcharts X is written as FGRAPH (X) then from the definition of the previous section:

$$\text{FGRAPH(D-charts)} \equiv \text{FGRAPH(WR-charts)} \equiv \text{FGRAPH(TD}_1\text{-charts)}$$

The only type of flowgraph which does not fall into this class is one which contains at least one loop with more than one exit.

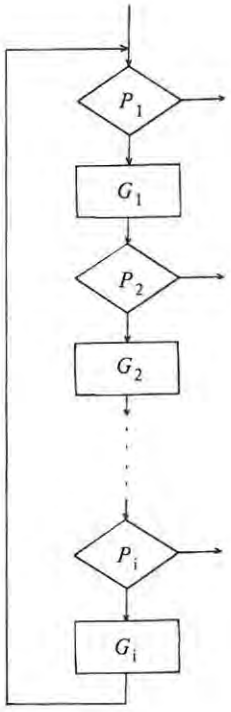


Figure 2. An i-exit loop.

A flowgraph is said to be *reducible*[11] if it has a proper subgraph (i.e. a set of nodes with one entry and one exit) and *irreducible* if it does not have such a subgraph. In this case reducibility is not concerned with the equivalence of two flowcharts but with whether or not the flowgraph corresponding to a given flowchart can be subdivided into smaller units.

If a flowgraph is irreducible and consists of more than one node then:

(a) It cannot be a member of FGRAPH(D-charts), FGRAPH(TD_1-charts), etc.
(b) It must contain at least one multi-exit loop.

2.3. Reducibility (3)

Prather and Giulieri[1] use the term reducible/irreducible in a similar way to Cowell, Gillies and Kaposi except that they apply it directly to the flowcharts rather than to the flowgraphs, viz. a flowchart is *reducible* if it has a proper subflowchart (set of nodes with one entry and one exit). In this case, if a flowchart is irreducible and consists of more than one node, it must contain at least one of the unstructured elements outlined in Section 3.

2.4. Reducibility (4)

Another interpretation of the terms reducible/irreducible is that given by Tarjan[12] and subsequently referred to by various authors, including Aho, Hopcroft and Ullman[13] and Hecht.[14] Following their definitions, an *elementary irreducible flowchart* is a flowchart with more than one node in which no pair of nodes X, Y can be found such that the only paths leading to X arise from Y. The simplest example of an elementary irreducible flowchart is given in Fig. 3.

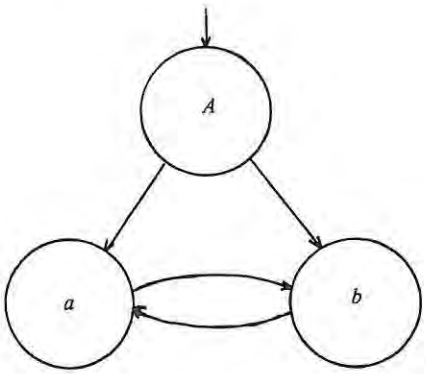


Figure 3. Simplest elementary irreducible flowchart (Tarjan).

In this connection the notion of an interval is important. Given a flowchart G and some node of the flowchart n, the *interval with header n*, $I(n)$, is the set of nodes consisting of n and all other nodes j such that the immediate predecessors of j are in $I(n)$ (i.e. all arcs entering node j arise from nodes in $I(n)$). The term *reduction by intervals* is used to refer to the process of partitioning a flowchart into intervals. An *irreducible flowchart* is any flowchart which, if reduced by intervals, will produce an elementary irreducible flowchart. A *reducible flowchart* on the other hand does not contain any elementary irreducible flowcharts.

2.5. Collapsibility

A concept which is closely related to reducibility is collapsibility, as proposed by Hecht.[15] This is defined in terms of two transformations:

(a) Transformation $T1$: If e is an edge (or arc) of some flowchart which emanates from some node X and leads back to the same node, then $T1$ is the removal of this edge.
(b) Transformation $T2$: If y is a node of some flowchart other than the entry node and x is the only direct predecessor of y then $T2$ is the replacement of the two nodes x and y and any edges between them by a single node z. The predecessors of x become the predecessors of z and the successors of y become the successors of z.

A flowchart is said to be *collapsible* if and only if it can be transformed into the trivial graph (consisting of a single node) by repeated application of $T1$ and $T2$. Hecht has shown that collapsibility is equivalent to the notion of reducibility in Tarjan's sense.

Figure 4 illustrates this concept. Transformation $T2$ can be applied at several points in Fig. 4(a) ($T2(A,a)$, $T2(A,b)$, $T2(B,d)$, $T2(e,C)$, $T2(D,f)$ to produce the flowchart given in Fig. 4(b). Transformation $T1$ can then be applied at node C to produce the flowchart in Fig. 4(c). Finally $T2$ can be applied once more ($T2(C,D)$) to produce the flowchart in Fig. 4(d). At this stage no further transformations can be applied. Hence none of these flowcharts is collapsible.

2.6. Structured collapsibility

A variation on the idea of collapsibility is that of *structured collapsibility*.[2] The principle behind structured collapsibility is to remove any operation node which is linked directly to another node (operation, decision, entry or exit) without any incoming flowline separating them. Any simple selections or single-exit loops (while, repeat or $n + \frac{1}{2}$) may be removed at the same time. If each operation node is written in the form

$$l\colon (p_1, \ldots p_n)x(s_1)$$

and each decision node as

$$l\colon (p_1, \ldots p_n)X(s_1, s_2)$$

where l denotes the number or label of the statement, $p_1, \ldots p_n$ denotes the predecessors of statement l (i.e. $p_1 \rightarrow l, \ldots, p_n \rightarrow l$) and s_1, s_2 the successors of statement l (i.e. $l \rightarrow s_1$, $l \rightarrow s_2$) then the transformations associated with structured collapsibility are as follows:

(1) $1\colon (0)a(i) \rightarrow$ (entry node is operation node) remove node 1
(2) $n\colon (i)\, a(n+1) \rightarrow$ (exit node is operation node) remove node n
(3) $i\colon (\alpha)a(j)$ and $j\colon (i)b(k) \rightarrow$ replace i, j by $i\colon (\alpha)a(k)$
(4) $i\colon (\alpha)a(j)$ and $j\colon (i)\, A(m,n) \rightarrow$ replace i, j by $i\colon (\alpha)A(m,n)$
(5) $i\colon (\alpha)A(j,x)$ and $j\colon (i)a(k) \rightarrow$ replace i, j by $i\colon (\alpha)A(k,x)$
(6) $i\colon (\alpha)A(x,j)$ and $j\colon (i)a(k) \rightarrow$ replace i, j by $i\colon (\alpha)A(x,k)$

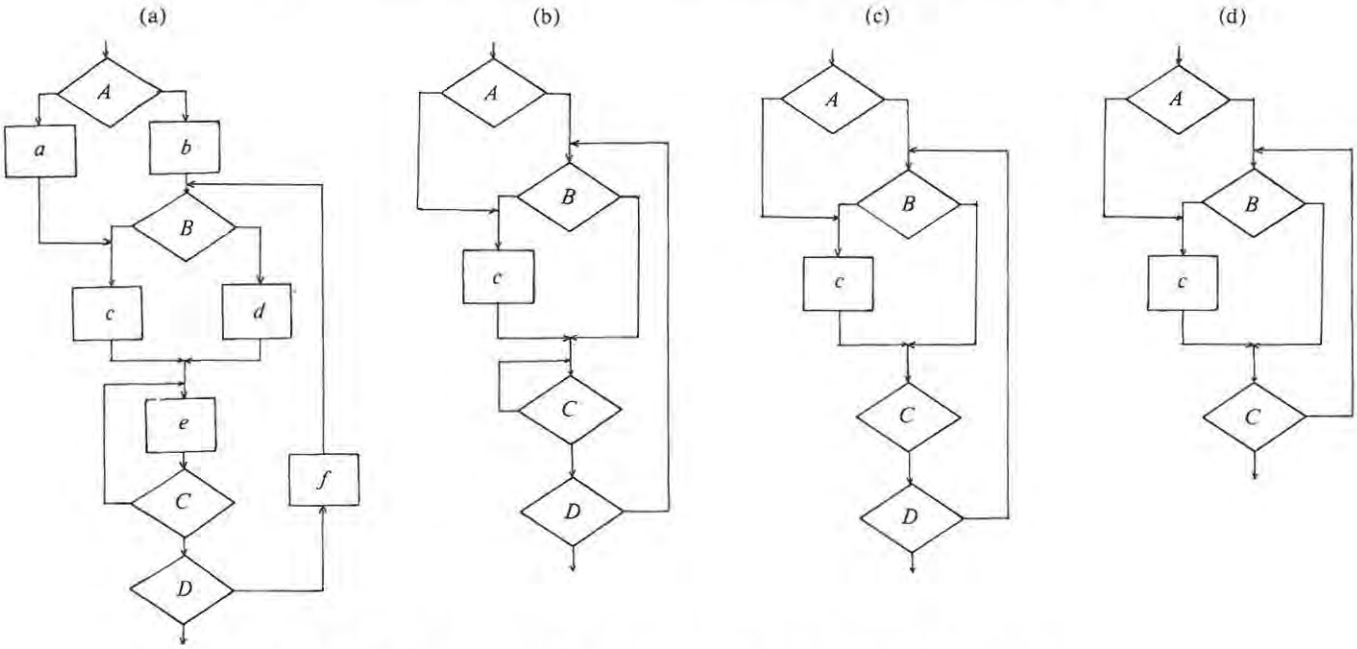


Figure 4. Applying Hecht's transformations to a flowchart to test for collapsibility.

(7) $i:(\alpha)A(i,x)$ or $i:(\alpha)A(x,i) \rightarrow$ (loop) replace i by $i:(\alpha)a(x)$
(8) $i:(\alpha)A(j,j), j \neq i \rightarrow$ (condition) replace i by $i:(\alpha)a(j)$

This process is illustrated in Fig. 5. Fig. 5(a) contains a flowchart before applying these transformations, Fig. 5(b) shows the flowchart after these transformations have been applied. Essentially what this process does is to remove each basic TD_1 subflowchart replacing it with an operation node, and remove each operation node which is adjacent to another node without any incoming edge separating them.

If this process is combined with Prather and Giulieri's reduction process[1] or that described in Ref. 2, the structure shown in Fig. 5(c) would be identified and removed from the flowchart and the remaining nodes would collapse until the flowchart was reduced to a single node. Thus the structure in Fig. 5(c) represents the only unstructured component—as will be discussed in the next section.

Note that the algorithm for structured collapsibility could be modified to remove only D-subflowcharts or WR-subflowcharts although this is more complicated to do.

3. STRUCTUREDNESS

One very important idea which is connected to reducibility and collapsibility is the concept of *structuredness*. This idea was introduced initially by Böhm and Jacopini

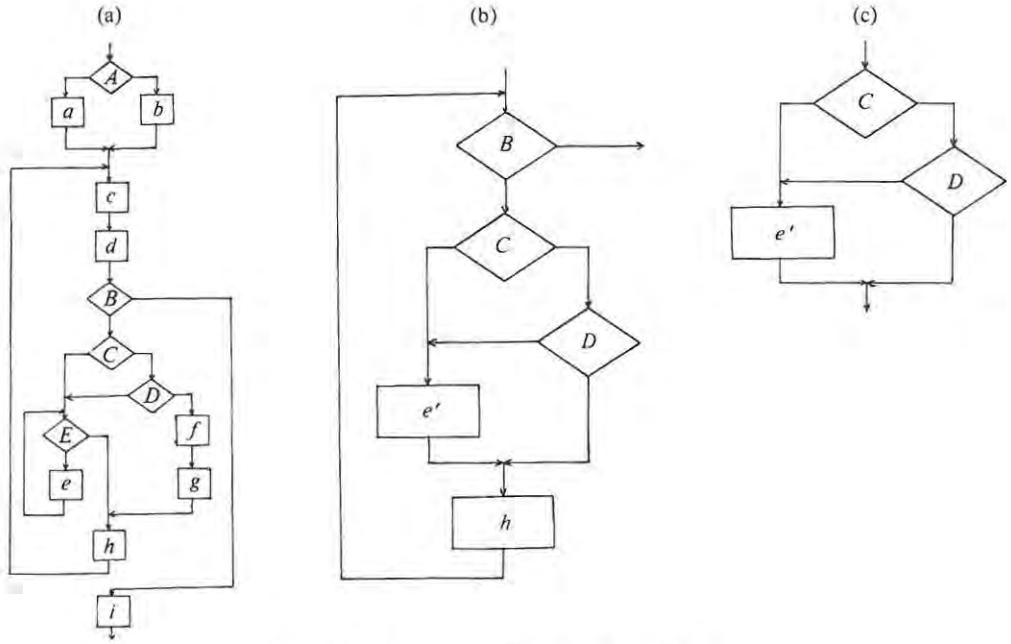


Figure 5. Illustration of structured collapsibility.

who showed how general flowcharts can be decomposed into a small set of base diagrams. However, not all flowcharts can be decomposed into 'a finite number of given base diagrams'. To overcome this problem they proposed two normalization methods for converting flowcharts which would not be decomposed into base diagrams from some finite set into equivalent flowcharts which can be decomposed thus. This idea, viewed from the point of view of synthesis instead of analysis, namely that flowcharts should be constructed initially from a small set of base diagrams has been termed structured programming.[6–8]

Arising from their work, a *structured flowchart* is one which is composed from (or can be decomposed into) a small set of base diagrams, an *unstructured flowchart* is one which cannot be decomposed into a small set of base diagrams without altering it in some way (normalizing it). As has been noted in the previous section, there are various possible choices for this set of base diagrams. To distinguish between these, a flowchart will be referred to as *D-structured* if it can be decomposed into D-charts, *WR-structured* if it can be decomposed into WR-charts and *TD_1-structured* if it can be decomposed into TD_1-charts. Similarly a flowchart may be D-unstructured, WR-unstructured or TD_1-unstructured.

With the recognition of the importance of structured programming, there has been considerable interest in identifying what makes a flowchart unstructured and how an unstructured flowchart can be transformed into a structured one. To understand the constituents of unstructured flowcharts, it is helpful to draw an analogy with the world of matter. Using this three levels can be distinguished:

(a) At the lowest level there are 'unstructured particles'. These are the building blocks of the next level.
(b) At the next level one has 'unstructured elements'. These are flowcharts constructed from structured and unstructured particles, but which may not exist in isolation if they have unsatisfied bonds (more than one entry or exit point).
(c) Unstructured elements may be combined into 'unstructured compounds' by coupling corresponding unsatisfied bonds. An unstructured compound is a 'stable' flowchart with a single entry point and a single exit point.

It must be stressed that this is a very crude analogy but one that does help to give perspective to the constituents of unstructuredness.

In a study of the complexity of programs and flowcharts, McCabe defined a flowchart as structured if it is a D-chart (i.e. D-structuredness) and identified four fundamental characteristics which characterize a D-unstructured flowchart. In terms of the above analogy these would be unstructured particles of D-unstructuredness. They are:

(a) branch out of a loop
(b) branch into a loop
(c) branch out of a selection
(d) branch into a selection.

Unfortunately he overlooked the fact that neither a repeat-loop nor an $n + \frac{1}{2}$-loop are D-structures, and any flowchart which contains either of these two types of loop will not be a D-chart. Thus the full catalogue of unstructured particles which characterize D-unstructuredness must also include:

(e) $n + \frac{1}{2}$ loop
(f) repeat loop.

Obviously the unstructured particles which characterize WR-unstructuredness are (a) to (e) of the above list, whereas those which characterize TD_1-unstructuredness consist of those in McCabe's original list (i.e. (a)–(d)).

When these unstructured particles are combined with structured particles (simple sequence, simple selection, etc.) this gives rise to unstructured elements. To illustrate these McCabe gave a diagrammatic representation of the four simplest elements. These are shown in Fig. 6. He also noticed that none of these constructs can exist in isolation in a flowchart (at least, none of the unstructured elements of TD_1-unstructuredness can exist in isolation) since each has more than one entry point or more than one exit point. Consequently he postulated a set of four basic unstructured compounds which can arise out of combining these elements; these are shown in Fig. 7(a)–(d). These same four compounds were recognized by Williams[6] in a paper submitted over a year before McCabe's paper. He also recognized another basic unstructured compound obtained when two multi-exit loops are combined into a pair of parallel loops as shown in Fig. 7(g).

Oulsnam has pointed out that the construct in Fig. 6(a) can be regarded as two separate constructs depending on which exit path is taken as the main exit and which the auxiliary exit. Likewise, the construct in Fig. 6(b) can be viewed as two separate constructs depending on which entry path is taken as the main entry path and which the

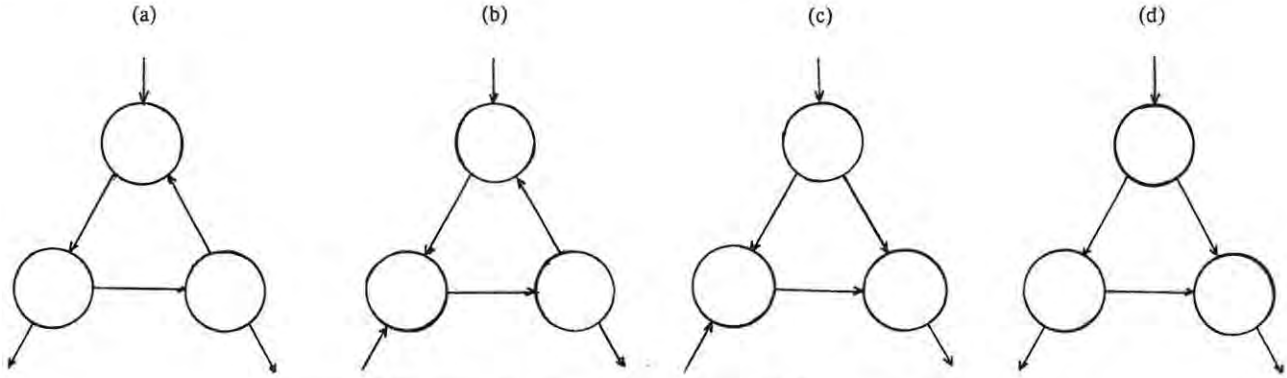


Figure 6. The simplest unstructured elements of TD_1-unstructuredness.

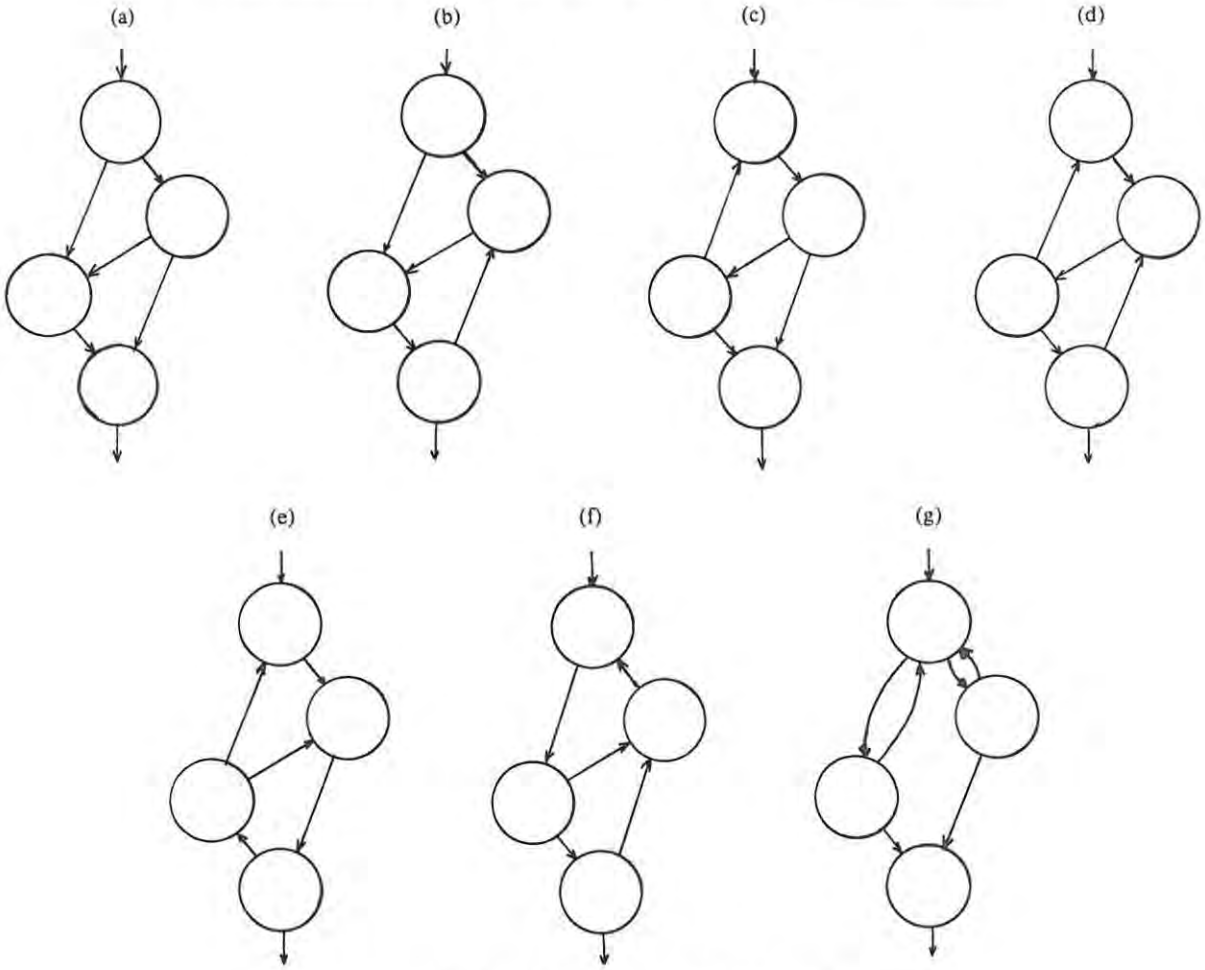


Figure 7. Some simple unstructured compounds.

auxiliary. However, when each of these elements is paired with another element (as in Fig. 7), the distinction disappears. Oulsnam has added two other basic unstructured compounds to the list given by McCabe. These are shown in Fig. 7(e) and (f).

Of course, these are not the only unstructured elements and compounds which are possible. Williams[6] and Williams and Ossher[19] have dealt with loops with more than two exit points or more than two entry points or indeed with n entry points and m exit points ($n \geq 2, m \geq 2$). Again such loops do not exist in isolation but occur in combination with other unstructured elements. Thus the number of combinations is endless.

Note that the combination of the concept of structured collapsibility together with the reduction process described by Prather and Giulieri as mentioned in Section 2.6 results in a process for identifying unstructured compounds. The isolation of unstructured elements or particles is more difficult and a typical method for doing this is that given by Williams and Ossher.[19]

It follows that any method which aims at transforming an unstructured flowchart into a structured one must deal with unstructured particles or elements rather than unstructured compounds. Various methods have been proposed, e.g. Ashcroft and Manna,[20] Wulf,[21] Williams and Ossher,[19] Oulsnam,[22,5] etc. Differences arise in the identification of the unstructured particles or elements, the transformations for converting them to structured form and in the structured form produced (D-structured or TD_1-structured).

4. CONCLUSIONS

Some interpretations of the related concepts of reducibility, collapsibility and structuredness as they apply to flowcharts have been presented in order to clarify the current position. The intention is not to provide a complete catalogue of interpretations but to study certain important ones.

It is hoped that the reader will have gained some insight from this into the similarities and differences between the interpretations and approaches adopted by different authors. In particular attention is drawn to the following:

(a) The similarity/difference between reducibility as defined by Knuth and Floyd and Kosaraju and the flowgraphs used by Cowell, Gillies and Kaposi.
(b) The similarity/difference between reducibility as defined by Cowell, Gillies and Kaposi and reducibility as used by Prather and Giulieri.
(c) In the case of flowgraphs (Cowell, Gillies and Kaposi) all operation nodes are removed, thereby camouflaging certain unstructured compounds (except those involving *multi-exit loops*). The process of collapsibility isolates elementary unstructured flow-

charts; when applied to the structures in Figs 6 and 7, only Figs 6(b) and 7(b) are not collapsible (these involve *multi-entry loops*). Structured collapsibility removes unnecessary operation nodes and basic structured forms or base diagrams, retaining only those nodes which preserve the unstructuredness of the flowchart.

(d) A distinction is drawn between unstructured particles, unstructured elements and unstructured compounds and between different types of structuredness (D-structuredness, TD_1-structuredness, etc.). The sets of unstructured particles/elements corresponding to different types of unstructuredness are slightly different.

REFERENCES

1. R. E. Prather and S. G. Giulieri, Decomposition of flowchart schemata. *The Computer Journal* **24** (No. 3), 258–262 (1981).
2. M. H. Williams, A Comment on the decomposition of flowchart schemata. *The Computer Journal* **25** (No. 3), 393–396 (1982).
3. A. P. Ershov, Theory of Program Schemata in *Proceedings of IFIP Congress 71*, pp. 28–45, North-Holland, Amsterdam (1972).
4. S. R. Kosaraju, Analysis of structured programs. *Journal of Computer and System Sciences* **9**, 232–255 (1974).
5. G. Oulsnam, Unravelling unstructured programs. *The Computer Journal* **25** (No. 3), 379–387 (1982).
6. M. H. Williams, Generating structured flow diagrams: the nature of unstructuredness. *The Computer Journal* **20** (No. 1), 45–50 (1976).
7. T. J. McCabe, A complexity measure. *IEEE Transactions on Software Engineering* **SE-2** (No. 4), 308–320 (1976).
8. D. E. Knuth and R. W. Floyd, Notes on avoiding 'goto' statements. *Information Processing Letters* **1**, 23–31 (1971); Errata **2**, 177 (1972).
9. D. E. Knuth, Structured programming with goto statements. *ACM Computing Survey* **6**, 261–301 (1974).
10. C. Böhm and G. Jacopini, Flow diagrams, Turing machines and languages with only two transformation rules. *Communications of the ACM* **9** (No. 5), 366–371 (1966).
11. D. F. Cowell, D. F. Gillies and A. A. Kaposi, Synthesis and structural analysis of abstract programs. *The Computer Journal* **23** (No. 3), 243–247 (1980).
12. R. E. Tarjan, Testing flowgraph reducibility in *Proceedings of the 5th Annual ACM Symposium on Theory of Computing*, pp. 96–107 (1973).
13. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts (1975).
14. M. S. Hecht, *Flow Analysis of Computer Programs*. North-Holland, New York (1977).
15. M. S. Hecht and T. D. Ullman, Flowgraph reducibility. *SIAM J. Computing* **1**, 188–202 (1972).
16. E. Yourdon, A brief look at structured programming. *Systems/Stelsels* **4** (No. 5), 7–12 (1974).
17. F. T. Baker and H. D. Mills, Chief programming teams. *Datamation* **19** (No. 12), 58–61 (1973).
18. E. F. Miller and G. E. Lindamood, Structured programming: top-down approach. *Datamation* **19** (No. 12), 55–57 (1973).
19. M. H. Williams and H. L. Ossher, Conversion of unstructured flow diagrams to structured form. *The Computer Journal* **21** (No. 2), 161–167 (1978).
20. E. Ashcroft and Z. Manna, The translation of 'goto' programs to 'WHILE' programs in *Proceedings of IFIP Congress 71*, pp. 250–255, North-Holland, Amsterdam (1972).
21. W. A. Wulf, Programming without the GOTO in *Proceedings of IFIP Congress 71*, pp. 408–413, North-Holland, Amsterdam (1971).
22. G. Oulsnam, Cyclomatic numbers do not measure complexity of unstructured programs. *Information Processing Letters* **9**, 207–211 (1979).

Received November 1982

5. M.H.WILLIAMS and G.CHEN, Restructuring Pascal programs containing goto statements, The Computer Journal, vol. 28, no. 2, 134-137 (1985).

Restructuring Pascal Programs Containing Goto Statements

M. H. WILLIAMS* AND G. CHEN
Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ

Besides their bad influence on programming style, the presence of* goto *statements in a program can complicate considerably the problem of mechanical translation into another programming language. In view of this an algorithm is presented which will restructure any Pascal program which contains* goto *statements to produce an equivalent program without* goto *statements. A system using this algorithm has been developed in Prolog.

1. INTRODUCTION

The processes by which an unstructured program can be automatically restructured to produce an equivalent structured program have been studied by a number of workers. This work has covered various aspects, from the determination of what constitutes unstructuredness in flow diagrams or programs,[1,2,3,4] and the isolation of these unstructured components,[5,6,7] to the transformations which may be applied in order to map unstructured flowcharts or programs into equivalent structured ones (e.g. Ashcroft and Manna,[8] Williams and Ossher,[9] Oulsnam,[3,10] to mention but a few).

Until now most of the work which has been conducted in this area has been of theoretical interest. Although systems have been constructed (e.g. Williams and Ossher)[9] which will accept as input an unstructured flow diagram (or program) and will transform this into an equivalent structured form, there has not been any real need for such mechanical translation, with the result that these systems have not been put to any significant practical use.

However, there is an area of growing importance in which mechanical restructuring could play a very useful role in the future – namely, the area of language translation. The process of translating a program in one imperative language into an equivalent program in another language can be made considerably more complicated if the program concerned is unstructured, because of the differences in the semantics associated with entry to and exit from programming constructs in different languages. Translation is therefore facilitated if the program to be translated is in structured form.

The problem of language translation is likely to become more acute in the future, particularly if the Japanese are successful with their Fifth Generation Computer Project.[11] The aim of the latter project is to develop the next generation of computer systems, systems which are considerably more powerful and cheaper than present-day equivalents. The underlying machine code of these new systems is to be a form of logic programming language based on the language Prolog. These languages will be very different from classical imperative languages, with none of the familiar programming constructs used in sequential programming.

In view of the radically different approach required to program such machines, there has been much concern about the problem of running existing software on this type of hardware. For this reason it is essential to look at different ways of handling conventional languages in this new type of environment. Our particular interest lies in the way in which Pascal programs might be handled.[12]

* To whom correspondence should be addressed.

One of the biggest problems in this regard is the **goto** statement. The only practical way of handling this statement is to remove it altogether – in other words, to restructure any program which contains **goto** statements to produce an equivalent structured program which does not. This paper describes a system which has been developed to do this.

2. BASIC TRANSFORMATIONS

This section makes use of the hierarchical structure of Pascal. For instance, a block consists of a sequence of statements which are at the same level and will be executed sequentially. If one of the statements is a **for**, **repeat** or **while** loop, then the body of this loop is at a level which is one lower than the loop statement, and the sequence of statements making up the body will be executed sequentially. The same is true of **if** or **case** statements. Thus a program can be regarded as a hierarchical tree structure with simple statements (such as assignment, read, write or procedure call) as leaves and structured statements (**if**, **case**, **for**, **while**, **repeat**, **procedure** or **function**) as branch nodes. In this context a **goto** statement is a leaf node in the tree.

Although the Pascal standard[12] is ambiguous on the question of jumping into a structured statement, it is clearly the intention (Jensen and Wirth[13]) that:

(*a*) A **goto** statement outside a structured statement should not be directed to a label within the structured statement, and

(*b*) a **goto** statement within one limb of a conditional statement (case list element of a **case** statement, then or else part of an **if** statement) should not be directed to a label within another limb of the same conditional statement.

It is therefore assumed that neither of these conditions will be present in a valid Pascal program.

In discussing the basic transformations to be applied, each **goto** statement will be assumed to have the form:

if *P* **then** (*A*; **goto** *L*) **else** *B*;

where *P* represents a condition and *A* and *B* represent statement sequences, either of which may be empty. If a **goto** statement is not in this form, it can always be transformed into this form. All the '**begin**' and '**end**' symbols will be replaced by '('and')'.

The algorithm (which will be described in detail later) proceeds roughly as follows. First an innermost block is found which contains at least one **goto** statement. This block is checked for any **goto** statement which jumps to a label in the same block. If such a **goto** statement is found, it is removed using one of the transformations T1

followed if necessary by one or more of the transformations T3. If no such **goto** statement can be found, then a **goto** statement which jumps to a label in an outer block is found and removed using one of the transformations T2. Once all **goto** statements to a particular label L have been removed from a block, the label itself if present in the block can be removed and discarded.

The three basic sets of transformations required are as follows.

T1. Replacing a **goto** statement which is at the same level as the corresponding label. Here there are two basic transformations.

(a) Forward jump

```
if P then (A; goto L)          if P then A
    else B;                        else (B; C);
C;                             L;
L:
```

If the code sequence represented by C contains any labels, remove these from the **else** clause immediately using the transformation T3(a) so that the resulting program will be a valid Pascal program.

(b) Backward jump

```
L:C;                           L:repeat
if P then (A; goto L)              C;
    else B;                        flag:= P;
                                   if flag then A
                               until not flag;
                               B;
```

If the statement sequence represented by A is empty it is not necessary to use a flag so that the transformation will be

```
L:C;                           L:repeat
if P then goto L                   C
    else B;                    until not P;
                               B;
```

In either case if the code sequence represented by C contains any labels, remove these from the **repeat** statement using transformation T3(b) to produce a valid Pascal program. On the other hand if C is empty, a **while** statement may be used instead of a **repeat** statement

```
L:if P then (A; goto L)    L:while P do A;
      else B;              B;
```

T2. Replacing a **goto** statement which branches out of a structure. Here there are four possibilities and in each case the transformation moves the **goto** statement out of the structure concerned.

(a) Branch out of a **repeat** loop

```
repeat                         flag1:= false;
  A;                           repeat
  if P then (B;goto L)           A; flag1:= P;
      else C;                    if flag1 then (B; flag2:= true)
  D                                    else (C;D;flag2:= Q)
until Q;                       until flag2;
                               if flag1 then goto L;
```

(b) Branch out of **while** loop

```
while Q do                     flag1:= false; flag2:= Q;
  (A;                          while flag2 do
  if P then (B;goto L)           (A; flag1:= P;
      else C;                    if flag1 then (B; flag2:= false)
  D);                                  else (C;D;flag2:= Q));
                               if flag1 then goto L;
```

(c) Branch out of **then** clause

```
if Q then                      flag:= false;
(A;                            if Q then
if P then (B; goto L)            (A;
    else C;                      if P then (B; flag:= true)
D)                                   else (C;D))
  else E;                        else E;
                               if flag then goto L;
```

(d) Branch out of **case** option

```
case I of                      flag:= false;
1:(A;                          case I of
  if P then (B; goto L)        1:(A;
      else C;                    if P then (B; flag:= true)
  D);                                else (C;D);
2:E;                           2:E;
3:F                            3:F
end;                           end;
                               if flag then goto L;
```

T3. Transferral of a label out of a structure. When either of the following two transformations is applied, any **goto** statement to the label concerned will be changed from

if P **then** (A; **goto** L) **else** B;

into

if P **then** (A; $flag$:= **true**; **goto** L) **else** B;

If there is more than one label to be removed, the transformation must be applied repeatedly, removing the labels one by one in order of last appearance.

(a) Move label out of an **else** clause

```
if P then A                    if P then (A; flag:= false)
    else                           else (B; flag:= true);
      (B; L:C);                L: if flag then C;
```

(b) Move label out of a **repeat** loop

```
repeat                         flag:= false;
  A;                           L: repeat
  L:B                              if not flag then A;
until P;                           B; flag:= P
                                 until flag;
```

In addition to these transformations a **for** statement is changed to a **while** statement if there is any jump out of its body, and a **with** statement may need to be subdivided into two or more smaller **with** statements.

In the standard transformations the only cost is the auxiliary variables and some operations on these variables. However, Ashcroft and Manna have proved that it is impossible to translate **goto** programs to **while** programs without using auxiliary variables.

3. ABNORMAL EXITS FROM PROCEDURES AND FUNCTIONS

One common use of the **goto** statement in Pascal programs is as an abnormal exit from procedures and functions. The classical example of this is the exit to error recovery routines in a Pascal compiler.

Since the algorithm to be described in the following section restructures one segment (procedure, function or main body) at a time, any jump which is directed to a label outside the current segment is easily detected. For each such label, a unique global flag is used (as opposed to local

flags used in the transformations of the previous section), and each jump to this label in the segment is replaced by an instruction setting the flag and a jump to the end of the segment.

```
procedure proc                    procedure proc
  (A;                               (gflag:= false;
  if P then                         A;
    (B; goto L)                     if P then
    else C;                           (B; gflag:= true)
  D);                                 else (C;D));
```

If *proc* is a procedure which has an abnormal exit to a label L outside it, then any call to *proc* must be followed by an instruction which tests the global *flag* and jumps to label L accordingly

```
proc(...);                        proc(...);
                                  if gflag then goto L;
```

In the case of a function call, the expression containing the function call must be rewritten so as to isolate the function call in a separate assignment statement, and once again an instruction which tests the flag and jumps to label L is inserted at this point. Every statement in which an expression may appear must be checked and rewritten as required (e.g. **repeat**, **while**, **if**, write and read, **case**, procedure call and assignment). For example, the following assignment statement

$a:=f(g)+b;$

may need to be rewritten as

```
dummy:= f(g);
if gflag then goto L;
a:= dummy+b;
```

Here *dummy* is a variable having the same type as function f. In each case if the label L does not occur in the calling segment, the whole process is repeated.

One problem which arises in the case of functions is how an expression containing a function call should be rewritten in order to isolate the function call. The reason for the difficulty is that the order of evaluation of operands in Pascal is implementation-dependent. For example, consider the statement

$a:=i+f(i);$

where $f(j)$ is a function which returns the value of j and increases j by 1. This statement may either be rewritten as

```
dummy:= f(i);
if flag then goto L;
a:= i+dummy;
```

or as

```
dummy1:= i;
dummy2:= f(i);
if flag then goto L;
a:= dummy1+dummy2;
```

depending on what the final value of a should be (i.e. $2i+1$ or $2i$). The latter approach has been adopted in the system described in this paper.

In the absence of any **forward** directives abnormal exits from procedures and functions can be dealt with at the same time as each segment is restructured in a single pass of the program. However, if **forward** directives are used, more than one pass may be required.

4. THE ALGORITHM

In this context the term 'segment' will be used to refer to a procedure body, function body or the main body of the program, while the term 'structure' will refer to a sequence of statements at the same level in the tree which will be executed sequentially (this includes the body of a **repeat** statement, **while** statement, **then** clause, **else** clause, etc., as well as the body of a segment).

In the course of scanning the source program the algorithm sets up a symbol table containing the name of each user-defined procedure or function appearing in the program and for each such segment a list (ABLIST) of labels to which an abnormal exit might occur when the procedure or function is called.

The algorithm takes each program segment in turn and performs the following:

(*a*) If any procedure or function X which is called within the current segment has a non-empty ABLIST then:

(1) if X is a function, break any statement in which it is called into two parts (if necessary);

(2) insert a conditional jump to each label in the ABLIST of X after each call to X in the current segment.

(*b*) While any **goto** statement remains in the current segment perform the following.

(1) Find an innermost structure which contains one or more **goto** statements (i.e. a structure which does not contain any inner structures with **goto** statements but does itself contain at least one **goto** statement).

(2) If this structure contains a label L and a **goto** statement directed to this label then one of the transformations T1 can be applied. If there is a choice of possible labels and **goto** statements, choose a pair (L, **goto** L) for which the code sequence C in the relevant transformation does not contain some other label L' and a **goto** statement directed to L' (i.e. an innermost pair). Apply the appropriate transformation T1. If any labels are contained in the code sequence represented by C, remove these from the conditional statement by applying transformations T3. If there is more than one such label, the transformation must be applied repeatedly, removing the labels one by one in order of last appearance. If there are no **goto** statements directed to label L, remove this label. Return to step 1.

(3) The structure contains a **goto** statement which jumps out of the structure. If it contains any labels these can be removed, since it is an innermost structure with **goto** statements and step 2 has failed. In this case select such a **goto** statement and apply the relevant transformation T2.

(*c*) If the current segment C calls itself and its ABLIST is not empty then:

(1) if C is a function, break any statement in which it is called into two parts (if necessary);

(2) insert conditional jumps to each label in the ABLIST of C after each call to C in the current segment.

(*d*) If any **goto** statements remain, perform step (*b*) to remove them.

This process is applied to each segment in turn until no further change is effected. In practice for a program without **forward** directives this will require a single pass of the program.

The proof of correctness of the algorithm has been derived and is available.

5. IMPLEMENTATION IN PROLOG

In the implementation of this system in Prolog all the Pascal reserved words are defined as operators and the various constructs in Pascal are mapped into Prolog data structures by means of a set of rules.

The implementation consists of four phases:

- preprocess.
- program reduction.
- deletion of **goto** statements.
- postprocess.

The system considers each program segment in turn and for each segment applies each of these four phases in sequence. Thus it will preprocess, reduce, delete **goto** statements and then postprocess one segment before proceeding to the next.

In the first phase, the declaration information is saved and the segment is scanned and changed slightly to convert it to a Prolog term composed of operators and operands. In the second phase, each section of program which contains no label or **goto** statement is extracted from the program and stored in a Prolog data structure. After this step, the segment consists of structures which contain only **goto** statements, labelled statements, and tokens which represent the information held in the data structure. While this is done, the expressions are parsed and the procedure and function calls rewritten if necessary. In the third phase this greatly reduced segment is restructured into a computationally equivalent one which contains no **goto** statement. The last phase retrieves the information from the data structure and rebuilds the program in Pascal form.

The remainder of this section will focus on steps 2 and 3, and ignore the preprocess and postprocess phases of this implementation.

5.1. Reduction of the source program

Since the repeated application of transformations in phase 3 can be very slow, efficiency is an important consideration. Since it is obvious that all sections of program which contain no **goto** statement and no labelled statement do not require to be processed by the algorithm, each one can be removed from the program and replaced by a token. As each such section of program is identified in turn and replaced by a token, the token together with the original section of code is stored in a dictionary where it is kept to rebuild the Pascal program in phase 4. The program's overall efficiency is greatly improved by this approach. It is also necessary to parse each statement and expression to identify the calls to procedures or functions containing abnormal exits from their bodies, and to rewrite them as described in section 3. For this purpose the information in the ABLIST is used.

5.2. Replacement of goto statements

In order to realise the algorithm presented in section 4, several important procedures are used.

(*a*) A set of predicates perform the basic transformations and may call each other recursively.

(*b*) Analysis of structured statements to determine an innermost **goto** statement and label pair. For this purpose the position of each **goto** statement and label in a structure is stored in a list. From this list a **goto** statement is chosen and is put into the following form:

A; **if** *P* **then** (*B*; **goto** *L*) **else** *C*; *D*

(*c*) If a **goto** statement and its label are both present in the same structured statement, transformations T1 and T3 will be applied. Once all such **goto** statements have been removed from a structured statement, any remaining **goto** statements are removed using the transformations T2.

(*d*) A special predicate is used to treat abnormal exit from procedure or function bodies.

6. DISCUSSION

As mentioned in Oulsnam's paper,[3] the structured schemas generally require an increase in space–time requirements. The algorithm presented here requires additional variables and additional operations on these variables since $n+\frac{1}{2}$ loops[14] are not a feature of Pascal. In the basic transformations each flag is necessary, but when a number of these flags are used this can sometimes be reduced. The algorithm can be modified slightly to perform such a reduction.

The algorithm has been implemented in Prolog on a PDP 11/34 and it is hoped to transfer the program to either a DEC 10 or VAX 11/750 shortly.

REFERENCES

1. M. H. Williams, Generating structured flow diagrams: the nature of unstructuredness. *The Computer Journal* **20**, 45–50 (1976).
2. T. J. McCabe, A complexity measure. *IEEE Transactions on Software Engineering* **SE 2**, 308–320 (1976).
3. G. Oulsnam, Unravelling unstructured programs. *The Computer Journal* **25**, 379–387 (1982).
4. M. H. Williams, Flowchart schemata and the problem of nomenclature. *The Computer Journal* **26**, 270–276 (1983).
5. R. E. Prather and S. G. Guilieri, Decomposition of flowchart schemata. *The Computer Journal* **24**, 258–262 (1981).
6. M. H. Williams, A comment on the decomposition of flowchart schemata. *The Computer Journal* **25**, 393–396 (1982).
7. A. Shafibegly-Gray and R. W. Whitty, Correspondence of flowchart schemata. *The Computer Journal* **25**, 495 (1982).
8. E. Ashcroft and Z. Manna, The translation of 'GOTO' programs to 'WHILE' programs. In *Proceedings of IFIP Congress 71*, pp. 250–255. North-Holland, Amsterdam (1972).
9. M. H. Williams and H. L. Ossher, Conversion of unstructured flow diagrams to structured form. *The Computer Journal* **21**, 161–167 (1978).
10. G. Oulsnam, Cyclomatic numbers do not measure complexity of unstructured programs. *Information Processing Letters* **9**, 207–211 (1979).
11. T. Moto-Oka, Challenge for knowledge information processing systems. In *Fifth Generation Computer Systems*, pp. 3–89. North-Holland, Amsterdam (1982).
12. Specification for computer programming language Pascal, British Standards Institute, BS 6192: 1982 (1982).
13. K. Jensen and N. Wirth, *Pascal User's Manual and Report*. Springer-Verlag, New York (1975).
14. D. E. Knuth, Structured programming with goto statements. *ACM Computing Surveys* **6**, 261–301 (1974).

6. M.H.WILLIAMS and A.R.BULMER, Array handling in Cobol compilers, Software Practice and Experience, vol. 7, 469-474 (1977).

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 7, 469–474 (1977)

Array Handling in Cobol Compilers

M. H. WILLIAMS AND A. R. BULMER
Computer Science Department, Rhodes University, Grahamstown, South Africa

SUMMARY

Since Cobol is essentially a byte-oriented language, certain practical problems arise when implementing it on a word-oriented machine. The particular problem of handling arrays with SYNCHRONIZED clauses is discussed here and a method put forward for solving it.

KEY WORDS Cobol Array handling Compilers

INTRODUCTION

In implementing a Cobol compiler for a word-oriented machine, one of the problems which one encounters concerns the handling of arrays containing SYNCHRONIZED clauses. Since the problem is not an obvious one and since it has escaped the attention of some compiler writers, it merits some attention.

The machine involved in this particular case is an ICL 1900, which has a word size of four characters. All examples given here are based on this.

STORAGE ALLOCATION FOR RECORDS AND FIELDS

If a record is composed of group and elementary fields without the presence of an OCCURS clause (e.g. Figure 1(a)), the position of each field can be expressed quite simply as a constant relative to the start of the record, i.e.

$$\text{Address of field X} = B + k$$

where B is the base address of the start of the record, and k is some constant.

Even the presence of SYNCHRONIZED clauses do not affect this other than by changing the value of k (see Figure 1(b)).

If an OCCURS clause is present in a record one would expect an address mapping function of the form,

$$\text{Address of field X(I)} = B + k_1 + k_2 * (I - 1)$$

where

k_1 accounts for the displacement of the first occurrence of field X from the start of the record,

and

k_2 is a measure of the distance between successive occurrences of field X in the array.

Received 13 September 1976

© 1977 by John Wiley & Sons, Ltd.

However, if a SYNCHRONIZED clause applies to a field contained within the scope of the OCCURS clause, an anomalous situation may arise in which for any subfield of the OCCURS clause, k_2 is no longer a constant, but has a different value for the first

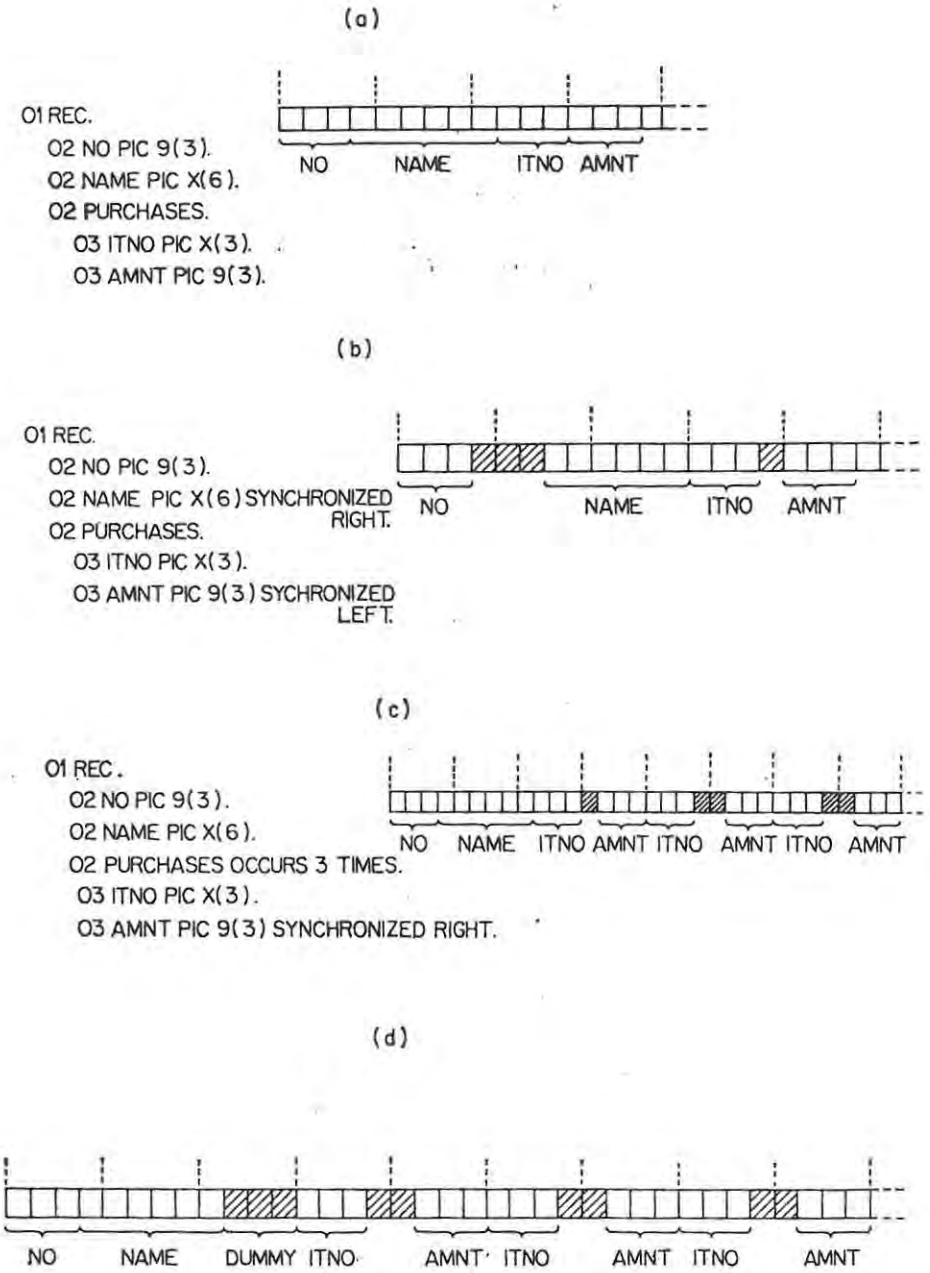


Figure 1. (a) A simple Cobol record without SYNCHRONIZED or OCCURS clauses, (b) a Cobol record with SYNCHRONIZED clauses, (c) a Cobol record with a SYNCHRONIZED clause within the scope of an OCCURS clause, (d) an alternative representation of the record in 1(c) using a dummy field

row from that for all other rows. For example, in Figure 1(c) the mapping function for ITNO(I) is

$$\text{Address of ITNO(I)} = \begin{cases} B+10 & \text{if } I=1 \\ B+10+7 & \text{if } I=2 \\ B+10+7+8*(I-2) & \text{if } I>2 \end{cases}$$

This mapping function becomes even more complex in the case of the two-dimensional and three-dimensional arrays.

DUMMY FIELDS

Since complex mapping functions such as the one above are very inconvenient from the point of view of code generation, this situation can be remedied by inserting a dummy field before the array to cause the first row to start on the same character position of a word as all other rows. This situation is illustrated in Figure 1(d). With this adjustment the mapping function for ITNO becomes

Address of ITNO(I) = B + 13 + 8*(I − 1)

In this case insertion of the dummy field means that the record occupies an extra word of memory. However, the insertion of a dummy field need not necessarily lead to a lengthening of the record, as illustrated by the example in Figure 2. Nevertheless in both cases the addition of a dummy field leads to a simplification of the address mapping functions.

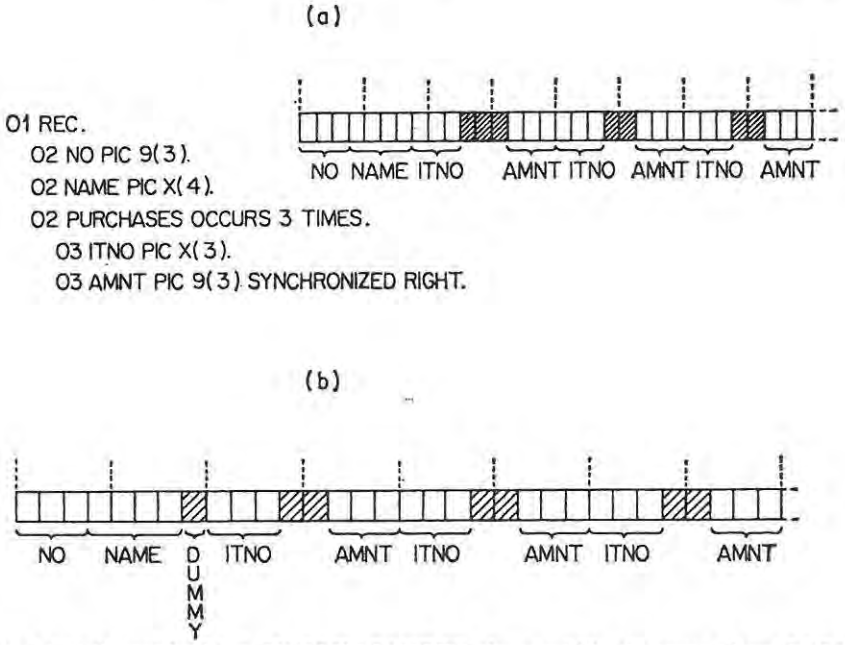


Figure 2. (a) A Cobol record with a SYNCHRONIZED clause within the scope of an OCCURS clause, (b) an alternative representation of (a) using a dummy field

ALGORITHM FOR DETECTION AND CORRECTION

In Cobol OCCURS clauses may be nested to a maximum depth of three. Thus in order to detect and correct for this condition three lists are used (L_1, L_2, L_3), one for each level of

nesting. The index i is used to refer to the current list L_i and pointers $FRONT_i$ and $BACK_i$ are used to keep track of the ends of each list. i is initially set to zero (i.e. no list) and incremented and decremented accordingly.

Fields outside the scope of an OCCURS clause are dealt with in the normal way, while fields affected by an OCCURS clause are handled by the algorithm shown in flowchart form in Figure 3. This algorithm associates with each list L_i a level number ($LEVEL_i$) which is the Cobol level number of the field with the OCCURS clause, and a flag ($FLAG_i$)

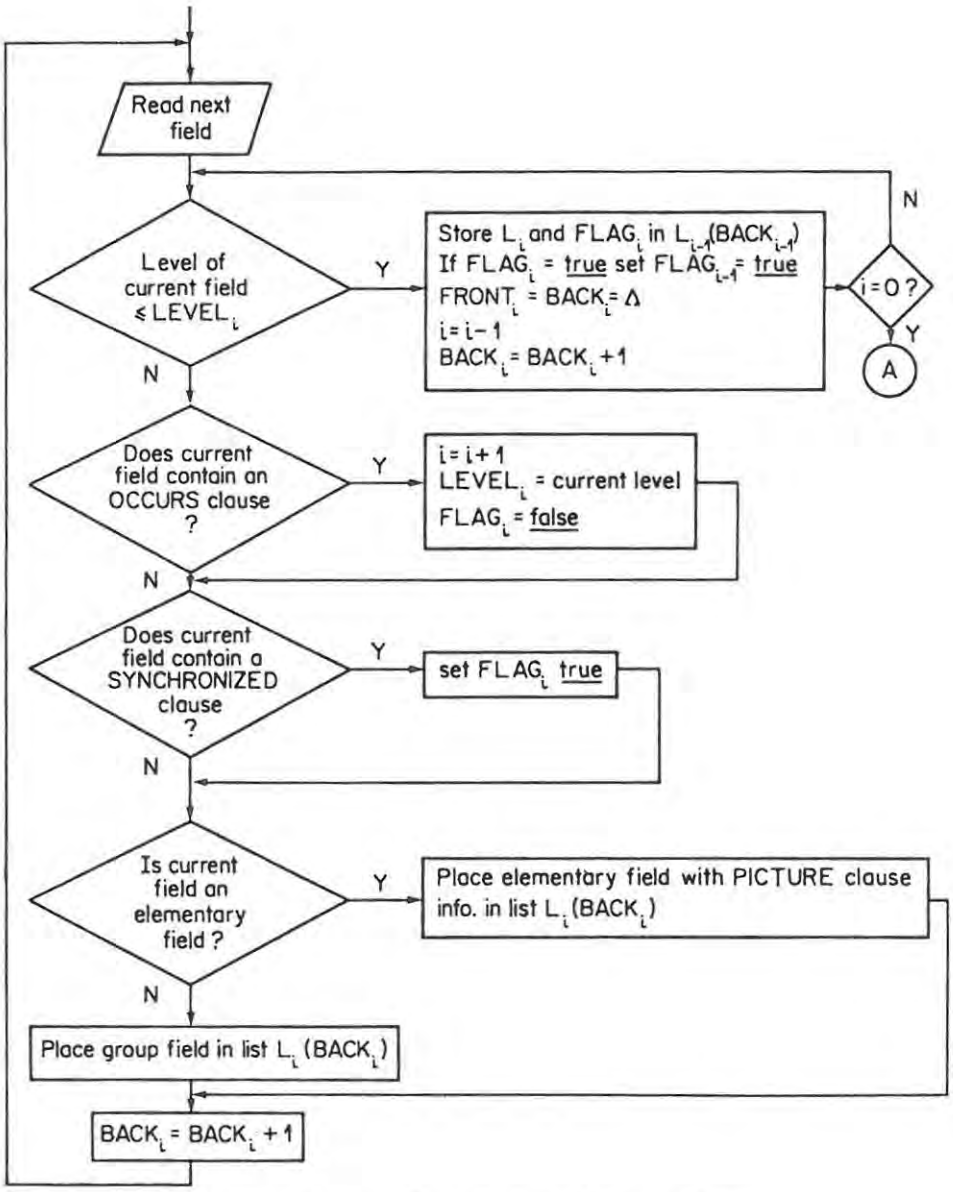


Figure 3. Flowchart for analyzing OCCURS clauses

which indicates whether any SYNCHRONIZED clauses have been encountered within the scope of the OCCURS clause. Each field is checked to see whether

(a) the end of the scope of an OCCURS clause has been reached,

(b) an inner OCCURS clause has been encountered, or

(c) it is SYNCHRONIZED.

On exit from the flowchart at point A, the following algorithm is applied to list L_1:

Algorithm A (Allocate Store)

1. Assume that space has been allocated up to some character position X − 1 and that one is about to allocate to position X. Beginning at position X, allocate space for each item in the list (starting from FRONT) as follows:
   (a) if it is an unsynchronized elementary field, allocate the space required;
   (b) if it is a synchronized elementary field, allocate space adjusting for alignment with boundaries;
   (c) if it is a group field (group fields may not be synchronized),[1,2] ignore it*;
   (d) if it is a list (in other words a nested OCCURS clause), apply this three step algorithm to this list.
2. When this is done, the first row is complete. Assume that the last character position occupied by the first row is position Y − 1. Now if the flag corresponding to this list is set (in other words there is at least one SYNCHRONIZED clause within the range of this OCCURS clause), one must determine whether a dummy field is required before the first row. This is done as follows:
   (a) if the first elementary field of the OCCURS clause is SYNCHRONIZED then no dummy field is necessary; otherwise
   (b) assume that the first character position of the first row (i.e. X) is the nth character position within a word, and that the first character position of the second row (i.e. Y) is the mth character position within a word. If $m = n$ then no dummy field is necessary, otherwise introduce a dummy field of length $m-n$ (if $m>n$) or $N+m-n$ (if $m<n$) where N = number of character positions per word.

   If a dummy field of length L characters is necessary, then shift the fields of the first row to the right in order to insert this dummy field at the beginning. This is done as follows:
   (a) All unsynchronized fields to the left of the first synchronized field are shifted L character positions to the right.
   (b) The first synchronized field and all fields to the right of it are shifted either zero or N places to the right depending on whether the shift in part (a) has caused an overlap to occur or not.
3. Allocate space for the remaining rows of the OCCURS clause.
   On completion of this algorithm all fields will have simple address mapping functions.

---

* Group fields are not used by this algorithm but are included here for completeness as it is necessary to know where group fields occur in a record.

ACKNOWLEDGEMENTS

One of the authors, Mr. A. R. Bulmer, was in receipt of a CSIR bursary when this work was carried out.

REFERENCES

1. International Computers Ltd., *Cobol*, 1971.
2. J. B. Maginnis, *Fundamental ANSI COBOL Programming*, Prentice-Hall Inc., Englewood Cliffs, N.J., 1975.

7. M.H.WILLIAMS, Is an exit statement sufficient?, Information Processing Letters, vol. 17, 47-51 (1983).

Information Processing Letters 17 (1983) 47–51
North-Holland

19 July 1983

IS AN EXIT STATEMENT SUFFICIENT?

M.H. WILLIAMS

Department of Computer Science, Heriot-Watt University, Edinburgh, EH1 2HJ, Scotland (U.K.)

Communicated by W.L. Van der Poel
Received February 1982

Keywords: Goto statement, exit statement, control structures

Introduction

The first control instruction to appear in a computer's instruction set was the goto instruction (or jump or branch instruction, as it is variously called). This primitive but powerful instruction has changed very little during the past three decades and is still the basic control instruction at machine level to this day.

When the first high level languages were developed, the concept of the goto instruction was carried over directly into these high level languages as the goto statement. At the same time three elementary forms of control structure were recognised: procedures, loops and choices (conditional statement or case statement).

From the outset the problems associated with entering a procedure at any point other than the beginning were acknowledged and languages were designed to prevent the user from attempting to do so by placing restrictions on the scopes of label names. Exit from a procedure is less hazardous and some languages permit one to exit from one or more nested procedures by means of a goto statement.

It was also recognised that a loop ought not to be entered through any point other than its first statement lest the control variable be set up incorrectly; but language designers had no strong feelings about this and restrictions were generally limited to verbal statements concerning the inadvisability of this manner of entry to loops, leaving it to the programmer to ensure that it did not happen. Exit from any point of the body of a loop appeared harmless enough, and again the goto statement provided the means of achieving this.

As far as the choice construct is concerned there was no apparent reason why entry to or exit from this construct should be restricted and so no restriction was placed on it.

But as language designers sought after greater generality, even the restriction on entry to procedures was relaxed and the concept of multiple entry points for subroutines was introduced.

This was the situation until the mid-1960s when the importance of simple program structures was realized and the idea of structured programming emerged. The fundamental tenet of the new doctrine is that the three control constructs (procedures, loops and choices) should each be restricted to have a single entry point and a single exit point, thereby making the structure of the program simpler.

But it was also recognised that the use of structured programming did not always produce the most concise or efficient programs [1], and as a result language designers have been loathe to exclude the goto statement from their languages. Thus even the well-structured language PASCAL [2] includes a goto statement in its repertoire of instructions.

This paper considers a generalized form of exit statement as an alternative to the goto statement. This statement is more restrictive than the goto statement but has the advantage that any program using it can easily be converted to a structured program.

0020-0190/83/$3.00 © 1983, Elsevier Science Publishers B.V. (North-Holland)

The exit statement

Consider a programming language in which each procedure has a single entry point (namely the beginning of the procedure body) and in which the normal exit point for a procedure is the end of the procedure body.

Assume that there are various forms of loop statement but that each loop must be entered via its first statement, and that the normal exit from a loop occurs when the loop test is satisfied appropriately.

Suppose that each conditional statement can only be entered via its initial condition and that exit normally takes place when execution of one of the option paths is completed.

However, suppose that there is also an exit statement with the following two forms:

(a) EXIT⟨procedure name⟩.

When this statement is executed, its effect is to return control from the current procedure to the calling procedure, and from this to each successive calling procedure until the named procedure is reached. Execution continues from immediately after the most recently executed procedure call in this procedure.

(b) EXIT⟨label⟩.

The label specified here must also be used to prefix a conditional statement or a loop, enclosing the exit statement. The effect of this statement is to pass control to the statement immediately following the labelled statement.

Suppose further that there are three standard Boolean variables:

PROCNORMAL	– which is initially true, and is set to false whenever an exit statement causes an exit from a procedure, and to true whenever a normal procedure exit occurs,
LOOPNORMAL	– same for loops,
CHOICENORMAL	– same for choices (conditional statements or cases).

Finally suppose that there is no goto statement (or its equivalent) in the language. The question is whether the exit statement can offer a reasonably simple and flexible alternative to the goto statement despite its constraints.

Some specific cases

The first example is taken from the area of compiler construction [3]. In a recursive descent parser there are many points in the program at which an error may be dectected. In each case one may want to print an error message, abandon what has been accumulated thus far and search ahead for the next beacon symbol. Fig. 1a shows part of such a program written using a structured programming approach without a goto or exit statement, Fig. 1b illustrates the same part rewritten using an exit statement. In practice, in a large recursive descent parser the overhead in space and time caused by the additional if statements, coupled with the resulting loss of clarity, make the

```
procedure statement;
begin var;
      if not err then
        if nextsym ≠ assign then error
        else begin lexan;
                   if not err then expression
             end
end;
procedure var;
begin recid;
      if not err then
        if nextsym = leftround then restfncall
        else if nextsym = leftsquare then subscripts
end;
procedure error;
begin
write ('error');
err:= true
end;
procedure master
begin
while not end do
  begin statement;
    if err then recover
  end
end
```

Fig. 1a. Part of a recursive descent analyzer without goto or exit statements.

```
procedure statement;
begin var;
      if nextsym ≠ assign then error
      else begin lexan; expression end
end;
procedure var;
begin recid;
      if nextsym = leftround then restfncall
      else if nextsym = leftsquare then subscripts
end;
procedure error;
begin
write ('error');
exit master
end;
procedure master;
begin
while not end do
  begin statement
    if not PROCNORMAL then recover
  end
end
```

Fig. 1b. The same program as in Fig. 1a, rewritten using exit statements.

structured programming approach very unattractive in this instance. As a result a goto statement is often used under these circumstances. An exit statement could equally well be used to produce the same effect.

The second example is the search program described by Knuth and Floyd [1], viz.

```
for i:=1 step 1 until n do
  if A[i] = x then goto found;
notfound: n:=i; A[i]:=x; B[i]:=0;
found: B[i]:=B[i]+1;
```

This can be expressed as a program using an exit statement as follows:

```
loop1: for i:=1 step 1 until n do
         if A[i] = x then exit loop1;
       if loopnormal then begin n:=i; A[i]:=x; B[i]:=0
         end;
       B[i]:=B[i]+1;
```

The third example is also taken from Knuth and is typical of a 'backtracking' application. The basic form of the program is as follows:

```
Start: m[1]:=0; k:=0;
up: k:=k+1; list(k), a[k]:=m[k];
try: if a[k] < m[k+1] then
  begin move (a[k]); goto up
  end;
down: k:=k-1;
  if k = 0 then goto done;
  unmove(a[k]);
  a[k]:=a[k]+1; goto try;
done :
```

This may be written using an exit statement instead of a goto statement as follows:

```
start: m[1]:=0; k:=0;
L1: repeat k:=k+1; list(k); a[k]:=m[k];
    while a[k] ⩾ m[k+1] do
      begin k:=k-1;
        if k = 0 then exit L1;
        unmove(a[k]);
        a[k]:=a[k]+1;
      end;
    move(a[k])
  until false;
```

The general approach

The goto statement as it appears in high level languages is usually restricted to jumping out of procedures or jumps within a procedure, but using a goto statement to jump into a procedure is not usually permissible. With this restriction in mind, any goto statement may be regarded as being made up of a sequence of zero or more returns from procedures, followed by a sequence of zero or more jumps out of loops and choices, followed by a simple jump which does not cross the boundary of any procedure, loop or choice construct. In other words it is equivalent to a possible procedure-type exit followed by a possible label-type exit followed by a simple jump. The latter may be implemented as an if statement in the case of a forward jump, or as a loop (such as a repeat statement) in the case of a backward jump.

The main difficulty when attempting to do without a goto statement lies in implementing unstructured forms. These are constructs which are not directly realizable with the normal tools of structured programming. The strict adherents of structured programming maintain that such con-

structs are unnecessary and that any algorithm containing such constructs should be redesigned to eliminate them. However, there are instances [1] when such constructs can be simple, efficient and clear, and when a user may feel justified in using them.

It has been shown [4] that there are five basic unstructured forms. These are the following:

(a) *Abnormal selection path*. This structure is illustrated in Fig. 2. It can be implemented using an exit statement as follows:

```
LAB: if a then
       begin B;
       if b then begin D; exit LAB end
       else C
       end
     else A;
     if CONDNORMAL then E
```

(b) *Loop with multiple exit points* (Fig. 3). This can be implemented as follows:

```
flag ← false;
LAB: repeat A;
       if a then exit LAB;
       B;
       if b then flag ← true else E
     until flag;
if LOOPNORMAL then D else C
```

(c) *Loop with multiple entry points* (Fig. 4). This

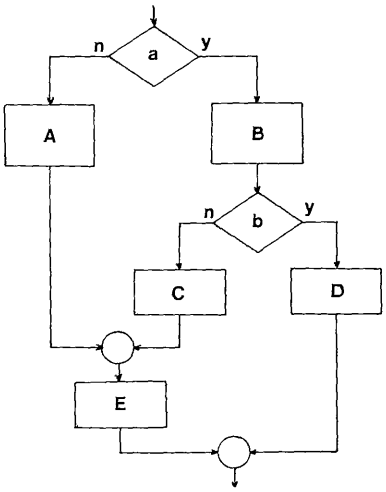


Fig. 2. Flow diagram of abnormal selection path.

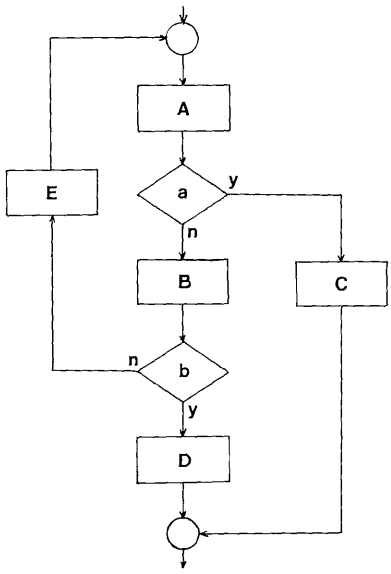


Fig. 3. Flow diagram of loop with multiple exit points.

can be implemented as:

```
LAB: if a then B else begin A; exit LAB end;
flag ← false;
repeat if CONDNORMAL then C;
       D;
       if b then flag ← true else E
until flag
```

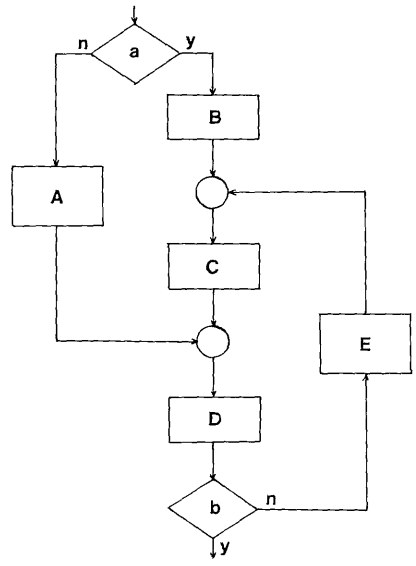


Fig. 4. Flow diagram of loop with multiple entry points.

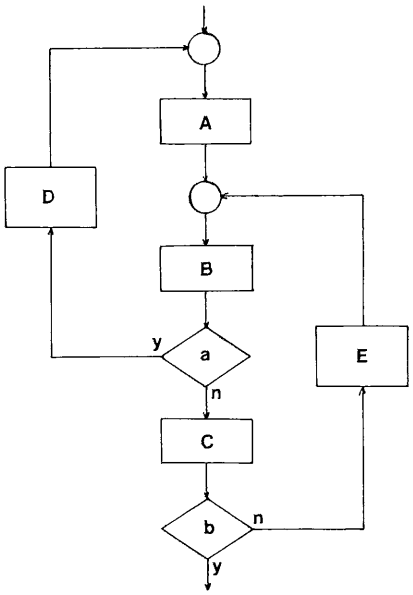


Fig. 5. Flow diagram of overlapping loops.

(d) *Overlapping loops* (Fig. 5). This structure can be realized using an exit statement as follows:

```
L1: repeat A;
  L2: repeat B;
      if a then exit L2;
      C;
      if b then exit L1;
      E
      until false;
      D
      until false
```

(e) *Parallel loops* (Fig. 6). This structure can be realized as follows:

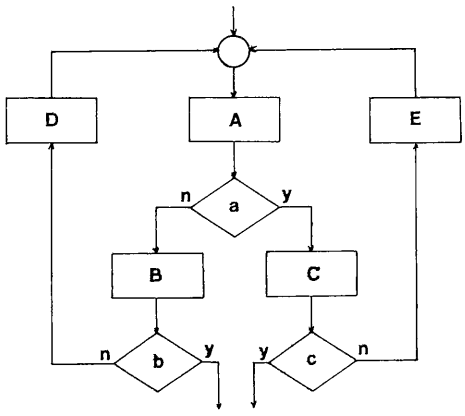


Fig. 6. Flow diagram of parallel loops.

```
LOOP: repeat A;
        if a then begin C;
          if c then exit LOOP;
          E
          end
        else begin B;
          if b then exit LOOP;
          D
          end
        until false
```

Conclusion

The exit statement described here is a control construct which is more limited than the goto statement. However, by judicious combination with conditional statements and loops it can be used to produce the same effect as any goto statement. In particular the five basic unstructured forms can be implemented using exit statements.

The chief advantage of the exit statement stems from its close relationship to the elements of structured programming, compared with the goto statement which bears no relation to structured programming. As a result a program employing exit statements but no goto statements can be transformed into a structured program relatively easily although the resulting program will inevitably be more cumbersome.

A less important advantage is that the exit statement lends itself to one-pass compilation moreso than the goto statement in that it does not require labels to be defined at the beginning of each procedure or the main body.

Whenever one encounters an exit statement which refers to a label, the definition of that label should have preceded it; otherwise the statement is in error.

References

[1] D.E. Knuth and R.W. Floyd, Notes on avoiding 'Goto' statements, Inform. Process. Lett. 1 (1971) 23–31.

[2] C.A.R. Hoare and N. Wirth, An axiomatic definition of the programming language PASCAL, Acta Inform. 2 (1973) 335–355.

[3] D.E. Knuth, Structured programming with goto statements, ACM Comput. Surveys 6 (1974) 261–301.

[4] M.H. Williams, Generating structured flow diagrams: The nature of unstructuredness, Comput. J. 20 (1) (1977) 45–50.

8. M.H.WILLIAMS, Yet another conversational programming language - BPL, Proceedings of the ACM 78 conference, ACM, New York, 516-521 (1978).

YET ANOTHER CONVERSATIONAL PROGRAMMING LANGUAGE - BPL

M.H. Williams

Dept. of Computer Science and Applied Mathematics
Rhodes University
Grahamstown, 6140, South Africa

BPL is an interactive language which is derived from the languages BASIC and Pascal. It was designed as a two-tier language to cater for beginners as well as advanced programmers. The lower level is very similar to BASIC with a few slight modifications particularly to the procedure/function declarations and control statements. The higher level incorporates most of the flexible data structures available in Pascal. The resulting language provides the advantages of both BASIC and Pascal.

Key words: Programming language, BASIC, Pascal, BPL.

1. Introduction

Many departments in educational institutions which are responsible for teaching computer programming in one form or another, are faced with the problem of what programming language to teach. If one is teaching Computer Science at an undergraduate level, the choice of first language is an important one since the language which a student learns first, does, to a greater or lesser extent, affect his programming habits and his approach to problem-solving generally. On the other hand if one is giving a service course to first-year Mathematics or Chemistry students, one's criteria for choosing a language might be somewhat different. And if one happens to be in the position of teaching both (Computer Science undergraduates and service courses), one may well find that one uses two different languages for the different groups.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery, Inc. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1978 ACM 0-89791-000-1/78/0012/0516 /$00.75

In attempting to decide on a suitable language three possibilities which suggest themselves, are : BASIC, FORTRAN and Pascal. The proponents of each are likely to put forward the following arguments in their favour :

BASIC : This language is very easy to learn and easy to teach. Students get off the ground quickly and are assisted by the system itself. In general after a limited period of time students who have learnt BASIC have a better understanding of what he/she/the computer is doing than students who are taught either of the other two languages.

FORTRAN : BASIC suffers from the fact that it is too simple-minded and does not include sufficient versatility. Fortran provides more versatility in the form of subroutines and functions, a wider range of data types, storage of compiled subroutines on libraries, etc.

Pascal : Both of the previous two languages are "unstructured languages" - i.e. languages which do not lend themselves to the writing of well-structured programs. Pascal is a "well-structured language" which lends itself to good programming habits, and has a wider range of data types than either FORTRAN or BASIC.

Ideally one would like a language which has the good features of all of these, viz. one which is easy to teach and learn, is structured, versatile and has a wide range of data types. It was for this reason that BPL was designed. BPL

(which stands for BASIC-Pascal-Liaison or Better Programming Language or simply a conversational language developed after APL) aims at combining the simplicity of BASIC with the structuredness and range of data types of Pascal.

BPL has been designed specifically as a two-tier language. The lower level is a subset of the language which is very close to BASIC. It is self-contained and very useful for teaching as service courses to other disciplines, partly because it is reasonably close to standard BASIC with its attendant simplicity and fairly universal availability, and partly because it is part of a more general language the facilities of which could be called upon if necessary.

The higher level is the complete language which is useful for teaching as a first language to Computer Science undergraduates. It too is simple but has a wide range of data types. Both levels are well-structured languages.

The concept of dividing a language into subsets is not new - other examples are SP/K[5] and SF/K[4]. What is important in this case is the nature of the two levels - the lower level being very similar to BASIC, the higher level being a natural extension of this with additional data types similar to those available in Pascal. As with other conversational languages such as JOSS[10], JEAN[7], BASIC[2], FORCON[3], etc., the language is very easy for beginners to learn but unlike many of the conversational languages, BPL is well-structured. Like APL[6] it is more powerful than most other conversational languages, in this case because of the wide range of data types available in the full language while it still retains much of the simplicity of structure inherent in Pascal.

This paper presents a brief outline of the language.

2. The Lower Level - a Simple Subset

As mentioned before this is a subset of the full language which is very similar to BASIC. Very briefly the language can be summarized as follows :

2.1. Variables and expressions

As in BASIC there are two data types : numbers and strings. However, variable names may be of any length : a numeric identifier consisting of any number of letters and digits the first of which is a letter, and a string identifier being similar except that it is followed by a dollar sign (e.g. A, A$,SUM, NAME$). Thus one can adhere to standard BASIC names for variables or one can use more meaningful ones.

Simple variables and arrays are referred to by unique identifiers of the appropriate type. Thus one cannot have a single variable with the same name as an array (a point which has always been a little confusing about BASIC). Simple variables are not declared but arrays must be declared in a DIM statement which has the same format as in BASIC.

The standard functions available are those which are common in BASIC, viz. SIN,SQR,INT, etc. Numeric expressions are the same as in BASIC, as are string expressions, using + to denote concatenation of strings and ** to denote exponentiation.

2.2. Simple program statements

The following statements are virtually the same as in standard BASIC [1,9].

{LET}$_0^1$<var>=<exp>

DATA <const>{,<const>}$_0^\infty$

FOR<var>=<exp>TO<exp>{STEP<exp>}$_0^1$

NEXT<var>

GOTO<line number>

STOP

REM

RESET

FILE#<numeric exp>:<string exp>

READ{#<channel no>:}$_0^1${USING<format>,}$_0^1$
<destination list>

PRINT{#<channel no>:}$_0^1${USING<format>,}$_0^1$
<print list>

FORMAT<format list>

If an array identifier without subscript(s) occurs in the destination list of a READ statement, data items are read into the array as in a MATREAD. Similarly if an unsubscripted array identifier occurs in the expression list of a PRINT statement, the contents of the array are printed as in a MATPRINT. Another BASIC statement which has been replaced is INPUT, the effect of which is achieved by

READ#CONS: ...

As in BASIC the function TAB(x) is used to obtain variable horizontal spacing on output. An additional function NL(x) is used to move x lines vertically.

2.3 Conditional statements

Two statements which have a slightly different form from standard BASIC are the two-way and multi-way branches, IF and ON. The BASIC IF statement has been replaced by a two-way conditional statement which is either an IF-THEN statement which has either of the forms

IF<Bool exp>THEN<uncond stm>ENDIF

or

IF<Bool exp>THEN{<uncond stm>}$_0^1$
{<statement>}$_0^\infty$
ENDIF

or an IF-THEN-ELSE statement which has either of the forms :

IF<Bool exp>THEN{<uncond stm>}$_0^1$
{<statement>}$_0^\infty$
ELSE<simple stm>ENDIF

or

IF<Bool exp>THEN{<uncond stm>}$_0^1$
{<statement>}$_0^\infty$
ELSE{<simple stm>}$_0^1$
{<statement>}$_0^\infty$
ENDIF

An <uncond stm> is any of the following : assignment statement, READ, PRINT, RESET, RETURN, STOP, CALL or GOTO. A <simple stm> is an <uncond stm> or an IF, ON or FOR statement.

The ON statement has the form

ON<numeric var>

$$\left\{\begin{array}{l} \text{CASE<num const>}\{,\text{<num const>}\}_0^\infty\text{:<simple stm>} \\ \qquad \{\text{<statement>}\}_0^\infty \end{array}\right\}_1^\infty$$

$$\left\{\begin{array}{l} \text{DEFAULT :<simple stm>} \\ \qquad \{\text{<statement>}\}_0^\infty \end{array}\right\}_0^1$$

ENDON<numeric var>

Its effect is to pass control to the case whose constant (or one of whose constants) matches the value of the variable. If the value of the variable does not match any of the cases, control is passed to the default section if one is present, or otherwise to the statement after the ENDCASE. The <numeric var> in the ENDCASE statement must match the <numeric var> in the ON statement as for FOR-NEXT loops. ON statements may be nested but as with loops, an ON statement or any of its CASE parts may not interleave with another ON statement, CASE part, conditional statement or loop.

2.4. While and repeat loops

Two additional types of loop which have been added are the WHILE and REPEAT loops. A WHILE loop has the form

WHILE<Bool exp>DO<loop id>
{<statement>}$_0^\infty$
ENDLOOP<loop id>

and the REPEAT loop has the form

REPEAT<loop id>
{<statement>}$_0^\infty$
UNTIL<Bool exp>FOR<loop id>

The loop id serves to identify the two parts of the loop. As with FOR-NEXT loops these loop ids must match and must be different from the loop ids of any enclosing loops.

2.5. Procedures and functions

A procedure definition has the form

PROCEDURE<pname>{(<f par>{,<f par>}$_0^\infty$)}$_0^1$
{<statement>}$_0^\infty$
ENDPROC

and is called by a statement of form

CALL<pname>{(<actual par>{,<actual par>}$_0^\infty$)}$_0^1$

A function definition has the form

FUNCTION<fn name>(<f par>{,<f par>}$_0^\infty$)
{<statement>}$_0^\infty$
ENDFN

and is called from within an expression in the usual way.

A formal parameter (<f par>) may be a string or numeric identifier (in which case the corresponding actual par must be a string or numeric expression) or it may be a string or numeric identifier followed by
: ARRAY(<integer>{,<integer>}$_0^\infty$)
(in which case the corresponding actual parameter must be the name of an array of the appropriate type whose bounds correspond to the integers in the formal parameter definition).

When a procedure or function is called, the statements within the procedure or function body are executed until either a statement of the form

RETURN

is reached or an ENDPROC or ENDFN is encountered, in which case control is returned to the statement after the call in the calling segment.

2.6. Statement formats

As in BASIC each indirect statement is preceded by a line number. Each procedure and function has its own set of line numbers and the numbers associated with the statements in any one procedure or function are independent of line numbers which occur in any other procedure or function or in the main body of the program (as in the case of labels in separate Fortran segments). A GOTO statement may only refer to a line number within the segment in which it occurs.

In order to handle the input and editing of procedures, two modes of operation are necessary :

- main body mode - in which all indirect statements are stored as part of the main program body
- procedure mode - in which all indirect statements are stored as part of the current procedure or function body.

When one starts a new program, the system is in main body mode. If one encounters a PROCEDURE or FUNCTION statement, the system checks to see whether it is already in procedure mode; if it is, it rejects the statement, otherwise it accepts it and enters procedure mode.

Once in procedure mode, indirect statements of the procedure body are entered. These may be entered in any order provided that the PROCEDURE or FUNCTION statement has the lowest line number and the ENDPROC or ENDFN statement has the highest. One remains in procedure mode until the ENDPROC or ENDFN statement is encountered whereafter the mode reverts to main body mode. If one wishes to return from main body mode to a procedure which has already been entered in order to alter it, one gives the command

EDITFN <proc or fn name >

This causes the system to be placed in procedure mode and the appropriate procedure is once again accessible. One may extract oneself from procedure mode once again by giving the appropriate ENDPROC or ENDFN statement, this time either as a direct statement or as an indirect statement.

Note that neither DIM nor DATA statements may occur in a procedure or function body - these must be part of the main program body.

2.7 Commands

The standard commands (including debugging commands [2]) are :

NEW	RESEQUENCE
OLD	LIST
SAVE	TRACE
REPLACE	UNTRACE
RUN	BREAK
BYE	UNBREAK

Other useful commands are

VTRACE<name>{,<name>}$_0^\infty$

UNVTRACE<name>{,<name>}$_0^\infty$

for tracing when the contents of the named variables (single variables or arrays) are altered.

3. The Higher Level - The Complete Language

3.1. Data types

The following simple data types are available

- Basic data types:
  - REAL
  - INTEGER
  - BOOLEAN
  - FIXED LENGTH STRINGS
  - VARIABLE LENGTH STRINGS
- declared data types:
  - scalars
  - subranges
  - pointers
  - records

A variable is a single instance of a simple data type, an array is a collection of instances of the same simple data types ordered in one or more dimensions.

A scalar (or definable scalar type) is a special type of string which can only assume certain strings as values. In fact scalar constants are written in a similar fashion to string constants except that single quotes ' ' encompass a scalar constant while double quotes " " enclose a string constant, e.g.

'MONDAY' is a scalar constant
"MONDAY" is a string constant

A subrange is a special type of integer which can only take on a limited range of integer values. A record is an aggregate of fields, each of which may be a simple data type. A pointer is an address of an object of a particular type.

Before one can declare variables or arrays of scalars, subranges, pointers or records, one must define the data type. This is done with a TYPE statement which has the form

TYPE<type id>=<type def>
<type def>=<int>..<int> for a subrange
or (<sc const>{,<sc const>}$_0^\infty$)
for a scalar
or POINTER TO <type>
for a pointer
or RECORD<f def>{;<f def>}$_0^\infty$END
for a record
<f def>=<field name>{:<type>}$_0^1$
<type>= any of basic types or
a type identifier or
POINTER TO followed by a type id

If no type is specified the type is assumed by default to be either REAL (if the field name is numeric) or variable length string (if the field name is a string identifier).

Facilities for variant parts (i.e. different sets of fields corresponding to the different values of one particular field) are also included but these will not be described here.
An example of a set of type declarations is :

```
10 TYPE AGETYP = 15..65
20 TYPE SEXTYP = ('MALE','FEMALE')
30 TYPE EMPLREC=RECORD NAME$;NO;
      AGE:AGETYP;SEX:SEXTYP;BASICWAGE
      END
```

3.2. Declarations

Simple variables are declared by statements of the form

VAR<identifier>{,<identifier>}$_0^1$:<type>

If a variable is not declared then it will be allocated automatically on encountering the first reference to it, and its type will be assumed by default to be REAL, if the variable name is a numeric identifier, or variable length string with default maximum length if the variable name is a string identifier.

Arrays must be declared by a DIM statement. The only extensions included are :
(a) the type of an array may be specified (if it is not, REAL or STRING is assumed as with simple variables), and
(b) each bound may be any of a single integer (upper bound only), a pair of integers, a subrange type or a scalar type.
Array elements are accessed as in BASIC.

When accessing a field (say X) of a record A, one writes

A.X

Any of the fields of a record may in turn be of type record. Thus to access a field X of a record field Y of a record A, one writes

A.Y.X

and so on. If one wishes to access a field I of a record which is the (I+1)st element of an array of records, one might have

A(I+1).X

3.3. Pointers

A variable which is declared explicitly or implicitly (by automatic allocation) and whose

scope is linked to the program structure, is referred to as a static variable. By contrast a dynamic variable is one whose scope is not linked to the program structure but may be created or destroyed at arbitrary points in the program [8].

A dynamic variable is created by a call to the standard procedure CREATE(p). This allocates a new variable v and places a pointer to this variable into the pointer variable p. Once created, a dynamic variable may only be accessed through the pointer p. Thus while <pointer id> refers to the pointer variable itself

<pointer id>↑

refers to the dynamic variable pointed to by the pointer variable. If the dynamic variable is a record and one wishes to refer to a field X of the record pointed to by the pointer variable P, one would write

P↑.X

and so on.

3.4. Statements

The assignment statement is generalized to cater for assignment to real, integer, subrange, Boolean, string, scalar or pointer variables as well as records and arrays.

The conditional statement, GOTO, STOP, REM, FILE, FORMAT and RESET statements remain unchanged. The READ and PRINT statements are the same as those available at the lower level except that any type of expression or array name, excluding pointer type, may occur in the destination or expression list. In addition if the format is specified as "BINARY", values will be transferred as binary numbers instead of being converted to or from ASCII.

The FOR statement has two forms - the form for numeric variables (integer, real or subrange), which is the same as the BASIC FOR statement, and the form for scalar variables, which is as follows :

FOR<scalar var>=<scal exp>TO<scal exp>

—

NEXT<scalar var>

The ON statement has three forms - for numeric variables, string variables and scalar variables.

3.5. Procedures and Functions

In view of the wider range of data types at the higher level, the types of formal parameters for procedures and functions, and the type returned by the function itself, may be declared. If not declared, REAL or STRING will be assumed.

Since procedures and functions may be compiled separately and stored in compiled form, one needs to distinguish between local and global variables. For this purpose one may have one or more global blocks. These resemble in some ways the COMMON blocks of Fortran. Global blocks may be named or if no name is specified, the blank global block is assumed. A variable occurring in a global block in one segment will be associated with a variable of the same name occurring in the same global block in another segment (irrespective of the order in which the variables are written in the GLOBAL statement) and the system will check that the types of the two occurrences are identical.

The format of the LOCAL statement is

$$\text{LOCAL<id>}\{\text{:<type>}\}_0^1\{\text{,<id>}\{\text{:<type>}\}_0^1\}_0^\infty$$

and that of the global statement is

$$\text{GLOBAL<glist>}\{(\text{<glob block name>})\}_0^1$$

$$\text{<glist>::=<gitem>}\{\text{,<gitem>}\}_0^\infty$$

$$\text{<gitem>::=<id>}\{\text{:<type>}\}_0^1$$

A variable which is not declared in a GLOBAL or LOCAL statement is assumed to be in the blank global block.

Procedures and functions may be called recursively.

3.6 Order of Statements

The first statements in a program must be TYPE, VAR and DIM (if they are required). The first statements in a procedure or function must be LOCAL and GLOBAL (if they are required).

4. Example - Monkey Puzzle Sort

```
10 REM ********************************************
20 REM *  THIS PROGRAM READS IN N NUMBERS AND     *
30 REM *- SORTS THEM INTO ASCENDING ORDER USING A *
40 REM *  MONKEY PUZZLE SORT                      *
50 REM ********************************************
60
70 TYPE TREEREC=RECORD LLINK:POINTER TO TREEREC;
            INFO; RLINK:POINTER TO TREEREC END
80 TYPE TREEPTR=POINTER TO TREEREC
90 VAR P,ROOT:TREEPTR

10 PROCEDURE TRAVERSE(Q: TREEPTR)
20 REM THIS TRAVERSES A BINARY TREE IN POSTORDER
30 IF Q↑.LLINK ≠ NIL THEN CALL TRAVERSE(Q↑.LLINK)
            ENDIF
40 PRINT Q↑.INFO
50 IF Q↑.RLINK ≠ NIL THEN CALL TRAVERSE(Q↑.RLINK)
            ENDIF
60 ENDPROC

10 PROCEDURE INSERT(Q:TREEPTR,X)
20 REM THIS INSERTS AN ITEM X INTO A BINARY TREE
30 IF Q = NIL THEN CREATE(P)
40 P↑.LLINK = NIL
50 P↑.RLINK = NIL
60 P↑.INFO = X
70 Q = P
80 ELSE IF X<Q↑.INFO THEN CALL INSERT(Q↑.LLINK,X)
90 ELSE CALL INSERT(Q↑.RLINK,X) ENDIF
100 ENDIF
110 ENDPROC

100 ROOT = NIL
110 READ N
120 FOR I = 1 TO N
130 READ NUMBER
140 CALL INSERT(ROOT,NUMBER)
150 NEXT I
160 CALL TRAVERSE(ROOT)
170 END
```

5. Conclusion

This paper gives a very brief-description of the interactive language BPL. Its purpose is not to define the language completely as a complete specification of the language will be published in due course, but rather to give the reader a flavour of the language. The two-tier approach used is aimed at providing for both beginners and more advanced programmers. The lower level language is a language very similar to standard BASIC, but with a slightly altered approach to procedures and functions and control statements; the higher level is an extension which includes a wide variety of data types.

References

1. Bull, G.M., Dynamic Debugging in BASIC, The Computer Journal, 15, 21 (1972).

2. Bull, G.M., Freeman, W., and Garland, S.J., Specification for Standard BASIC, (NCC Publications, Manchester, 1973).

3. FORCON, (International Computers Ltd., Letchworth, England, 1972).

4. Holt, R.C., and Hume, J.N.P., Fundamentals of Structured Programming using Fortran with SF/K and Watfiv-S, (Reston Publishing Co., Reston, Virginia, 1977).

5. Hume, J., and Holt, R.C., Structured Programming using PL/1 and SP/K, (Reston Publishing Co., Reston, Virginia, 1976)

6. Iverson, K.E., A Programming Language, (John Wiley, New York, 1962).

7. JEAN, (International Computers Ltd., Letchworth, England, 1968).

8. Jensen, K., and Wirth, N., Pascal User Manual and Report, (Springer-Verlag, Berlin, 1976).

9. Lee, J.A.N., The Formal Definition of the BASIC Language, The Computer Journal, 15, 37 (1972).

10. Shaw, J.C., JOSS : A Designer's View of an Experimental On-Line Computing System, in Proc FJCC, 26, 455 (1964).

9. M.H.WILLIAMS, Design principles of the language BPL, Quaestiones Informaticae, vol. 1, no. 2, 23-25 (1979).

Design Principles of the Language BPL

M.H. Williams

Dept. of Computer Science and Applied Maths, Rhodes University, Grahamstown, South Africa.

Abstract

BASIC and Pascal are two rather different programming languages, both of which have proved to be very popular. A new language BPL has been designed as an interactive programming language based on these two languages, combining advantages of each. This paper discusses some of the criteria used in the design of the language.

1. Introduction

The programming language BASIC was originally created by J.G. Kemeny and T.E. Kurtz [4] of Dartmouth College, U.S.A., as a language for teaching programming in an educational environment [2, 4]. Its simplicity has led to its popularity and currently it is widely used for teaching purposes. In addition it is the high-level language which is most common on microcomputers and mini-computers today.

Pascal was also created as a language for teaching programming [3, 7] and while it is very different from BASIC, it too has proved to be very popular. The main reasons for this are undoubtedly the wide range of data types and the neat control structures available in the language [5].

Both languages do, however, have certain disadvantages and it was for this reason that the language BPL has been proposed — once again as teaching language. BPL [6] aims at combining the simplicity of BASIC with the wide range of data types and neat control structures of Pascal to produce an interactive language which on the one hand is easy to teach and learn while on the other offers a powerful and flexible tool for solving problems. BPL is currently being implemented in Pascal on an ICL 1900 computer. The purpose of this paper is to examine some of the design decisions embodied in BPL.

2. General Principles

The chief advantages of the language BASIC are considered to be the following:

(a) The concept of combining programming language and system command language into one system is a very useful one especially for beginners who are learning to use an interactive system. It is also very useful in simple microprocessor systems where savings can be effected by combining these two. The particular set of commands available in most BASIC systems is based on the commands NEW, OLD, SAVE, LIST, RESEQUENCE, TRACE, BREAK, etc., as proposed by Kemeny and Kurtz and extended by Bull [1]. These are simple and easy to use, and are generally ideal for most purposes.

(b) Input-output (via the console or disc files) is simple for the beginner but is also sufficiently flexible to handle most advanced applications too.

(c) The notion of only two data types (reals and strings), the association of a data type with the form of the identifier and the automatic allocation of variables is considered to be an advantage by many users.

(d) The modus operandi of the system, viz. syntax checking as each statement is entered into the system and the ability to correct it immediately if it is incorrect, the ability to stop a program and access its variables and then continue execution, and so on, is very useful from the programmer's point of view.

(e) String variables are simple to use and yet the operations available are powerful.

On the other hand Pascal has the following important advantages:

(a) The neat control structures (with the exception of GOTO) which are available in the language make it easy for the user to write neat and well-structured programs.

(b) The wide range of data types available assists the user in producing more readable programs. Indeed everything that one can do with defined scalar variables, pointer variables, or even record variables can be achieved in other ways in other languages — however, the result is not necessarily as readable as it is in the case of Pascal.

(c) The notion of having to declare any variable before it is used provides a useful check on the programmer's intent.

(d) The length of an identifier in Pascal is much more convenient than that in BASIC and also contributes towards producing more readable programs.

The aim of BPL is to combine the advantages of both of these systems. BPL was designed as two-tier language with the following levels:

(a) The lower level is a subset of the complete language intended for beginners. It is very similar to BASIC and has two data types — real and variable length strings. All variables are allocated automatically although all arrays must be declared.

(b) The higher level is the complete language, in which a much wider range of data types is available, viz. real, integer, Boolean, defined scalar, subrange, record, pointer, set, fixed-length string or variable-length string. One may also have arrays of any of these types.
The set of statements available at the higher level is almost identical to that available at the lower level, with the exception of a few additional statements to cater for the extra data types. The set includes many of the standard statements from BASIC with the exception of the control statement which are replaced by structured forms.

3. Some Aspects of the Language

3.1 Identifiers

As in Pascal identifiers may be of any length although only the first n characters are significant (where n is an implementation dependent limit value ≥ 8 in Standard BPL). In a similar fashion to BASIC an identifier may be one of two types:
(i) A plain identifier — which consists of a sequence of letters and digits beginning with a letter.
(ii) A string identifier — which consists of a sequence of letters and digits beginning with a letter, and followed by a $ sign.
String identifiers are used for variables, arrays or functions of type string while plain identifiers are used in all other instances.
The idea in BASIC of having a single variable and an array bearing the same name has always caused confusion amongst beginners and thus in BPL variable, array and procedure names must be unique. One consequence of this requirement is that one need not distinguish between READ, PRINT, etc., and MATREAD, MATPRINT, etc. If any array name without subscripts appears in the destination list of a READ statement, values are to be read into the whole array; likewise for a PRINT statement the whole array is to be printed.

3.2 Control Statements

The statement numbers in BASIC serve a dual purpose — they act as line numbers for the purpose of editing programs and they serve as destinations for GOTO and other statements. This aspect of BASIC has been criticised since it is contended that the presence of labels encourages the use of GOTO statements which in the long run leads to poorly structured programs. To avoid this it has been decided to omit the GOTO statement altogether from BPL's repertoire of statements.
Structured forms of IF-THEN and IF-THEN-ELSE statements are already available in many BASIC systems and these have been adopted in BPL. The FOR-NEXT loop in BPL is a simple extension of the BASIC loop (this is in turn more powerful than the for loop in Pascal which does not permit the controlled variable to be of type real nor allows for any increment other than $+1$ or -1). Both WHILE and REPEAT loops similar to those available in some BASIC systems have also been included.

The multiway branch is an ON statement with a form similar to the case statement in Pascal except that it permits a default case, e.g.

```
100  ON I
110    CASE 1: LET A=1
120    B=2
130    CASE 2: READ A,B
140    I=0
150    DEFAULT: A=0
160  ENDON I
```

To compensate for the lack of a GOTO statement, an EXIT statement is included to enable one to exit from within a loop or a set of nested loops. This has the form

EXIT loop variable — exits from the loop with this loop variable
or EXIT — exits from the innermost loop without a loop variable

3.3 Procedures and Functions

Another obvious feature which aids good programming and which is lacking in most BASIC system, is a good procedure/function mechanism. This is essential for top-down programming and for increasing the readability of programs.
In BPL a procedure or function is introduced by a procedure declaration statement of form:

PROCEDURE<proc id>{(<formal par>
{,<formal par>}$_0^\infty$)}$_0^1$
or FUNCTION<function id>({<formal par>
{,<formal par>}$_0^\infty$ }$_0^1$)
{OF<type>}$_0^1$

This is followed by the statements comprising the procedure body and terminated by ENDPROC.
Unlike BASIC the statement numbers of statements within a procedure are completely independent of any statement numbers occuring outside that procedure. This facilitates the inclusion of procedures into a program, separate compilation of procedures onto a library, etc. On the other hand it does complicate system commands such as TRACE, BREAK, LIST, etc.
Procedures cannot be nested as in Pascal, but they may have their own local variables and they may call each other recursively. Parameters are called by reference as in BASIC — there is no call by value as there is in Pascal.

3.4 Declarations

The approach towards declarations adopted in BPL is similar to that taken in Pascal in that types, global variables and arrays are declared at the beginning of the program. This is followed by any procedures and functions and the main body of the program.
As will be noticed from the discussion on the advantages of BASIC and PASCAL, it is regarded by some to be an advantage to have automatically allocated variables, while others prefer that all variables should be declared before use. BPL attempts to cater for both points of view by providing three modes of operation:
(a) Declaration mode, in which every identifier must be declared before it is used,
(b) Automatic allocation with cross-reference, in which any identifier encountered which has not been declared, will be assumed to be either a procedure or a variable or function of type variable-length string (if the identifier is a string identifier) or real (if it is not). At the end of each segment (procedure, function or main body) a list of all automatically allocated variables is given. No arrays are allocated automatically.
(c) Automatic allocation without cross-reference, which is similar to (b) except that cross-reference tables are not produced.
In BPL two mechanisms are provided for array declaration, viz.
(a) An array may be declared using an array type, e.g.

```
10 TYPE ARRTYPE = ARRAY[1 . . 5]OF REAL
20 VAR X:ARRTYPE
```

This declares static arrays as in Pascal.
(b) The DIM statement may be used, e.g.

```
50 DIM A(5),B(N)
```

This may be used to declare dynamic arrays. Once an array has

been set up in a segment by a DIM statement, its size and bounds are fixed until execution of that segment is complete and control is returned to the calling segment. The DIM statement is an extension of the BASIC form of DIM.

3.5 Input-output

There are five input-output statements based on those available in BASIC. These are: READ, PRINT, DATA, FILE and RESET. In addition there are three functions: TAB, NL and EOF.
The statement READ can have various forms such as

10 READ A,B,C	— read 3 values from DATA statement into A,B,C
10 READ#1: A,B,C	— read 3 values from channel 1 into A,B,C
10 READ#1: USING BINARY: A,B,C	— read from channel 1 into A,B,C using binary format

The items in the destination list may be simple variables, subscripted variables, fields, records or arrays of any type except pointer or set.
Output is achieved by means of the PRINT statement which also has a format similar to that in BASIC, e.g.

```
10 PRINT "RESULT=";A+B,PAY
20 PRINT#1:MATRIX
30 PRINT#1:USING BINARY:EMPLREC(I)
```

If input/output is not associated with the console, a FILE statement must be used to assign a channel number to the particular file or device required. This has format

FILE#<numeric exp>:<string exp>{,<mode>}$_0^1$

where <string exp> reduces either to a device name (e.g. "CR","LP", etc.) or a file name.

3.6. Expressions

The way in which string manipulation is handled in BASIC is both simple and powerful. String variables and constants may be concatenated (using the operator "+"), substrings may be accessed or assigned to (e.g. A$[1,3]), strings or substrings compared lexicographically, etc. In BPL the same facilities are provided but for two types of string, viz.

(a) fixed-length strings of predetermined length, declared as follows:

```
10 VAR A$,B$:STRING(8)
```

(b) variable-length strings with specified maximum length, e.g.

```
20 VAR C$,D$:STRING(<=8)
```

If a string variable is not declared, it is assumed to be a variable-length string with maximum length equal to a default value.
Arithmetic expressions are similar to those in BASIC and include the operations +, −, ★, / and exponentiation which is denoted by ★★ to avoid confusion with the pointer operation ↑. The absence of an operator for exponentiation is one of the criticisms levelled against Pascal.
Boolean expressions are much the same as in Pascal or BASIC. The order of precedence of operators in BPL is as follows:

```
★★
★/
+ −
> >= <> = <<=
NOT
AND
OR
```

This is more natural than that used in Pascal.

3.7 Miscellaneous

The fact that each statement starts on a separate line simplifies syntax checking and error recovery. This coupled with the fact that each statement is checked for syntax errors as it is read into the system and each segment (procedure, function or main body) is checked for static semantic errors when the end of the segment is reached, assists the user by enabling him to correct errors as the program is entered instead of permitting the user to make a number of errors and then attempting to unravel them all.
Furthermore the dynamic debugging facilities available in BASIC viz. TRACE, BREAK, etc., have been included in BPL to make it easier for the user to debug his program.

4. Conclusion

The object in designing BPL was to produce a language which has, amongst others, the following properties:

(a) It is simple to learn and simple to use
(b) It is an interactive language with consequent benefits in the areas of syntax checking and debugging.
(c) It has neat control structures and lends itself to the production of readable, well-structured programs.
(d) It has a wide range of data types to facilitate programming a wide variety of applications.

At the end of the exercise it was discovered that a number of the facilities specifically excluded from Pascal because they might invite inefficient programming, had been included in BPL, whereas the dreaded GOTO (excluded from BPL) has been included in Pascal as a compromise to efficiency. Thus one might sum up by concluding that BPL places less stress on efficiency and more on assisting the programmer to produce a program which is readable and well-structured.

References

[1] BULL, G.M. (1972). Dynamic Debugging in BASIC. *The Computer Journal*, 15, 21–24.
[2] BULL, G.M., FREEMAN, W. and GARLAND, S.J. (1973). *Specification for Standard BASIC*, NCC Publications, Manchester.
[3] JENSEN, K. and WIRTH, N. (1976). *PASCAL User Manual and Report*, Springer-Verlag, Berlin.
[4] KEMENY, J.G. and KURTZ, T.E. (1971). *BASIC Programming*, John Wiley, New York.
[5] SCHWARTZ, A.M. (1978). Pascal versus BASIC: An Exercise. *BYTE*, 3, 8, 168–176.
[6] WILLIAMS, M.H. The Programming Language BPL, to be published in The Computer Journal.
[7] WIRTH, N. (1971). The Programming Language Pascal, *Acta Informatica*, 1, 35–63.

10. M.H.WILLIAMS, The programming language BPL, The Computer Journal, vol. 25, no. 3, 289-306 (1982).

The Programming Language BPL

M. H. Williams
Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ, UK

BPL is a programming language which was developed from the two languages BASIC and Pascal. The chief aim in designing BPL was to create a language as simple to use as BASIC with the additional data types and control structures which make Pascal attractive to use. It is intended in the first instance as a language to teach programming but it can be used for any application where BASIC might have been used, and in many such cases it is simpler to use and the result more readable than if BASIC had been used. For instance, the record facilities make the language more amenable to commercial applications, and the pointer facilities to manipulating complex linked data structures.

1. INTRODUCTION

With the abundance of programming languages already available, it is with great trepidation that one dares to announce yet another one. Nevertheless the language described here, BPL, has a number of merits and for this reason the reader is exhorted to read on and judge it for himself.

The language originates from a search for a suitable language to teach to two different types of undergraduate student—a first language for undergraduate Computer Science students and a language for other undergraduate science (or non-science) students. Two languages which suggested themselves for this purpose are:

(a) BASIC. This is simple to teach and simple to learn. The BASIC system is geared to helping the user. After a very limited period of time the student who has been taught BASIC has a better understanding of (or, if you prefer it, is less confused about) the process of computing than a student who has learnt a conventional batch language. Input/output is particularly simple and the notion of numeric and string variables is ideal for the beginner; file handling and matrix manipulation facilities enable more sophisticated applications to be handled very simply.

(b) Pascal. This is a well-structured language (i.e. one which lends itself to the construction of well-structured programs) and is thus a better language from the point of view of teaching good programming habits and style. In addition Pascal has a rich collection of data types which lend themselves to programming a wide range of applications. In particular, pointer, record and definable scalar types and more flexible array subscripts enable one to write more readable programs.

Both languages have very appealing features and either language could be used to teach both groups of students, although in general BASIC would probably be preferred for the non-Computer Science student while Pascal might be more attractive for teaching the Computer Science student. However, it would be nice to have a language which has the advantages of both BASIC and Pascal. It was for this reason that BPL was developed.

BPL (which stands for BASIC-Pascal-Liaison or Better Programming Language or simply a conversational programming language which was designed after APL) is designed as a two-tier language which combines the advantages of BASIC and Pascal. The lower level, a self-sufficient subset of BPL, is very similar to BASIC and provides a language suitable for teaching to service courses. It has all the advantages of BASIC together with those of the structured control structures of Pascal; in addition the facilities of the full language can be called upon if necessary. The full language has the advantage of the simplicity of BASIC combined with the flexibility of Pascal.

Since BASIC and Pascal are two rather different languages, BPL was designed to be as compatible as possible with BASIC but with the concepts of Pascal superimposed onto it. As both BASIC and Pascal have a very simple syntactic structure with unique keywords at the start of most statements and declarations, BPL has been constructed likewise. This simplifies matters both for the compiler-writer and for the user. Furthermore, BPL is designed to be compiled or interpreted as desired.

BPL is not intended to be just a teaching language. Any programmer familiar with BASIC should find BPL very similar to BASIC and in many cases easier to use because of the additional facilities. In particular the notions of records, pointers and definable scalar types can make programming simpler and programs more readable.

2. DEFINITION OF THE LANGUAGE

2.1 Statements and commands

Like BASIC, BPL is intended as a conversational programming language. The user interacts with the system and may give (a) commands, (b) indirect statements, and (c) direct statements.

A command is a directive to the system, concerned with the filing, retrieving, listing, tracing, running, etc., of programs.

An indirect statement is a program statement which is preceded by a statement number (an integer). When an indirect statement is encountered, it is stored away for execution at a later stage.

1. ⟨indirect statement⟩ : : = ⟨statement no⟩ [2]
⟨statement part⟩ [73]

CCC-0010-4620/82/0025-0289 $9.00

© Heyden & Son Ltd, 1982

2. ⟨statement no⟩::=⟨non-zero digit⟩ {⟨digit⟩}$_0^\infty$ [4,3]
3. ⟨digit⟩::=0|⟨non-zero digit⟩ [4]
4. ⟨non-zero digit⟩::=1|2|3|4|5|6|7|8|9

The extended form of BNF described by Lee and others is used here.[1] Any portion of a production enclosed in braces { }$_i^j$ indicates repetition of the contents at least i and not more than j times. Production numbers for nonterminals occurring on the right hand sides of productions are given in square brackets at the end of each line.

A direct statement is a program statement which is not preceded by a statement number. Such a program statement is executed when it is encountered.

5. ⟨direct statement⟩::=⟨stm⟩ [74]

In BASIC indirect statements may be entered in any order and the system puts them into ascending order of statement number; likewise in BPL the order of entering indirect statements is unimportant except for the restrictions discussed in the section on Procedures and Functions (Section 2.9).

A statement number is any unsigned integer in the range 1 to *maxstmno*, where *maxstmno* is an implementation dependent limit.

One important difference between BPL and BASIC as regards format is that in BASIC spacing in a program statement or command is generally unimportant whereas in BPL keywords, numbers and identifiers must be separated by at least one space or some nonalphanumeric symbol. Whereas in BASIC a program statement may not overflow over the end of a line, in BPL a continuation indicator (&) at the end of a line indicates that the statement continues on the next line. The number of continuation lines permitted will in general be system dependent but a minimum of 500 characters should be permitted.

2.2 Variables, types and identifiers

Each data item of a program may be stored in a 'cell' which may be either (a) a simple variable, (b) a field of a record or (c) an element of an array.

A simple variable is a simple cell capable of containing a single data item of a particular type; a record is a composite cell consisting of a collection of one or more simple cells not necessarily of the same type; an array is a collection of cells (simple or composite), all of the same type.

The type of a data item may be any of the following:

(a) Simple data types—REAL, INTEGER, BOOLEAN;
(b) Strings—these are divided into *fixed length strings* which are declared as STRING(n) where n is an integer denoting the number of characters in the string, and *variable length strings* which are declared as STRING (⟨=n) where n is an integer denoting the maximum number of characters in the string;

6. ⟨string type⟩::=STRING ({<=}$_0^1$⟨integer⟩) [14]

(c) Definable scalar type—this is a data type which can only assume values from some ordered set of identifiers specified in the type declaration;
(d) Subrange type—this is a special type of integer or scalar type which can only take on values in a limited range;
(e) Pointer type—this represents an address of a data item and is used mainly with dynamic storage allocation.

In addition there is a record type which is a collection of one or more named components or *fields*; the type of each field may be any of the above five types or may be a record type.

In what follows, one must distinguish between an identifier used as the name of a variable, record or array, and an identifier used as the name of a new data type. There are two types of identifiers used in the language—these are plain identifiers

7. ⟨plain identifier⟩::=⟨letter⟩ {⟨letter or digit⟩}$_0^\infty$ [10] [8]
8. ⟨letter or digit⟩::=⟨letter⟩|⟨digit⟩ [10, 3]

and string identifiers

9. ⟨string identifier⟩::=⟨letter⟩ {⟨letter or digit⟩}$_0^\infty$$ [10] [8]
10. ⟨letter⟩::=A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
11. ⟨identifier⟩::=⟨plain identifier⟩|⟨string identifier⟩ [7] [9]

Plain identifiers are used for all identifiers referring to types and to non-string variables and arrays; string identifiers refer to string variables and arrays. These identifier forms are compatible with names of variables and arrays in BASIC while at the same time permitting longer and more meaningful names too. As the Ent in *The Lord of the Rings*[2] so wisely commented, 'Real names tell you the story of the things they belong to in my language'.

One important point to note is that simple variables may either be declared or may be allocated automatically on encountering the first reference to the identifier (as in BASIC). In this language there are three modes of operation: (a) No declaration by default—in this mode all identifiers must be declared before they are used.

(b) Declaration by default with check—in this mode any identifier not declared will be allocated automatically as a variable of appropriate type on encountering the first reference to it. At the end of the segment, the system lists all automatically allocated identifiers and the lines where they are encountered.

(c) Declaration by default without check—same as (b) but without obtaining a listing of automatically allocated variables for each segment.

The type of an automatically allocated variable is determined by the type of the identifier—if it is a plain identifier, the variable is assumed to be real; if it is a string identifier, the variable is assumed to be a variable length string whose maximum length is some implementation dependent constant, *defaultstrlen* (this should be at least 64 characters).

This same default rule for determining the type of automatically allocated simple variables also applies to other situations. For example, arrays, fields and formal parameters must all be declared but if no type is specified the type is assumed to be either real or variable length string according to whether the identifier is plain or string.

© Heyden & Son Ltd, 1982

2.3 Constants

A numeric constant is a signed or unsigned integer or real number.

12. ⟨numeric constant⟩::={+|−}$_0^1$⟨number⟩ [13]
13. ⟨number⟩::=⟨integer⟩|⟨real number⟩ [14, 15]
14. ⟨integer⟩::={⟨digit⟩}$_1^\infty$ [3]
15. ⟨real number⟩::={⟨digit⟩}$_0^\infty$.{⟨digit⟩}$_1^\infty$ [3, 3]
{E⟨exponent⟩}$_0^1$|{⟨digit⟩}$_1^\infty$E⟨exponent⟩ [16, 3, 16]
16. ⟨exponent⟩::={+|−}$_0^1${⟨digit⟩}$_1^\infty$ [3]

A string constant is a string of characters enclosed in double quotation marks. If a double quotation mark itself forms part of the string, it is written twice.

17. ⟨string constant⟩::=
{"{⟨non-quote character⟩}$_0^\infty$"}$_1^\infty$

where ⟨non-quote character⟩ is any character other than a quotation mark.

A Boolean constant is one of the values TRUE or FALSE.

18. ⟨Bool constant⟩::=TRUE|FALSE

A scalar constant is a plain identifier.

19. ⟨scalar constant⟩::=⟨plain identifier⟩ [7]

The only pointer constant is NIL:

20. ⟨pointer constant⟩::=NIL

2.4 Type definitions

A data type represents a class of values which a data item can have. The class of values associated with the standard data types REAL, INTEGER and BOOLEAN are fairly obvious. On the other hand, definable scalar types, record types, etc., have no obvious class of values associated with them and these must be spelled out in type definitions. Thus a type definition defines an identifier to be associated with a class of values.

21. ⟨type definition⟩::=TYPE⟨type identifier⟩= [22]
⟨defined type⟩ [23]
22. ⟨type identifier⟩::=⟨plain identifier⟩ [7]
23. ⟨defined type⟩::=⟨scalar type⟩ [24]
|⟨subrange type⟩|⟨pointer type⟩| [25, 35]
⟨record type⟩|⟨array type⟩ [26, 36]

A scalar type defines an ordered set of scalar constant values.

24. ⟨scalar type⟩::=(⟨scalar constant⟩ [19]
{, ⟨scalar constant⟩}$_1^\infty$) [19]

For example

```
10 TYPE SEX = (MALE, FEMALE)
20 TYPE DAY = (MON, TUES, WED, THURS, FRI,
                                SAT, SUN)
```

A subrange type defines a subset of the set of integers or a subset of a set of scalars

25. ⟨subrange type⟩::=⟨integer⟩..⟨integer⟩| [14, 14]
⟨scalar constant⟩..⟨scalar constant⟩ [19, 19]

For example

```
10 TYPE DIGIT = 0..9
20 TYPE WEEKDAY = MON..FRI
```

A record type defines a composite type consisting of a number of fields, each with a type and an identifier associated with it.

A record definition may include one or more variant parts. A variant part consists of a set of different subrecords corresponding to different values of a particular field known as the tag field. The value of the tag field determines the subrecord assumed at any instant. A tag field must be of type INTEGER, a defined scalar type or subrange.

26. ⟨record type⟩::=RECORD⟨field def⟩ [27]
{;⟨field def⟩}$_0^\infty$END [27]
27. ⟨field def⟩::=⟨field name⟩{:⟨type⟩}$_0^1$| [33, 31]
⟨variant part⟩ [28]
28. ⟨variant part⟩::=
ON⟨tag field name⟩:⟨variant type⟩ [29, 30]
{CASE⟨constant⟩{, ⟨constant⟩}$_0^\infty$: [34, 34]
⟨field def⟩{;⟨field def⟩}$_0^\infty$}$_1^\infty$ [27, 27]
{DEFAULT:⟨field def⟩{;⟨field def⟩}$_0^\infty$}$_0^1$ [27, 27]
ENDON
29. ⟨tag field name⟩::=⟨field name⟩ [33]
30. ⟨variant type⟩::=INTEGER|⟨type identifier⟩ [22]
31. ⟨type⟩::={POINTER TO|↑}$_0^1$⟨simple type⟩ [32]
32. ⟨simple type⟩::=REAL|INTEGER|BOOLEAN|
⟨string type⟩|⟨type identifier⟩ [6, 22]
33. ⟨field name⟩::=⟨identifier⟩ [11]
34. ⟨constant⟩::=⟨numeric constant⟩| [12]
⟨Bool constant⟩| [18]
⟨scalar constant⟩|⟨pointer constant⟩| [19, 20]
⟨string constant⟩ [17]

ON, CASE, DEFAULT, ENDON, etc., may each appear at the start of a new continuation line. The space allocated for a record with variant parts will be determined by the maximum size of each variant part.

If a field is of type string, the field name must be a string identifier, otherwise it must be a plain identifier. If the ⟨type⟩ part is omitted from a field definition, the type of the field is assumed according to the previously mentioned rule: if the identifier is plain, the type of the field is assumed to be REAL, if it is string, the type is assumed to be variable length string with the default maximum length. For example,

```
10 TYPE SEXTYPE = (MALE, FEMALE)
20 TYPE MARTYPE = (SINGLE, MARRIED,
                   DIVORCED, SEPARATED)
30 TYPE EMPLREC = RECORD EMPLNO:
                              INTEGER;  &
     NAME$:STRING(< = 20);
                        ADDRESS$;WAGE;  &
     ON SEX:SEXTYPE                     &
     CASE MALE:ARMYNO$:STRING(15)       &
     CASE FEMALE:MAIDENNAME$:
                        STRING(< = 20)  &
     ENDON;                             &
     ON MARSTAT:MARTYPE                 &
     CASE SINGLE:NEXTOFKIN$:
                        STRING(< = 20)  &
     DEFAULT:NOOFCHILDREN:INTEGER       &
     ENDON                              &
END
```

A diagram of this record is shown in Fig. 1.

© Heyden & Son Ltd, 1982

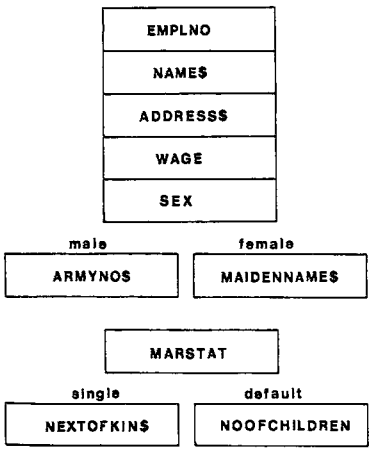


Figure 1. Diagram of record with variant parts.

A pointer type defines a pointer to a data item of a particular type.

35. ⟨pointer type⟩::=POINTER TO ⟨simple type⟩|↑⟨simple type⟩ [32, 32]

There is no semantic difference between these two forms. An example of pointer definitions is

```
100 TYPE INTPTR = POINTER TO INTEGER
110 TYPE REC = RECORD LLINK:POINTER TO
    & REC;INFO;RLINK:POINTER TO REC END
```

An array type defines an ordered collection of items of the same type.

36. ⟨array type⟩::=ARRAY[⟨bound⟩ {,⟨bound⟩}$_0^\infty$] OF⟨type⟩ [37] [37, 31]
37. ⟨bound⟩::={⟨integer⟩..}$_0^1$⟨integer⟩| ⟨type identifier⟩ [14, 14] [22]

If a bound is given as a single integer, this represents the upper bound, the lower bound being zero; if it is given as a pair of integers (subrange), the first represents the lower bound, the second the upper bound. If the bound is a type identifier, it may only be a scalar or subrange type. For example

```
20 TYPE ARR1 = ARRAY [DAY] OF INTEGER
30 TYPE ARR2 = ARRAY [1..10, 1..20]
                    OF STRING (< = 20)
```

All type identifiers and field names must be unique identifiers.

2.5 Variable and array declarations

Simple variables need not necessarily be declared if their types are REAL or variable length string (with maximum length equal to the default) as such variables will be allocated automatically on encountering a reference to the variable. However, if one does wish to declare such variables, or if a variable has type other than these two default types, variable declarations are used.

38. ⟨variable declaration⟩::=VAR⟨identifier⟩ {,⟨identifier⟩}$_0^\infty$:⟨type⟩ [11] [11, 31]

For example

```
200 VAR I,J,K:INTEGER
210 VAR REC1:EMPLREC
220 VAR A$,B$:STRING(< = 30)
```

Static arrays are declared using a TYPE and VAR statement as outlined; for example

```
10 TYPE ARRTYP = ARRAY[1..10]OF INTEGER
21 VAR SUMARR:ARRTYP
```

However, in BPL one may also use dynamic arrays, i.e. arrays whose bounds can be determined at run-time. In order to declare a dynamic array one uses a DIM statement which has the form

39. ⟨dim stm⟩::=DIM⟨array part⟩ {;⟨array part⟩}$_0^\infty$ [40] [40]
40. ⟨array part⟩::=⟨id list⟩ (⟨dbound⟩{,⟨dbound⟩}$_0^\infty$){OF⟨type⟩}$_0^1$ [41] [42, 42, 31]
41. ⟨id list⟩::=⟨array name⟩ {,⟨array name⟩}$_0^\infty$ [43, 43]
42. ⟨dbound⟩::={⟨numeric exp⟩..}$_0^1$⟨numeric exp⟩ [56, 56]
43. ⟨array name⟩::=⟨identifier⟩ [11]

Each array name must be a string identifier if the elements of the array are strings otherwise it must be a plain identifier. Once again if an 'OF⟨type⟩' clause is omitted, the types of the arrays are determined by default—REAL arrays if the identifiers are plain, variable length string arrays if the identifiers are string identifiers. Whether the 'OF⟨type⟩' clause is present or not, all array names in an identifier list must be of the same type. For example

```
10 DIM MATRIX,IJ(A,B) OF
                    INTEGER;NAME$(1..B)
```

declares three arrays, two two-dimensional integer arrays MATRIX and IJ whose row subscript runs from 0 to *A* and column subscript from 0 to *B*, and one one-dimensional variable length string array NAME$ whose subscript ranges from 1 to *B*.

A DIM statement may occur in any segment of a program. When it is executed, the appropriate arrays will be set up. On exit from the segment to the calling segment the arrays are lost so that on re-entry to the segment one must execute a DIM statement once again before the arrays will be set up. The values of the bounds of the array may be different on each entry to the segment; however, within a segment the bounds of an array will remain fixed from the time that the DIM statement is executed to the time that one exits from the segment. The scope and uniqueness of variable and array names is discussed in Section 2.9.

2.6 Reference to data items

A data item may be a simple variable in which case it may be referred to directly by using its identifier. If the data item is a pointer variable then using its identifier will yield the pointer value contained in the variable, while if the pointer variable identifier is followed by the symbol '↑', the value of the data item pointed to by this variable will be used, e.g.

```
110 VAR IPTR:POINTER TO INTEGER
```

reference to IPTR yields the pointer value,
reference to IPTR↑ yields the integer value pointed to.

© Heyden & Son Ltd, 1982

If the data item is a record variable (say X), then reference to X refers to the whole record; if one wishes to access a field Y of record X, one writes 'X.Y'. Likewise a data item may be an array element in which case it is referred to by a subscripted variable and if the array is of type pointer or record, this may in turn be followed by a sequence of ↑s or .⟨field name⟩s. For example,

```
TYPE AREC = RECORD FIELD1;FIELD2:
                   POINTER TO AREC END
DIM ARECARR (10) OF POINTER TO AREC
ARECARR(I)↑.FIELD2↑.FIELD1
```

In the case of a string data item, a substring may be accessed by specifying the first and last character positions of the substring in square parentheses after the string identifier, e.g.

STR$ [2,I + 1]

Thus a variable is defined by the following BNF definition:

44. ⟨variable⟩::=⟨array name⟩(⟨expression⟩ [43, 48]
{,⟨expression⟩}$_0^\infty$)⟨rest var⟩| [48, 45]
⟨identifier⟩⟨rest var⟩ [11, 45]
45. ⟨rest var⟩::={⟨field or pointer part⟩}$_0^\infty$ [46]
{⟨substring part⟩}$_0^1$ [47]
46. ⟨field or pointer part⟩::=.⟨field name⟩|↑ [33]
47. ⟨substring part⟩::=[⟨numeric exp⟩, [56]
⟨numeric exp⟩] [56]

2.7 Expressions

An expression consists of variables, constants and function calls separated by operators and parentheses. A BNF definition of an expression is as follows:

48. ⟨expression⟩::=⟨simple exp⟩|⟨Bool exp⟩ [55, 49]
49. ⟨Bool exp⟩::=⟨Bool term⟩ [50]
{OR⟨Bool term⟩}$_0^\infty$ [50]
50. ⟨Bool term⟩::=⟨Bool factor⟩{AND [51]
⟨Bool factor⟩}$_0^\infty$ [51]
51. ⟨Bool factor⟩::={NOT}$_0^1$⟨Bool primary⟩ [52]
52. ⟨Bool primary⟩::=⟨variable⟩| [44]
⟨Bool constant⟩|⟨fn designator⟩ [18, 62]
|(⟨Bool exp⟩)|⟨relation⟩ [49, 53]
53. ⟨relation⟩::=⟨simple exp⟩ [55]
⟨relational op⟩⟨simple exp⟩ [54, 55]
54. ⟨relational op⟩::=< =|<|> =|>|= |<>
55. ⟨simple exp⟩::=⟨numeric exp⟩ [56]
|⟨scalar exp⟩| [63]
⟨pointer exp⟩|⟨string exp⟩ [65, 64]
56. ⟨numeric exp⟩::={⟨adding op⟩}$_0^1$⟨term⟩ [61, 57]
{⟨adding op⟩⟨term⟩}$_0^\infty$ [61, 57]
57. ⟨term⟩::=⟨factor⟩{⟨multiplying op⟩ [58, 60]
⟨factor⟩}$_0^\infty$ [58]
58. ⟨factor⟩::=⟨primary⟩{**⟨primary⟩}$_0^\infty$ [59, 59]
59. ⟨primary⟩::=⟨numeric var⟩|⟨number⟩| [71, 13]
⟨fn designator⟩|(⟨numeric exp⟩) [62, 56]
60. ⟨multiplying op⟩::=*|/
61. ⟨adding op⟩::=+|−
62. ⟨fn designator⟩::=⟨fn name⟩ [72]
({⟨actual par⟩{,⟨actual par⟩}$_0^\infty$}$_0^1$) [67, 67]
63. ⟨scalar exp⟩::=⟨scalar var⟩ [68]
|⟨scalar constant⟩|⟨fn designator⟩ [19, 62]
64. ⟨string exp⟩::=⟨string primary⟩ [66]
{+⟨string primary⟩}$_0^\infty$ [66]
65. ⟨pointer exp⟩::=⟨pointer var⟩| [69]
⟨pointer constant⟩|CREATE(⟨type⟩)| [20, 31]
ADDRESSOF(⟨variable⟩)| [44]
⟨fn designator⟩ [62]
66. ⟨string primary⟩::=⟨string var⟩ [70]
|⟨string constant⟩|⟨fn designator⟩ [17, 62]
67. ⟨actual par⟩::=⟨expression⟩ [48]
68. ⟨scalar var⟩::=⟨variable⟩ [44]
69. ⟨pointer var⟩::=⟨variable⟩ [44]
70. ⟨string var⟩::=⟨variable⟩ [44]
71. ⟨numeric var⟩::=⟨variable⟩ [44]
72. ⟨fn name⟩::=⟨identifier⟩ [11]

A summary of the operators is given in Table 1.

The functions CREATE and ADDRESSOF are two pointer-valued functions whose effect is as follows:

(a) CREATE is equivalent to the 'new' function in Pascal. It creates an instance of the data type specified (using the concept of a 'heap' as in Pascal) and returns a pointer to this item;

Table 1. Summary of operators in BPL expressions

Operator	Operation	No. of operands	Type of operands	Type of result
NOT	Boolean NOT	1	Boolean	Boolean
AND	Boolean AND	2	Boolean	Boolean
OR	Boolean OR	2	Boolean	Boolean
= <>	comparison	2	numeric, scalar, string, pointer — both of same type	Boolean
< < = > > =	comparison	2	numeric, scalar, string — both of same type	Boolean
+	addition	2	numeric	integer unless either or both of operands is real
−	subtraction	2	numeric	
*	multiplication	2	numeric	
/	division	2	numeric	real
**	exponentiation	2	numeric	real
+	unary plus	1	numeric	real if operand is real
−	sign inversion	1	numeric	otherwise integer
+	concatenation	2	string	string

© Heyden & Son Ltd, 1982

(b) ADDRESSOF takes as argument the name of a simple variable, array element or record and returns the address of this item.

In each case the type of object pointed to must match the type expected (e.g. in an assignment statement, a relation or a procedure call) and if it does not match, an error must be flagged.

The standard functions available include all the standard BASIC functions (SQR, LOG, EXP, INT, LEN, MOD etc.) as well as SUCC(scalar exp)—yields the value succeeding the value of the scalar expression; PRED(scalar exp)—yields the value preceding the value of the scalar expression.

2.8 Statements

The statements available in the language are as follows:

73. ⟨statement part⟩::=⟨stm⟩|⟨proc or fn stm⟩| [74, 131]
⟨type definition⟩|⟨variable declaration⟩ [21, 38]
74. ⟨stm⟩::=⟨assign stm⟩|⟨input–output stm⟩| [75, 87]
⟨dim stm⟩|⟨control stm⟩|⟨rem stm⟩ [39, 104, 128]

Assignment statement. The assignment statement (⟨assign stm⟩) enables a numeric, Boolean, string, scalar or pointer expression to be assigned to the corresponding destination. It also permits the copying of a record variable to another variable of the same type or the performance of certain array manipulations and the assignment of the result to an array. It has the form

75. ⟨assign stm⟩::={LET}$_0^1$⟨numeric dest⟩ [76]
=⟨numeric exp⟩| [56]
{LET}$_0^1$⟨Bool dest⟩=⟨Bool exp⟩| [77, 49]
{LET}$_0^1$⟨string dest⟩=⟨string exp⟩| [78, 64]
{LET}$_0^1$⟨scalar dest⟩=⟨scalar exp⟩| [79, 63]
{LET}$_0^1$⟨pointer dest⟩=⟨pointer exp⟩| [80, 65]
{LET}$_0^1$⟨record dest⟩=⟨record var⟩| [81, 85]
{LET}$_0^1$⟨array name⟩ [43]
={⟨array exp⟩|⟨numeric exp⟩} [82, 56]
76. ⟨numeric dest⟩::=⟨numeric var⟩|⟨fn name⟩ [71, 72]
77. ⟨Bool dest⟩::=⟨Bool var⟩|⟨fn name⟩ [86, 72]
78. ⟨string dest⟩::=⟨string var⟩|⟨fn name⟩ [70, 72]
79. ⟨scalar dest⟩::=⟨scalar var⟩|⟨fn name⟩ [68, 72]
80. ⟨pointer dest⟩::=⟨pointer var⟩|⟨fn name⟩ [69, 72]
81. ⟨record dest⟩::=⟨record var⟩ [85]
82. ⟨array exp⟩::=⟨array term⟩ [83]
{+⟨array term⟩}$_0^\infty$ [83]
83. ⟨array term⟩::=⟨array factor⟩ [84]
{*⟨array factor⟩}$_0^\infty$ [84]
84. ⟨array factor⟩::=⟨array name⟩ [43]
|TRANSPOSE(⟨array name⟩) [43]
85. ⟨record var⟩::=⟨variable⟩ [44]
86. ⟨Bool var⟩::=⟨variable⟩ [44]

If the right hand side is a single array name, the contents of the array are copied to the array on the left hand side. If the right hand side is a numeric expression its value is copied to every element of the array. If addition or multiplication or TRANSPOSE is specified, array addition/multiplication/transpose is performed. If a real expression is assigned to an integer or subrange variable, the value is rounded to the nearest integer before assignment. Any value assigned to a subrange variable is also checked to ensure that it lies within the permissible limits for the subrange type.

Input–output statements. Input–output statements are as follows:

87. ⟨input–output stm⟩::=⟨data statement⟩ [88]
|⟨read statement⟩|⟨print statement⟩| [90, 94]
⟨file statement⟩ [101]
|⟨reset statement⟩ [103]

As in BASIC, items of data may be stored in a DATA statement which has the form

88. ⟨data statement⟩::=DATA⟨data constant⟩ [89]
{,⟨data constant⟩}$_0^\infty$ [89]
89. ⟨data constant⟩::=⟨numeric constant⟩| [12]
⟨string constant⟩|⟨scalar constant⟩| [17, 19]
⟨Bool constant⟩ [18]

There is only one input statement which has the form

90. ⟨read statement⟩::=READ
{#⟨input channel⟩:}$_0^1$ [91]
{USING BINARY:}$_0^1$⟨destination list⟩ [92]
91. ⟨input channel⟩::=DATA|CONS|⟨numeric exp⟩ [56]

If DATA is specified, input is taken from DATA statements (i.e. standard BASIC READ), if CONS, input is taken from the console (equivalent to BASIC INPUT) and if an expression with n as value, input is taken from file n. If the channel specification is omitted, #DATA: is assumed.

Data is read from the specified source and unpacked into the destinations specified in the destination list—these may be simple variables, subscripted variables, fields, arrays (in which case data items are read into each element of the array) or records (in which case data items are read into each field of the record) provided that all destinations have the same type.

92. ⟨destination list⟩::=⟨dest⟩{,⟨dest⟩}$_0^\infty$ [93, 93]
93. ⟨dest⟩::=⟨numeric var⟩|⟨Bool var⟩| [71, 86]
⟨string var⟩|⟨scalar var⟩| [70, 68]
⟨record var⟩|⟨array name⟩ [85, 43]

The USING option will be described under the PRINT statement below. Note that one may not read a value into a pointer variable nor into a record containing a field of type pointer nor into an array of pointers.

The only output statement has the form

94. ⟨print statement⟩::=PRINT
{#⟨output channel⟩:}$_0^1$ [98]
{USING BINARY:}$_0^1$⟨print list⟩ [95]
95. ⟨print list⟩::={{⟨print item⟩}$_0^1$ [96]
⟨print separator⟩}$_0^\infty${⟨print item⟩}$_0^1$ [97, 96]
96. ⟨print item⟩::=⟨print exp⟩⟨print format⟩ [100, 99]
|TAB(⟨numeric exp⟩) [56]
|NL(⟨numeric exp⟩)|⟨record var⟩ [56, 85]
|⟨array name⟩⟨print format⟩ [43, 99]
97. ⟨print separator⟩::=,|;
98. ⟨output channel⟩::=CONS|LP|⟨numeric exp⟩ [56]
99. ⟨print format⟩::={:⟨numeric exp⟩ [56]
{:⟨numeric exp⟩}$_0^1$}$_0^1$ [56]

© Heyden & Son Ltd, 1982

100. ⟨print exp⟩::=⟨numeric exp⟩|⟨scalar exp⟩ [56, 63]
|⟨string exp⟩|⟨Bool exp⟩ [64, 49]

If CONS is specified, output is sent to the user's console (i.e. standard BASIC PRINT), if LP, output is sent to the line printer, and if an expression with *n* as value, output is sent to file *n*. If the channel specification is omitted, #CONS: is assumed. When a PRINT statement is executed the contents of the print list are sent to the output device. As in BASIC the output medium is divided into zones of length *j* characters (*j* is implementation dependent—typically 15). Whenever a print separator is encountered in the print list, it is treated as follows:

if it is a semicolon, it has no effect
if it is a comma, skip to the next zone boundary

The function TAB(X) causes the print position to move forward to position Y where Y = X modulo the number of character positions per line of the output medium. If the print position is already beyond this position, the TAB function has no effect. The function NL(X) causes the output to move X lines vertically. If the print list ends on a separator, the line will not be printed and the next PRINT statement will continue output on the same line; otherwise the line will be printed and the output medium moved to a new line. If any item in a print list overflows over the end of a line, the line is printed and the remainder of the print list is continued on the following line.

For example

```
10 PRINT TAB(10);"RESULTS";NL(1);TAB(10);
                  "-------",NL(2);TAB(10);
20 PRINT "NO","NAME"
```

will cause

```
VVVVVVVVVRESULTS
VVVVVVVVV-------
VVVVVVVVVNOVVVNAME
```

to be output (where each occurrence of the symbol V is used to denote a space in the final output). If an expression or array name is used as a print item without a print format, the system decides on the best format to use to print the value(s)[3]. If an expression (or array name) is followed by '*:n*' where *n* is a numeric expression, then the value of the expression (values in the array) is printed so that it occupies *n* print positions (right justified). If it is too long to fit into *n* positions the full number is printed despite the number of positions required. If a numeric expression *e* is followed by '*:n:m*' where *n* and *m* are numeric expressions, then the value of *e* is printed so that it occupies *n* print positions and has *m* digits to the right of the decimal point.

Note that just as pointer values cannot be read, likewise they cannot be printed. Thus it is not permissible to print the contents of a pointer variable, or a record containing a field of type pointer, or an array of pointer values.

The clause USING BINARY may be used with files. If this clause is used in a PRINT statement, numeric items are transmitted in binary instead of being converted to decimal, scalar items are transmitted as binary numbers representing their position in the list of scalar constants, etc. If the clause is used in a READ statement the system will expect binary values.

The file statement establishes a relationship between a particular channel number and a file. It has the form

101. ⟨file statement⟩::=FILE#⟨numeric exp⟩: [56]
⟨string exp⟩{,⟨mode⟩}$_0^1$ [64, 102]
102. ⟨mode⟩::=READ|WRITE|READWRITE

When a file statement is executed the numeric expression is evaluated and rounded to the nearest integer to give the channel number. If a file is already assigned to that channel, this file is closed. The string expression is evaluated to give the new file name. This file is opened and assigned to that channel. The mode specifies the mode in which the file must be opened—it may be omitted if the file is to be opened for reading.

The reset statement restores the data pointer to the beginning of the first DATA block or to the beginning of a file (corresponds to rewinding a magnetic tape file). It has the form

103. ⟨reset statement⟩::=RESET{⟨numeric exp⟩}$_0^1$ [56]

Without the numeric expression this statement resets the DATA block pointer, with it it resets the file pointer for the appropriate file.

The Boolean function EOF(X) indicates whether or not the file on channel X is currently positioned at the end of the file.

Control statements. The control statements available include the following

104. ⟨control stm⟩::=⟨conditional statement⟩| [105]
⟨loop statement⟩|⟨multi-way branch⟩|
[110, 119]
⟨exit statement⟩|⟨ stop statement⟩ [126, 125]

A conditional statement may be part of an IF–THEN construct or part of an IF–THEN–ELSE construct. There are two forms of IF–THEN construct, a single line IF–THEN statement having the format

IF ⟨Bool exp⟩ THEN ⟨simple statement⟩ ENDIF

and a multi-line construct of form

IF ⟨Bool exp⟩ THEN {⟨simple statement⟩}$_0^1$
{⟨stm⟩}$_0^\infty$
ENDIF

where the IF part, the ENDIF part and each intervening statement besides the simple statement appears on a separate line with a separate statement number (the IF part having the lowest statement number of the sequence and the ENDIF part the highest). An IF–THEN–ELSE construct also has two forms, a single line ELSE part:

IF ⟨Bool exp⟩ THEN {⟨simple statement⟩}$_0^1$
{⟨stm⟩}$_0^\infty$
ELSE ⟨simple statement⟩ ENDIF

and a multi-line ELSE part:

IF ⟨Bool exp⟩ THEN {⟨simple statement⟩}$_0^1$
{⟨stm⟩}$_0^\infty$
ELSE {⟨simple statement⟩}$_0^1$
{⟨stm⟩}$_0^\infty$
ENDIF

© Heyden & Son Ltd, 1982

where the IF part, ELSE part, ENDIF part and each intervening statement besides the two simple statements appears on a separate line with a separate statement number.

105. ⟨conditional statement⟩::=⟨if stm⟩| [106]
⟨else stm⟩|⟨endif stm⟩ [107, 108]
106. ⟨if stm⟩::=IF⟨Bool exp⟩THEN [49]
{⟨simple statement⟩{ENDIF}$_0^1$}$_0^1$ [109]
107. ⟨else stm⟩::=ELSE{⟨simple statement⟩
{ENDIF}$_0^1$}$_0^1$ [110]
108. ⟨endif stm⟩::=ENDIF
109. ⟨simple statement⟩::=⟨assign stm⟩| [75]
⟨read statement⟩|⟨print statement⟩| [90, 94]
⟨file statement⟩|⟨reset statement⟩| [101, 103]
⟨return statement⟩|⟨call statement⟩| [137, 135]
⟨stop statement⟩|⟨exit statement⟩|⟨dim stm⟩|
[125, 126, 39]
⟨if stm⟩|⟨on statement⟩|⟨while statement⟩|
[106, 120, 111]
⟨repeat statement⟩|⟨for statement⟩ [113, 115]

If the Boolean expression (⟨Bool exp⟩) is true, the sequence of statements following the symbol THEN is executed; if it is false then either the statement sequence following the symbol ELSE is executed (if there is an ELSE part), or control is passed to the first statement after the ENDIF symbol.

There are three formats for loops: a while loop, a repeat loop and a for–next loop. The while loop has the form

WHILE ⟨Bool exp⟩ DO ⟨loop id⟩
{⟨stm⟩}$_0^\infty$
ENDWHILE ⟨loop id⟩

where the WHILE part, the ENDWHILE part and each intervening statement is placed on a separate line with its own statement number. The effect of this statement is to repeatedly execute the sequence of statements until the Boolean expression is found to be false. If its value is false initially, the statement sequence is not executed at all. The loop identifier may be any plain identifier or the null string and serves mainly to identify the two parts of the loop. If a plain identifier is used, an inner loop may not use as its loop identifier the same identifier as is used by an enclosing loop.

The repeat loop has the form

REPEAT ⟨loop id⟩
{⟨stm⟩}$_0^\infty$
UNTIL ⟨Bool exp⟩ FOR ⟨loop id⟩

where the REPEAT part, the UNTIL part and each intervening statement occurs on a separate line with its own statement number. In this form of loop the statement sequence forming the body of the loop is executed at least once. On reaching the UNTIL part, the Boolean expression is evaluated and if it is false, the body of the loop is executed once more. This pattern is repeated until on testing the Boolean expression it is found to be true. As for the while loop, the loop identifier may be null, in which case the keyword FOR may be omitted.

The for–next loop causes a set of statements to be executed a number of times while a sequence of values is assigned to a variable (known as the *control variable*). It has two forms. A numeric for–next loop has the format

FOR ⟨numeric var⟩=⟨numeric exp⟩
⟨to-downto⟩⟨numeric exp⟩{STEP⟨increment⟩}$_0^1$
{⟨stm⟩}$_0^\infty$
NEXT⟨numeric var⟩

where the FOR part, the NEXT part and the intervening statements all occur on separate lines with their own statement numbers. A loop of form

FOR v = exp$_1$ TO exp$_2$ STEP exp$_3$
S
NEXT v

will be interpreted as

LET v = exp$_1$
LET *tempfin* = exp$_2$
LET *tempstep* = exp$_3$
WHILE (v − *tempfin*)**tempstep* ⟨0 DO *LOOP1*
S
LET $v = v$ + *tempstep*
ENDWHILE *LOOP1*

Since the representation of decimal fractions in binary machines is necessarily approximate, the loop will operate within the limits of such arithmetic approximations in the case of a REAL controlled variable. If the STEP clause is omitted, an increment of 1 is assumed if TO is used, and −1 if DOWNTO is used.

A scalar for–next loop has the form

FOR⟨scalar var⟩=⟨scalar exp⟩⟨to-downto⟩⟨scalar exp⟩
{⟨stm⟩}$_0^\infty$
NEXT⟨scalar var⟩

The loop

FOR v = sexp$_1$ TO sexp$_2$
S
NEXT v

will be interpreted as

LET v = sexp$_1$
LET *temp* = sexp$_2$
WHILE v < = *temp* DO *LOOP1*
S
LET v = SUCC(v)
ENDWHILE *LOOP1*

Similarly

FOR v = sexp$_1$ DOWNTO sexp$_2$
S
NEXT v

will be interpreted as

LET v = sexp$_1$
LET *temp* = sexp$_2$
WHILE v > = *temp* DO *LOOP1*
S
LET v = PRED(v)
ENDWHILE *LOOP1*

Loops of all three types may be nested provided that the loop id (if there is one) or controlled variable used in an inner loop is different from any used in enclosing loops.

110. ⟨loop statement⟩::=⟨while statement⟩| [111]
⟨endwhile statement⟩|⟨repeat statement⟩|
[112, 113]

© Heyden & Son Ltd, 1982

⟨until statement⟩|⟨for statement⟩|
⟨next statement⟩ [114, 115, 116]

111. ⟨while statement⟩::=WHILE⟨Bool exp⟩ [49]
DO{⟨loop id⟩}$_0^1$ [117]

112. ⟨endwhile statement⟩::=ENDWHILE
{⟨loop id⟩}$_0^1$ [117]

113. ⟨repeat statement⟩::=REPEAT{⟨loop id⟩}$_0^1$ [117]

114. ⟨until statement⟩::=UNTIL⟨Bool exp⟩ [49]
{FOR⟨loop id⟩}$_0^1$ [117]

115. ⟨for statement⟩::=
FOR⟨numeric var⟩=⟨numeric exp⟩
⟨to-downto⟩ [56, 118]
⟨numeric exp⟩{STEP⟨numeric exp⟩}$_0^1$| [56, 56]
FOR⟨scalar var⟩=⟨scalar exp⟩ [63]
⟨to-downto⟩⟨scalar exp⟩ [108, 63]

116. ⟨next statement⟩::=NEXT⟨numeric var⟩| [71]
NEXT⟨scalar var⟩ [68]

117. ⟨loop id⟩::=⟨plain identifier⟩ [7]

118. ⟨to-downto⟩::= TO|DOWNTO

The multi-way branch has the form

$$\begin{array}{l} \text{ON}\langle\text{simple var}\rangle \\ \left\{\begin{array}{l} \text{CASE}\langle\text{expression}\rangle \\ \quad \{..\langle\text{expression}\rangle\}_0^1\{,\langle\text{expression}\rangle \\ \quad \{..\langle\text{expression}\rangle\}_0^1\}_0^\infty:\langle\text{simple statement}\rangle \\ \quad \{\langle\text{stm}\rangle\}_0^\infty \end{array}\right\}_1^\infty \\ \left\{\begin{array}{l} \text{DEFAULT}:\langle\text{simple statement}\rangle \\ \quad \{\langle\text{stm}\rangle\}_0^\infty \end{array}\right\}_0^1 \\ \text{ENDON}\langle\text{simple var}\rangle \end{array}$$

where the ON Part, each CASE part, the DEFAULT part (if one is present), the ENDON part and each intervening statement besides the simple statements shown occur on separate lines, each with its own statement number. The simple var may be of type REAL, INTEGER, string, subrange or scalar. If it is a numeric variable (REAL, INTEGER or subrange), the expressions must all be numeric, if it is a string variable, the expressions must all be strings and if it is a scalar variable, the expressions must be scalar constants.

The effect of the multi-way branch is to evaluate each case expression (⟨expression⟩) and each case range (⟨expression⟩..⟨expression⟩) in turn until it finds a case expression whose value matches the value of the ⟨simple var⟩, or a case range such that the value of the ⟨simple var⟩ lies within the range denoted by the two expressions. If such a case is found, control is passed to the sequence of statements immediately following the case expressions; if the value of the variable does not match any of the cases, control is passed to the default section if one is present or otherwise to the statement after the ENDON. If control is passed to one of the case alternatives, execution continues until the next CASE, DEFAULT or ENDON is encountered, at which point control is transferred to the statement after the ENDON.

As with the loops the simple var in the ENDON statement must match that in the ON statement.

119. ⟨multi-way branch⟩::=⟨on statement⟩| [120]
⟨case statement⟩|⟨default statement⟩|[121, 122]
⟨endon statement⟩ [123]

120. ⟨on statement⟩::=ON⟨simple var⟩ [124]

121. ⟨case statement⟩::=CASE⟨expression⟩ [48]
{..⟨expression⟩}$_0^1${,⟨expression⟩ [48, 48]
{..⟨expression⟩}$_0^1$}$_0^\infty$:⟨simple statement⟩
[48, 109]

122. ⟨default statement⟩::=DEFAULT:
⟨simple statement⟩ [109]

123. ⟨endon statement⟩::=ENDON⟨simple var⟩
[124]

124. ⟨simple var⟩::=⟨identifier⟩ [11]

The STOP statement causes execution of a program to cease.

125. ⟨stop statement⟩::=STOP

There are two forms of exit statement, viz.

126. ⟨exit statement⟩::=EXITL{⟨loop id⟩}$_0^1$| [117]
EXITP{⟨proc or fn name⟩}$_0^1$ [127]

127. ⟨proc or fn name⟩::=⟨proc name⟩|⟨fn name⟩
[132, 72]

The first form (EXITL) causes an exit from the statically enclosing loop specified to the statement immediately following that loop. If the loop id is omitted, the innermost enclosing loop with a null loop id is exited. The second form (EXITP) is used to return to the most recent invocation of the procedure or function specified, unstacking the details of any procedures or functions encountered in the interim. This statement assumes that the procedure or function concerned has been entered but not yet returned from at the time—if not an error will be flagged on execution of the statement. If no procedure or function name is specified the system will return to the main program.

REM statement. As in BASIC the REM statement is defined as

128. ⟨rem stm⟩::=REM{⟨character⟩}$_0^\infty$ [129]

129. ⟨character⟩::=⟨non-quote character⟩|″

It is used to insert comments into a program.

2.9 Procedures and functions

In BPL one may define procedures and functions. Statements related to procedures and functions are:

130. ⟨proc or fn stm⟩::=⟨call statement⟩| [135]
⟨return statement⟩|⟨procedure statement⟩|
[137, 131]
⟨function statement⟩|⟨endproc statement⟩|
[138, 134]
⟨endfn statement⟩ [139]

A procedure definition has the form

⟨procedure statement⟩
{⟨variable declaration⟩}$_0^\infty$
{⟨stm⟩}$_0^\infty$
⟨endproc statement⟩

where

131. ⟨procedure statement⟩::=PROCEDURE
⟨proc name⟩ [132]
{(⟨formal par⟩{:⟨type⟩}$_0^1$ [133, 31]
{,⟨formal par⟩{:⟨type⟩}$_0^1$}$_0^\infty$)}$_0^1$ [133, 31]

132. ⟨proc name⟩::=⟨plain identifier⟩ [7]

© Heyden & Son Ltd, 1982

133. ⟨formal par⟩::=⟨identifier⟩ [11]
134. ⟨endproc statement⟩::=ENDPROC

and the procedure heading, the intervening variable declarations and statements, and the endproc statement all occur on separate lines with their own statement numbers.

A procedure is called by a statement of form

135. ⟨call statement⟩::=⟨proc name⟩ [132]
{(⟨actual par⟩{,⟨actual par⟩}$_0^\infty$)}$_0^1$ [136, 136]
136. ⟨actual par⟩::=⟨expression⟩ [48]

A formal parameter may be a plain or string identifier, depending on its type. If the type part is omitted, the parameter is assumed to be variable length string if it has a string identifier, or real if it has a plain identifier. When the procedure is called, the actual parameters must correspond in number, type and order to the formal parameters in the procedure definition. If an actual parameter is a single variable or an array name, the address of this variable is passed across to the formal parameter; otherwise the actual parameter is taken as an expression, is evaluated and a pointer to this value associated with the formal parameter. After assigning actual parameters to formal parameters, the statements in the body of the procedure are executed until a return statement of form

137. ⟨return statement⟩::=RETURN

or the end of the procedure (ENDPROC) is reached. In either case control is returned to the statement after the call in the calling segment (A segment being taken to be a procedure, a function or the main body of the program).

A function definition is very similar and has the form

⟨function statement⟩
{⟨variable declaration⟩}$_0^\infty$
{⟨stm⟩}$_0^\infty$
⟨endfn statement⟩

where

138. ⟨function statement⟩::=FUNCTION
⟨fn name⟩({⟨formal par⟩{:⟨type⟩}$_0^1$ [72, 133, 31]
{,⟨formal par⟩{:⟨type⟩}$_0^1$}$_0^\infty$}$_0^1$){OF ⟨type⟩}$_0^1$ [133, 31, 31]
139. ⟨endfn statement⟩::=ENDFN

Within the body of the function definition, the function identifier may be used (without parentheses following it) as a destination, but any other access to the function identifier implies a recursive call of the function and must be so written. Within the function body there must be at least one statement which assigns a value to the function identifier, and at least one such statement must be executed at every entry of the function. The final value assigned to the function identifier is the value returned by the function. A function name may be a plain or string identifier depending on its type. As with formal parameters, if the 'OF⟨type⟩' clause is omitted, the function is taken to be real or variable length string depending on the function name. A function call has the form

⟨fn name⟩({⟨actual par⟩{,⟨actual par⟩}$_0^\infty$}$_0^1$)

(see production 62) and may be used in an expression in the usual way. As with a procedure, control is returned to the calling segment on encountering a RETURN statement or the ENDFN statement.

Variables and arrays may be classified into two types: local or global. Any variable or array which is declared in a variable declaration or DIM statement in a procedure or function is local to that procedure or function and is accessible only within that procedure or function segment. The name of such a variable or array must be different from that of any other variable or array within that segment. Any other variables, i.e. those declared in variable declarations in the main body of the program and those allocated automatically anywhere in a program are taken to be global variables and are accessible throughout the main body and in any procedure or function in which the same identifier is not used as a local variable. Once again the names of such variables or arrays must be different from those of any other global variables or arrays or type identifiers.

Procedures and functions may be called recursively. Each procedure and function has its own set of statement numbers which are independent of statement numbers occurring in any other procedure or function or in the main body of the program. In order to handle input and editing of procedures, two modes of operation are used: (a) normal mode—in which all indirect statements are stored as part of the main program; (b) procedure mode—in which all indirect statements are stored as part of the current procedure or function.

Initially the system is in normal mode. If a PROCEDURE or FUNCTION statement is encountered then, provided that the system is not already in procedure mode (for if it is, an error will be flagged), the mode is switched to procedure mode. In procedure mode indirect statements of the procedure body are entered. These may be entered in any order provided that the PROCEDURE or FUNCTION statement has the lowest line number and the ENDPROC or ENDFN statement has the highest. One remains in procedure mode until the ENDPROC or ENDFN statement is encountered whereafter the mode reverts to normal mode.

If one wishes to return from normal mode to a procedure which has already been entered in order to alter it, this is done by means of the command:

140. ⟨editfn command⟩::=EDITFN⟨proc or fn name⟩ [127]

This causes the system to be placed in procedure mode and the appropriate procedure is once again accessible. One may extract oneself from procedure mode once again by giving the appropriate ENDPROC or ENDFN statement, this time either as a direct statement or as an indirect one.

Note that DATA statements may not occur in a procedure or function body—they may only occur in the main body.

2.10 Commands

The standard commands include

141. ⟨command⟩::=⟨editfn command⟩| [140]
⟨new command⟩|⟨old command⟩| [142, 144]
⟨save command⟩|⟨replace command⟩| [146, 147]
⟨run command⟩| [148]
⟨bye command⟩|⟨resequence command⟩| [149, 150]

© Heyden & Son Ltd, 1982

⟨list command⟩| [155]
⟨trace command⟩|⟨break command⟩| [156, 158]
⟨vtrace command⟩| [166]
⟨untrace command⟩|⟨unbreak command⟩| [157, 159]
⟨unvtrace command⟩| [167]
⟨compile command⟩|⟨continue command⟩| [170, 163]
⟨mode command⟩| [171]
⟨last command⟩|⟨unlast command⟩ [164, 165]

142. ⟨new command⟩::=NEW⟨prog name⟩ [143]
143. ⟨prog name⟩::=⟨plain identifier⟩ [7]

NEW causes the current program to be erased from main memory in preparation for input of a new program.

144. ⟨old command⟩::=OLD⟨prog name⟩ {,⟨file name⟩}$_0^1$ [143] [145]
145. ⟨file name⟩::=⟨plain identifier⟩ [7]

OLD erases the current program from main memory and fetches the program specified by ⟨prog name⟩ from the file ⟨file name⟩ (or if no file name is given then the file currently open is used).

146. ⟨save command⟩::=SAVE {⟨file name⟩}$_0^1$ [145]

SAVE stores the current program in a subfile of a file. The name of the subfile into which the program is stored is that given in the most recent NEW or OLD command. The file used is either the one specified in the SAVE command, or if none is specified, the file currently open.

147. ⟨replace command⟩::=REPLACE {⟨file name⟩}$_0^1$ [145]

REPLACE causes the current program to replace the contents of an existing subfile. The name of the subfile is that given in the latest NEW or OLD command; the file used is the one specified (or the one currently open).

148. ⟨run command⟩::=RUN

RUN causes the current program to be interpreted.

149. ⟨bye command⟩::=BYE

BYE causes an exit from the BPL system.

150. ⟨resequence command⟩::= RESEQUENCE{⟨proc fn or all⟩}$_0^1$ [151] {⟨old first stm no⟩{,⟨new first stm no⟩[152, 153] {,⟨stm step⟩}$_0^1$}$_0^1$}$_0^1$ [154]
151. ⟨proc fn or all⟩::=⟨proc or fn name⟩|ALL [127]
152. ⟨old first stm no⟩::=⟨statement no⟩ [2]
153. ⟨new first stm no⟩::=⟨statement no⟩ [2]
154. ⟨stm step⟩::=⟨integer⟩ [14]

RESEQUENCE changes the value of some or all of the statement numbers of a program segment. If a procedure or function name is specified the statement numbers of that segment are resequenced; if ALL is specified all segments are resequenced; if neither a procedure or function name nor ALL is specified the statement numbers of the main program are resequenced. Resequencing starts at the old first statement number or if no such statement number exists in the specified segment, the next highest statement number is taken as the starting point. If omitted, the old first statement number is taken as one.

The new first statement number specifies the new value of the statement number of the first line to be resequenced (if omitted this defaults to 10). The statement step specifies the increment in value between successive new statement numbers (if omitted this defaults to 10). Resequencing continues to the end of the segment.

155. ⟨list command⟩::=LIST⟨arg⟩ [160]
156. ⟨trace command⟩::=TRACE⟨arg⟩ [160]
157. ⟨untrace command⟩::=UNTRACE⟨arg⟩ [160]
158. ⟨break command⟩::=BREAK⟨arg⟩ [160]
159. ⟨unbreak command⟩::=UNBREAK⟨arg⟩ [160]
160. ⟨arg⟩::={⟨proc fn or all⟩}$_0^1$ [151] {⟨first line no⟩{−⟨last line no⟩}$_0^1$}$_0^1$ [161, 162]
161. ⟨first line no⟩::=⟨statement no⟩ [2]
162. ⟨last line no⟩::=⟨statement no⟩ [2]

For each of the above commands, if a procedure or function name is specified, the command applies only to that segment; if ALL is specified, it applies to all segments; if neither, it applies only to the main program body. If a single statement number is specified, the command applies only to the program statement with this statement number in the appropriate segment(s); if a pair of statement numbers is specified, all statements with statement numbers in the inclusive range of this pair in the appropriate segment(s) are concerned; if no statement number is specified, the command applies to all statements of the appropriate segment(s).

LIST causes the specified statements to be listed, TRACE sets the trace bits on the appropriate statements to enable tracing,[4] UNTRACE unsets the trace bits. BREAK sets the break bits on the appropriate statements causing execution to stop whenever it reaches such a statement,[4] UNBREAK unsets the break bits. If the program has stopped at a break point, execution may be continued by typing the command CONTINUE, defined as

163. ⟨continue command⟩::=CONTINUE

Another debugging command which is used in conjunction with TRACE is

164. ⟨last command⟩::=LAST⟨integer⟩ [14]

Under normal operation the TRACE facility will give a continuous trace of the execution of the program. However, when the command LAST *n* is given, the trace lines are not printed out but stored on a circular list with capacity for *n* entries. On encountering an error or the end of the program, the most recent *n* entries of this TRACE are printed out. Similarly

165. ⟨unlast command⟩::=UNLAST

is used to switch off this facility.

166. ⟨vtrace command⟩::=VTRACE⟨varlist⟩ [168]
167. ⟨unvtrace command⟩::=UNVTRACE {⟨varlist⟩}$_0^1$ [168]
168. ⟨varlist⟩::=⟨trace var⟩{,⟨trace var⟩}$_0^\infty$ [169, 169]
169. ⟨trace var⟩::=⟨simple var⟩|⟨array name⟩ [124, 43]

VTRACE sets the trace bits on the specified variables and arrays. UNVTRACE unsets the trace bits; if no variable list is specified, it unsets the trace bits on all variables and arrays. If on execution a value is assigned to a variable or an element of an array (in an assignment statement, READ, FOR or NEXT) for which the trace

© Heyden & Son Ltd, 1982

bit is set, the statement number, the name of the variable and its new value are printed out.

170. ⟨compile command⟩::=COMPILE ⟨prog name⟩{,⟨file name⟩}$_0^1$ [143, 145]

COMPILE causes the current program to be compiled and stored in the subfile specified under ⟨prog name⟩ in the specified file (or if no file is specified, the current file).

The form of the MODE command is given by:

171. ⟨mode command⟩::=MODE⟨declaration mode⟩ [172]

172. ⟨declaration mode⟩::=DEF|LIST|NOLIST

This command sets the mode (as described in Section 2.2) according to whether all variables must be declared, or variable declarations may be omitted and the variables automatically allocated and listed at the end of each segment, or variables are allocated automatically without such a listing.

3. EXAMPLES

Three examples of simple BPL programs are given in Figs 2–4. Figure 2 contains a simple program for playing a simulated game of craps. This is based on the example of Dwyer.[5] The function RND is used to generate random numbers—RND(−1) initializes the random number generator and generates a starting value (which should be different each time) and RND(0) returns the next random number (in the range 0 to 1, excluding 1). Figure 3 illustrates a simple file application—updating a master file with amendments contained in an amendment file, while Fig. 4 shows how a monkey puzzle sort may be implemented.

```
1    REM ************************************************
2    REM **  SIMULATED CRAPS GAME AS DESCRIBED IN        **
3    REM **  CREATIVE COMPUTING,VOL.3,NO.6,PP.83-84(1977)**
4    REM ************************************************
10   TYPE WINSTATUS=(WIN,LOSE,UNCERTAIN)
20   TYPE ROLLTYPE=2..12
30   VAR RESULT:WINSTATUS
40   VAR ROLL1,ROLL2:ROLLTYPE

10   FUNCTION ROLL( ) OF ROLLTYPE
20   LET ROLL=INT(6*RND(0)+1)+INT(6*RND(0)+1)
30   ENDFN

100  LET X=RND(-1)
110  PRINT"SIMULATED CRAPS GAME - YOU START WITH $10"
120  LET YOURMONEY=10
130  REPEAT
140     PRINT"HOW MUCH DO YOU WANT TO BET?"
150     READ#CONS:BET
160     LET ROLL1=ROLL( )
170     PRINT"ROLL IS";ROLL1
180     ON ROLL1
190        CASE 4,5,6,8,9,10:PRINT "YOUR POINT IS";ROLL1
200           LET RESULT=UNCERTAIN
210           WHILE RESULT=UNCERTAIN DO
220              LET ROLL2=ROLL( )
230              PRINT"NEXT ROLL IS";ROLL2
240              IF ROLL1=ROLL2 THEN LET RESULT=WIN
250              ELSE IF ROLL2=7 THEN LET RESULT=LOSE ENDIF
260              ENDIF
270           ENDWHILE
280        CASE 2,3,12:LET RESULT=LOSE
290        CASE 7,11: LET RESULT=WIN
300     ENDON ROLL1
310     IF RESULT=WIN THEN LET YOURMONEY=YOURMONEY+BET
320        PRINT"YOU WIN! YOU NOW HAVE $";YOURMONEY
330     ELSE LET YOURMONEY=YOURMONEY-BET
340        PRINT "TOUGH..YOU LOSE. YOU NOW HAVE $";YOURMONEY
350     ENDIF
360     PRINT"WANT TO PLAY AGAIN?"
370     READ#CONS:ANSWER$
380  UNTIL ANSWER$ <> "YES"
390  PRINT"YOU ENDED UP WITH $";YOURMONEY
400  END
```

Figure 2. BPL program for simulating the game of craps.

```
1    REM ************************************************
2    REM ** PROGRAM TO READ IN FILE OF AMENDMENTS TO **
3    REM ** WAGES OR OVERTIME HOURS AND UPDATE A     **
4    REM ** MASTER FILE CONTAINING EMPLOYEE RECORDS  **
5    REM ************************************************
10   TYPE AMENDT=(WAGE,OVERTIME)
20   TYPE EMPREC=RECORD EMPNO:INTEGER;NAME$:STRING(20);   &
                        OTIME;BASICPAY END
30   TYPE AMENDREC=RECORD NO:INTEGER;                     &
                  ON AMTYPE:AMENDT                       &
                  CASE WAGE:NEWPAY                       &
                  CASE OVERTIME:NEWOT                    &
                  ENDON                                  &
                  END
40   VAR EMP:EMPREC
50   VAR AMEND:AMENDREC
60   VAR NOERRYET:BOOLEAN
70   VAR AMENDFILE,MASTER,NEWMASTER:INTEGER
120  NOERRYET=TRUE
130  FILE#1:"AMENDFILE"
140  FILE#2:"MASTERFILE"
150  FILE#3:"NEWMASTER",WRITE
160  AMENDFILE=1
170  MASTER=2
180  NEWMASTER=3
190  READ#MASTER:USING BINARY:EMP
200  WHILE NOT EOF(AMENDFILE) AND NOERRYET DO LOOP1
210     READ#AMENDFILE:AMEND
220     WHILE EMP.EMPNO<>AMEND.NO AND NOT EOF(MASTER)DO
230        PRINT#NEWMASTER:USING BINARY:EMP
240        READ#MASTER:USING BINARY:EMP
250     ENDWHILE
260     IF EOF(MASTER) THEN
270        PRINT"INVALID EMPLOYEE NO";AMEND.NO
280        NOERRYET=FALSE
290     ELSE ON AMEND.AMTYPE
300           CASE WAGE: EMP.BASICPAY=AMEND.NEWPAY
310           CASE OVERTIME: EMP.OTIME=AMEND.NEWOT
320        ENDON
330     ENDIF
340  ENDWHILE LOOP1
350  REM IF STILL MASTER LEFT,COPY TO END OF FILE
360  WHILE NOT EOF(MASTER)DO
370     PRINT#NEWMASTER:USING BINARY:EMP
380     READ#MASTER:USING BINARY:EMP
390  ENDWHILE
400  REM WRITE LAST RECORD
410  PRINT#NEWMASTER:USING BINARY:EMP
420  END
```

Figure 3. Program for updating a master file with amendments contained in an amendment file.

```
10   REM ******************************************
20   REM *  THIS PROGRAM READS IN N NUMBERS AND    *
30   REM *  SORTS THEM INTO ASCENDING ORDER USING A *
40   REM *  MONKEY PUZZLE SORT                      *
50   REM ******************************************
60
70   TYPE TREEREC=RECORD LLINK:POINTER TO TREEREC;        &
          INFO;  RLINK:POINTER TO TREEREC END
80   TYPE TREEPTR=POINTER TO TREEREC
90   VAR P,ROOT:TREEPTR

10   PROCEDURE TRAVERSE(Q:  TREEPTR)
20   REM THIS TRAVERSES A BINARY TREE IN POSTORDER
30   IF Q↑.LLINK<> NIL THEN TRAVERSE(Q↑.LLINK) ENDIF
40   PRINT Q↑.INFO
50   IF Q↑.RLINK <> NIL THEN TRAVERSE(Q↑.RLINK) ENDIF
60   ENDPROC

10   PROCEDURE INSERT(Q:TREEPTR,X)
20   REM THIS INSERTS AN ITEM X INTO A BINARY TREE
30   IF Q = NIL THEN P:=CREATE(TREEREC)
40   P↑.LLINK = NIL
50   P↑.RLINK = NIL
60   P↑.INFO = X
70   Q = P
80   ELSE IF X<Q↑.INFO THEN INSERT(Q↑.LLINK,X)
90   ELSE INSERT(Q↑.RLINK,X) ENDIF
100  ENDIF
110  ENDPROC

100  ROOT = NIL
110  READ N
120  FOR I = 1 TO N
130  READ NUMBER
140  INSERT(ROOT,NUMBER)
150  NEXT I
160  TRAVERSE(ROOT)
170  END
```

Figure 4. BPL program for performing a monkey puzzle sort.

An interpreter for BPL is currently being implemented in Pascal for an ICL 1900 computer. Work is also being done on a compiler for the language (also in Pascal).

Since the initial submission of this paper several other languages have appeared which have similar characteristics, e.g. COMAL. These languages have varying degrees of complexity, and slightly differing objectives. For example, the chief objective in the design of BPL is to provide a language which can be taught at two levels—at the lower level, a subset of BPL is very similar to a structured form of BASIC, at the higher level, the

© Heyden & Son Ltd, 1982

complete language is an interactive language which is probably more powerful than Pascal.

In view of this common interest it might be useful to establish a working group to study developments in this area.

Acknowledgements

The author wishes to thank the referee for his helpful suggestions, and Marianne Collett, Peter Clayton, Gavin Hall and Grant Pote for their respective parts in implementing the language. The work was done at Rhodes University, Grahamstown, South Africa.

REFERENCES

1. J. A. N. Lee, The formal definition of the BASIC language. *The Computer Journal* **15**, 37–41 (1972).
2. J. R. R. Tolkien, *The Lord of the Rings, Part II: The Two Towers.* Methuen, Toronto (1971).
3. G. M. Bull, W. Freeman and S. J. Garland, *Specification for Standard BASIC.* NCC Publications, Manchester (1973).
4. G. M. Bull, Dynamic debugging in BASIC. *The Computer Journal* **15** (No. 1), 21–24 (1972).
5. T. A. Dwyer, The 8-hour wonder. *Creative Computing* **3** (No. 6), 78–85 (1977).

Received November 1978

© Heyden & Son Ltd, 1982

APPENDIX 1

List of reserved words for BPL statements

AND	ELSE	FILE	OF	REPEAT	TRANSPOSE
ARRAY	END	FOR	ON	REM	TRUE
BINARY	ENDFN	FUNCTION	OR	RESET	TYPE
BOOLEAN	ENDIF	IF	POINTER	RETURN	UNTIL
CASE	ENDON	INTEGER	PRED	STEP	USING
CONS	ENDPROC	LET	PRINT	STOP	VAR
DATA	ENDWHILE	LP	PROCEDURE	STRING	WHILE
DEFAULT	EOF	NEXT	REAL	SUCC	WRITE
DIM	EXITL	NIL	READ	TAB	
DO	EXITP	NL	READWRITE	THEN	
DOWNTO	FALSE	NOT	RECORD	TO	

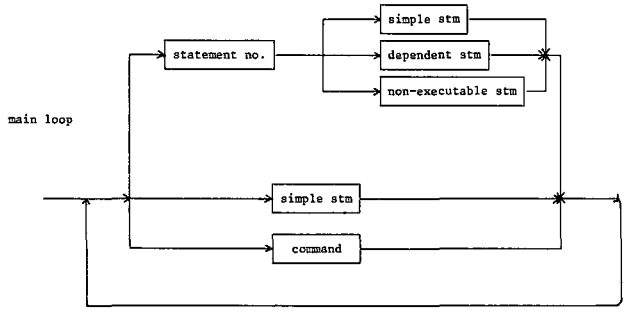
List of reserved words for commands

ALL	CONTINUE	LIST	OLD	SAVE	UNTRACE
BREAK	DEF	MODE	REPLACE	TRACE	UNVTRACE
BYE	EDITFN	NEW	RESEQUENCE	UNBREAK	VTRACE
COMPILE	LAST	NOLIST	RUN	UNLAST	

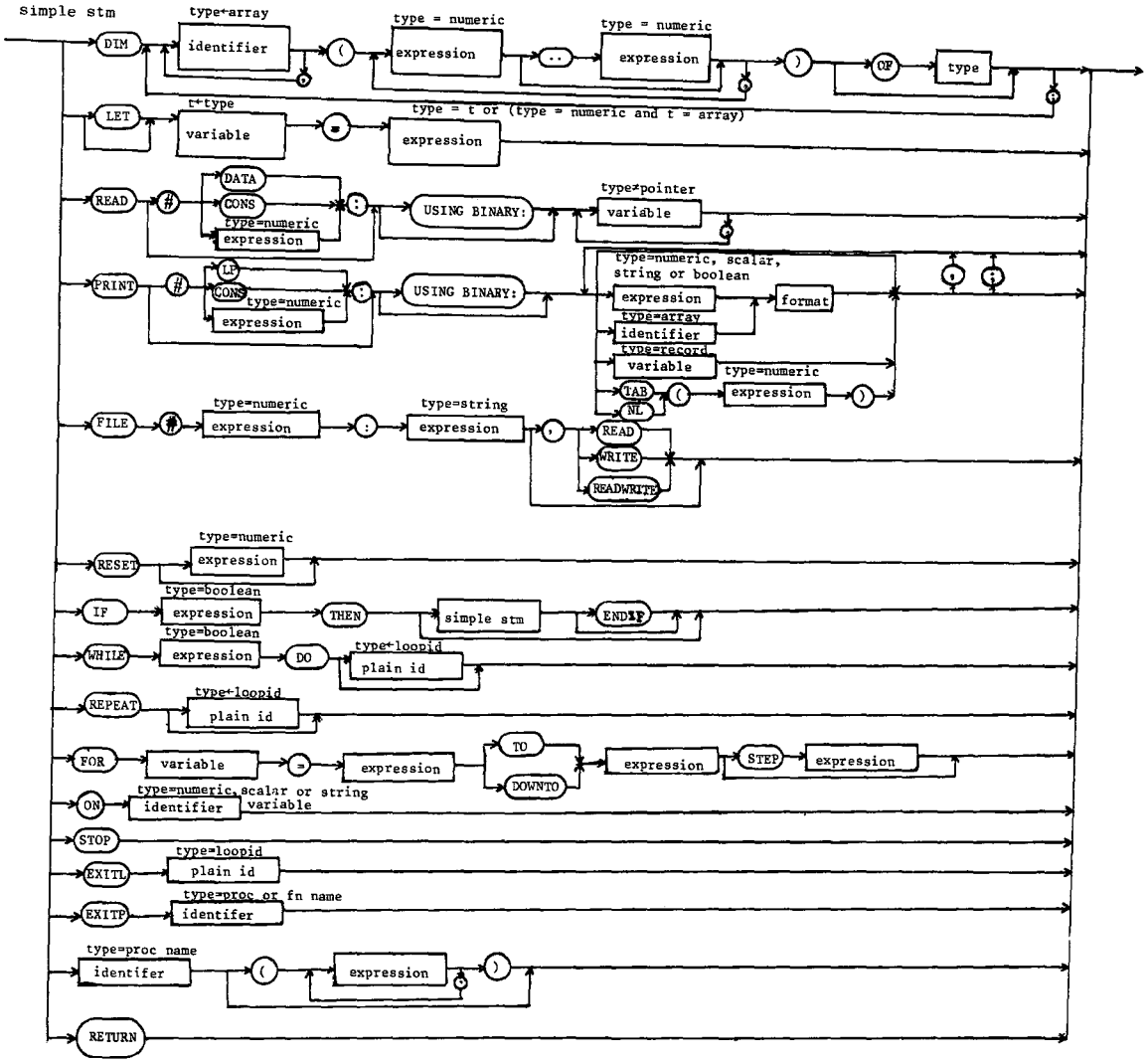
APPENDIX 2

Syntax diagrams for BPL

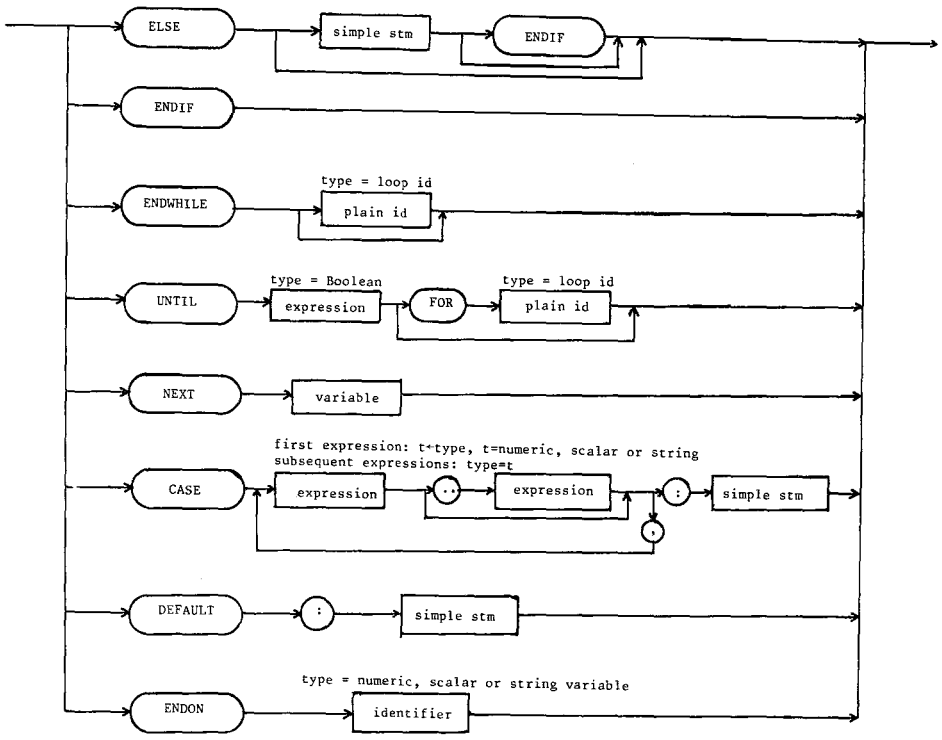
The comments regarding assignment of and checks on type which occur above some of the rectangular boxes are not exhaustive but do give some additional information which may assist the reader.



© Heyden & Son Ltd, 1982

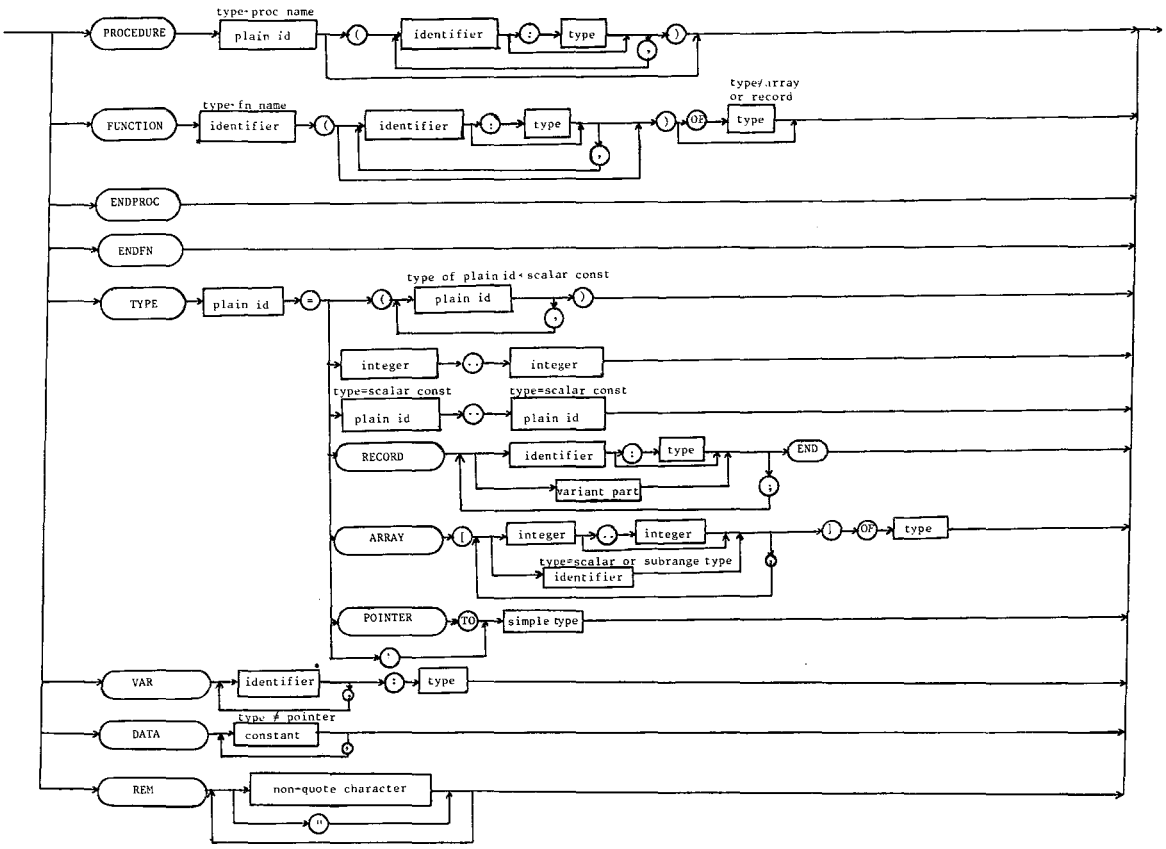


dependent stm

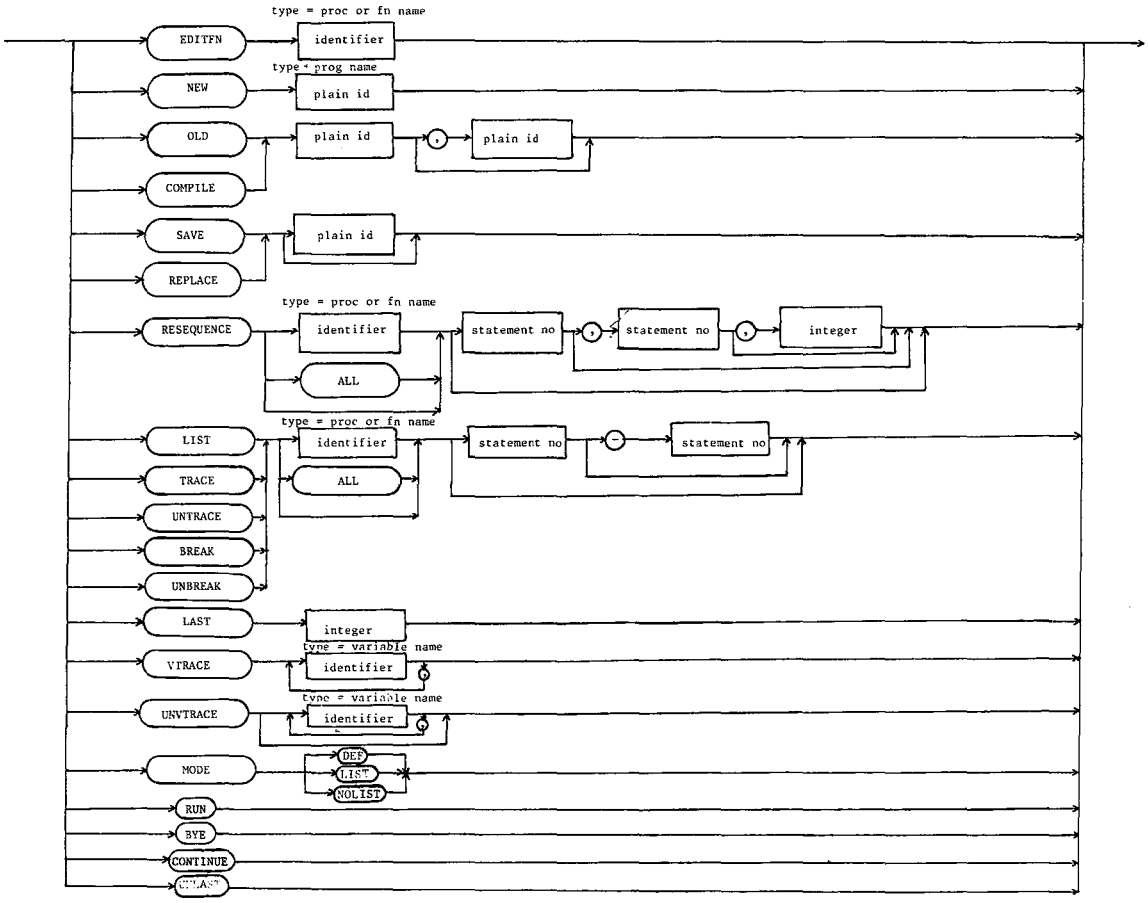


© Heyden & Son Ltd, 1982

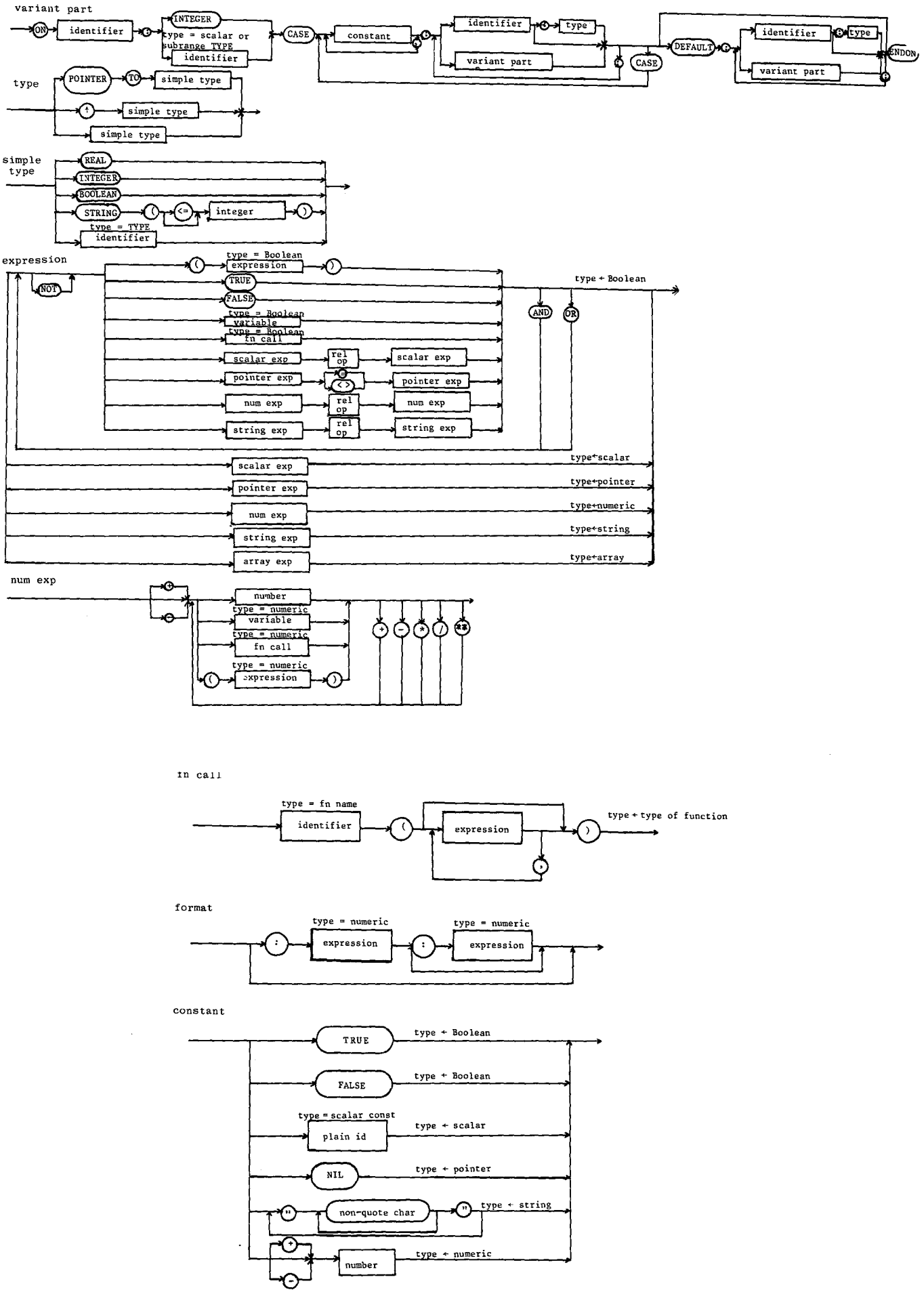
non-executable stm



command



© Heyden & Son Ltd, 1982



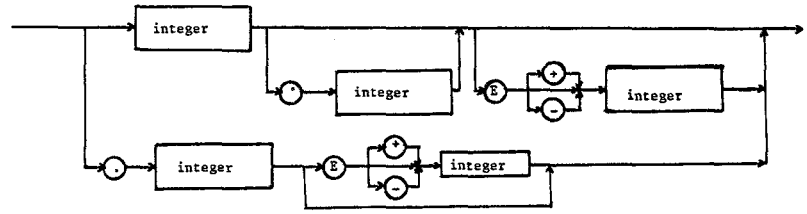
fn call

format

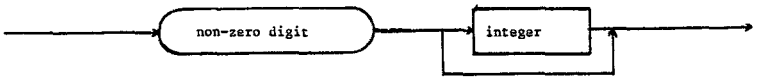
constant

© Heyden & Son Ltd, 1982

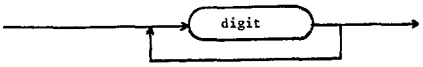
number



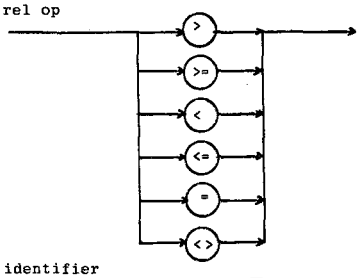
statement number



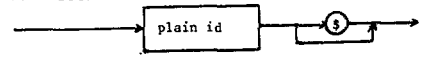
integer



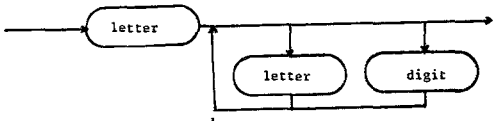
rel op



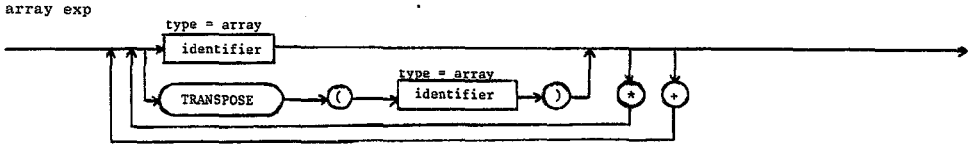
identifier



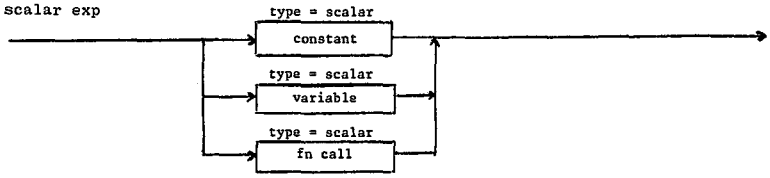
plain id



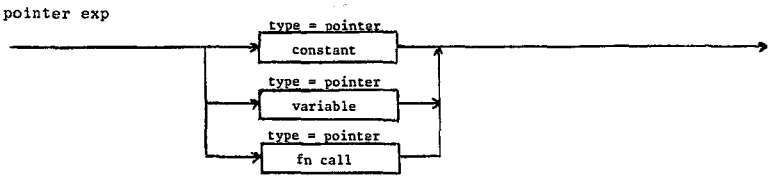
array exp



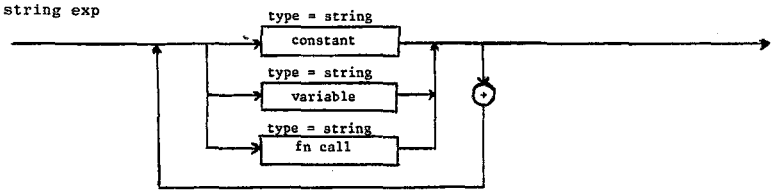
scalar exp



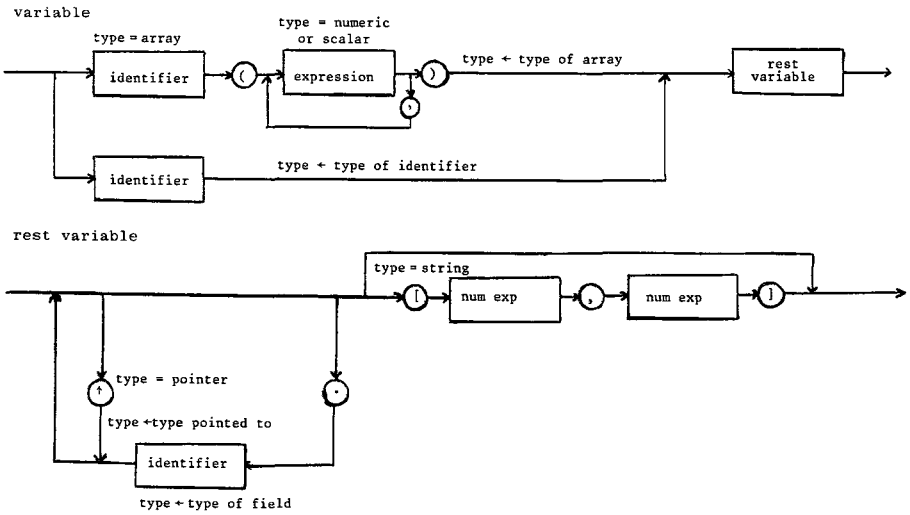
pointer exp



string exp



© Heyden & Son Ltd, 1982



AN INVENTORY OF SOFTWARE PACKAGES FOR INFORMATION WORK

In early Fall 1981 an announcement was made that COSTI (National Center of Scientific and Technological Information) has contracted with UNESCO, for the preparation of an inventory of software packages written for information work. Developers and vendors of such packages were invited to register with COSTI, in order to be put on the mailing list to receive a detailed questionnaire.

The response to our announcement has been overwhelming. More than 150 packages written for mainframes, minicomputers and microprocessors are already on file and more letters are coming in daily. The questionnaire, authorized by an international panel set up by UNESCO, was sent to all respondents early in 1982.

Some questions naturally arise during this initial data collection phase. The inventory intends to describe all software packages which serve or may serve in textual and alphanumeric information work, library and documentation systems, SDI and online searches, information and fact storage, retrieval and distribution, text processing and publication, etc. In order to keep the inventory within reasonable limits, only systems which will be installed and/or be operational by December 1982 will be incorporated. Systems which are produced by non-commercial institutions are actively sought out. There is a special interest in packages which are potentially available and transferable to developing countries.

The inventory is scheduled for publication by the end of 1982. Considerable work is going into the preparation of a format to maximize its usefulness. There will be a number of tables which will enable the user to identify the packages which are closest to his specific requirements. There will be a concise summary for each package detailing its principal features, availability, costs, etc.

Any software producer or vendor who has not yet registered with COSTI is kindly invited to do so, always considering that this inventory will be a very useful tool in advancing the state-of-the-art of information work.

For more details, please contact the National Center of Scientific and Technological Information (COSTI), P.O. Box 20125, Tel-Aviv 61201, Israel, phone number 03–297781/292766, telex number 3–2332 CSTI IL.

In your initial reply, please refer to *UNESCO Inventory of Software Packages* and indicate name(s) of packages(s) to be submitted, with full mailing details.

© Heyden & Son Ltd, 1982

11. M.H.WILLIAMS and G.CHEN, Translating Pascal for execution on a Prolog-based system, The Computer Journal, vol. 29, no. 3, 246-250 (1986).

Translating Pascal for Execution on a Prolog-based System

M. H. WILLIAMS* AND G. CHEN
Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ

One of the objectives of the Japanese Fifth Generation Computer Project is to develop computer systems whose kernel languages are based on logic programming rather than on the conventional imperative languages which have been in general use until now. This has led to conjecture about the problem of the large base of existing software which is implemented in imperative languages. To this end a study has been conducted into the possibility of translating programs written in Pascal-S into Prolog. Because of the radically different control and data structures in these two languages, the translation process is not straightforward. The problems associated with this process are discussed and its performance assessed.

Received October 1984

1. INTRODUCTION

In recent years Prolog,[1,2] a programming language based on symbolic logic, has been used to develop programs in a variety of application areas (e.g. natural language processing,[3] expert systems,[4] database query languages,[5] CAD modellers,[6] etc.). In the process logic programming languages have been found to have many advantages over conventional imperative languages.

The Japanese have recognised the potential of logic programming and have selected it as one of the foundation stones on which to establish their Fifth Generation Computer Project.[7] One of their aims is to develop several different computer systems whose underlying kernel languages will be variants of logic programming.

Since the announcement of the details of the Japanese plan, research on logic programming and its applications has intensified. At the same time concern has been voiced over the problem of the large base of existing software which is implemented in imperative languages, and what might happen to it if computer systems with these radically different architectures were to replace existing systems.

In order to assess the seriousness of this problem a study has been conducted to investigate the possibility of implementing conventional languages on a Prolog machine. For this purpose Pascal was chosen as representative of the class of imperative programming languages. Besides its virtues in the area of well-structured program constructs and rich data structures, Pascal has the advantage of a reasonably precise formal specification,[8–10] which is generally accepted and adhered to.

The radically different semantics (involving both data structures and program constructs) between Pascal and Prolog gives rise to several major problems in the translation process. These include the translation of **goto** statements, the evaluation of expressions, the execution of assignments, the translation of procedure or function bodies and the translation of control statements.

As a first step in preparing a Pascal program for execution on a Prolog machine, all **goto** statements can be removed from it by mapping it into an equivalent Pascal program without **goto** statements. The process for accomplishing this is described in Ref. 11.

To simplify the problem it was decided to concentrate on a subset of Pascal, called Pascal-S.[12] This does not include variant fields within record types, pointer types or packed variables. Set and file types will be discussed briefly but their implementation is not dealt with in full.

The next section discusses the main features of the two languages and the general principles of translation between them. Subsequent sections concentrate on different aspects of the problem.

* To whom correspondence should be addressed.

2. GENERAL PRINCIPLES

In order to understand the problems of translation from a conventional imperative language like Pascal to a declarative language like Prolog, consider briefly the main features of the two languages and the differences between them.

2.1. Pascal

In an imperative programming language such as Pascal, the fundamental mode of operation is based on changing the state of variables through assignments or other similar language constructs. These variables are used to imitate the storage of the underlying machine. This basic dependence upon variables and the association of values with variables is characteristic of a von Neumann architecture.[13]

The execution of a Pascal program which realises some algorithm may be regarded as a sequence of state transformations in which the state of the store or the current point of control or both may change. This is reflected in the three main constructs of the language, namely,

(*a*) declarations which define the attributes of storage locations,

(*b*) assignment statements which effect transformations from one state to another by updating storage locations, and

(*c*) control statements which determine the point of control at any instant and hence the order of state transformations.

2.2. Prolog

By contrast Prolog is a high-level programming language based on a subset of first-order predicate calculus formulae known as Horn clauses. A well-formed

first-order predicate calculus formula in clausal form has the form

$$P_1 \vee P_2 \vee \ldots \vee P_n \vee \neg Q_1 \vee \neg Q_2 \vee \ldots \vee \neg Q_m$$

or $P_1 \vee P_2 \vee \ldots \vee P_n \leftarrow Q_1 \wedge Q_2 \ldots \wedge Q_m$

or $P_1 \vee P_2 \vee \ldots P_n$ if $Q_1 \wedge Q_2 \wedge \ldots \wedge Q_m$

A Horn clause is a restricted case in which $0 \leqslant n \leqslant 1$ and has one of the following three forms:

$$P_1 \leftarrow Q_1 \wedge Q_2 \wedge \ldots \wedge Q_m$$

$$P_1$$

$$\leftarrow Q_1 \wedge Q_2 \wedge \ldots \wedge Q_m$$

where the symbol '$\leftarrow$' is read as 'if' or 'is implied by'. The first clause is an implication or rule, the second is an assertion or fact and the third a query.

These three forms of Horn clauses form the basic constructs in Prolog. The first form is that of a Prolog rule in which the predicate on the left-hand side is defined to be the conjunction of goals on the right-hand side (the rule body), e.g.

```
parent(X, Y):- father(X, Y).
```

which may be read as 'X is a parent of Y if X is a father of Y'. The second form reflects a Prolog fact; an example of this might be

```
father(john, mary).
```

which may be read as 'john is the father of mary'. The third form reflects a question to be answered. For example

```
?- father(john, X).
```

which may be read as 'of whom is john the father?'.

The basic data structures used in Prolog are numeric and string constants, variables and compound terms. The concept of a variable in Prolog is totally different from that in Pascal. A variable in Prolog stands for a single object; it will be undefined before it is instantiated (its value is determined), but after it has been instantiated its value may not be altered except when backtracking returns to the point where the variable was instantiated and undoes this instantiation.

2.3. Translating from Pascal to Prolog

Following the state transformation approach a block in Pascal can be regarded as an action which transforms some initial state S_0 into some final state S_n. Each statement ST_i ($1 \leqq i \leqq n$) of the block will transform state S_{i-1} into state S_i, so that the execution of the block as a whole may be represented by the sequence

$$\{S_0\}ST_1\{S_1\}ST_2\ldots\{S_{n-1}\}ST_n\{S_n\}$$

in which ST_i is the ith statement of the block and S_{i-1} is the pre-state vector of ST_i and S_i its post-state vector. Alternatively this may be written as

$$ST_1(S_0, S_1), ST_2(S_1, S_2), \ldots, ST_n(S_{n-1}, S_n)$$

where each ST_i represents a function which maps state S_{i-1} into state S_i.

If each statement ST_i is thought of as being represented by a Prolog rule, a block consisting of statements ST_1, ST_2, ..., ST_n may be translated as

$$block(S_0, S_n) \leftarrow st_1(S_0, S_1), st_2(S_1, S_2), \ldots, st_n(S_{n-1}, S_n)$$

where the subgoals st_1, st_2, ..., st_n are the names of predicates defining the effects of statements ST_1, ST_2, ..., ST_n. These are similar to procedure calls in conventional languages.[14]

Control statements are treated in a similar manner except that some control component is required to test the condition part of the control statement and transfer control accordingly. For example, a **while** statement

while c = 1 do begin ST_1; ST_2; ...; ST_n end

will be translated as the goal

...$statement_i(S_{i-1}, S_i)$...

in the sequence of statements in which it appears and will result in a definition of the form

$$statement_i(S_0, S_{n+1}) \leftarrow C = 1, st_1(S_0, S_1), st_2(S_1, S_2), \ldots, st_n(S_{n-1}, S_n), statement_i(S_n, S_{n+1}).$$

$$statement_i(S, S).$$

Simple statements such as assignment statements, input–output statements and procedure calls do not necessarily require separate rules to define them; they may be incorporated directly in the parent rule.

In practice, it is not necessary to pass the complete state as a parameter from statement to statement; usually a subset of the state, a substate vector, will suffice. This will be discussed in detail in later sections.

3. DECLARATIONS AND STATEMENTS

In order to translate a goto-less Pascal program into Prolog, the program must first be parsed and converted into a suitable internal form.

In the approach adopted the declaration part and the statement part are treated separately. The translation for each part makes use of the ability in Prolog to define a wide range of operators, and to specify for each its precedence and associativity. If all Pascal reserved words are defined as operators, a section of a Pascal program may be treated as a Prolog term and parsed accordingly. The resulting program will have the form of a Prolog term. For example, a **while** statement of form **while** C **do** X may be represented as

```
while(C, X),
```

a sequence of statements as

$;(S_1, S_2)$

where S_1 is the first statement of the sequence and S_2 the remainder of the sequence, and so on.

One set of rules is used to analyse the declarations in a program and will parse each of the definitions for constants, types, variables, etc. Information such as the type and scope of each variable is stored for use in parsing the statement part. The second set of rules is used to parse the statements in the body of a block. The statements are parsed in top-down fashion with information being retrieved and stored as required.

As intimated in the previous section, when a control statement is translated into Prolog, it is not always necessary to pass the complete state vector as a parameter

to it. In order to determine for each statement ST_i the smallest subset of the state vector which will suffice, two sets of variables, M_i and A_i, are defined.

The set M_i consists of those legally accessible variables in the scope of statement ST_i whose values may be modified when ST_i is executed, e.g. destinations of assignment statements, parameters for input statements and actual parameters associated with formal variable parameters whose values are modified in procedures or functions called within ST_i.

The set A_i consists of those legally accessible variables in the scope of statement ST_i whose values may be accessed in any expressions in ST_i, excluding any variables whose values are set within ST_i before being used. For example, in the following **repeat** statement (ST_i)

```
repeat v: = a; w: = w+1; x: = v+w;...until c;
```

the set A_i will contain the variable w but not the variable v.

During the parse a syntax tree is constructed for each body of a block. Each leaf node of this tree represents a simple statement (assignment statement, input–output statement or procedure call); each branch node corresponds to a compound or structured statement in the block. Each branch node has associated with it the two variable sets A and M.

By the end of the parse the source program will be decomposed into a constant table, a variable definition table (which maps each variable to its type), a procedure table (in which each procedure or function name is associated with a parameter list and a syntax tree) and the syntax tree for the main body of the program.

4. VARIABLES AND ASSIGNMENTS

The basic structure of the Prolog output code produced by the translation process is described in Section 2. The body of each Pascal block will be translated into a Prolog rule which calls the subgoals corresponding to the statements in this block. All Pascal variables required in this block must be accessible in this rule, and the proper subsets of the variables will be passed to and updated by the subgoals corresponding to each statement in the block. Thus the compound and structured statements will call the statements in their bodies until a simple statement such as an assignment, procedure call or input–output statement is encountered.

The simple variables in a source program can be handled by being passed as parameters between rules. Each time a variable is updated by an assignment in Pascal, a new variable will be created to substitute for the old one in Prolog. To accomplish this the compiler keeps a record of each Pascal variable name paired with its corresponding Prolog token (a Prolog variable) at any point in the compilation process. The current token for a Pascal variable will be changed whenever the latter is updated by an assignment or input statement or substituted by an actual parameter in a procedure or function call. For example,

```
a: = b+c; b: = b+1;
...a...b...;
```

will be translated as

```
A is B+C,  B1 is B+1,
...A...B1...
```

where '**is**' is a built-in predicate in Prolog which sets the first argument equal to the result of the expression in the second argument.

The structured type array is more difficult to represent since Prolog does not support arrays as such. In the approach adopted a predicate **access** is used to select an element from an array and return this element in a Prolog variable, while a predicate **update** is used to update a single specified element in an array.

Initially Prolog lists were used to represent arrays. The predicates **access** and **update** operated on lists, finding the appropriate element and performing the operation required. However, this proved to be very inefficient. Each access to (or update of) an array required time proportional to the number of elements in the array to perform.

As an alternative to simple lists, height-balanced binary trees were tried (see Section 7). These reduced the access time for an array of n elements from o(n) to $o(\log_2 n)$ which produced a considerable improvement in the execution time of the resulting programs.

It is natural to represent a variable of type **record** as a Prolog term of arity n corresponding to n fields. Such a representation can be nested if the fields of a record type are in turn structured data types such as arrays or other records. A variable of type **set** can be handled as a list if special operations are defined for performing the test of membership of a set, union, intersection and so on. These can be provided as built-in functions to be used in the Prolog program. The file type can also be handled in Prolog without difficulty.

Simple statements such as assignments, input–output statements and procedure calls will not be defined as separate rules, instead they will be translated directly in the context of their parent rules. In translating an expression the code for calling the functions in the expression and the code for evaluating the elements of structured variables will be generated first, the results being stored in temporary variables. Then the predicate **is** is used to evaluate the expression and to instantiate a variable with this value. This variable may be a new version of a simple variable, or the value with which a tree is updated or simply an intermediate variable used, for instance, in testing a condition. When a new version of a variable is created, the compiler will update the present token corresponding to the variable and this token will be used in the following context until the variable is again updated.

5. HANDLING PASCAL PARAMETERS

The approach used to treat variables discussed in the previous section applies equally to variables in the main body of the program as it does to local variables or formal value parameters within a procedure or function. However, the treatment of formal variable parameters and global variables referred to within a procedure or function body poses rather different problems.

The major problem is caused by the fact that two distinct variable names may refer to the same location in a Pascal program, i.e. variable aliasing. This may arise in Pascal-S when the same variable is used as actual parameter for two different formal variable parameters, e.g.

```
var i:integer;
procedure ex1(var x,y:integer);
  begin
    ...
  end;
...
ex1(i,i);
...
```

or when a variable used as an actual parameter in a procedure call is also referred to directly as a global variable within the procedure or function body, e.g.

```
var i:integer;
procedure ex2(var x:integer);
  begin
    ...
    i:=i+1;
    x:=x+i;
    ...
  end;
...
ex2(i);
...
```

The problem may be further complicated by several levels of indirection, e.g.

```
var i:integer;
procedure ex3a(var x:integer);
  procedure ex3b(var y:integer);
    procedure ex3c(var u, v:integer);
      begin     (* body of ex3c *)
        ...
      end;
    begin     (* body of ex3b *)
      ...
      ex3c(y,i);
      ...
    end;
  begin     (* body of ex3a *)
    ...
    ex3b(x);
    ...
  end;
begin     (* main program *)
  ...
  ex3a(i);
  ...
end.
```

or may be disguised as in the case of a subscripted variable, e.g.

```
procedure ex4(var x, y:integer);
  begin
    ...
    x:=...;
    y:=...;
    ...
  end;
ex4(a[i],a[j]);
```

In the approach described in the previous section each Pascal variable x is translated as a sequence of Prolog variables $\{X_1, X_2, \ldots, X_n\}$ in the compilation process. However, when variable aliasing may arise, it is necessary to split the translation into two mappings, one from the domain of variables to that of locations and the other from locations to values.

For each procedure or function P whose set of formal parameters contains at least one variable parameter, a set S can be defined which consists of all variable parameters from the formal parameter set and any global variables referred to in the procedure or function body. If the set S contains more than one member then:

(*a*) When a call to P is compiled, code is generated for each element x in S defining a map @: Int → Value from a location N in the Prolog database to the present value of x (@(N, X)), and a reference to N will be the initial value of x when P is activated. This is followed by the call to P and finally code to collect the value of x from the data base and destroy this map.

(*b*) In compiling the body of P whenever a reference to an element x in S is encountered, a reference is made to the data base (@(N)) using special predicates ac_fp and up_fp defined as:

```
ac_fp(@(N), V) ← @(N, V).
up_fp(@(N), V) ← retract(@(N,_)), assert(@(N, V)).
```

For the sake of efficiency, this process can be improved in two ways:

(*a*) if no call to P can give rise to variable aliasing, the elements of S can be treated in the same way as variables dealt with in the previous section;

(*b*) if calls to P can give rise to variable aliasing, it may still be possible to define for each procedure call of P a subset of S which is relevant.

To handle the latter case where some calls to P may cause variable aliasing, the predicates ac_fp and up_fp are redefined as follows:

```
ac_fp(@(N), V) ← @(N, V).
ac_fp(V, V).
up_fp(@(N), V, @(N)) ← retract(@(N, _)), assert
                       (@(N, V)).
up_fp(_, V, V).
```

In each case the first rule involves a double mapping while the second uses a single map.

When a call to P is compiled, if an element x in S is an alias for at least one other element of S, the map for x and its initial value will be defined and inserted before the call, and code generated to collect the value of x and destroy this map after the call. If x is not an alias for at least one other element of S, the present value of x will be passed to P directly.

The process of detecting variable aliasing can be described briefly as follows.

(*a*) For each call to a procedure or function an entry is created which records the formal variable parameters and the corresponding actual parameters used in the call. In addition, for each procedure or function definition an entry is set up to record each global variable used in the body of the definition.

(*b*) Once the whole program has been scanned and all calls analysed, the closure is constructed, yielding the chain of possible actual parameters corresponding to each formal variable parameter.

(*c*) For each procedure or function P the chains corresponding to the formal variable parameters of P are searched for any elements which occur in more than one chain, or for any global variables which are used in the body of P. If any such element is found, variable aliasing may occur.

(*d*) If two array elements with variable subscripts a[i], a[j], are used as actual parameters, it is, in general, far more difficult to determine at compile time whether aliasing occurs (i.e. . i = j) or not. In this instance it is simply assumed that aliasing may take place.

6. CONTROL STATEMENTS

Before the Prolog code for each compound or control statement is generated, two state vectors, IN and OUT, are constructed for the statement. The variable sets A_i and M_i are used for this purpose as follows.

(1) For the main body of the program the two vectors will both be empty and the corresponding Prolog rule generated is:

$$\text{program} \leftarrow st_1(IN_1, OUT_1), st_2(IN_2, OUT_2) \ldots, st_n(IN_n, OUT_n)$$

where n is the number of control statements in the main body which are not nested within other control statements.

(2) For each procedure or function body, the variables in IN will be those in the formal parameter list together with any global variables accessed in the body. The variables in OUT will be those formal variable parameters and global variables modified in the body. In the case of a function, the function name will also belong to OUT. The Prolog code generated is similar to that for the main body except that two parameter vectors are used in this case.

(3) For each **if** or **case** statement the variables in set OUT are those in M_i whose values are used in subsequent statements at the same level (ST_{i+1}, ..., ST_n), i.e.

$$\{x \mid x \in M_i \wedge x \in (A_{i+1} \cup A_{i+2} \cup \ldots \cup A_n)\},$$

together with any variables which will be used in its parent's successor statements (belonging to its parent's OUT set) but which will not be modified by its own successor statements, i.e.

$$\{x \mid x \in M_i \wedge x \in (OUT(parent(ST_i)) \setminus (M_{i+1} \cup \ldots \cup M_n))\}$$

where OUT(parent(ST_i)), the OUT set of ST_i's parent statement, has already been constructed. Thus the OUT set for conditional statement ST_i will be:

$$OUT = M_i \cap (A_{i+1} \cup \ldots \cup A_n \cup (OUT(parent(ST_i)) \setminus (M_{i+1} \cup \ldots \cup M_n)))$$

The set IN is defined as:

$$IN = A_i \cup OUT$$

i.e. all variables used in ST_i(in set A_i) together with any variables in OUT which return values if the variables are not updated.

The Prolog rule generated for an **if** statement is:

st_i(IN, OUT) ← condition, ...goals for then clause...
st_i(IN, OUT)← ...goals for else clause...

If the condition is satisfied and found to be true then the goals for the **then** clause will be executed. Otherwise the second rule will be attempted and the goals for the **else** clause executed. If the **else** clause is absent, the following rules will be generated:

st_i(IN, OUT) ← condition, ...code for then clause...
st_i(IN, IN) ←

In this case if the condition fails, the post-state vector will be the same as the pre-state vector.

The code generated for a **case** statement ST_i is similar, namely,

$st_i(Label_1, IN, OUT)$ ← ...code for option 1...
$st_i(Label_2, IN, OUT)$ ← ...code for option 2...
...
$st_i(Label_n, IN, OUT)$ ← ...code for option n...
$st_i(_, _, _)$ ← ...error occurs...

(4) For an iterative statement (**for**, **while** or **repeat**) the sets IN and OUT will be defined in the same way as those for conditional statements.

The code generated for a **for** statement will be:

st_i(IN, OUT) ← ...bound condition,
...goals for loop body...,
...modify bound condition...,
st_i(IN′, OUT).
st_i(IN, IN) ←

which states that to execute a **for** statement it is necessary to satisfy the bound condition first. If it holds, the goals for the body will be called one by one, the bound condition will be modified and the goal itself will be called to perform further iterations. If the condition fails, the current state will be returned to the post-state vector of this rule by the second rule.

The format of the code generated for a **while** or a **repeat** statement will be similar to that for a **for** statement.

Example

A simple example is given to demonstrate the translation process. The procedure 'mean' reads a group of integers terminated by a zero and finds the maximum, minimum and average values. The main body simply calls procedure 'mean' and prints the results.

```
program exam(input, output);
  var a,b,c:integer;
  procedure mean(var min,max,ave:integer);
    var a, t, n:integer;
    begin
      read(a); max: = a; min: = a; n: = 0; t: = 0;
      while a > 0 do
        begin
          if a > max
            then max: = a
            else if a < min
              then min: = a;
          n: =n+1; t: = t+a; read(a)
        end;
      ave: = t div n
    end;
  begin
    mean(a,b,c);
    writeln(a); writeln(b); writeln(c)
  end.
```

In the topmost level of procedure 'mean' there are six simple statements and one structured statement (the **while** statement). In the **while** body there are three simple

statements and one structured statement, an **if** statement (called if1), within which another **if** statement (called if2) is nested. For the **while** statement, A = {a, max, min, t, n}, M = {max, min, t, n}. Because the variables max, min, t, n will all be used after the **while** statement, OUT = M, and IN = (A ∪ OUT) = A. For if1, A = {a, max, min}, M = {max, min}. Since variables max and min occur in the OUT set of the parent rule (OUT for while), OUT = M and IN = A. The same applies to if2, where A = {a, min}, M = {min}, OUT = M, IN = A. Thus the Prolog program will be:

```
exam:- mean (_, _, _, A, B, C),
     write(A), nl, write (B), nl,
     write(C), nl.
mean(_, _, _, Min, Max, Ave):- read(A),
     while(A, A, A, 0, 0, Max, Min, T, N),
     Ave is T/N.
while (A, Max, Min, T, N, Max1, Min1, T1, N1):-
     A > 0, if1(A, Max, Min, Max0, Min0),
     N0 is N+1, T0 is T+A, read(A1),
     while(A1, Max0, Min0, T0, N0, Max1, Min1, T1,
     N1).
while(_, Max, Min, T, N, Max, Min, T, N).
if1(A, Max, Min, A, Min):- A > Max.
if1(A, Max, Min, Max, Min1):- if2(A, Min, Min1).
if2(A, Min, A):- A < Min.
if2(_, Min, Min).
```

7. DISCUSSION

The chief differences between Pascal and Prolog from the point of view of translation are the following.

(*a*) Iterative control structures. Whereas most algorithms in conventional languages employ iterative constructs, these have to be achieved by recursive constructs in Prolog.

(*b*) Procedure calls. In general Prolog programs make much heavier use of procedure calls than do Pascal programs. Furthermore, each procedure call in Prolog requires more time than a Pascal procedure call, partly because of the pattern matching which is performed in seeking a matching rule and partly because of the additional control stack management which is required for backtracking.

(*c*) Data representations. In view of the fundamental differences between variables and parameters in Pascal and those in Prolog, each Pascal variable or parameter may be translated as a pair of entries, (name,loc) and (loc,value), in the Prolog database. However, this is very inefficient. A more efficient solution which uses Prolog variables and parameters requires a much greater degree of processing at compile time.

(*d*) Data structures. A fundamental structured data type available in virtually all conventional programming languages is the array. However, arrays are not provided in Prolog and their effect must be achieved by other means. Initially lists were used, but these proved to be too inefficient. Several different alternative approaches involving Prolog terms were tried but problems were encountered due to limitations on the size of such a term and methods of updating it. The best solution tried was a binary tree representation. This requires $o(\log_2 n)$ time to access, which proved more acceptable, although even with this representation the execution time is large for programs which make heavy use of large arrays. Several simple examples have been tested on the C-Prolog system to assess this bottleneck problem. Of six examples tested, three use an array as the main data structure. The sizes of the arrays range from 5 to 400 elements. The time efficiency of compiled code compared with the Unix Pascal interpreter EM1 and with Prolog written by hand is shown in Table 1. CODE/PASC is the ratio of the execution time of code generated to that of the original program running under Pascal EM1, CODE/PROL is that of execution time of generated code to that of an equivalent Prolog program written by hand. Each of the test programs was executed with a number of different sets of data, and the range of ratios presented in the table reflects the range of ratios observed. The results show clearly the bottleneck which arises due to array variables. Although this is not the only source of inefficiency (e.g. heavy recursion is also inefficient), it did appear to be the most important single factor.

Table 1. Ratio of the execution time of the Prolog code generated by this system to that of the original Pascal program running under the EM1 interpreter (CODE/PASC), and of the Prolog code generated by this system to that of an equivalent Prolog program written by hand (CODE/PROL) for a number of sample programs.

Example no.	Uses arrays	CODE/PASC	CODE/PROL
1	Yes	10–25	11–19
2	Yes	10–23	6–10
3	Yes	8–18	3–5.5
4	No	0.9–1.6	1.5–1.9
5	No	1.4–2.4	1.3
6	No	1.3	1.4

The method discussed above has been applied to the translation from Pascal-S into Prolog. In Pascal-S, the data structures are the basic data types integer, real, char and boolean and the structured data types **array** and **record** (without variants). Omitted are the **set** and **file** structures (apart from the standard textfiles input and output), the pointer type and packing options. For the dynamic data type, one solution might be to use a tree to hold all the objects and to pass it as a parameter to the rules.

The 'real' data type was omitted from this particular study since real numbers are not available in some Prolog implementations. However, this does not affect the basic method and algorithm.

Another simplification which Pascal-S makes is the omission of procedure parameters. Although these may be handled by the predicate '=..' ('univ'), their use complicates the analysis of variables and parameters outlined in the paper and may necessitate the use of the simpler and less efficient approach mentioned in (*c*) above.

Acknowledgement

The authors would like to thank Stuart Anderson for helpful discussions and suggestions and Greg Michaelson for reading and providing comments on this paper. Mr G. Chen is currently supported by a scholarship from the British Council.

REFERENCES

1. W. F. Clocksin and C S. Mellish, *Programming in Prolog*. Springer-Verlag, Heidelberg (1981).
2. P. Roussel, *Prolog: Manual de Référence et d'Utilisation*, Groupe d'Intelligence artificielle, Université d'Aix-Marseille, Luminy (1975).
3. F. C. N. Pereira and D. H. D. Warren, Definite clause grammars for language analysis – a survey of the formalism and a comparison with argumented transition networks. *Artificial Intelligence* **13** (3), pp. 231–278 (May 1980).
4. K. L. Clark and F. G. McCabe, *Prolog: A Language for Implementing Expert Systems*. Department of Computing, Imperial College, London (November 1980).
5. J. C. Neves, S. O. Anderson and M. H. Williams, A Prolog implementation of Query-by-Example. *Proceedings of the 7th International Computing Symposium*, edited H. J. Schneider, pp. 318–332. B. G. Teubner, Stuttgart (1983).
6. J. Camacho Gonzalez, M. H. Williams and I. E. Aitchison, Evaluation of the effectiveness of Prolog for a CAD application. *IEEE Computer Graphics and Applications* **4** (3), 67–75 (March 1984).
7. T. Moto-Oka, Challenge for knowledge information processing systems. In *Fifth Generation Computer Systems*, pp. 3–89. North-Holland, Amsterdam (1982).
8. N. Wirth, The programming language Pascal. *Acta Informatica* **1** (1) 35–63, (1971).
9. *Specification for Computer Programming Language Pascal*. British Standards Institute, BS 6192:1982 (1982).
10. K. Jensen and N. Wirth, *Pascal User's Manual and Report*. Springer-Verlag, New York (1975).
11. M. H. Williams and G. Chen, Restructuring Pascal programs containing GOTO statements, *The Computer Journal*, **28** (2) 134–137 (1985)
12. N. Wirth, Pascal-S: a subset and its implementation. In *PASCAL: the Language and its Implementation*. Wiley-Interscience, New York (1981).
13. J. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM* **21** (8) 613–641 (August 1978).
14. H. Gallaire, A study of Prolog. In *Computer Program Synthesis Methodologies*, edited A. W. Bierman and G. Guiho, pp. 173–212. Reidel, Dordrecht (1983).

12. G.CHEN and M.H.WILLIAMS, Executing Pascal programs on a Prolog architecture, Information and Software Technology, vol. 29, no. 6, 285-290 (1987).

Executing PASCAL programs on a PROLOG architecture

by G CHEN and M H WILLIAMS

Abstract: The decision by the Japanese to build their Fifth Generation Computer Project around systems based on logic programming rather than on conventional imperative languages is a significant departure from the style of computers developed in the past. The effect which this might have on computer systems in the future has led to concern about what might happen to the large base of existing software which is implemented in imperative languages. To allay fears on this score an investigation has been conducted into the feasibility of translating conventional languages like PASCAL into PROLOG. The results of this study are reported.

Keywords: programming languages, PASCAL, PROLOG.

Since its development in 1970 as a tool for research in artificial intelligence, the use of the programming language PROLOG[1,2] has grown relatively slowly. Nevertheless it has been successfully used to develop programs in a number of different application areas, including natural language processing[3], expert systems[4], database query languages[5], CAD modellers[6], etc.

Recently, the importance of the language has received a considerable boost as a result of the decision by the Japanese to establish their Fifth Generation Computer Project on logic programming languages[7]. One of the objectives of this project is to develop several different computer systems whose kernel languages are variants of PROLOG. Subsequently other national and international research programmes provoked by the Japanese undertaking have also recognized the importance of logic programming.

Although there has been considerable debate over the likelihood of success of these research programmes and their effects over the next decade or two, there has also been some concern regarding the problem of the large base of existing software which is implemented in imperative languages. Also, there has been concern about the software if computer systems with these radically different architectures were to replace existing systems.

Ideally, programs which have been developed in imperative languages should be able to be run, albeit inefficiently, on architectures designed for logic programming languages. To assess the feasibility of this, a study has been conducted into the possibility of translating programs in a conventional imperative language into equivalent programs in PROLOG. For the purpose of this exercise PASCAL was chosen as representative of the class of imperative languages, partly because of its simplicity and neat control and data structures and partly because it has a reasonably precise formal specification[8-10].

Because of the fundamentally different approach to control in the two languages (PASCAL and PROLOG), the **goto** statement is particularly difficult to translate. To handle this, the translation process has been divided into two stages. In the first stage all **goto** statements are removed from the PASCAL program. This is achieved by mapping it into an equivalent PASCAL program without **goto** statements. The second stage converts a structured PASCAL program into an equivalent PROLOG program. To simplify the problem, it was decided at this stage to concentrate on a subset of PASCAL, PASCAL-S[11]. This does not include pointer types, file types, set types, variant record fields or packed variables.

Besides the handling of **goto** statements, there are a number of major problems in the translation process. These include:

- differences in the nature of variables in the two languages,
- approach to parameters,
- difference in structured data types,
- handling of assignments,
- translation of procedure or function bodies.

This paper reviews briefly the approaches adopted in the two stages of the translation process and the problems currently under investigation.

Department of Computer Science, Heriot-Watt University, Edinburgh, UK

0950-5849/87/060285-06$03.00 © 1987 Butterworth & Co (Publishers) Ltd.

Restructuring the PASCAL program

PROLOG is a logic programming language based on a subset of first order predicate calculus. The execution of a PROLOG program corresponds to the proof of some goal, i.e. that some particular relation is satisfied. This is done by recursively matching the goal against the relation definitions until either a match is found or the attempt fails. In sequential PROLOG the order of evaluation is strictly left to right and depth first. When more than one match can be made, the clauses are selected in top down order.

This approach to control is fundamentally different from that in imperative languages. In the latter case control constructs can be divided into two categories.

Simple structured constructs. This includes simple sequence, single-entry single-exit conditions and loops, and procedure or function calls. These are relatively 'well-behaved' and can be mapped onto PROLOG without much difficulty.

Unstructured constructs. In the case of PASCAL, the only statement which leads to unstructured constructs is the **goto** statement. There is no corresponding statement which is equivalent to this in PROLOG, nor is it possible to produce the effect of one without great difficulty. Thus, when a PASCAL program is to be translated into PROLOG, the only practical way of handling **goto** statements is to remove them altogether by mapping the program into an equivalent one without **goto** statements.

In formulating an algorithm to remove all **goto** statements from a PASCAL program, it is convenient to regard a PASCAL program as a hierarchical structure. Thus a block is made up of a sequence of statements which are at the same level (one lower than the block itself) and will be executed sequentially. Likewise the body of a loop statement (**for**, **repeat** or **while**) consists of a sequence of statements which are at a level one below that of the loop statement and which will be executed sequentially. The same is true of **if** or **case** statements. Hence each program can be regarded as a hierarchical tree structure with simple statements (such as assignment, input-output or procedure call) as leaves and structured statements as branch nodes. By this reasoning a **goto** statement is treated as a leaf node in the tree.

Following Jensen and Wirth[12], in a valid PASCAL program:

- A **goto** statement which is outside a structured statement will not be directed to a label within the structured statement.
- A **goto** statement which is within one limb of a conditional statement will not be directed to a label within another limb of the same conditional statement.

Since this study is only concerned with correct PASCAL programs, neither of these conditions should be violated.

The process of restructuring is based on mapping unstructured constructs in the PASCAL program into equivalent ones until a structured program is arrived at. The transformations used can be divided into three categories. First, a branch to a point in the program at the same level as the **goto** statement. This will be transformed into a **repeat** or an **if–then** statement depending on whether it is a backward or forward jump respectively. In doing so this will create a new level corresponding to the body of the loop or limb of the conditional statement. Some labels may become further separated from the **goto** statements which refer to them by being embedded into the new **repeat** or **if** statement. Additional transformations are used to move labels out of such structures. Normally one Boolean variable is introduced as a flag for each label moved.

Second, a branch from within a structured statement to a point outside this statement. By using one flag, a **goto** statement can be moved out of the structured statement, thereby decreasing the level of that **goto** statement by one.

Third, a branch from within a procedure or function body to some point outside the body (abnormal exit). This can be replaced by assigning the value true to a global flag to indicate that the abnormal exit takes place in the body of the procedure or function, and inserting a jump instruction to the end of the body to terminate the execution. The abnormal exit is therefore translated into a normal jump instruction and an instruction setting the global flag. On the other hand, for each function or procedure call which may be affected by such an abnormal exit, an instruction must be inserted after the invocation of the function or procedure, which tests the global flag and branches to the corresponding label if the flag is true. Thus, any expression containing such functions must be split up so that the evaluation of a function is isolated and a dummy variable is introduced to keep the value of the function temporarily. After all functions have been evaluated, the value of the expression is calculated by substituting the values of the dummy variables in place of the function calls.

Based on these three groups of transformations, the algorithm to remove all **goto** statements operates briefly as follows. First, all abnormal exits in the procedure and function definitions need to be removed using the third group of transformations. These transformations are applied repeatedly until all the abnormal exits are translated into normal jump instructions which do not cross procedure body boundaries. Second, all remaining **goto** statements are removed by systematic application of the first two groups of transformations.

A system for restructuring PASCAL programs in this way has been implemented in PROLOG. The only overhead incurred in restructuring in this way is the flags and

the operations on them. Normally a single pass of the PASCAL source program is sufficient to achieve this.

Translating structured PASCAL-S to PROLOG

Because of the fundamental differences in data and control structures between the two languages concerned, the translation process is not straightforward. Some critical problems are discussed briefly in this section. For further details, see Williams and Chen[13].

The variables in these two languages are conceptually different. A PASCAL variable is a token of a piece of von Neumann storage which can be accessed and updated in the process of state transformations. By contrast a PROLOG variable stands for a single object and, once instantiated with a value, may not be altered, unless by backtracking the system returns to the point where the variable is instantiated and undoes the instantiation. Based on this conceptual difference the translation process creates a sequence of PROLOG variables to stand for a PASCAL variable at various points in a computation. Denotationally, whenever a PASCAL variable is updated, the translation process will create a new PROLOG variable to stand for this new value in the subsequent execution and abandon the old one.

The mapping of a single PASCAL variable into different PROLOG variables at different stages of state transformation is direct and has no side effect. However, since two distinct PASCAL variables may refer to the same location, referred to as variable aliasing, it is necessary to split the translation into two mappings, one from the domain of variables to that of locations and the other from locations to values. For example, when the same variable is used as actual parameter for two different formal variable parameters or when a variable used as an actual parameter is also referred to directly as a global variable within the procedure or function body. Thus, all access to or update of this kind of variable will be performed by two predicates to treat these special cases. An algorithm has been derived to detect possible variable aliasing at compile time.

A PASCAL block is translated as a rule to call each statement making up the block in turn. Each structured statement is translated as a rule which operates on a subset of the variables and yields another subset as the result, for example:

```
if a > b
  then if a > c
    then max:=a
    else max:=c
  else if b > c
    then max:=b
    else max:=c
```

the rule corresponding to this **if** statement will operate on the input set $\{a, b, c\}$ and output set $\{\max\}$. Since a block or structured statement may in turn contain other blocks or structured statements, such a translation can be performed recursively until simple statements such as assignment, procedure calls or input-output statements are encountered. These can be coded directly. The necessary set of parameters for each rule can be determined by constructing a data flow analysis framework at compile time.

In the case of array variables, there is no direct equivalent of arrays in PROLOG. To deal with these, two predicates called 'update' and 'access' were defined to modify or retrieve the relevant value from an array, and various different representations were experimented with. The most suitable was found to be a height-balanced binary tree. Thus an assignment in PASCAL can be translated as an instantiation of a PROLOG variable in the sequence corresponding to the destination of the assignment. The same is true for the values passed to the parameters of procedure or function calls.

The fundamental differences in the approach for handling arrays proved to be a particular stumbling block to efficiency in the PROLOG analogues of PASCAL programs. Even with a binary tree representation, updating and accessing an array element requires $o(\log_2 N)$ time which is very inefficient compared with the time to access an array element in PASCAL.

To assess the extent of this problem six simple examples were tested using the C-PROLOG system. Of these, three used an array as the main data structure. In each case variations of the program were produced using arrays of different sizes in the range 5 to 400 elements. The remaining three examples did not use arrays at all.

In Table 1 the execution times are compared by showing the ratio of the time taken to execute the PROLOG code which is generated by this approach (Code) to the time taken by the original PASCAL program running under the Unix PASCAL interpreter EM1 (PASC), and the ratio of Code to the time taken to execute an alternative PROLOG program which was handwritten to produce the same effect. The results show clearly that, although not the only

Table 1. Execution times

Example no.	Uses arrays	Code/PASC	Code/Prol
1	yes	10–25	11–19
2	yes	10–23	6–10
3	yes	8–18	3–5.5
4	no	0.9–1.6	1.5–1.9
5	no	1.4–2.4	1.3
6	no	1.3	1.4

source of inefficiency, the representation of array variables is a major cause of inefficiency.

For programs which do not involve arrays the execution times of the translated programs are much closer to the execution times of the PASCAL programs.

Adding an array facility to PROLOG

In the previous section the translation from PASCAL-S into PROLOG was seen to lead to very inefficient code when array variables are used in the PASCAL-S program. This was due to the fact that in PROLOG a list or binary tree structure was used to simulate an array, with the result that the time required to access an element depends either linearly or logarithmically on the size of the array concerned. This problem may be alleviated if an efficient array facility could be provided in PROLOG. To this end an extension of PROLOG has been implemented which provides such a facility and its performance has been evaluated.

Any array can be mapped onto a one dimensional vector with lower and upper bounds which are non-negative numbers. Three basic operations have been provided to manipulate arrays in the implementation. These are: *isarr*(Lower,Upper,Array) – which is used to create an array or to test if an object is unified with an array. When it is used to create an array, the first two arguments specify the lower and upper bounds of the array and the third argument stands for the array created. Second is *access*(Array,Index,Element), this selects an element of an array. Third is *update*(Old-Array,Index,NewValue,NewArray). This is used to modify an array element. A new array (fourth argument) is created from an existing array (first argument) in which the element at the specified index value is given the value 'New Value'.

While the operations isarr and access are relatively straightforward to implement, the operation update can be handled in a number of different ways. Two such ways which have been catered for, are that: it can create a new copy of the array, assigning the selected element to this new array and copying the remainder of the values from the old array (update by copying). Also, it can update the array *in situ*, assigning the value to the selected element directly (selective update).

The second interpretation defines a destructive operator which will increase the efficiency considerably. However, to maintain the applicative character of PROLOG, it is important to preserve one of its main features: *referential transparency*. That is, an element of an array can be updated directly only if it is not referred to in the subsequent execution. Otherwise the element must be updated by copying.

To decide when the more efficient selective update can be used, a data flow analysis framework for PROLOG programs has been implemented which will give the correlative referential states of the variables in each goal. Thus, for each array variable to be updated, the system can decide whether the update can be achieved destructively (i.e. if there is no further reference to this variable in the subsequent execution) or whether it must be done by copying.

The results of this investigation did not yield as significant an improvement as might have been expected, and are reported more fully in Chen and Williams' paper[14].

Parallel logic programming languages

Another area of interest is how this work may be extended to translate PASCAL programs into either Concurrent PROLOG[15] or PARLOG[16].

Concurrent PROLOG and PARLOG are logic programming languages similar to PROLOG except that they have the ability to execute rules and goals in parallel. Both languages provide 'and-parallelism' (to execute goals in parallel rather than in the left to right order in sequential PROLOG) and 'or-parallelism' (to execute alternative clauses in parallel rather than the top-to-bottom order of PROLOG). They incorporate features such as guards, stream communication via shared variables and committed choice non-determinism as the 'and-parallel' component. PARLOG uses *mode declarations* and *producer* and *consumer* annotations as the basic synchronization mechanism to achieve 'and- parallelism', whereas Concurrent PROLOG uses the read-only annotation to achieve this.

By performing a data flow analysis on a PASCAL program, it is possible to decide which parts of a program represented by different PROLOG goals can be executed in parallel. For example, in the case of translation to PARLOG, when a PASCAL program is analysed and each statement is represented as a PROLOG rule, it is natural to derive the mode declarations for this rule and to construct two sets of PROLOG variables corresponding to the set of PASCAL variables consumed in the goal and the set produced in the goal. If goal *B* follows goal *A* in the equivalent PROLOG program and the intersection of the variable set produced by *A* and that consumed by *B* is empty, goals *A* and *B* can be executed in parallel.

One difficulty lies in the maximum parallelism achievable for an iterative statement by using 'divide and conquer' techniques. For example, the algorithm to calculate the inner product of two arrays (using a **for** statement) can be understood as halving the arrays each time, determining the inner products for each half and then adding them together. Thus the **for** statement can be translated into two calls to the same PROLOG rule with different parameters and executed in parallel.

Thus with an extension to the method discussed previously, the system might generate code in Concurrent PROLOG or PARLOG which can be executed in parallel.

Conclusion

The study described in this paper has shown how a program in the conventional programming language PASCAL can be restructured and translanted to execute on a PROLOG architecture. The translantion discussed herein presented the following results.

First, a program in a conventional programming language, a realistic subset in the form of PASCAL-S, was translated as a set of PROLOG rules. For any PASCAL-S program, a PROLOG program was obtained by applying the translation algorithm to the PASCAL-S source program. From this the extension to full PASCAL has been looked at and no significant problems encountered other than ones of efficiency.

Second, the syntax-directed translation used associates each PASCAL BNF syntax definition with a Horn clause representation in a way which is natural both in respect of the translation process and the PROLOG representation of PASCAL programs.

Third, the translation algorithm produced a representation which takes account both of the basic program constructs and of conventional variables. The basic principles of the translation algorithm are suited to other conventional programming languages as well.

Fourth, the PASCAL source program was translated into a group of PROLOG procedures in which the interfaces between individual procedures relied solely on variables as parameters. This approach changes the dominance of control flow in the source program into that of data flow in the object program.

Several problems emerged from this study, the most serious among these being the efficiency of the object code. The object PROLOG program can run very efficiently in many cases, but in some applications, the speed of the translated code was drastically reduced. It is expected that this might be improved in two respects. From the translation, it is expected that a refined flow analysis of the data flow and control flow of the source program might be required not only to obtain information on the logical relationship between program objects, but also to explore the additional information needed to help the translation process to analyse, reorganize and optimize these relations in order to produce more efficient code. On the other hand, further study of logic programming may be required to improve the performance in general.

Although the study of this translation process is primarily concerned with the problem of how a PASCAL program can be handled on a PROLOG architecture, a secondary interest lies in how PROLOG itself can be further developed to achieve its goal, as a high level system programming language and as a kernel language for the fifth generation computer systems. The study revealed some problems in this respect.

First, a powerful conventional structured data type, the array data type, cannot be handled efficiently in PROLOG through the mechanism of system predicates. Although the result proved that the introduction of an array facility cannot match the efficiency of a list in PROLOG without a fundamental change to the PROLOG interpreter, it revealed that in handling problems which depend on arrays (editing, searching, matrix multiplication), the performance of PROLOG is drastically inferior to that of conventional programming languages. PROLOG has already shown merit in many areas such as artificial intelligence, databases, natural language processing and knowledge representation, but it is still questionable whether it can achieve as much in numerical computation and system programming. Many proposals have been made to adopt PROLOG as a high level general purpose system programming language; however, the problems with the data structures available in PROLOG severely limit the effectiveness of PROLOG in achieving this purpose.

Second, the system overhead to achieve nondeterminism and backtracking is very high, although it is not always necessary for certain applications. Most conventional computation algorithms do not require these features when implemented in PROLOG even when they can be efficiently implemented in PROLOG. For instance, most implementations of algorithms tested in this work such as searching, sorting and editing do not make use of the features of nondeterminism or backtracking; the same must be true of a wide range of numerical computation algorithms. This overhead greatly reduces the efficiency of the language when these classes of applications are required. Some improvements which could be made to assist the PROLOG system in reducing this overhead when the features of nondeterminism or backtracking are not required includes more general exploitation of mode declaration. This remains a significant problem for the language designer to overcome.

References

1 **Clocksin, W F and Mellish, C S** *Programming in PROLOG* Springer-Verlag (1981)

2 **Roussel, P** *PROLOG: Manual de reference et d'utilisation* Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy France (1975)

3 **Pereira, F C N and Warren, D H D** 'Definite clause grammars for language analysis – a survey of the formalism and a comparison with argumented transl-

ation networks' *Artif. Intell.* Vol 13 No 3 (May 1980) pp 231–278

4 **Clark, K L and McCabe, F G** PROLOG: A language for implementing expert systems' *Research report* Dept of Computing, Imperial College of Science and Technology London UK (November 1980)

5 **Neves, J C, Anderson, S O and Williams M H** 'A PROLOG implementation of query-by-example' *Proc. 7th Int. Comput. Symp.* (**Eds Schneider, H J and Teubner, B G**) (1983) pp 318–332

6 **Camacho-Gonzalez, J, Williams, M H and Aitchison, I E** 'Evaluation of the effectiveness of PROLOG for a CAD application' *IEEE Comput. Graph. Applic.* Vol 4 No 3 (March 1984) pp 67–75

7 **Moto-oka, T** Challenge for knowledge information processing systems' *Fifth Generation Computer Systems* pp 3–89 North-Holland, Amsterdam, The Netherlands (1982)

8 **Wirth, N** The programming language PASCAL' *Acta Inf.* Vol 1 No 1 (1971) pp 35–63

9 Specification for computer programming language PASCAL *BS 6192:1982* British Standards Institute (1982)

10 **Williams, M H and Chen, G** 'Restructuring PASCAL programs containing GOTO statements' *Comput. J.* Vol 28 No 2 (1985) pp 134–137

11 **Wirth, N** 'PASCAL-S: A subset and its implementation' (**Ed Barron, D W**) pp 199–259 Wiley-Interscience Publication (1981)

12 **Jensen, K and Wirth, N** *PASCAL user's manual and report* Springer-Verlag New York (1975)

13 **Williams, M H and Chen, G** 'Translating PASCAL for execution on a PROLOG-based system' *Comput. J.* Vol 29 No 3 (1986) pp 246–252

14 **Chen, G and Williams, M H** 'The value of an array facility in PROLOG' *Inf. Process. Lett.* Vol 23 No 5 (November 1986) pp 247–251

15 **Shapiro, E** 'A subset of concurrent PROLOG and its interpreter' *Report CS83-06* Dept Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel (February 1983)

16 **Clark, K L and Gregory, S** PARLOG: Parallel programming in logic' *Research report DOC84/4* Dept of Computing, Imperial College of Science and Technology London UK (April 1984) □

13. G.CHEN and M.H.WILLIAMS, The value of an array facility in Prolog, Information Processing Letters, vol. 23, no. 5, 247-251 (1986).

Information Processing Letters 23 (1986) 247–251
North-Holland

24 November 1986

THE VALUE OF AN ARRAY FACILITY IN PROLOG

G. CHEN and M.H. WILLIAMS

Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ, United Kingdom

Communicated by W.L. Van der Poel
Received 21 November 1985

Keywords: Prolog, array, evaluation

1. Introduction

In 1980, the Japanese embarked on a national programme aimed at developing fifth generation computer systems based on logic programming languages [2]. The prospect of such a development raises the question as to what will happen to imperative programming languages on such systems. A study into the feasibility of translating Pascal programs into Prolog for execution on a Prolog-based machine has demonstrated how this problem may be overcome [3].

However, the study also revealed that whereas some of the programs translated from Pascal to Prolog showed little or no degradation in performance, others ran markedly slower. The differences appear to be due to differences in the data and control structures between the two languages. It was observed that programs which were affected most, all made significant use of arrays. Since Prolog does not support the data type array, the effect of one had been achieved through the use of compound logical terms such as lists and binary trees. This appeared to be a significant disadvantage compared with virtually all conventional imperative languages. Consequently, it was felt that the introduction of an array data type into Prolog as a basic data structure of the language might improve the performance of programs translated from Pascal while at the same time enhance the facilities of the language in general. In particular, such a facility might be valuable for those applications which make heavy use of random access to the elements of a large compound data structure (e.g., an editor).

In view of this it was decided to add a facility to handle arrays to the C-Prolog interpreter in order to assess its effect on the performance of these examples. As far as the user is concerned this entails the definition of three basic operations, viz. creation of an array, access to an array element, and update of an array element. This is similar to the interface used by Eriksson and Rayner [1] in their implementation of arrays in LM-Prolog. The essential difference between the two systems lies in the way in which the update operation is handled. In the LM-Prolog version, a complex implementation is used to maintain the purity of the applicative features of Prolog at runtime. The system described here uses two different update operations and the applicative semantics of Prolog is preserved by performing a flow analysis of the Prolog program to determine which operations should be used at which points.

It was expected that with the optimization of the use of destructive update operations, the performance of the resulting programs in general should be improved. However, the results were unexpectedly disappointing.

2. Implementing arrays in Prolog

An array is defined as a sequence of objects with a lower and upper bound. Each element is

0020-0190/86/$3.50 © 1986, Elsevier Science Publishers B.V. (North-Holland)

directly accessible with a unique index within the range of the two bounds. The three basic operations which are required are: to create an array, to access an array element, and to update an element of an array. These three operations are catered for by three predicates, viz.:

(1) mk_arr(Lower, Upper, Array) .

The predicate "mk_arr" is used to create an array. The first two arguments specify the lower and upper bounds of the array while the third denotes the array itself. Thus, if the two bounds are instantiated when the predicate is called and the third argument is uninstantiated, it creates an array with N elements and unifies it with the third argument. Alternatively, if the array is given, it unifies the lower and upper bounds with the first two arguments. The predicate fails if any of the instantiations or unifications fails.

(2) access(Array, Index, Element) .

An element of an array can be accessed via the predicate "access". This predicate requires three arguments, an array, an index, and the element in the array specified by the index. The array and the index must both be instantiated when this predicate is called. In turn, it unifies the third argument with the specified element in the array. The predicate will fail if this unification fails or the given index is out of range.

(3) update(OldArray, Index, NewValue, NewArray) .

An array element can be modified by using the predicate "update". The first two arguments of this predicate must be instantiated when the predicate is called. A new array is then created and unified with the fourth argument "NewArray". The element of the new array with the index specified by "Index" will be unified with "NewValue" and the remaining elements will be copied from the old array. The "NewValue" can be any object, from an uninstantiated variable to any other data structure including another array. The predicate will fail if the instantiation required fails or the index is out of range.

In C-Prolog, these three operations have been added as system predicates. An array is represented as a record consisting of a lower bound, an upper bound, and a list of pointers initialised with nil. When mk_arr is encountered, if the third argument is uninstantiated, a function is activated to allocate N nil pointers and to create the record; if the third argument is already instantiated with an array, it unifies the bounds with the first two arguments.

The function corresponding to the predicate "access" simply checks the instantiation state and unifies the required element with the third argument; alternatively, if the unification fails or the index is out of range, the function as a whole fails.

Two alternative forms of update operation have been provided. The first (update by copy) simply creates a new array with the same lower and upper bounds, unifies the element indicated by "Index" with the value provided, and copies the remaining values from the old array. Naturally, this is very inefficient, especially for large arrays. The second form (destructive update) is an in situ update similar to that used in conventional imperative languages. This can be used whenever it is known that the old array will not need to be referred to again.

After a destructive update has been performed on an array variable, backtracking over the update operator cannot undo the effect of the update and recover the old value, since this would not be consistent with the meaning of destructive update. It is therefore assumed that each destructive update is implicitly followed by a cut symbol.

3. Automatic determination of type of update

In order to assist the user, a flow analysis can be used to determine which update operations can safely be handled by destructive update. This avoids the necessity for the user to make this distinction, thereby improving the efficiency without detracting from the applicative approach.

The main objective of this flow analysis is to determine the variable sharing state of a program. An instruction which updates an array variable to produce a new array can definitely be performed destructively when there are no further references

to the old array, nor to any arrays or array elements which share values with that array during the subsequent execution. In other words, an array variable can be updated destructively if it is not explicitly or implicitly referred to after the update is performed.

In order to analyse a Prolog program, the correlations of the variables in the program must be determined. Two variables can be sharing values when they have the same name, when they are unified with each other explicitly by an operator such as " = ", or when they are both unified with the same object.

An informal description of the flow analysis is as follows. Each clause in which an "update" predicate is called can be assumed to have the following general form:

p ← q, update(A, _, _, _), r.

For each such clause, part r is checked first. If there is any variable in part r which shares values with A, the analysis will fail, indicating that this update must not be carried out destructively. If none of the variables in r shares values with A, the arguments of the head of the clause p must be checked for any which may share values with A and any such argument positions are marked. If no such argument positions can be found (A is local to the predicate P), then the update can be performed destructively. Otherwise, for each such argument position the analysis extends to all clauses in which p is called. For a clause of the form

t ← a, p', b,

in which p' is a call to the predicate p, the same analysis is conducted recursively for each variable in a marked argument position. Each test may produce one of the following outcomes:

(a) The test fails if array A is further referred to in one of the ancestral goals of p after A is updated.

(b) It succeeds when the variable concerned is local to a predicate.

(c) The recursion reaches the top-most goal and the variable concerned is not local to it so that the result is dependent on the form of the call to the top goal. In this case, the result is undecided.

If, at any stage during the analysis, a test fails, the destructive update cannot be used and the analysis can be abandoned. However, if all tests succeed, a destructive update can be used. If one of the tests is undecided, it will be treated in the same way as a test failure.

To avoid repeated analysis of the same clause (caused by recursive or mutually recursive definitions), a stack may be used to keep a record of the analysis process so that each such position in a clause is searched at most once. This also assists termination of the analysis process.

This flow analysis has been implemented in Prolog and is used as a preprocessor to the Prolog interpreter.

4. Evaluation of array data structure

A number of examples were selected to assess the performance of the array data structure. In general, those chosen should have been suited to the use of arrays. In each case, comparisons were made between implementations using arrays and equivalent ones using lists. Four of these examples will be considered here.

The first example is a simple version of a line editor which provides some simple functions such as reading, writing, printing lines, inserting lines, and deleting lines. Table 1 shows the execution times of two programs, one based on the use of a list to hold the file being processed, the other on an array.

The editor problem should be more suited to the use of an array since every access to a line is random. If a list is used to store the processing file and each line of the file is stored as an element in the list (i.e., the processing file is represented as a list of lists), then in order to access the N*th* line, the list needs to be searched recursively N times. However, the average execution times (in milliseconds) given in Table 1 show that with the exception of the print command all the commands require greater execution time when an array is used.

The second example is one of multiplication of

Table 1
Execution times for editor program (TEST FILE: 2000 lines, 12000 chars)

Operation	Using list	Using array
READ 12000 chars	30750	34000
PRINT the first, middle and last line	7533	700
DELETE the first, middle and last line	6383	14616
READ and WRITE 2000 lines, 12000 chars	52083	55816
INSERT after first, middle and last line	11800	13683

two matrices A and B, each with N * N elements, to produce a third matrix C. When lists are used, each row is represented as a list and once again the matrix is treated as a list of lists. The Prolog program using lists transposes the second matrix and updates C[i, j] by the inner product of the *ith* element of list A with the *jth* element of the transposed of B. The Prolog program using the array representation closely follows the conventional algorithm to generate matrix C. Table 2 shows the statistics of execution time when N is 10, 20, and 30, respectively.

The third example considered here is that of insertion sort. This example is not particularly suited to the use of an array since both implementations need to search for a position in the list (or array) where an element should be inserted. Table 3 shows that the time in executing the program using an array is greater than using the list.

In the light of these statistics a more detailed analysis was performed to establish the cause of the inefficiency. This revealed that the main source of inefficiency is not the mechanism used in the implementation of the array, but the different control structures required by the object program when different data structures are used to solve the same problems.

Table 4 gives a breakdown of the execution

Table 2
Matrix multiplication

N	matrix_list	matrix_array
10	4750	10116
20	31033	69700
30	98233	227033

Table 3
Insert sort

N	sort_list	sort_array
100	5330	17366
200	17433	59183
300	39733	138716
400	71216	242450

times for the matrix multiplication program discussed in Table 2. In this table, the execution time is divided into three parts, the 'user process' time (i.e., the time spent executing the object program clauses), the 'function entry' time (i.e., the time spent executing function calls), and the 'function body' time.

The first of these is obtained by removing all array predicates "access" and "update" (the time for creating arrays is not included in the execution time). Thus, unifying a variable with an element of the array will be replaced by unifying it with a constant number, and updating an element of the array will not be carried out. The execution time thus obtained reflects the 'user process' or control time.

In order to obtain the function entry time, two new predicates are created corresponding to "access" and "update" whose function bodies are empty. Once again a constant number is used as the value of the array element obtained by calling "access". The difference between the execution time of this program and the user process time is the function entry time and the difference between the execution time of the full program and that obtained using empty functions is the real time spent in the functions of "access" and "update" predicates for suitably chosen sets of data.

The statistics in Table 4 clearly show that more than 80 per cent of the execution time is spent in

Table 4
Matrix multiplication

	N = 10	N = 20
% of time in user process	81.06	88.95
% of time entering function	10.38	5.47
% of time executing function	8.56	5.58

Table 5
Binary search

N	Sequential search (using list)	Binary search (using list)	Binary search (using array)
1000	4483	7466	2183
2000	8766	14783	4166
3000	13133	21966	6050
4000	17416	30050	8266
5000	23950	38216	9950

the user process. In absolute terms, this is more than the total time required for the version which uses lists to solve the same problem. Other examples investigated yield much the same results as presented here. For the purposes of these statistics, all updates were treated as destructive update thereby producing the lowest execution times.

Finally, the only examples which did produce a considerable improvement in performance when using the array data structure were certain classes of search problems. The results for a simple binary search are shown in Table 5. Although the binary search technique is not well suited to lists and execution times for simple sequential search are better than those for binary search in this case, the execution times obtained using arrays are markedly better than either of the values obtained using lists.

5. Conclusion

In most conventional programming languages, the data type array is one of the most efficient data structures. However, the introduction of arrays to Prolog does not give the same benefit to the language in general. The reasons for this may be as follows:

(1) An algorithm using arrays as the main data structures often has a very different control framework from one using lists, that is to say, the control mechanism is dependent upon the data structures it employs. The test algorithms designed above reflect this tendency clearly. When array data types are used, the resulting algorithms are more like conventional imperative programs than a set of logic relations.

(2) The language overhead in Prolog to support a data type like array is significant. This is due partly to the overhead of system predicates and partly to the fact that the system has to deal with nondeterminacy and backtracking, which inevitably requires a substantial amount of time to maintain. In comparing lists with arrays, the main overhead incurred when using lists is the time spent in the unification of the head and tail, whereas in the case of arrays each call to the evaluable predicates "array", "access" and "update" incurs not only the overhead associated with a function call but also that due to the additional "and" operation which must be performed by the interpreter.

For these reasons, the efficiency of random accessing of an array element available in most conventional programming languages is reduced by the system overhead to support such a data facility and the different control structures required to use it in Prolog. On the other hand, it has been shown that compound logical terms are not adequate in some cases where the large number of random accesses to an element of a term is used. In general, this study has shown that an array facility in Prolog may be of less benefit than expected.

References

[1] L.H. Eriksson and M. Rayner, Incorporating mutable arrays into logic programming, Proc. 2nd Internat. Logic Programming Conf., Uppsala, Sweden (1984) 101–114.

[2] T. Moto-Oka, Challenge for knowledge information processing systems, in: T. Moto-Oka, ed., Fifth Generation Computer Systems (Proc. Internat. Conf. Tokyo, 19–22 October 1981) (North-Holland, Amsterdam, 1982) 3–89.

[3] M.H. Williams and G. Chen, Translating Pascal for execution on a Prolog-based system, The Computer J. 29 (3) (1986) 246–250.

14. M.H.WILLIAMS, A formal notation for specifying static semantic rules, Computer Languages, vol. 5, no. 1, 37-55 (1980).

Computer Languages, Vol. 5, pp. 37 to 55
© Pergamon Press Ltd 1980. Printed in Great Britain
0096-0551/80/0401-0037$02.00/0

A FORMAL NOTATION FOR SPECIFYING STATIC SEMANTIC RULES

M. H. WILLIAMS

Department of Computer Science and Applied Maths, Rhodes University, Grahamstown, South Africa

(*Received* 1 *August* 1978; *revision received* 20 *July* 1979)

Abstract—The syntax rules of a programming language are generally fairly well understood and may easily be expressed formally as a grammar using BNF. However, the static semantic rules are less well understood and considerably more difficult to express formally. All notations for expressing static semantic rules either formally or informally involve actions being associated with the productions of the grammar. This paper presents such a formal notation which has been applied to various languages and found to be very useful both for the designer and the compiler writer.

Static semantics Complete syntax SDTS Formal specification

INTRODUCTION

THE *COMPLETE SYNTAX* of a programming language consists of a set of rules which determine the valid strings of the language. These rules are of two types:

(a) those which describe the form of a string (e.g. block = *begin* declarations statements *end*) which will be referred to as *syntax rules* or *syntax form rules*, and
(b) those which are concerned with the interpretation of components of a string (e.g. identifiers used in a statement must be declared somewhere) which are referred to as *static semantic* or *syntax interpretation rules*.

The syntax rules of languages are fairly well understood and are usually expressed in Backus Naur Form, a notation which has the advantage of being both simple and widely accepted. Furthermore, methods exist for constructing syntax analysers from the BNF definition of a language. However, BNF does not cater for the static semantic features of a language.

A number of systems have been proposed for specifying static semantic rules in a formal or informal manner, e.g. canonic systems [1], two-level grammars [2], etc. But none of these systems have gained general acceptance and thus the static semantic rules for programming languages are usually expressed as a set of English sentences.

All systems thus far proposed for dealing with static semantic rules, either formally or informally, involve actions being associated with the productions. In the case of formal systems such as canonic systems the actions are centred on lists which are associated with the variables in the canons (equivalent of productions); in the case of two-level grammars the actions are built into the non-terminal names themselves as they are passed from one production to the next, and so on. The less formal systems take the form of syntax directed translation schemes or SDTSs (e.g. Aho and Ullman [3, 4]; Griffiths [5]) in which actions are written in a suitable notation and inserted at appropriate points in the set of productions. These systems are referred to here as "less formal" since the notation for writing the actions is often not defined formally and left to the user to introduce whatever forms he likes. Furthermore I have seen no publication in which the static semantic rules of a major language are completely specified in such a notation.

This paper describes another SDTS notation but in this case one which has been defined formally and which has been used to specify the static semantic rules of several languages [6, 7].

THE EXTENDED BNF NOTATION

Although several extensions to the basic BNF notation (as used to define the syntax of ALGOL 60 [8]) have been put forward to simplify the task of expressing syntax form, only the basic BNF notation will be used here for syntax rules, i.e. metasymbols ⟨ and ⟩ are used to enclose metalinguistic variable names, :: = stands for "is defined as" and | is interpreted as "or" (see Appendix 1).

The static semantic rules can be specified as a series of actions to be taken at various points in the recognition of a program as a valid or invalid string of the grammar. These actions are enclosed in curly brackets; for example,

⟨actual parameter⟩:: = ⟨id⟩{a_1}|⟨arithmetic exp⟩{a_2}|⟨Boolean exp⟩

is interpreted as: if ⟨id⟩ is recognised then perform action a_1, if ⟨arithmetic exp⟩ is recognised then perform action a_2 and if ⟨Boolean exp⟩ then no action is required. If the same action applies to two or more successive alternatives on the right-hand side of a production (e.g. if a_1 and a_2 were identical), the action need only be specified once after the first alternative and the action *ditto* used in subsequent cases, e.g.

⟨actual parameter⟩:: = ⟨id⟩{a_1}|⟨arithmetic exp⟩{*ditto*}|⟨Boolean exp⟩

Before considering the actions which can be performed, consider the data structures used in the notation:

(a) Simple items—strings
  (i) the strings represented by metalinguistic variables in the BNF specification (e.g. ⟨array id⟩, ⟨label⟩, etc.);
  (ii) string constants represented by "string" (e.g. "FP");
  (iii) the constant 0 (which is a shorthand notation for the string "0");
  (iv) the null item λ.

(b) Complex items—stacks and variables
  (i) Tree-structured stacks. For the purposes of this notation a stack is regarded as a linked structure, whose elements consist of n-tuples of pointers, each of which may point either to a single item or to another linked stack (see Fig. 1). Insertion and deletion refer to the top element of a stack, although it is possible to access and alter an element in the middle of the stack. A stack name consists of a sequence of upper-case letters (e.g. FPS, DUMP, etc.).
  (ii) System variables. A system variable consists of a pointer which may point either to a simple item or to a stack entry (i.e. an n-tuple) or to a stack. System variable names consist of sequences of lower-case letters (e.g. *x*, *ijk*, etc.).
  (iii) Components of stack entries or system variables may be accessed individually, using the notation

Stack name·field name	—refers to the named field of the current entry of the stack in question
Stack name·field name (V)	—refers to the named field of the entry in the stack in question which is pointed to by variable V.
·field name (V)	—refers to the named field of an n-tuple stored in variable V.

Field names, stack names and system variable names must be unique. (Note that this replaces the form Comp*n* which was used in previous versions [6, 7] to reference the *n*th component of a stack entry.)

The actions which one may want to perform are:

(i) Push down an item i onto stack S: $i \downarrow$ S
(ii) Pop up stack S placing resulting item in i: $i \uparrow$ S
(iii) Pop up the whole stack S placing the result as a single item in i: $i \Uparrow$ S

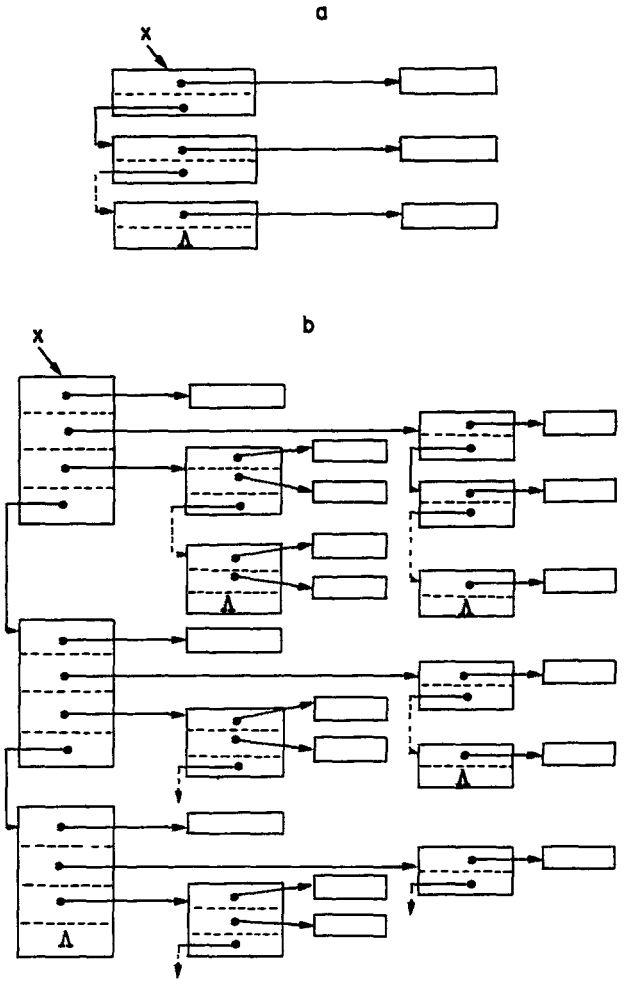
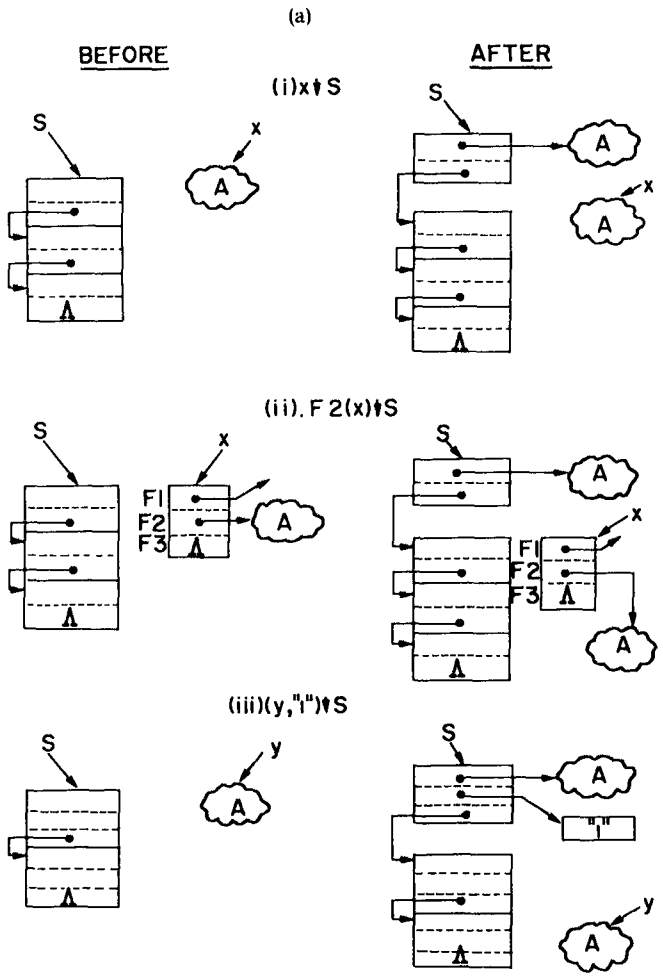


Fig. 1. (a) A stack X with three simple entries. (b) A stack X with n entries of form (Comp 1, Comp 2, Comp 3) where: Comp 1—simple entry; Comp 2—stack of simple entries; Comp 3—stack of entries of form (Component 1, Component 2) where both components are simple entries.

(iv) Add items in stack pointed to by i to those in stack S: $i \Downarrow S$
(v) Search a stack S for a particular item i: $\mathscr{S}$ (S, i, success action, fail action)
(vi) Alter a system variable or a component of a stack entry j: $i \rightarrow j$
(vii) Remove each item from a stack S in turn and perform some action a on it
$\forall S$: ($x \uparrow S$; a)
(viii) Conditional actions: *if* Boolean expression *then* action *fi*
or *if* Boolean expression *then* action *else* action *fi*
(ix) Signal an error (error number n): En (n an integer)
(x) Insert a call to a run-time routine into the object code generated: "r(p)" (r = routine name, p = parameters)
(xi) Sort stack S into ascending or descending order and place result in T: $\mathscr{A}$(S, T) or $\mathscr{D}$(S, T)
(xii) The null action: –
(xiii) Call an action rule (or action procedure): action name
(xiv) Loop action: *while* Boolean exp *do* action *od*
or *repeat* action *until* Boolean exp

Items which one may push down onto a stack include any simple item, system variable or component of a stack entry, or any compound object consisting of a set of such items. The action of pushing down an item onto a stack causes a copy of the item to be


(a)
BEFORE
AFTER
(i)x↑S
S
x
A
(ii). F 2(x)↑S
F1
F2
F3
(iii)(y,"1")↑S
y
"1"

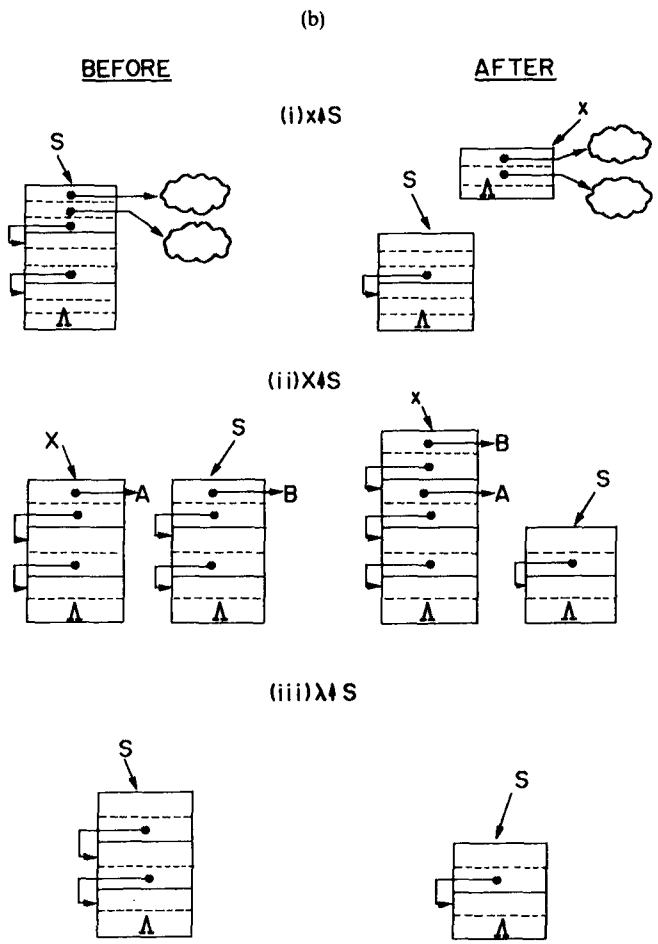

(b)
BEFORE
AFTER
(i)x↑S
S
x
(ii)X↑S
X
A
B
(iii)λ↑S

Fig. 2(a) and (b).

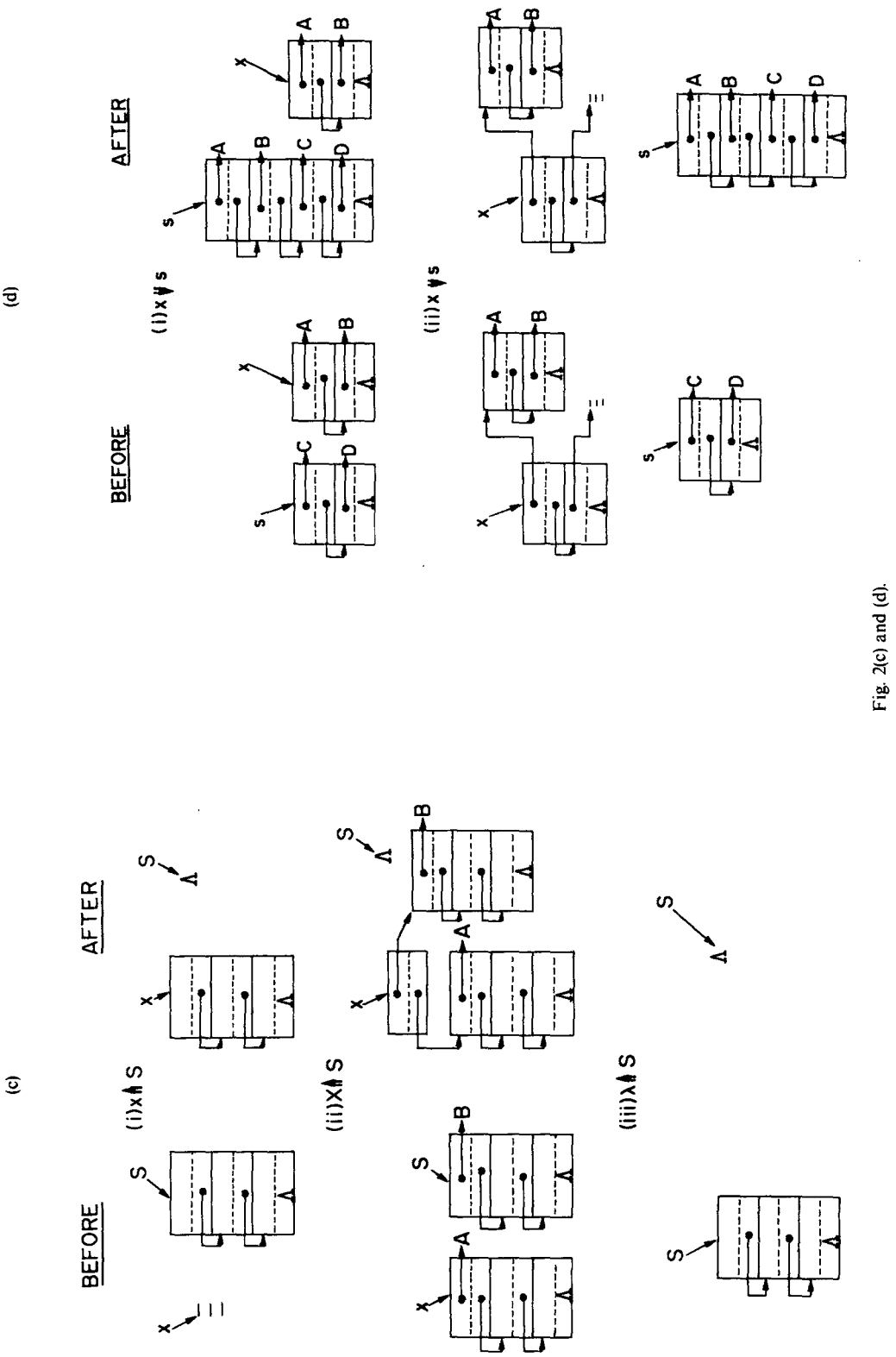


Fig. 2(c) and (d).

(e)

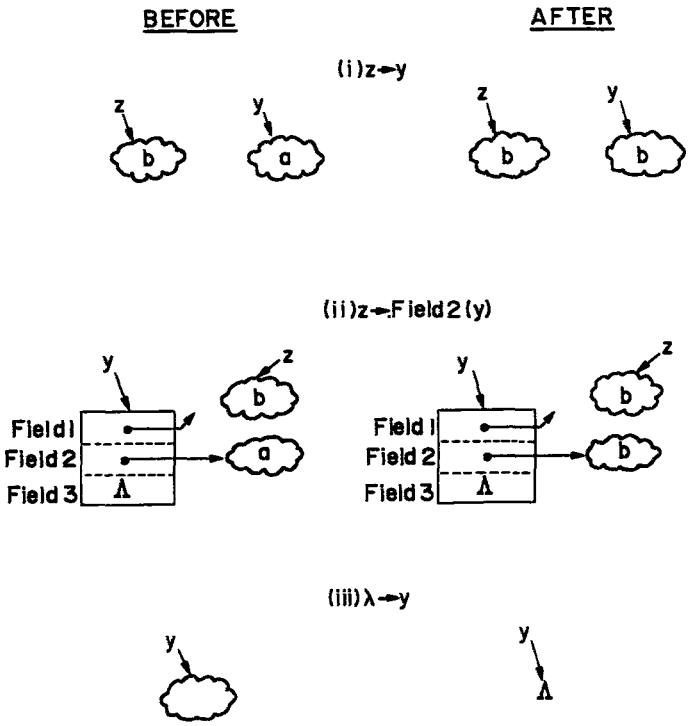


Fig. 2. (a) The stack operation ($i \downarrow$ S). (b) The unstack operation ($i \uparrow$ S). (c) Whole stack transfer operation ($i \Uparrow$ S). (d) Whole stack return operation ($i \Downarrow$ S). (e) The alter operation ($i \rightarrow j$).

constructed so that the original item is left unaffected. The effect of this operation is illustrated in Fig. 2(a).

If a stack S is popped up, the item removed may be allocated to a system variable (e.g. $x \uparrow$ S) in which case the previous contents of the system variable will be replaced by a pointer to this new item, or the item may be discarded (denoted by $\lambda \uparrow$ S), or it may be pushed down onto another stack (e.g. X $\uparrow$ S) (Fig. 2b).

The operation of popping up a whole stack can be used to transfer the contents of a stack to a system variable (e.g. $x \Uparrow$ S) or to a component of a system variable or stack entry (e.g. ST.Type(x) $\Uparrow$ S) or to another stack (e.g. X $\Uparrow$ S places the contents of S as a single item on stack X). Alternatively, this process may simply be used to clear a stack (e.g. $\lambda \Uparrow$ S) (Fig. 2c). The reverse operation takes a stack of items from a system variable (e.g. $x \Downarrow$ S) or from a component (e.g. ST.Type(x) $\Downarrow$ S)) or from the top of a stack (e.g. X $\Downarrow$ S—assuming that the top element of stack X is in turn a stack) and adds them to the stack as a number of separate entries (Fig. 2d).

The search operation $\mathscr{S}$(S, i, SA, FA) searches stack S from the top down for an entry whose first component is i. If an entry is encountered which is in turn a stack, it searches all items in this sub-stack for an entry with i as first component. If it succeeds in finding such an item, it performs action SA (success action) otherwise action FA (fail action). These actions may be any of the fourteen basic actions listed above. The notation

Stack name·field name

refers to the named field of the item most recently found in a search operation or added to the stack in a push-down operation.

The operation of altering a system variable or a component of a system variable or stack entry, denoted by $i \rightarrow j$, is illustrated in Fig. 2(e). i may be any simple or compound object or a string expression (two or more strings joined by the concatenation

operator$_+$); j may be a simple variable or a metalinguistic variable (used for altering the string being analysed; for example, in stripping comments from program text).

If one wishes to perform some operation on each item from a stack in turn, the form

$\forall$⟨stack⟩: (⟨unstack operation⟩; ⟨action⟩)

is used. For example,

$\forall$L: ($x \uparrow$ L; $\mathscr{S}$(ST, x, $-$, $x \downarrow$ ST)) adds items of L not already present in ST, to ST;

or

$\forall$L: ($\lambda \uparrow$ L; $-$) is equivalent to $\lambda \Uparrow$ L

Conditional actions have the form:

if ⟨Boolean expression⟩ *then* ⟨action⟩ *fi*

or

if ⟨Boolean expression⟩ *then* ⟨action⟩ *else* ⟨action⟩ *fi*

A Boolean expression consists of one or more "Boolean secondaries" connected by the Boolean operators $\wedge$ or $\vee$. A Boolean secondary consists either of a Boolean expression in parentheses or of a relation of form

⟨simple object⟩ ⟨relational op⟩ ⟨simple object⟩

where ⟨relational op⟩ is usually $=$ or $\neq$ but may be $<$, $\leqslant$, $>$ or $\geqslant$ if an ordering relation is defined for the simple objects. This may be followed by zero or more terms of the form

and ⟨simple object⟩ or *or*⟨simple object⟩

where simple object is any simple item, system variable or field, and

$x = y$ *and* z

is interpreted as

$x = y \wedge x = z$.

Similarly *or* is a shorthand notation for $\vee$.

In a language such as Pascal there is some standardisation even at the level of the error messages which the compiler produces. Thus it is desirable that error conditions should, where possible, be integrated into the specification of the syntax. Hence an action such as

E12

signals an error corresponding to error message 12.

The two sort operations, $\mathscr{A}$(S, T) which sorts elements of stack S and places them in ascending order in stack T, and $\mathscr{D}$(S, T) which sorts them into descending order in T, both sort on the first component of each element.

The facility to insert a call to a run-time routine into the generated object code of the program is essential for dynamic type checking. A run-time routine name always consists of an upper-case letter followed by a sequence of lower-case letters. Parameters may be any simple objects or stack names. As with pushing down an item onto a stack, a copy of each parameter is constructed so that the original items are left unaffected.

Three functions which each take as argument a stack name, are:

$\mathscr{N}$(S) which produces as value a string representing the number of items in stack S. If stack S is empty, the value produced is 0.
$\mathscr{T}$(S) which produces as value a copy of the top item on stack S.
$\mathscr{P}$(S) which produces as value a pointer to the current item in the stack.

All three functions may be used as simple objects.

The two loop actions have the form:

while Boolean exp *do* action *od*

or

repeat action *until* Boolean exp

In general, in place of a single action one may have a sequence of actions separated by

semicolons, i.e.

$a_1; a_2; \ldots; a_n$

For example

$\forall$L:(x ↑ L; $\mathscr{S}$(ST, x, *if* ST·Type = 0 *then* "I" → ST·Type; x ↓ DUMP *fi*, E22); x ↓ S)

The complete syntax of this notation is given in Appendix 1. A few static semantic restrictions have been included, namely:

(i) that a metalinguistic variable appearing on the right-hand side of a production must also appear on the left-hand side of some production (stacks MVS and MVDS are used for this);
(ii) that a metalinguistic variable appearing in an action sequence must also appear in the metalinguistic expression preceding that action sequence (stack TEMP).

APPLICATION OF EXTENDED BNF

In this section the way in which the Extended BNF notation can be used, is illustrated by means of an example. The context-free grammar defined in Fig. 3 can be used to generate programs such as that shown in Fig. 4. However, in order for these programs to be meaningful certain static semantic restrictions must be imposed (such as requiring that all identifiers used in a program are declared, etc.). The set of static semantic rules which will be considered, are given in Fig. 5.

Top-down or bottom-up approach

The specification of static semantic rules may be approached from two different points of view—that of the compiler writer and that of the user. The type of specification suitable for the compiler writer is the bottom-up type which reflects the way in which a compiler may work, while the user may prefer a top-down type of specification.

To illustrate the difference between these two approaches, consider the rule which states that the type of each non-label identifier appearing in a statement sequence must be declared at the head of the block or at the head of an enclosing block (Rule 2 in Fig. 5). In the bottom-up approach, this check is performed at the lowest level. Thus one might define

⟨var⟩:: = ⟨id⟩{$\mathscr{S}$(ST,⟨id⟩, *if* ST·Type ≠ "I" *and* "FP" *then* E10 *fi*, E5)}

which is interpreted as: if some string which satisfies the rules for ⟨id⟩ is to be taken as satisfying the rules for ⟨var⟩ then search the stack ST (i.e. Symbol Table) for an item with this string as its first component; if such an item is found, check that the field "Type" is either "I" (i.e. an integer) or "FP" (i.e. a formal parameter of undeclared type)—if not, flag error 10 (identifier has incorrect type). If such an item is not found in the list ST then flag error 5 (undeclared identifier). Similarly

⟨array id⟩:: = ⟨id⟩{$\mathscr{S}$(ST,⟨id⟩, *if* ST·Type ≠ "IA" *and* "FP" *then* E11 *fi*, E6)}

In the top-down approach, the check might be performed at a higher level. Thus:

⟨var⟩:: = ⟨id⟩{⟨var⟩ ↓ VS}
⟨array id⟩:: = ⟨id⟩{⟨array id⟩ ↓ AS}
⟨block⟩:: = ⟨begin symbol⟩⟨dec sequence⟩;⟨stm sequence⟩END
{$\forall$VS: (x ↑ VS; $\mathscr{S}$(ST, x, *if* ST·Type ≠ "I" *and* "FP" *then* E10 *fi*, E5));
$\forall$AS: (x ↑ AS; $\mathscr{S}$(ST, x, *if* ST·Type ≠ "IA" *and* "FP" *then* E11 *fi*, E6)); ...}

While the bottom-up approach is more practical from the point of view of implementation, one is faced with the problem of identifiers being used before they are declared (labels are a classic example of this). The top-down approach does not suffer from this problem since all checking is done at the end of each block.

If only one or two different types of identifier are affected then one could use stacks of forward references to solve this problem. Alternatively, one can divide the static semantic

```
<digit>::=1|2|3
<id>::=A|B|C|D|E|F|G|H
<label>::=<id>
<var>::=<id>
<array id>::=<id>
<proc id>::=<id>
<function id>::=<id>
<subscript list>::=<arith exp>|<subscript list>,<arith exp>
<actual parameter>::=<arith exp>|<id>
<actual parameter list>::=<actual parameter>|<actual parameter list>,
    <actual parameter>
<actual par part>::=<empty>|(<actual parameter list>)
<formal parameter list>::=<id>|<formal parameter list>,<id>
<formal par part>::=<empty>|(<formal parameter list>)
<subscripted var>::=<array id>[ <subscript list>]
<primary>::=<digit>|<var>|<subscripted var>|<function id><actual par part>
<arith exp>::=<primary>|<arith exp>+<primary>
<ass st>::=<subscripted var>:=<arith exp>|<id>:=<arith exp>
<goto st>::=GOTO<label>
<proc st>::=<proc id><actual par part>
<unlabelled stm>::=<ass st>|<goto st>|<proc st>|<block>
<stm>::=<unlabelled stm>|<label>:<stm>
<stm sequence>::=<stm>|<stm sequence>;<stm>
<identifier list>::=<id>|<identifier list>,<id>
<value part>::=<empty>|VALUE<identifier list>;
<specifier>::=INTEGER|INTEGER ARRAY|INTEGER PROCEDURE|PROCEDURE|LABEL
<formal par dec>::=<specifier><identifier list>;
<spec part>::=<empty>|<formal par dec>|<spec part><formal par dec>
<procedure name>::=<id>
<procedure heading>::=<procedure name><formal par part>;<value part><spec part>
<bound pair>::=<arith exp>:<arith exp>
<bounds>::=<bound pair>|<bounds>,<bound pair>
<type list>::=<id>|<type list>,<id>
<procedure type>::=INTEGER PROCEDURE|PROCEDURE
<dec>::=INTEGER<type list>|INTEGER ARRAY<type list><bounds>|
   <procedure type><procedure heading><stm>
<dec sequence>::=<dec>|<dec sequence>;<dec>
<begin symbol>::=BEGIN
<block>::=<begin symbol><dec sequence>;<stm sequence>END
<program>::=<block>
```

Fig. 3. Syntax (or syntax form) rules for a simple language.

specification into two parts corresponding to the two passes of a compiler. The first part (Pass I) scans the program, collecting all identifiers (variables, procedures, labels, formal parameters, etc.) *declared* in each block or procedure and places them on a stack in a suitable format; the second part (Pass II) checks each identifier *used* in each block and procedure. This approach will be followed for the example given.

```
BEGIN INTEGER PROCEDURE A(B,C,F);INTEGER B;INTEGER PROCEDURE C;A:=C(B,F);
PROCEDURE D(E,C,H);VALUE E; INTEGER E; LABEL C;
          BEGIN INTEGER F;F:=A(E,G,H)+B;  GOTO C END;
INTEGER PROCEDURE G(A,B); G:=A + B [ 1] ;
INTEGER B,E;
INTEGER ARRAY C [ 1:2] ;
B:=1;
F:B:=B+2;
D(B,F,C);
GO TO F
END
```

Fig. 4. Example of program generated from rules in Fig. 3.

Definition of Pass I

Consider the definition of Pass I for the case of the grammar in Fig. 3. It scans the program setting up for each block and procedure of the program a stack of entries of the form:

(Name, Type, Formaloractual, X, Y)

For example

(id, "I", "A", 0, 0)	if the id is an integer,
(id, "IA", "A", n, 0)	if the id is an integer array (n = string representing number of dimensions),
(id, "IP", "A", n, 0)	if the id is an integer procedure (n = string representing number of formal parameters),
(id, "P", "A", n, 0)	if the id is a procedure (n = string representing number of formal parameters),
(id, "L", "A", 0, 0)	if the id is a label,
(id, "I", "FP", x, y)	if the id is a formal parameter of type integer,
(id, "IA", "FP", x, y)	if the id is a formal parameter of type integer array,
(id, "IP", "FP", x, y)	if the id is a formal parameter of type integer procedure,
(id, "P", "FP", x, y)	if the id is a formal parameter of type procedure,
(id, "L", "FP", x, y)	if the id is a formal parameter of type label,
(id, 0, "FP", x, y)	if the id is a formal parameter of undeclared type,

where

x = string representing position in the formal parameter list at which the formal parameter occurs, and

y = name of the procedure or integer procedure.

One entry is set up for each symbol declared in the current block (or each formal parameter declared in the current procedure). These entries are placed on a stack called the ICBS (Identifiers in Current Block Stack). If an inner block is encountered, the contents of the ICBS are preserved until the inner block has been dealt with and then restored once again.

When dealing with an individual declaration a stack L is used to contain identifiers until the end of the declaration is reached, i.e.

⟨type list⟩::= ⟨id⟩{⟨id⟩↓ L}|⟨type list⟩, ⟨id⟩{*ditto*}

At the end of each declaration this stack is copied to the stack ICBS together with information on the number of dimensions (stack SL—subscript list) or formal parameters (stack FPS—formal parameter stack). Thus

⟨formal parameter list⟩::= ⟨id⟩{⟨id⟩ ↓ FPS}|⟨formal parameter list⟩, ⟨id⟩{*ditto*}
⟨bounds⟩::= ⟨bound pair⟩ {"1" ↓ SL}|⟨bounds⟩, ⟨bound pair⟩{*ditto*}.

In copying the identifiers to the ICBS a check is made against redeclaration of an identifier in the same block (Rule 1 in Fig. 5).

⟨dec⟩::= INTEGER ⟨type list⟩{∀L:(x ↑ L; $\mathscr{S}$(ICBS, x, E2, (x, "I", "A", 0, 0) ↓ ICBS))}|
INTEGER ARRAY ⟨type list⟩⟨bounds⟩{∀L:(x ↑ L;$\mathscr{S}$(ICBS, x, E2,
(x, "IA", "A", $\mathscr{N}$(SL), 0) ↓ ICBS)); A ⇑ SL}
|⟨procedure type⟩⟨procedure heading⟩⟨stm⟩
⟨procedure type⟩::= INTEGER PROCEDURE{"IP" ↓ T}|PROCEDURE{"P" ↓ T}
⟨procedure name⟩::= ⟨id⟩{$\mathscr{S}$(ICBS, ⟨id⟩, E2, ⟨id⟩ ↓ PNS)}
⟨procedure heading⟩::= ⟨procedure name⟩⟨formal par part⟩;⟨value part⟩⟨spec part⟩{x ↑ PNS; y ↑ T; (x, y, "A", $\mathscr{N}$(FPS), 0) ↓ ICBS;λ ⇑ FPS}
⟨stm⟩::= ⟨unlabelled stm⟩|⟨label⟩:⟨stm⟩{$\mathscr{S}$(ICBS, ⟨label⟩, E3, (⟨label⟩, "L", "A", 0, 0) ↓ ICBS)}

1. An identifier cannot be declared more than once in the same block (although it may be redeclared in an inner block).
2. The type of each non-label identifier (other than formal parameters) appearing in a statement sequence must be declared at the head of the block or at the head of an enclosing block.
3. The way in which an identifier is used must conform with its type as declared.
4. The scope of an identifier is the block in which it is declared and any inner blocks in which it is not redeclared. If an identifier is redeclared in an inner block, the new definition holds until the end of the inner block (and any enclosed blocks), after which the definition of the outer block holds once again. Thus it follows that an identifier cannot be used for two different purposes within the same block.
5. A non-label identifier may be used before it is declared if the identifier reference occurs in a function or procedure body and the identifier declaration occurs at some later stage in the same declaration sequence.
6. Every label referred to in a GOTO statement must be "declared" in the same or an enclosing block. Hence it is not possible to jump into a block.
7. The number of subscripts in the subscript list of a subscripted variable must correspond to the number of dimensions in the array declaration.
8. Every formal parameter in a formal parameter list must be different.
9. Every formal parameter declared in the specification part of a function or procedure declaration must appear in the corresponding formal parameter list.
10. Each parameter identifier may not appear more than once in the specification part of a function or procedure declaration.
11. Every formal parameter appearing in a VALUE clause must be different and must also appear in an INTEGER or INTEGER ARRAY declaration.
12. The number and type of actual parameters in the parameter list of a procedure or function call must correspond to the number and type of formal parameters in the procedure or function declaration.
13. The type of a formal parameter need not be declared in the specification part. In this case the type of the corresponding actual parameter must be checked wherever the formal parameter is used in the procedure or function body.
14. A function must contain at least one assignment statement with the function identifier on the left-hand side.

Fig. 5. Static semantic (or syntax interpretation) rules for grammar in Fig. 3.

In order to handle formal parameters, entries will be set up in the stack FPS with the same format as the ICBS. At the same time a check is made that every formal parameter identifier in a formal parameter list is different (Rule 8).

⟨formal parameter list⟩::= ⟨id⟩{$\mathscr{S}$(FPS, ⟨id⟩, E15, (⟨id⟩, 0, "FP", $\mathscr{N}$(FPS), $\mathscr{T}$(PNS)) ↓ FPS)}|⟨formal parameter list⟩, ⟨id⟩{*ditto*}.

Every parameter appearing in a VALUE clause must be different (Rule 12):

⟨identifier list⟩::= ⟨id⟩{⟨id⟩ ↓ L}|⟨identifier list⟩, ⟨id⟩{*ditto*}
⟨value part⟩::= ⟨empty⟩|VALUE⟨identifier list⟩;{∀L:(x ↑ L; $\mathscr{S}$(VL, x, E19, x ↓ VL))}

Every formal parameter type declared in the specification part of a procedure declaration must appear in the corresponding formal parameter list (Rule 9). Each parameter identifier may not appear more than once in the specification part (Rule 10). A formal parameter appearing in a VALUE clause must also appear in either an INTEGER declaration or an INTEGER ARRAY declaration in the specification part (Rule 11).

⟨specifier⟩::= INTEGER{"I" ↓ T}|INTEGER ARRAY{"IA" ↓ T}|
INTEGER PROCEDURE{"IP" ↓ T}|PROCEDURE{"P" ↓ T}|
LABEL{"L" ↓ T}
⟨formal par dec⟩::= ⟨specifier⟩⟨identifier list⟩;
{y ↑ T; ∀L: (x ↑ L; $\mathscr{S}$(FPS, x, *if* FPS·Ftype ≠ 0 *then* E1 *else* y → FPS·Ftype *fi*;
if y = "I" *or* "IA" *then* $\mathscr{S}$(VL, x, 0 → VL·Name, −) *fi*, E20))}
⟨procedure heading⟩::= ⟨procedure name⟩⟨formal par part⟩;⟨value part⟩⟨spec part⟩
{∀VL:(x ↑ VL; *if* x ≠ 0 *then* E21 *fi*); x ↑ PNS; y ↑ T; (x, y, "A", $\mathscr{N}$(FPS), 0)↓ ICBS;
x ⇑ FPS; x ⇓ ICBS}

Whenever an inner block or procedure is encountered, the contents of the ICBS must be preserved while the inner block is handled. The stack DUMP is used for this purpose. Two other stacks used are BS (Block Stack) which has entries of form (Blockno, Icbsdump)

where Blockno = string representing the number of the block or procedure
and Icbsdump = identifiers declared in this block or procedure (i.e. the ICBS for this block or procedure),

and IBS (Incomplete Block Stack) which has single entries containing the block number of each incomplete block or procedure, either on the DUMP or in the ICBS.

⟨begin symbol⟩::= BEGIN{DUMP ⇑ ICBS; $\mathscr{N}$(BS) ↓ IBS; ($\mathscr{N}$(BS, 0) ↓ BS}
⟨procedure heading⟩::= ⟨procedure name⟩⟨formal par part⟩;⟨value part⟩⟨spec part⟩
{∀VL:(x ↑ VL; *if* x ≠ 0 *then* E21 *fi*); x ↑ PNS; y ↑ T; (x, y, "A", $\mathscr{N}$(FPS), 0) ↓ ICBS;
DUMP ⇑ ICBS; $\mathscr{N}$(BS) ↓ IBS; ($\mathscr{N}$(BS), 0) ↓ BS; x ⇑ FPS; x ⇓ ICBS}
⟨block⟩::= ⟨begin symbol⟩⟨dec sequence⟩;⟨stm sequence⟩END
{x ↑ IBS; y ⇑ ICBS; $\mathscr{S}$(BS, x, y → BS·Icbsdump, −); DUMP ⇓ ICBS; λ ↑ DUMP}
⟨dec⟩::= .....|⟨procedure type⟩⟨procedure heading⟩⟨stm⟩
{x ↑ IBS; y ⇑ ICBS; $\mathscr{S}$(BS, x, y → BS·Icbsdump, −); DUMP ⇓ ICBS; λ ↑ DUMP}

On completing the scan of the whole program, all stacks will be empty except for the stack BS, which contains the identifiers declared in each block—but in the wrong order. Thus this stack is copied into another stack SS (symbol stack), reversing the order of the entries and removing the block number from each entry (as this is no longer required).

⟨program⟩::= ⟨block⟩{∀BS:(x ↑ BS;·Icbsdump(x) ↓ SS)}

The complete specification of Pass I is given in Appendix 2.

Definition of Pass II

For the remaining static semantic rules the following additional stacks will be used:

ST — symbol table: contains all symbols declared in the current block and all enclosing blocks;

APS — actual parameter stack: contains the type of each actual parameter;
FASL — function assignment statement list: keeps track of whether a function contains an assignment statement with the function identifier on the left-hand side;
RTFPS — run-time formal parameter stack: contains the declared type of each formal parameter of all functions for run-time checking;
RTAPS — run-time actual parameter stack: contains the type of each actual parameter for all function calls (nested) in effect.

The remaining static semantic rules are handled as follows:

(1) The type of each non-label identifier (other than formal parameters) must be declared before it is used (Rule 2). The way in which such an identifier is used must conform with its type (Rule 3). Thus each identifier that occurs in an assignment statement or procedure call must be checked to see that it has been declared and that its type is correct.

⟨var⟩:: = ⟨id⟩{$\mathscr{S}$(ST, ⟨id⟩, *if* ST·Type ≠ "I" and 0 then E10 *fi*, E5)}
⟨array id⟩:: = ⟨id⟩{$\mathscr{S}$(ST, ⟨id⟩, *if* ST·Type ≠ "IA" *and* 0 *then* E11 *fi*, E6)}
⟨proc id⟩:: = ⟨id⟩{$\mathscr{S}$(ST, ⟨id⟩, *if* ST·Type ≠ "P" *and* 0 *then* E12 *fi*, E7)}
⟨function id⟩:: = ⟨id⟩{$\mathscr{S}$(ST, ⟨id⟩, *if* ST·Type ≠ "IP" *and* 0 *then* E13 *fi*, E8)}
⟨ass st⟩:: = ⟨subscripted var⟩: = ⟨arith exp⟩|⟨id⟩: = ⟨arith exp⟩
{$\mathscr{S}$(ST, ⟨id⟩, *if* ST·Type ≠ "I" *and* 0 *and* "IP" *then* E10 *fi*, E5)}

(2) The scope of an identifier is the block in which it is declared and any inner blocks in which it is not redeclared (Rule 4). Similarly, the scope of a formal parameter is the procedure in whose formal parameter part it occurs. Thus one needs to enter all new declarations in the symbol table ST on entry to a block or procedure, and remove them on exit from the block or procedure. Since the search operation searches from the top down it will always locate the most recent definition of an identifier. Thus

⟨begin symbol⟩:: = BEGIN {ST ↑ SS}
⟨procedure name⟩:: = ⟨id⟩{ST ↑ SS}
⟨block⟩:: = ⟨begin symbol⟩⟨dec sequence⟩;⟨stm sequence⟩END{λ ↑ ST}
⟨dec⟩:: = ...|⟨procedure type⟩⟨procedure heading⟩⟨stm⟩{λ ↑ ST}

This also accounts for cases where a non-label identifier is used before it is declared (Rule 5).

(3) Every label referred to in a block must be "declared" in the same or an enclosing block (Rule 6).

⟨goto st⟩:: = GOTO⟨label⟩{$\mathscr{S}$(ST, ⟨label⟩, *if* ST·Type ≠ "L" *and* 0 *then* E9 *fi*, E4)}

(4) The number of subscripts in the subscript list of a subscripted variable must correspond to the number of dimensions in the array declaration (Rule 7).

⟨subscript list⟩:: = ⟨arith exp⟩{"1"↓ SL}|⟨subscript list⟩, ⟨arith exp⟩{*ditto*}
⟨array id⟩:: = ⟨id⟩{$\mathscr{S}$(ST, ⟨id⟩, *if* ST·Type ≠ "IA" *and* 0 *then* E11 *fi*, E6); DUMP ⇑ SL}
⟨subscripted var⟩:: = ⟨array id⟩[⟨subscript list⟩]
{$\mathscr{S}$(ST, ⟨array id⟩, *if* ST·Formaloractual = "A" ∧ ST·X ≠ $\mathscr{N}$(SL) *then* E16 *fi*, −);
λ ⇑ SL; DUMP ⇓ SL; λ ↑ DUMP}

The stack SL is preserved on the stack DUMP to take account of cases where subscripted variables are used in subscript expressions.

(5) The number and types of actual parameters in the parameter list of a procedure or function call must correspond to the number and types of formal parameters in the procedure or function declaration (Rule 12). However, since it is not always possible to check the types of actual parameters at compile time, three run-time routines (Rtcheck, Rtparcheck and Rtremove) are introduced and calls to these routines are inserted into the object code. Rtparcheck checks the type of actual parameters in an actual parameter stack (APS) against the declared formal parameter types in RTFPS, and constructs an

entry in RTAPS; Rtcheck checks that a specific formal parameter is replaced by an actual parameter with a particular type and number of dimensions (or formal parameters); and Rtremove removes the top entry of RTAPS.

⟨proc id⟩::= ⟨id⟩{$\mathscr{S}$(ST, ⟨id⟩, *if* ST·Type ≠ "P" *and* 0 *then* E12 *fi*, E7); DUMP ⇑ APS}
⟨function id⟩::= ⟨id⟩{$\mathscr{S}$(ST, ⟨id⟩, *if* ST·Type ≠ "IP" *and* 0 *then* E13 *fi*, E8); DUMP ⇑ APS}
⟨actual parameter⟩::= ⟨arith exp⟩{("I", 0, 0, 0) ↓ APS} |
⟨id⟩{$\mathscr{S}$(ST, ⟨id⟩, *if* ST·Type = "IP" *or* "P" ∧ ST·Formaloractual = "A" *then* (ST·Type, ST·X, ST·Y, ⟨id⟩) ↓ APS *else* (ST·Type, ST·X, ST·Y, 0) ↓ APS, E14)}
⟨primary⟩::= ⟨digit⟩|⟨var⟩|⟨subscripted var⟩|⟨function id⟩⟨actual par part⟩
{$\mathscr{S}$(ST, ⟨function id⟩, *if* ST·Formaloractual = "A" *then*
(*if* ST·X ≠ $\mathscr{N}$(APS) *then* E17 *fi*; 'Rtparcheck(⟨function id⟩, 0, APS)') *else*
('Rtcheck(ST·X, ST·Y, "IP", $\mathscr{N}$(APS))'; 'Rtparcheck(ST·X, ST·Y, APS)') *fi*, −);
λ ⇑ APS; DUMP ⇓ APS; λ ↑ DUMP}
⟨proc st⟩::= ⟨proc id⟩⟨actual par part⟩
{$\mathscr{S}$(ST, ⟨proc id⟩, *if* ST·Formaloractual = "A" *then* (*if* ST·X ≠ $\mathscr{N}$(APS) *then* E17 *fi*; 'Rtparcheck(⟨proc id⟩, 0, APS)') *else* ('Rtcheck(ST·X, ST·Y, "P", $\mathscr{N}$(APS))'; 'Rtparcheck(ST·X, ST·Y, APS)') *fi*, −); λ ⇑ APS; DUMP ⇓ APS; λ ↑ DUMP}
⟨dec⟩::= ....|⟨procedure type⟩⟨procedure heading⟩⟨stm⟩{λ ↑ ST;'Rtremove()'}

(6) The type of a formal parameter need not be declared in the specification part. In this case the type of the corresponding actual parameter must be checked at run-time wherever the formal parameter is used in the procedure or function body.

⟨go to st⟩::= GOTO⟨label⟩{$\mathscr{S}$(ST, ⟨label⟩, *if* ST·type = 0 *then* 'Rtcheck (ST·X, ST·Y, "L", 0)' *else if* ST·Type ≠ "L" *then* E9 *fi fi*, E4)}
⟨var⟩::= ⟨id⟩{$\mathscr{S}$(ST, ⟨id⟩, *if* ST·Type = 0 *then* 'Rtcheck (ST·X, ST·Y, "I", 0)' *else if* ST·Type ≠ "I" *then* E10 *fi fi*, E5)}
⟨subscripted var⟩::= ⟨array id⟩[⟨subscript list⟩]
{$\mathscr{S}$(ST, ⟨array id⟩, *if* ST·Formaloractual = "A" *then* (*if* ST·X ≠ $\mathscr{N}$(SL) *then* E16 *fi*) *else* 'Rtcheck(ST·X, ST·Y, "IA", $\mathscr{N}$(SL))' *fi*, −); λ ⇑ SL; DUMP ⇓ SL; λ ↑ DUMP}
⟨ass st⟩::= ⟨subscripted var⟩:= ⟨arith exp⟩|⟨id⟩:= ⟨arith exp⟩
{$\mathscr{S}$(ST, ⟨id⟩, *if* ST·Type = 0 *then* 'Rtcheck (ST·X, ST·Y, "I", 0)' *else if* ST·Type ≠ "I" *and* "IP" *then* E10 *fi fi*, E5)}

(7) A function body must contain at least one assignment statement with the function identifier on the left-hand side (Rule 14). An assignment statement with a function identifier on the left-hand side can only occur within the definition of that function.

⟨procedure heading⟩::= ⟨procedure name⟩⟨formal par part⟩;⟨value part⟩⟨spec part⟩
{$\mathscr{S}$(ST, ⟨procedure name⟩, *if* ST·Type = "IP" *then* (⟨procedure name⟩, 0) ↓ FASL *else* (⟨procedure name⟩, "I") ↓ FASL *fi*, −)}
⟨ass st⟩::= ⟨subscripted var⟩:= ⟨arith exp⟩|⟨id⟩:= ⟨arith exp⟩
{$\mathscr{S}$(ST, ⟨id⟩, *if* ST·Type = "IP" ∧ ST·Formaloractual = "A" *then*
$\mathscr{S}$(FASL, ⟨id⟩, "I" → FASL·Funtype, E18) *else if* ST·Type = 0 *then*
'Rtcheck (ST·X, ST·Y, "I", 0)' *else if* ST·Type ≠ "I" *then* E10 *fi fi fi*, E5}
⟨dec⟩::= ....|⟨procedure type⟩⟨procedure heading⟩⟨stm⟩{λ ↑ ST; 'Rtremove()';
x ↑ FASL; *if* ·Funtype (x) = 0 *then* E22 *fi*}

(8) Finally in order to handle run-time checks a run-time formal parameter stack, RTFPS, must be set up with entries of the form (Fnid, Fptype)
where

Fnid = function or procedure identifier,
Fptype = stack of entries containing type of each formal parameter.

This stack is retained during the execution phase of the program in order to check the actual parameter list in each function or procedure call.

⟨formal parameter list⟩:: = ⟨id⟩{$\mathcal{S}$(ST,⟨id⟩, ST·Type ↓ FPS, −)}|
⟨formal parameter list⟩,⟨id⟩{*ditto*}
⟨procedure heading⟩:: = ⟨procedure name⟩⟨formal par part⟩;
⟨value part⟩⟨spec part⟩{$\mathcal{S}$(ST, ⟨procedure name⟩, *if* ST·Type = "IP" *then*
(⟨procedure name⟩, 0) ↓ FASL *else* (⟨procedure name⟩, "1") ↓ FASL *fi*, −);
y ⇑ FPS; (⟨procedure name⟩, *y*) ↓ RTFPS}

The complete definition of Pass II and the error messages corresponding to this syntax are given in Appendix 2.

Definition of run-time routines

The routine Rtparcheck checks the type of the actual parameters for a function or procedure call (APS) against the corresponding formal parameter definition in RTFPS, and constructs an entry in RTAPS. The format of entries in APS is (Ptype, Norx, Procname, Id) where

Ptype = "I", "IA", "IP", "P", "L", or 0
Norx = no. of subscripts or parameters (if type = "IA", "IP" or "P")
or position in formal parameter list (if type = 0)
Procname = name of procedure or function (if type = 0)
Id = name of function or procedure passed as an actual parameter,
while entries in RTAPS have form (Rtid, Stack)
where Rtid = function or procedure identifier
Stack = stack of entries of form (I, Stype, Sn, Sid)

where

I	= position of actual parameter in list
Sn	= no. of subscripts or parameters (if Stype = "IA", "IP" or "P")
Sid	= name of function or procedure used as actual parameter.

Rtparcheck can be defined using the same static semantic notation as follows:

Rtparcheck (*i*, *j*, APS) =

{*if* *j* = 0 *then* *i* → *k* *else* $\mathcal{S}$(RTAPS, *j*, RTAPS·Stack ⇓ Z, −); $\mathcal{S}$(Z, *i*, Z·Sid → *k*, −);
λ ⇑ Z *fi*; $\mathcal{S}$(RTFPS, *k*, RTFPS·Pars ↓ X, −); ∀X: (*x* ↑ X; *y* ↑ APS;
if ·Ptype (*y*) ≠ 0 *then* *if* *x* ≠ 0 *and* ·Type (*y*) *then* E23 *else*
($\mathcal{N}$(X), ·Ptype (*y*), ·Norx (*y*), ·Id (*y*)) ↓ Y *fi*) *else*
($\mathcal{S}$(RTAPS, ·Procname (*y*), RTAPS·Stack ↓ Z, −); $\mathcal{S}$(Z, ·Norx (*y*),
if *x* ≠ 0 *and* Z·Stype *then* E24 *else* ($\mathcal{N}$(X), Z·Stype, Z·Sn, Z·Sid) ↓ Y *fi*, −);
λ ⇑ Z) *fi*) *fi*; *x* ⇑ Y; (*k*, *x*) ↓ RTAPS}

Similarly the function Rtcheck (*x*, *y*, *i*, *j*) which checks that the actual parameter in position *x* of the most recent call to function or procedure *y* is of type *i* with number of dimensions or parameters *j*, can be defined as follows:

Rtcheck (*x*, *y*, *i*, *j*) = {$\mathcal{S}$(RTAPS, *y*, RTAPS· Stack ↓ Z, −); $\mathcal{S}$(Z, *x*, *if* Z·Stype ≠ *i* *then* E25
else *if* Z·Sn ≠ *j* *then* E26 *fi* *fi*, −); λ ⇑ Z}

Finally the function Rtremove is defined as:
Rtremove () = {λ ↑ RTAPS}

SUMMARY

The notation presented in this paper is sufficient to describe the complete syntax of any programming language. A simple example has been given to demonstrate the way in which it is used.

The approach concentrated on in this paper is the bottom-up approach which reflects the way in which a compiler may function. The notation may also be used in a top-down fashion which may be more suitable for the user.

REFERENCES

1. H. F. Ledgard, A Formal System for Defining the Syntax and Semantics of Computer Languages. Ph.D. Thesis, M.I.T. (1969).
2. C. H. A. Koster, Two-level Grammars, in *Compiler Construction An Advanced Course*, pp. 146–156. Springer-Verlag, Berlin (1974).
3. A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation and Compiling* Vol. II. Prentice-Hall, Englewood Cliffs (1973).
4. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*. Addison-Wesley, Reading, MA (1977).
5. M. Griffiths, Relationship between definition and implementation of a language, in *Software Engineering, An Advanced Course*, pp. 76–110. Springer-Verlag, New York (1975).
6. M. H. Williams, Static semantic features of ALGOL60 and BASIC, *Comput. J.* **21**, 234–242 (1978).
7. M. H. Williams and A. R. Bulmer, Use of a formal notation for static semantics in compiler design, *Software Practice and Experience* **8**, 579–584 (1978).
8. P. Naur *et al.*, Revised report on the algorithmic language ALGOL60, *Comput. J.* **5**, 349–367 (1963).

About the Author—MORGAN HOWARD WILLIAMS received the B.Sc. degree in Physics and Mathematics from Rhodes University, and then proceeded to an Honours degree and a Ph.D. degree in ionospheric physics. However, as has happened to many physicists, particularly ionospheric physicists, he became more and more deeply involved with computing, and when undergraduate courses in Computer Science were started at Rhodes University in 1970, he was appointed as a lecturer in Computer Science. At present he is Professor of Computer Science at Rhodes University. He is also a co-editor of the new journal Quaestiones Informaticae, the journal of the Computer Society of South Africa, and the South African representative on the TC2 committee of IFIP.

APPENDIX 1: DEFINITION OF NOTATION

First define
⟨metalinguistic open bracket⟩ as ⟨
⟨metalinguistic close bracket⟩ as ⟩
⟨metalinguistic equals⟩ as ::=
⟨metalinguistic or⟩ as |
⟨open curly bracket⟩ as {
⟨close curly bracket⟩ as }
⟨production terminator⟩ as some suitable character to separate productions, and
⟨character⟩ as any terminal symbol of the grammar.

The simple BNF notation for syntax rules can be defined as follows.

⟨upper case letter⟩::= A|B|...|Z
⟨lower case letter⟩::= a|b|...|z
⟨letter⟩::= ⟨upper case letter⟩|⟨lower case letter⟩
⟨letter string⟩::= ⟨letter⟩|⟨letter string⟩⟨letter⟩|⟨letter string⟩␣
⟨metaling var⟩::= ⟨metalinguistic open bracket⟩⟨letter string⟩
⟨metalinguistic close bracket⟩
⟨metalinguistic expression⟩::= ⟨metaling var⟩{⟨metaling var⟩↓ MVS}|
⟨metalinguistic expression⟩⟨metaling var⟩{*ditto*}|
⟨character⟩|⟨metalinguistic expression⟩⟨character⟩
⟨production⟩::= ⟨metaling var⟩⟨metalinguistic equals⟩⟨metalinguistic expression⟩{⟨metaling var⟩↓ MVDS}
|⟨production⟩⟨metalinguistic or⟩⟨metalinguistic expression⟩
⟨production sequence⟩::= ⟨production⟩|
⟨production sequence⟩⟨production terminator⟩⟨production⟩
⟨grammar rules⟩::= ⟨production sequence⟩
{∀MVS: (*x* ↑ MVS; *if* *x* ≠ "⟨empty⟩" *then* $\mathcal{S}$(MVDS, *x*, −, E1))}

The generalized productions including static semantic rules are defined as follows:

⟨underlined letter⟩::= A|B|....|Z
⟨digit⟩::= 0|1|2...|9
⟨integer⟩::= ⟨digit⟩|⟨integer⟩⟨digit⟩
⟨character string⟩::= ⟨character⟩|⟨character string⟩⟨character⟩
⟨string constant⟩::= "⟨character string⟩"
⟨small letter string⟩::= ⟨lower case letter⟩⟨small letter string⟩⟨lower case letter⟩
⟨underlined letter string⟩::= ⟨underlined letter⟩|⟨underlined letter string⟩⟨underlined letter⟩
⟨variable name⟩::= ⟨small letter string⟩
⟨capital letter string⟩::= ⟨upper case letter⟩|⟨capital letter string⟩⟨upper case letter⟩
⟨stack name⟩::= ⟨capital letter string⟩
⟨simple string⟩::= ⟨metaling var⟩{$\mathcal{S}$(TEMP,⟨metaling var⟩, −, E2)}|⟨string constant⟩|0
⟨function⟩::= $\mathcal{N}$(⟨stack name⟩)|$\mathcal{T}$(⟨stack name⟩)|$\mathcal{P}$(⟨stack name⟩)
⟨null object⟩::= λ
⟨field name⟩::= ⟨capital letter⟩|⟨field name⟩⟨capital letter⟩|
⟨field name⟩⟨small letter⟩|⟨field name⟩⟨digit⟩
⟨pointer part⟩::= λ|(⟨variable name⟩)
⟨stack part⟩::= λ|⟨stack name⟩
⟨component object⟩::= ⟨stack part⟩·⟨field name⟩⟨pointer part⟩
⟨simple variable⟩::= ⟨variable name⟩|⟨component object⟩

⟨simple object⟩:: = ⟨simple string⟩|⟨function⟩|⟨null object⟩|⟨variable name⟩|⟨component object⟩
⟨string expression⟩:: = ⟨simple object⟩ + ⟨simple object⟩|⟨string expression⟩ + ⟨simple object⟩
⟨object sequence⟩:: = ⟨simple object⟩|⟨object sequence⟩, ⟨simple object⟩|
⟨compound object⟩|⟨object sequence⟩, ⟨compound object⟩
⟨compound object⟩:: = (⟨object sequence⟩)
⟨object⟩:: = ⟨simple object⟩|⟨compound object⟩
⟨stack operation⟩:: = ⟨object⟩ ↓ ⟨stack name⟩
⟨unstack operation⟩:: = ⟨simple variable⟩ ↑ ⟨stack name⟩|⟨null object⟩ ↑ ⟨stack name⟩|
⟨stack name⟩ ↑ ⟨stack name⟩
⟨whole stack transfer operation⟩:: = ⟨simple variable⟩ ⇑ ⟨stack name⟩|
⟨null object⟩ ⇑ ⟨stack name⟩|⟨stack name⟩ ⇑ ⟨stack name⟩
⟨whole stack return operation⟩:: = ⟨simple variable⟩ ⇓ ⟨stack name⟩|
⟨stack name⟩ ⇓ ⟨stack name⟩
⟨receptor⟩:: = ⟨simple variable⟩|⟨metaling var⟩
⟨alter operation⟩:: = ⟨compound object⟩ → ⟨simple variable⟩|⟨simple object⟩ → ⟨receptor⟩|
⟨string expression⟩ → ⟨receptor⟩
⟨whole stack unpack operation⟩:: = ∀⟨stack name⟩:(⟨unstack operation⟩;⟨action⟩)
⟨error action⟩:: = E⟨integer⟩
⟨routine name⟩:: = ⟨upper case letter⟩⟨small letter string⟩
⟨parameter⟩:: = ⟨simple object⟩|⟨stack name⟩
⟨parameter sequence⟩:: = ⟨parameter⟩|⟨parameter sequence⟩,⟨parameter⟩
⟨parameter part⟩:: = ⟨parameter sequence⟩|⟨empty⟩
⟨runtime action⟩:: = '⟨routine name⟩(⟨parameter part⟩)'
⟨searched object⟩:: = ⟨simple string⟩|⟨simple variable⟩
⟨search action⟩:: = $\mathcal{S}$(⟨stack name⟩,⟨searched object⟩,⟨action⟩,⟨action⟩)
⟨action name⟩:: = ⟨underlined letter string⟩
⟨null action⟩:: = –
⟨unconditional action⟩:: = ⟨search action⟩|⟨runtime action⟩|⟨error action⟩|
⟨whole stack unpack operation⟩|⟨whole stack transfer operation⟩|
⟨whole stack return operation⟩|⟨alter operation⟩|⟨unstack operation⟩|
⟨stack operation⟩|⟨null action⟩|(⟨action⟩)|⟨action name⟩
⟨relation⟩:: = ⟨simple object⟩⟨relational op⟩⟨simple object⟩
⟨relation op⟩:: = =|≠|<|≤|>|≥
⟨Boolean primary⟩:: = ⟨relation⟩|⟨Boolean primary⟩ *and* ⟨simple object⟩|
⟨Boolean primary⟩ *or* ⟨simple object⟩
⟨Boolean secondary⟩:: = ⟨Boolean primary⟩|(⟨Boolean expression⟩)
⟨Boolean factor⟩:: = ⟨Boolean secondary⟩|⟨Boolean factor⟩ ∧ ⟨Boolean secondary⟩
⟨Boolean expression⟩:: = ⟨Boolean factor⟩|⟨Boolean expression⟩ ∧ ⟨Boolean factor⟩
⟨conditional action⟩:: = *if* ⟨Boolean expression⟩ *then* ⟨action⟩ *fi*|
if ⟨Boolean expression⟩ *then* ⟨action⟩ *else* ⟨action⟩ *fi*
⟨loop action⟩:: = *while* ⟨Boolean expression⟩ *do* ⟨action⟩ *od*|
repeat ⟨action⟩ *until* ⟨Boolean expression⟩
⟨simple action⟩:: = ⟨conditional action⟩|⟨unconditional action⟩|⟨loop action⟩
⟨action⟩:: = ⟨simple action⟩|⟨action⟩; ⟨simple action⟩
⟨action sequence⟩:: = ⟨open curly bracket⟩⟨action⟩⟨close curly bracket⟩|
⟨open curly bracket⟩ *ditto* ⟨close curly bracket⟩|⟨empty⟩
⟨metalinguistic expression⟩:: = ⟨metaling var⟩{⟨metaling var⟩ ↓ MVS;
⟨metaling var⟩ ↓ TEMP}|
⟨metalinguistic expression⟩⟨metaling var⟩{*ditto*}|
⟨character⟩|⟨metalinguistic expression⟩⟨character⟩
⟨production⟩:: = ⟨metaling var⟩⟨metalinguistic equals⟩⟨metalinguistic expression⟩
⟨action sequence⟩{⟨metaling var⟩ ↓ MVDS; λ ⇑ TEMP}|
⟨production⟩⟨metalinguistic or⟩⟨metalinguistic expression⟩⟨action sequence⟩
{λ ⇑ TEMP}
⟨production sequence⟩:: = ⟨production⟩|
⟨production sequence⟩⟨production terminator⟩⟨production⟩
⟨action rule⟩:: = ⟨action name⟩ = ⟨action⟩
⟨action rules⟩:: = λ|⟨action rule⟩⟨action rules⟩
⟨grammar rules⟩:: = ⟨production sequence⟩⟨action rules⟩
{∀MVS: (x ↑ MVS; *if* x ≠ "⟨empty⟩" *then* $\mathcal{S}$(MVDS, x, –, E1))}

APPENDIX 2: STATIC SEMANTIC RULES FOR GRAMMAR IN FIG. 3

Pass 1

⟨formal parameter list⟩:: = ⟨id⟩{$\mathcal{S}$(FPS, ⟨id⟩, E15, (⟨id⟩, 0, "FP", $\mathcal{N}$(FPS), $\mathcal{F}$(PNS)) ↓ FPS)}|
⟨formal parameter list⟩, ⟨id⟩ {*ditto*}
⟨stm⟩:: = ⟨unlabelled stm⟩|⟨label⟩:⟨stm⟩{$\mathcal{S}$(ICBS, ⟨label⟩, E3, (⟨label⟩, "L", "A", 0, 0) ↓ ICBS)}
⟨identifier list⟩:: = ⟨id⟩{⟨id⟩ ↓ L}|⟨identifier list⟩, ⟨id⟩ {*ditto*}
⟨value part⟩:: = ⟨empty⟩|VALUE⟨identifier list⟩; {∀L: (x ↑ L; $\mathcal{S}$(VL, x, E19, x ↓ VL))}
⟨specifier⟩:: = INTEGER {"I" ↓ T}|INTEGER ARRAY{"IA" ↓ T}|INTEGER PROCEDURE{"IP" ↓ T}|
PROCEDURE{"P" ↓ T}|LABEL{"L" ↓ T}
⟨formal par dec⟩:: = ⟨specifier⟩⟨identifier list⟩

{$y \uparrow$ T; ∀L: ($x \uparrow$L; $\mathscr{S}$(FPS, x, *if* FPS·Ftype ≠ 0 *then* E1 *else* $y \rightarrow$ FPS·Ftype *fi*; *if* y = "I" *or* "IA" *then* $\mathscr{S}$(VL, x, 0 → VL·Name, −) *fi*, E20))}
⟨procedure name⟩:: = ⟨id⟩{$\mathscr{S}$(ICBS, ⟨id⟩, E2, ⟨id⟩ ↓ PNS)}
⟨procedure heading⟩:: = ⟨procedure name⟩⟨formal par part⟩;⟨value part⟩⟨spec part⟩
{∀VL: ($x \uparrow$ VL; *if* $x \neq 0$ *then* E21 fi); $x \uparrow$ PNS; $y \uparrow$ T; (x, y, "A", $\mathscr{N}$(FPS), 0) ↑ ICBS; DUMP ⇑ICBS; $\mathscr{N}$(BS) ↓ IBS; ($\mathscr{N}$(BS), 0) ↓ BS; $x \Uparrow$ FPS; $x \Downarrow$ ICBS}
⟨bounds⟩:: = ⟨bound pair⟩{"1" ↓ SL}|⟨bounds⟩,⟨bound pair⟩{*ditto*}
⟨type list⟩:: = ⟨id⟩{⟨id⟩ ↓ L}|⟨type list⟩,⟨id⟩{*ditto*}
⟨procedure type⟩:: = INTEGER PROCEDURE{"IP" ↓ T}|PROCEDURE{"P" ↓ T}
⟨dec⟩:: = INTEGER⟨type list⟩{∀L:($x \uparrow$L;$\mathscr{S}$(ICBS, x, E2, (x, "I", "A", 0, 0) ↓ ICBS))}|
INTEGER ARRAY⟨type list⟩⟨bounds⟩{∀L: ($x \uparrow$ L; $\mathscr{S}$(ICBS, x, E2, (x, "IA", "A", $\mathscr{N}$(SL), 0) ↓ ICBS)); $\lambda \Uparrow$SL}|
⟨procedure type⟩⟨procedure heading⟩⟨stm⟩{*INSERTICBSINTOIBS*}
⟨begin symbol⟩:: = BEGIN{DUMP ⇑ ICBS; $\mathscr{N}$(BS) ↓ IBS; ($\mathscr{N}$(BS), 0) ↓ BS}
⟨block⟩:: = ⟨begin symbol⟩⟨dec sequence⟩;⟨stm sequence⟩END
{*INSERTICBSINTOIBS*}
⟨program⟩:: = ⟨block⟩{∀BS: ($x \uparrow$ BS; ·Icbsdump(x) ↓ SS)}
INSERTICBSINTOIBS = $x \uparrow$ IBS; $y \Uparrow$ ICBS; $\mathscr{S}$(BS, x, $y \rightarrow$ BS·Icbsdump, −); DUMP ⇓ ICBS; $\lambda \uparrow$ DUMP

Pass 2

⟨var⟩:: = ⟨id⟩{$\mathscr{S}$(ST, ⟨id⟩, *if* ST·Type = 0 *then* 'Rtcheck (ST·X, ST·Y, "I", 0)' *else* *if* ST·Type ≠ "I" *then* E10 *fi fi*, E5)}
⟨array id⟩:: = ⟨id⟩{$\mathscr{S}$(ST, ⟨id⟩, *if* ST·Type ≠ "IA" *and* 0 *then* E11 *fi*, E6); DUMP ⇑ SL}
⟨proc id⟩:: = ⟨id⟩{$\mathscr{S}$(ST, ⟨id⟩, *if* ST·Type ≠ "P" *and* 0 *then* E12 *fi*, E7); DUMP ⇑ APS}
⟨function id⟩:: = ⟨id⟩{$\mathscr{S}$(ST, ⟨id⟩, *if* ST·Type ≠ "IP" *and* 0 *then* E13 *fi*, E8); DUMP ⇑ APS}
⟨subscript list⟩:: = ⟨arith exp⟩{"1" ↓ SL}|⟨subscript list⟩,⟨arith exp⟩{*ditto*}
⟨actual parameter⟩:: = ⟨arith exp⟩{("I", 0, 0, 0) ↓ APS}|
⟨id⟩{$\mathscr{S}$(ST, ⟨id⟩, *PLACEDETAILSONAPS*, E14)}
⟨formal parameter list⟩:: = ⟨id⟩{$\mathscr{S}$(ST, ⟨id⟩, ST·Type ↓ FPS, −)}|
⟨formal parameter list⟩,⟨id⟩{*ditto*}
⟨subscripted var⟩:: = ⟨array id⟩[⟨subscript list⟩]
{$\mathscr{S}$(ST, ⟨array id⟩, *if* ST·Formaloractual = "A" *then* *CHECKSUBSCRIPTSNOW* *else* *CHECKSUBSCRIPTSATRT*, −); $\lambda \Uparrow$ SL; DUMP ⇓ SL; $\lambda \uparrow$ DUMP}
⟨primary⟩:: = ⟨digit⟩|⟨var⟩|⟨subscripted var⟩|⟨function id⟩⟨actual par part⟩
{$\mathscr{S}$(ST, ⟨function id⟩, *if* ST·Formaloractual = "A" *then* *CHECKPARSNOW* *else* *CHECKPARSATRT fi*, −); $\lambda \Uparrow$ APS; DUMP ⇓ APS; $\lambda \uparrow$ DUMP}
⟨ass st⟩:: = ⟨subscripted var⟩: = ⟨arith exp⟩|⟨id⟩: = ⟨arith exp⟩
{$\mathscr{S}$(ST, ⟨id⟩, *if* ST·Type = "IP" ∧ ST·Formaloractual = "A" *then* $\mathscr{S}$(FASL, ⟨id⟩, "1" → FASL·Funtype, E18) *else if* ST·Type = 0 *then* 'Rtcheck (ST·X, ST·Y, "I", 0)' *else if* ST·Type ≠ "I" *then* E10 *fi fi fi*, E5)}
⟨goto st⟩:: = GOTO⟨label⟩{$\mathscr{S}$(ST, ⟨label⟩, *if* ST·Type = 0 *then* 'Rtcheck (ST·X, ST·Y, "L", 0)' *else if* ST·Type ≠ "L" *then* E9 *fi fi*, E4)}
⟨proc st⟩:: = ⟨proc id⟩⟨actual par part⟩{$\mathscr{S}$(ST, ⟨procid⟩, *if* ST·Formaloractual = "A" *then* *CHECKPROCPARSNOW* *else* *CHECKPROCPARSATRT* *fi*, −); $\lambda \Uparrow$ APS; DUMP ⇓ APS; $\lambda \uparrow$ DUMP}
⟨procedure name⟩: = ⟨id⟩{ST ↑ SS}
⟨procedure heading⟩:: = ⟨procedure name⟩⟨formal par part⟩; ⟨value part⟩⟨spec part⟩
{$\mathscr{S}$(ST, ⟨procedure name⟩, *if* ST·Type = "IP" *then* (⟨procedure name⟩, 0) ↓ FASL *else* (⟨procedure name⟩, "1") ↓ FASL *fi*, −); $y \Uparrow$ FPS; (⟨procedure name⟩, y) ↓ RTFPS}
⟨dec⟩:: = INTEGER⟨type list⟩|INTEGER ARRAY⟨type list⟩⟨bounds⟩|
⟨procedure type⟩⟨procedure heading⟩⟨stm⟩{$\lambda \uparrow$ ST; 'Rtremove()'; $x \uparrow$ FASL;
if ·Funtype (x) = 0 *then* E22 *fi*}
⟨begin symbol⟩:: = BEGIN{ST ↑ SS}
⟨block⟩:: = ⟨begin symbol⟩⟨dec sequence⟩;⟨stm sequence⟩END{$\lambda \uparrow$ ST}
PLACEDETAILSONAPS = *if* ST·Type = "IP" *or* "P" ∧ ST·Formaloractual = "A" *then* (ST·Type, ST·X, ST·Y, ⟨id⟩) ↓ APS *else* (ST·Type, ST·X, ST·Y, 0) ↓ APS
CHECKSUBSCRIPTSNOW = *if* ST·X ≠ $\mathscr{N}$(SL) *then* E16 *fi*
CHECKSUBSCRIPTSATRT = 'Rtcheck (ST·X, ST·Y, "IA", $\mathscr{N}$(SL))'
CHECKPARSATRT = 'Rtcheck (ST·X, ST·Y, "IP", $\mathscr{N}$(APS))'; 'Rtparcheck (ST·X, ST·Y, APS)'
CHECKPARSNOW = *if* ST·X ≠ $\mathscr{N}$(APS) *then* E17 *fi*; 'Rtparcheck(⟨function id⟩, 0, APS)'
CHECKPROCPARSNOW = *if* ST·X ≠ $\mathscr{N}$(APS) *then* E17 *fi*; 'Rtparcheck(⟨procid⟩, 0, APS)'
CHECKPROCPARSATRT = 'Rtcheck (ST·X, ST·Y, "P", $\mathscr{N}$(APS))'; 'Rtparcheck (ST·X, ST·Y, APS)'

Runtime routines

Rtparcheck (i, j, APS) = {*PLACEPROCNAMEINK*;
$\mathscr{S}$(RTFPS, k, RTFPS·Pars ↓ X, −); ∀X: ($x \uparrow$X; $y \uparrow$ APS; *if* ·Ptype(y) ≠ 0 *then* *APENTRYISACTUAL* *else* (*APENTRYISFORMAL*; $\lambda \Uparrow$ Z) *fi*); $x \Uparrow$ Y; (k, x) ↓ RTAPS}
PLACEPROCNAMEINK = *if* j = 0 *then* $i \rightarrow k$ *else* ($\mathscr{S}$(RTAPS, j, RTAPS·Stack Z, −); $\mathscr{S}$(Z, i, Z·Sid → k, −); $\lambda \Uparrow$ Z) *fi*
APENTRYISACTUAL = *if* $x \neq 0$ *and* ·Type (y) *then* E23 *else* ($\mathscr{N}(x)$, ·Ptype (y), ·Norx (y), ·Id (y)) ↓ *yfi*

APENTRYISFORMAL = $\mathscr{S}$(RTAPS, ·Procname (*y*), RTAPS·Stack ↓ Z, −); $\mathscr{S}$(Z, ·Norx (*y*), *if* $x \neq 0$ *and* Z·Stype *then* E24 *else* (. ↓ (X), Z·Stype, Z·Sn, Z·Sid) ↓ Y *fi*, −)
Rtcheck (*x*, *y*, *i*, *j*) = {$\mathscr{S}$(RTAPS, *y*, RTAPS·Stack ↓ Z, −); $\mathscr{S}$(Z, *x*, *if* Z·Stype $\neq$ *i then* E25 *else* *if* Z·Sn $\neq$ *j then* E26 *fi fi*, −); λ ⇑ Z}
Rtremove() = {λ ↑ RTAPS}

Errors

1. Formal parameter defined twice in the specification part of a procedure declaration.
2. Identifier defined more than once in the same block.
3. Identifier used as a label is already defined in the same block.
4. Label referred to has not been declared.
5. Variable identifier referred to has not been declared.
6. Array identifier referred to has not been declared.
7. Procedure identifier referred to has not been declared.
8. Integer procedure identifier referred to has not been declared.
9. Identifier used as label but declared to be of some other type.
10. Identifier used as a simple variable but declared to be of some other type.
11. Identifier used as an array but declared to be of some other type.
12. Identifier used as a procedure but declared to be of some other type.
13. Identifier used as a integer procedure but declared to be of some other type.
14. Identifier used as an actual parameter has not been declared.
15. Formal parameter appears twice in the same formal parameter list.
16. Number of subscripts in the subscript list of a subscripted variable does not match the number of dimensions in the corresponding array declaration.
17. Number of parameters in an actual parameter list does not match the number of formal parameters in the formal parameter list of a procedure declaration.
18. Assignment to a function identifier outside the body of the function.
19. Formal parameter identifier occurs more than once in VALUE list.
20. Parameter encountered in specification part which does not appear in the formal parameter list.
21. VALUE list contains identifiers not declared as INTEGER or INTEGER ARRAY.
22. Function does not contain assignment statement with function identifier on left-hand side.
23. Type of actual parameter in a procedure or function call does not match the type of the corresponding formal parameter in the called routine.
24. The type of a formal parameter used as an actual parameter in a procedure or function call does not match the type of the corresponding formal parameter in the called routine.
25. The type of an actual parameter, which is substituted for a formal parameter of undeclared type, does not match its use.
26. The number of dimensions associated with an array identifier or the number of parameters associated with a procedure or function identifier used as an actual parameter which is substituted for a formal parameter, does not match the number of subscripts or parameters used.

15. M.H.WILLIAMS, Static semantic features of Algol60 and Basic, The Computer Journal, vol. 21, no. 3, 234-242 (1978).

Static semantic features of ALGOL60 and BASIC

M. H. Williams

Department of Computer Science, Rhodes University, Grahamstown 6140, South Africa

The static semantic rules of ALGOL60 and BASIC are expressed in a formal notation to demonstrate that the notation can be used for such languages. This is seen as an aid to both the compiler writer and the user, partly in that the intentions of the language designer can be communicated clearly and unequivocally and partly in that the specification may be used as a guide to producing a correct compiler or in verifying the correctness of a compiler.

(Received October 1975)

. Introduction

n a previous paper (Williams, 1978) a formal notation was ›resented for describing the static semantic rules, or 'context-ensitive' rules (Ledgard, 1969) of a language. This notation vas developed to interface with the BNF specification of the ;yntax of a language. The notation is based on the use of ;trings (either those represented by metalinguistic variables n the BNF specification or literal strings) and stacks. It is ntended that such a notation should provide the compiler vriter with a guide for the implementation of the language or give the user an understanding of the details of the language.

In this paper the notation is used to describe the static semantic rules of two programming languages: ALGOL60 and BASIC. In the case of ALGOL60, the static semantic rules are complex and have never been completely specified in a useful notation for the compiler writer. BASIC, on the other hand, has relatively simple static semantic rules although there is no general agreement on the syntax rules.

The fact that the static semantic rules for these two languages can be expressed completely in this notation means that the language designer can communicate his intentions directly and unambiguously to the compiler writer and the user, rather than attempting to convey his ideas by means of a set of English sentences which may often give rise to misinterpretation. This notation also gives the compiler writer a clear guide as to how these rules may be implemented.

2. BASIC

Unlike ALGOL60 there is no generally accepted syntax specification for BASIC. Bull *et al.* (1973) have produced a detailed specification which includes all the facilities of a full BASIC compiler. However, as many BASIC compilers do not provide all these facilities and since the specification is a little large to handle here, a specification of a 'basic' subset of the language was sought.

Lee (1972) provides such a specification. The subset of BASIC which he defines does not include strings, matrix statements, subprograms, file handling or advanced input/output facilities. After correction of a few minor errors in Lee's specification and conversion of his context-sensitive rules to a context-free form, one arrives at a specification which is not a strict subset of Bull's more general definition. Thus changes were made to bring the specification more into line with Bull's definition (e.g. null programs or empty DATA statements are not valid, whereas parameterless functions and generalised forms of IF and ON statements are permitted).

The syntax specifications given by both Lee and Bull rely on statements being in correct order of line number. However, this is not very helpful for the compiler writer who is implementing an interactive BASIC system. He needs a specification which will check that each statement has the correct form as it is read in (whatever the order in which statements are read). Thus the syntax rules were changed slightly to cater for this. Finally the resulting set of rules was converted into 'standard' BNF notation.

The static semantic rules for BASIC can be summarised as follows:

1. Line numbers may be entered in any order; duplicate line numbers and null lines may occur. The lines must be sorted into line number order, and in the case of duplicate lines, the last supplied is used.
2. The same letter may be used both as a simple variable and as a one or two dimensional array. An array is declared either explicitly in a DIM statement or implicitly by its first appearance in the program. In the case of explicit declaration all references to the array (subscripted variables) must have the same number of subscripts as there are dimensions in the declaration; if declaration is implicit all references to the array must have the same number of subscripts as the first reference to the array. A DIM statement can occur anywhere in a program.
3. The same array name cannot be used to apply to more than one array in a program.
4. If a function is called in a program, the actual parameters supplied must correspond in number, type (relevant when extending the definition of BASIC to include strings and string variables) and order to the formal parameters in the function definition. A function definition may occur anywhere in a program, not necessarily at a lower-numbered line than that of the function call.
5. A function name may be used as a destination for a LET statement only within the definition block for that function.
6. Function definitions may not be nested.
7. Control may not be transferred from outside a function definition to an intermediate line within it, nor may control be transferred out of a function definition. However, control may be passed to the first or only line of a definition block.
8. Each FOR statement must have a corresponding NEXT statement which refers to the same control variable. FOR-NEXT loops may not overlap.
9. A FOR-NEXT loop may not overlap with a function definition.
10. For each multiline definition statement DEF <user function>, there must be a corresponding FNEND statement.
11. If a user function is called in a program, the function must be defined somewhere in the program.
12. If a line number is used as a destination line number in a

program, it must also occur as a line no def somewhere in the program.

The latter two rules may be regarded as unnecessary. However, as some BASIC compilers perform these checks at compile time and some do not, they have been included in this specification on the grounds that they can easily be removed if not required.

The syntax interpretation rules are divided into two passes. The first pass handles rule 1 above. This accepts lines in any order, checks that the form of each line is correct and stores it (overwriting any previous line with the same number). On encountering the END statement, the program is complete and lines are sorted into ascending order for the second pass. The concatenation operation is used to assemble the program in the correct order for the second pass. In practice the level of reconstruction will obviously depend on the compiler writer.

The second pass uses the following stacks:

(*a*) AS (array stack) with entries of form:
(array name, number of dimensions, declared or undeclared).

(*b*) APL (actual parameter list) with entries of form:
type of actual parameter.

(*c*) PARS (parameter stack) with entries of form:
(number of parameters, stack of parameter types, declared or undeclared).

(*d*) FS (function stack) with entries of form:
(function name, number of parameters, stack of parameter types, declared or undeclared).

(*e*) FPL (formal parameter list) with entries of form:
type of formal parameter.

(*f*) UFS (user function stack) should either be empty or contain the name of the current user function being defined.

(*g*) ULOD (undefined line numbers outside definitions)
(*h*) LOD (line numbers outside definitions)
(*i*) ULWCD (undefined line numbers within current definition)
(*j*) LWCD (line numbers within current definition)
} have entries of form: line number.

(*k*) CURLINE (current line) contains the line number of the current line.

(*l*) LS (line number stack) contains an entry for each line number in the program.

(*m*) FORS (for stack) has an entry of form: (simple variable) for each unmatched FOR statement encountered thus far in the current function definition or outer block.

(*n*) DUMP contains the contents of the for stack while a function definition is being handled.

Each entry (parameter type) in APL and FPL will always be 'A' (arithmetic) for this specification so that mismatched types cannot occur. However, when extending the language to include strings, entries will be either 'A' or 'S' and hence matching parameter types becomes important.

The complete specification of syntax and static semantic rules follows.

2.1 *Syntax rules for BASIC*

1. <program> ::= <statement sequence> <terminal st>
2. <statement sequence> ::= <statement> |
<statement> <statement sequence>
3. <statement> ::= <line no def> <unnumbered st>
<new line> | <line no def> <new line>
4. <unnumbered st> ::= <simple statement> |
<single line def st> | <multiline def st> | <fn end st>
5. <simple statement> ::= <let st> | <read st> |
<data st> | <print st> |
<goto st> | <on st> | <if st> |
<for st> | <next st> | <dim st> |
<gosub st> | <return st> |
<restore st> | <stop st> |
<rem st>
6. <terminal st> ::= <line no def> END <new line>
7. <line no def> ::= <line no>
8. <line no> ::= <digit> | <digit> <digit> | <digit> <digit> <digit> | <digit> <digit> <digit> <digit> | <digit> <digit> <digit> <digit> <digit>
9. <expression> ::= <plusminus> <term> | <term> | <expression> <plusminus> <term>
10. <term> ::= <factor> | <term> <timesover> <factor>
11. <factor> ::= <primary> | <factor> ↑ <primary>
12. <plusminus> ::= + | −
13. <timesover> ::= × |/
14. <primary> ::= <constant> | <variable> | <function ref> |(<expression>)
15. <variable> ::= <simple variable> | <subscripted variable>
16. <simple variable> ::= <letter> | <letter> <digit>
17. <subscripted variable> ::= <letter>(<expression>) | <letter>(<expression>,<expression>)
18. <integer> ::= <digit> | <integer> <digit>
19. <number> ::= <integer> |.<integer> | <integer>. <integer>
20. <exponent> ::= E <plusminus> <expt> |E <expt>
21. <expt> ::= <digit> | <digit> <digit>
22. <constant> ::= <number> <exponent> | <number>
23. <digit> ::= 0|1|2|3|4|5|6|7|8|9
24. <function ref> ::= <function name> | <function name> (<actual parameter list>)
25. <function name> ::= <library function> | <user function>
26. <actual parameter list> ::= <expression> | <actual parameter list>,<expression>
27. <user function> ::= FN <letter>
28. <library function> ::= SIN|COS|TAN|ATN|EXP|ABS|LOG|SQR|INT|RND
29. <dim st> ::= DIM <array dimension list>
30. <array dimension list> ::= <array dim> | <array dim>, <array dimension list>
31. <array dim> ::= <letter>(<integer>) | <letter> (<integer>,<integer>)
32. <let st> ::= LET <destination> = <expression> | <destination> = <expression>
33. <destination> ::= <variable> | <user function>
34. <signed number> ::= <plusminus> <constant> | <constant>
35. <data list> ::= <signed number> | <data list>,<signed number>
36. <data st> ::= DATA <data list>
37. <restore st> ::= RESTORE
38. <goto st> ::= GOTO <destination line no>
39. <on st> ::= ON <expression> <designator> <line number list>
40. <designator> ::= THEN|GOTO|GOSUB
41. <line number list> ::= <destination line no> | <line number list>,<destination line no>
42. <gosub st> ::= GOSUB <destination line no>
43. <return st> ::= RETURN
44. <if st> ::= IF <condition> <designator> <destination line no>
45. <destination line no> ::= <line no>

6. <condition> ::= <expression> <relation> <expression>
7. <relation> ::= = | > = | < = | > | < | < >
8. <for st> ::= FOR <simple variable> = <expression> TO <expression> | FOR <simple variable> = <expression> TO <expression> STEP <expression>
9. <next st> ::= NEXT <simple variable>
10. <rem st> ::= REM <comment> | REM
11. <comment> ::= <character> | <comment> <character>
12. <stop st> ::= STOP
13. <read st> ::= READ <destination list>
14. <destination list> ::= <variable> | <destination list>, <variable>
15. <print st> ::= PRINT <print list>
16. <item sep print list> ::= <item sequence> <separator sequence> | <item sep print list> <item sequence> <separator sequence>
17. <sep item print list> ::= <separator sequence> <item sequence> | <sep item print list> <separator sequence> <item sequence>
18. <item list> ::= <item sequence> | <item sep print list> | <item sep print list> <item sequence>
19. <sep list> ::= <separator sequence> | <sep item print list> | <sep item print list> <separator sequence>
20. <print list> ::= <item list> | <sep list>
21. <separator sequence> ::= <print separator> | <separator sequence> <print separator>
22. <print separator> ::= , | ;
23. <item sequence> ::= <message> | <print expression> | <message> <print expression> | <item sequence> <message> | <item sequence> <message> <print expression>
24. <print expression> ::= <expression> | TAB (<expression>)
25. <message> ::= " <character string> "
26. <character string> ::= <character> | <character string> <character>
27. <character> ::= <letter> | <digit> | <special character>
28. <def head> ::= DEF <user function> (<formal par list>) | DEF <user function>
29. <formal par list> ::= <simple variable> | <formal par list>, <simple variable>
30. <single line def st> ::= <def head> = <expression>
31. <multiline def st> ::= <def head>
32. <fn end st> ::= FNEND
33. <letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
34. <special character> ::= + | − | × | / | = |) | (| > | < | . | ; | <blank> | $ | ↑ | ?, |
35. <newline> ::= **CR**

2.2 *Static semantic rules for pass I*

3. <statement> ::= <line no def> <unnumbered st> <newline> {$\mathcal{S}$(SS, <line no def>, <unnumbered st> → Comp2, (<line no def>, <unnumbered st>)↓SS)} | <line no def> <newline> {$\mathcal{S}$(SS, <line no def>, λ→ Comp2, (<line no def>, λ)↓SS)}
6. <terminal st> ::= <line no def> END <newline> {$\mathcal{S}$(SS, <line no def>, 'END'→Comp2, (<line no def>, 'END')↓SS)}
1. <program> ::= <statement sequence> <terminal st> {$\mathcal{A}$(SS,SS); λ→x; ∀SS:(y↑SS; x.Comp1(y).Comp2(y). **'CR'**→x; Comp1(y)↓LS); x→ <program>}

2.3 *Static semantic rules for pass II*

17. <subscripted variable> ::= <letter> (<expression>) {$\mathcal{S}$(AS, <letter>, **if** Comp2≠'1' **then** (**if** Comp3='U' **then** E1 **else** E2), (<letter>,'1','U')↓AS)} | <letter> (<expression>, <expression>) {$\mathcal{S}$(AS, <letter>, **if** Comp2≠'2' **then** (**if** Comp3='U' **then** E1 **else** E2), (<letter>,'2', 'U')↓AS)}
31. <array discussion> ::= <letter> (<integer>) {$\mathcal{S}$(AS, <letter>, **if** Comp3='D' **then** E3 **else if** Comp2≠"1" **then** E4 **else** 'D'→Comp3, (<letter>, '1', 'D')↓AS)} | <letter> (<integer>, <integer>) {$\mathcal{S}$(AS, <letter>, **if** Comp3='D' **then** E3 **else if** Comp2≠'2' **then** E4 **else** 'D'→Comp3, (<letter>,'2','D')↓AS)}
26. <actual parameter list> ::= <expression> {'A'↓APL} | <actual parameter list>, <expression> {**ditto**}
28. <library function> ::= SIN{('1',('A'),'D')↓PARS} | COS{**ditto**} | TAN{**ditto**} | ATN{**ditto**} | EXP{**ditto**} | ABS{**ditto**} | LOG{**ditto**} | SQR{**ditto**} | INT{**ditto**} | RND{**ditto**}
25. <function name> ::= <library function> | <user function> {$\mathcal{S}$(FS, <user function>, (Comp2, Comp3, Comp4)↓PARS, ('U',0,0)↓PARS)}
24. <function ref> ::= <function name> {x↑PARS; **if** Comp1(x)='U' **then** (<function name>,0,0,'U')↓FS **else if** Comp1(x)≠0 **then** (**if** Comp3='D' **then** E5 **else** E6)} | <function name> (<actual parameter list>){x↑PARS; **if** Comp1(x)='U' **then** (<function name>, $\mathcal{N}$(APL), APL,'U')↓FS **else if** Comp1(x)≠$\mathcal{N}$(APL) **then** (**if** Comp3='D' **then** E5 **else** E6) **else** (Comp2(x)→y; ∀y:(u↑y; v↑APL; **if** u≠v **then** E7)); λ ⇑ APL}
69. <formal par list> ::= <simple variable> {'A'↓FPL; $\mathcal{S}$(T, <simple variable>, E20, <simple variable>↓T)} | <formal par list>, <simple variable> {**ditto**}
68. <def head> ::= DEF <user function> (<formal par list>) {**if** $\mathcal{N}$(UFS)≠0 **then** E8; $\mathcal{S}$(FS, <user function>, **if** Comp4='D' **then** E9 **else** 'D'→Comp4; **if** Comp2≠$\mathcal{N}$(FPL) **then** E10; Comp3→y; ∀y:(x↑y; z↑ FPL; **if** x≠z **then** E11), (<user function>, $\mathcal{N}$(FPL), FPL,'D')↓FS); <user function>↓UFS; λ ⇑ T} | DEF <user function> {**if** $\mathcal{N}$(UFS)≠0 **then** E8; $\mathcal{S}$(FS, <user function>, **if** Comp4='D' **then** E9 **else** 'D'→Comp4; **if** Comp2≠0 **then** E10, (<user function>, 0,0,'D')↓FS); <user function>↓UFS}
70. <single line def st> ::= <def head> = <expression> {λ↑UFS}
71. <multiline def st> ::= <def head> {DUMP ⇑ FORS}
7. <line no def> ::= <line no> {**if** $\mathcal{N}$(UFS)=0 **then** ($\mathcal{S}$(ULOD, <line no>, λ→Comp1, −); <line no>↓LOD) **else** ($\mathcal{S}$(ULWCD, <line no>, λ→Comp1, −); <line no>↓LWCD); **if** $\mathcal{N}$(CURLINE)≠0 **then** λ↑CURLINE; <line no>↓CURLINE}
45. <destination line no> ::= <line no> {$\mathcal{S}$(LS, <line no>, −, E12); **if** <line no> < = CURLINE **then** (**if** $\mathcal{N}$(UFS)=0 **then** $\mathcal{S}$(LOD, <line no>, −, E13) **else** $\mathcal{S}$(LWCD, <line no>, −, E14)) **else** (**if** $\mathcal{N}$(UFS)=0 **then** $\mathcal{S}$(ULOD, <line no>, −, <line no>↓ULOD) **else** $\mathcal{S}$(ULWCD, <line no>, −, <line no>↓ ULWCD)}
72. <fn end st> ::= FNEND{λ↑UFS; ∀ULWCD: (x↑ULWCD; **if** x≠λ **then** E14); λ ⇑ LWCD; **if** $\mathcal{N}$(FORS)

≠0 **then** E17;DUMP ⇓ FORS;λ↑DUMP}

13. <destination> ::= <variable> | <user function>
{**if** <user function> ≠ℱ(UFS) **then** E15}

18. <for st> ::= FOR <simple variable> = <expression>
TO <expression> {<simple variable> ↓FORS} |
FOR <simple variable> = <expression>
TO <expression> STEP <expression> {**ditto**}

19. <next st> ::= NEXT <simple variable>
{**if** 𝒩(FORS)=0 **then** E16 **else** (x↑FORS; **if** x≠ <simple variable> **then** E16)}

1. <program> ::= <statement sequence> <terminal st>
{∀FS:(x↑FS; **if** Comp4(x)='U' **then** E18); **if** 𝒩(FORS) ≠0 **then** E17; **if** 𝒩(ULOD)≠0 **then** E13; **if** 𝒩(UFS)≠0 **then** E19}

2.4 *Error numbers*

1. Number of subscripts of subscripted variable does not match number of subscripts previously used.
2. Number of subscripts of subscripted variable does not match number of subscripts declared in DIM statement.
3. Array defined twice.
4. Number of subscripts in array declaration does not match number of subscripts previously used.
5. Number of actual parameters in a function call does not match number of formal parameters in function definition.
6. Number of actual parameters in a function call differs from number of actual parameters used in a previous call to the same function.
7. Type of an actual parameter in a function call differs from type previously encountered or defined in that position.
8. Function definition within a function definition.
9. Function defined twice.
10. Number of formal parameters in a function definition does not match the number of actual parameters previously used in calls for the function.
11. Type of a formal parameter in a function definition does not match type previously encountered in that position in function calls.
12. Jump to an undefined line number.
13. Jump into a function definition.
14. Jump out of a function definition.
15. Function name used as a destination outside the definition of that function.
16. Next statement does not have corresponding For statement.
17. For statement does not have corresponding Next statement.
18. Function called is not defined.
19. FN END statement is missing.
20. Formal parameters in a formal parameter list are identical.

3. ALGOL60

In the case of ALGOL60 the syntax rules for the language were clearly defined in BNF notation by Naur, *et al.* (1963). However, the static semantic rules were expressed rather vaguely as a set of English sentences. As a result there are situations where the validity of an ALGOL program cannot be determined from the ALGOL60 report. This makes the problem of implementation more difficult for the compiler writer and the problem of standardisation becomes impossible.

The specification given here defines the static semantic rules completely. In cases where it was not clear from the original report what the outcome should be, a decision was taken one way or another. The specification again consists of two passes and requires in addition a number of run-time routines for dynamic checking of parameter types.

3.1 *Syntax rules for ALGOL60*

1. <expression> ::= <arith exp> | <bool exp> | <des exp>
2. <variable id> ::= <id>
3. <simple var> ::= <variable id>
4. <subs exp> ::= <arith exp>
5. <subs list> ::= <subs exp> | <subs list>, <sub exp>
6. <array id> ::= <id>
7. <array ref> ::= <id>
8. <subs var> ::= <array ref> [<subs list>]
9. <var> ::= <subs var> | <simple var>
10. <proc id> ::= <id>
11. <function id> ::= <id>
12. <actual par> ::= <string> | <id> | <expression>
13. <letter string> ::= <letter> | <letter string> <letter>
14. <par delimiter> ::= , |) <letter string> :(
15. <actual par list> ::= <actual par> |
<actual par list> <par delimiter> <actual par>
16. <actual par part> ::= <empty> |(<actual par list>)
17. <function des> ::= <function id> <actual par part>
18. <adding op> ::= + | −
19. <primary> ::= <unsigned integer no> |
<unsigned real no> | <var> | <function des> |
(<arith exp>)
20. <factor> ::= <primary> | <factor> ↑ <primary>
21. <term> ::= <factor> | <term> × <factor> |
<term> / <factor> | <term> ÷ <factor>
22. <simple a e> ::= <term> | <adding op> <term> |
<simple a e> <adding op> <term>
23. <if clause> ::= **if** <bool exp> **then**
24. <arith exp> ::= <simple a e> |
<if clause> <simple a e> **else** <arith exp>
25. <relational op> ::= < | ≤ | = | ≥ | > | ≠
26. <relation> ::=
<simple a e> <relational op> <simple a e>
27. <boolean prim> ::= <logical value> | <var> |
<function des> | <relation> |(<bool exp>)
28. <boolean sec> ::= <boolean prim> |
¬ <boolean prim>
29. <boolean factor> ::= <boolean sec> |
<boolean factor> ∧ <boolean sec>
30. <boolean term> ::= <boolean factor> |
<boolean ter> ∨ <boolean factor>
31. <implication> ::= <boolean term> |
<implication> ⊃ <boolean term>
32. <simple boolean> ::= <implication> |
<simple boolean> ≡ <implication>
33. <bool exp> ::= <simple boolean> |
<if clause> <simple boolean> **else** <bool exp>
34. <label> ::= <id> | <unsigned integer>
35. <label ref> ::= <id> | <unsigned integer>
36. <switch id> ::= <id>
37. <switch ref> ::= <id>
38. <switch des> ::= <switch ref> [<subs exp>]
39. <simple d e> ::= <label ref> | <switch des> |
(<des exp>)
40. <des exp> ::= <simple d e> |
<if clause> <simple d e> **else** <des exp>
41. <unlabelled basic stm> ::= <ass st> | <goto st> |
<dummy st> | <proc st>
42. <basic stm> ::= <unlabelled basic stm> |
<label> : <basic stm>
43. <unconditional stm> ::= <basic stm> |
<compound stm> | <block>
44. <stm> ::= <unconditional stm> | <conditional stm> |
<for st>
45. <compound tail> ::= <stm> **end** |
<stm> ; <compound tail>
46. <begin symbol> ::= **begin**

7. <block head> ::= <begin symbol> <declaration> |
<block head> ; <declaration>
8. <unlabelled compound> ::= **begin** <compound tail>
9. <unlabelled block> ::=
<block head> ; <compound tail>
10. <compound stm> ::= <unlabelled compound> |
<label> : <compound stm>
11. <block> ::= <unlabelled block> | <label> : <block>
12. <program> ::= <block> | <compound stm>
13. <left part> ::= <subs var> := | <id> :=
14. <left part list> ::= <left part> |
<left part list> <left part>
15. <ass st> ::= <left part list> <arith exp> |
<left part list> <bool exp>
16. <goto st> ::= **goto** <des exp>
17. <dummy st> ::= <empty>
18. <if st> ::= <if clause> <unconditional stm>
19. <conditional stm> ::= <if st> | <if st> **else** <stm> |
<if clause> <for st> | <label> : <conditional stm>
20. <for list element> ::= <arith exp> |
<arith exp> **step** <arith exp> **until** <arith exp> |
<arith exp> **while** <bool exp>
21. <for list> ::= <for list element> | <for list>, <for list element>
22. <for clause> ::= **for** <var> := <for list> **do**
23. <for st> ::= <for clause> <stm> | <label> : <for st>
24. <proc st> ::= <proc id> <actual par part>
25. <declaration> ::= <type decl> | <array decl> |
<switch decl> | <proc decl>
26. <type list> ::= <simple var> |
<simple var>, <typelist>
27. <type> ::= **real** | **integer** | **Boolean**
28. <local or own type> ::= <type> | **own** <type>
29. <type decl> ::= <local or own type> <type list>
30. <lower bound> ::= <arith exp>
31. <upper bound> ::= <arith exp>
32. <bound pair> ::= <lower bound> : <upper bound>
33. <bound pair list> ::= <bound pair> |
<bound pair list>, <bound pair>
34. <array seg> ::= <array id> [<bound pair list>] |
<array id>, <array seg>
35. <array list> ::= <array seg> |
<array list>, <array seg>
36. <array specifier> ::= **array** | <local or own type> **array**
37. <array decl> ::= <array specifier> <array list>
38. <switch list> ::= <des exp> | <switch list>, <des exp>
39. <switch decl> ::= **switch** <switch id> := <switch list>
40. <formal par> ::= <id>
41. <formal par list> ::= <formal par> |
<formal par list> <par delimiter> <formal par>
42. <formal par part> ::= <empty> | (<formal par list>)
43. <id list> ::= <id> | <id list>, <id>
44. <value part> ::= <empty> | **value** <id list> ;
45. <specifier> ::= **string** | <type> | **array** | <type> **array** |
label | **switch** | **procedure** | <type> **procedure**
46. <spec part> ::= <empty> | <specifier> <id list> ; |
<spec part> <specifier> <id list> ;
47. <proc name> ::= <id>
48. <proc heading> ::= <proc name> <formal par part> ;
<value part> <spec part>
49. <proc body> ::= <stm> | <code>
50. <proc decl> ::=
procedure <proc heading> <proc body> |
<type> **procedure** <proc heading> <proc body>

3.2 *Static semantic rules for pass 1*

The first pass is concerned with the construction of a symbol table with entries for each identifier in the program. This is done using:

(*a*) ICBS (Identifiers in Current Block Stack) which has entries of form: (identifier, type, structure, actual/formal, n, x, y)
where type = 'R', 'I', 'B' or 0,
structure = 'V' (variable), 'A' (array), 'F' (function), 'P' (procedure), 'S' (switch), 'L' (label) or 'STR' (string),
actual/formal = 'A' (actual variable) or 'F' (formal parameter),
n = number of dimensions (array) or parameters (procedure or function),
x = position of formal parameter in formal parameter list, and
y = name of procedure or function with formal parameter list;

(*b*) DUMP which stores the contents of the ICBS when entering an inner block;

(*c*) BS (Block Stack) which has entries of form: (No of level, ICBS for this level)

(*d*) NBS (Nested Block Stack) which contains level numbers of each incomplete (nested) block; and

(*e*) SS (Symbol Stack) which contains the ICBS for each level in order.

Other stacks used are:

(*a*) T (Type stack) contains entries of form: type;

(*b*) SL (Subscript List) contains count of number of subscripts;

(*c*) L is a stack of identifiers (id or array id);

(*d*) FPL (Formal Parameter List) is a stack of formal parameter identifiers;

(*e*) PNL (Procedure Name List) contains the current procedure (or function) name;

(*f*) VL (Value List) contains identifiers called by value;

(*g*) S (Structure stack) contains entries of form: structure ('V', 'A', etc.).

34. <label> ::= <id> {$\mathcal{S}$(ICBS, <id>, E2, (<id>, 0, 'L', 'A', 0,0,0)↓ICBS)} |
<unsigned integer> {$\mathcal{S}$(ICBS, <unsigned integer>, E2, (<unsigned integer>, 0, 'L', 'A', 0,0,0)↓ICBS)}
67. <type> ::= **real** {'R'↓T} | **integer** {'I'↓T} | **Boolean** {'B'↓T}
66. <type list> ::= <simple var> {$\mathcal{S}$(ICBS, <simple var>, E1, (<simple var>, $\mathcal{T}$(T), 'V', 'A', 0,0,0)↓ICBS)} |
<simple var>, <type list> {**ditto**}
69. <type decl> ::= <local or own type> <type list> {λ↑T}
73. <bound pair list> ::= <bound pair> {'1'↓SL} |
<bound pair list>, <bound pair> {**ditto**}
74. <array seg> ::= <array id> [<bound pair list>]
{<array id>↓L; ∀L:(y↑L; $\mathcal{S}$(ICBS, y, E1, (y, $\mathcal{T}$(T), 'A', 'A', $\mathcal{N}$(SL), 0,0)↓ICBS)); λ ⇑ SL} |
<array id>, <array seg> {<array id>↓L}
76. <array specifier> ::= **array** {'R'↓T} |
<local or own type> **array**
77. <array decl> ::= <array specifier> <array list> {λ↑T}
79. <switch decl> ::= **switch** <switch id> := <switch list> {$\mathcal{S}$(ICBS, <switch id>, E1, (<switch id>, 0, 'S', 'A', 0,0,0)↓ICBS)}
87. <proc name> ::= <id> {$\mathcal{S}$(ICBS, <id>, E1, <id>↓PNL)}
80. <formal par> ::= <id> {$\mathcal{S}$(FPL, <id>, E3, (<id>, 0, 'FP', 'F', 0, $\mathcal{N}$(FPL), $\mathcal{T}$(PNL))↓FPL)}
83. <id list> ::= <id> {<id>↓L} | <id list>, <id> {**ditto**}
84. <value part> ::= <empty> | **value** <id list> ; {∀L:(x↑L; $\mathcal{S}$(VL, x, E4, x↓VL))}
85. <specifier> ::= **string** {'STR'↓S; 0↓T} | <type> {'V'↓S} |
array {'A'↓S; 'R'↓T} | <type> **array** {'A'↓S} |
label {'L'↓S; 0↓T} | **switch** {'S'↓S; 0↓T} |
procedure {'P'↓S; 0↓T} | <type> **procedure** {'P'↓S}

6. <spec part> ::= <empty> | <specifier> <id list>;
{y↑T;z↑S;∀L:(x↑L;'F'→u; **if** z='V' **or** 'A' **or** 'L' **then** $\mathcal{S}$(VL,x,λ→VL;'V'→u,−);$\mathcal{S}$(FPL,x, **if** Comp3≠'FP' **then** E5 **else** (y→Comp2;z→Comp3;u→Comp4),E6))} |
<spec part> <specifier> <id list>; {**ditto**}

8. <proc heading> ::= <proc name> <formal par part>;
<value part> <spec part> {y↑PNL;x↑T;
if $\mathcal{N}$(T)=0 **then** (x↓T;(y,0,'P','A',$\mathcal{N}$(FPL),0,0)↓ICBS)
else (y,x,'F', 'A',$\mathcal{N}$(FPL),0,0)↓ICBS; DUMP ⇑ ICBS;
$\mathcal{N}$(BS)↓NBS; ($\mathcal{N}$(BS),0)↓BS;x ⇑ FPL;x ⇓ ICBS;
∀VL:(x↑VL; **if** x≠λ**then** E7)}

10. <proc decl> ::= **procedure** <proc heading> <proc body> {x↑NBS;y ⇑ ICBS;$\mathcal{S}$(BS,x,y→Comp2,−);
DUMP ⇓ ICBS;λ↑DUMP} |
<type> **procedure** <proc heading> <proc body> {**ditto**}

46. <begin symbol> ::= **begin** {DUMP ⇑ ICBS;$\mathcal{N}$(BS)↓NBS;
($\mathcal{N}$(BS),0)↓BS}

49. <unlabelled block> ::= <block head>;<compound tail> {x↑NBS;y ⇑ ICBS;$\mathcal{S}$(BS,x,y→Comp2,−);
DUMP ⇓ ICBS;λ↑DUMP}

52. <program> ::= <block> {∀BS:(x↑BS;Comp2(x)↓SS)} |
<compound stm> {**ditto**}

3.3 *Static semantic rules for pass II*

The result of the first pass is a complete symbol table SS containing the identifiers declared at each level. This is retained for use in the second pass. The second pass checks the uses of identifiers (arrays, procedures, formal parameters, labels, etc.) against their definitions to ensure that each identifier is used correctly.

Stacks used in the second pass are the following:

(*a*) ST (Symbol Table) which contains symbol table entries from SS for current blocks

(*b*) SL (Subscript List) which counts the number of subscripts in the subscript list of a subscripted variable

(*c*) APL (Actual Parameter List) which counts the number of actual parameters in a procedure or function call

(*d*) DUMP which stores SL and APL when nesting occurs

(*e*) STDUMP which stores one level of symbol table entries while evaluating dynamic array bound expressions in an array declaration

(*f*) FASL (Function Assignment Statement List) which stores the name of each function while it is being defined, together with a marker to indicate whether it contains an assignment statement with the function name as left part

(*g*) L which is used as a temporary stack

(*h*) TS (Type Stack) which has entries of form
(type, structure, n, x, y, APL, formal par stack)
where type = 'R', 'I', 'B', 'FP', 'FPS' or 0 (if structure='L')
structure = 'VE', 'V', 'A', 'P' or 'L',
n = number of dimensions or parameters,
x = position in formal parameter list,
y = name of procedure (function) to which formal parameter 'belongs',
APL = actual parameter list (in case of function call)
and formal par stack = stack of all formal parameters whose type is not determined at any point in an expression; this has entries of form:
(structure, n, x, y, APL)

(*i*) LPS (Left Part Stack) which has entries of the same form as TS

(*j*) RTFPS (Run Time Formal Parameter Stack) which has entries of form:
(procedure or function name, stack)
where stack is a stack of entries (one for each formal parameter) of form:
(type, structure, value)
where type = 'R', 'I', 'B' or 0,
structure = 'V', 'A', 'F', 'P', 'S', 'STR', 'L' or 'FP',
and value = 'V' or 0.

When an identifier is encountered in an expression an entry is placed on the Type Stack. If the identifier is not a formal parameter, or if it is a formal parameter whose type is specified, the Type Stack entry will have 'R', 'I' or 'B' in Component 1 and zeros in all other components. If the identifier is a formal parameter of unspecified type, the entry has 'FP' as first component and non-zero Components 4 and 5. From the context in which the formal parameter is used, parameters 2 and 3 are deduced.

When a production is encountered which determines the type of its dependent formal parameter(s), the appropriate Run Time Check(s) can be inserted into the object code and the type stack entry removed or replaced by one with 'R', 'I' or 'B' in Component 1 and zeros in the other components. If the two top items of TS are both of type 'FP' and the production cannot deduce what type either should be, the two items are replaced by a single entry which has 'FPS' as Component 1, zeros in Components 2 to 6 and Components 2 to 6 of the original two items are placed in the stack in Component 7 of the resulting item.

The specification for pass II is as follows:

1. <expression> ::= <arith exp> | <bool exp>
{('B',0,0,0, 0,0,0)↓TS} | <des exp> {(0,'L',0,0,0,0,0)↓TS}

5. <subs list> ::= <subs exp> {'1'↓SL} | <subs list>, <subs exp> {**ditto**}

7. <array ref> ::= <id> {DUMP ⇑ SL}

8. <subs var> ::= <array ref>[<subs list>]
{$\mathcal{S}$(ST,<array ref>, **if** Comp3='FP' **then** ('FP','A', $\mathcal{N}$(SL),Comp6,Comp7,0,0)↓TS **else if** Comp3≠'A' **then** E15 **else** ((Comp2,0,0,0,0,0,0)↓TS; **if** Comp4='A' **then** (**if** Comp5≠$\mathcal{N}$(SL) **then** E30) **else** 'Rtcheck(Comp6, Comp7, Comp2,'A', $\mathcal{N}$(SL))'),E10);λ ⇑ SL;DUMP ⇓ SL; λ↑DUMP}

9. <var> ::= <subs var> | <simple var>
{$\mathcal{S}$(ST,<simple var>, **if** Comp3='FP' **then** ('FP', 'VE', 0,Comp6,Comp7,0,0)↓TS **else if** Comp3='V' **then** (**if** Comp4='F' **then** (Comp2,'VE',0,Comp6,Comp7,0,0)↓TS **else** (Comp2,0,0,0,0,0,0)↓TS) **else** E14,E9)}

19. <primary> ::= <unsigned integer no> {('I',0,0,0,0,0,0) ↓TS} |
<unsigned real no> {('R',0,0,0,0,0,0)↓TS} |
<var> {$\mathcal{T}$(TS)→x; **if** Comp1(x)='B' **then** E21} |
<function des> {$\mathcal{T}$(TS)→x; **if** Comp1(x)='B' **then** E22} |
(<arith exp>)

20. <factor> ::= <primary> | <factor> ↑ <primary> {①}

21. <term> ::= <factor> | <term> × <factor> {①} |
<term> / <factor> {x↑TS;②;x↑TS;②;('R',0,0,0,0,0,0) ↓TS} |
<term> ÷ <factor> {x↑TS; **if** Comp1(x)='R' **then** E23 **else** ③;x↑TS; **if** Comp1(x)='R' **then** E23 **else** ③; ('I',0,0,0,0,0,0)↓TS}

22. <simple a e> ::= <term> | <adding op> <term> |
<simple a e> <adding op> <term> {①}

24. <arith exp> ::= <simple a e> |
<if clause> <simple a e> **else** <arith exp> {①}

26. <relation> ::= <simple a e> <relational op>
<simple a e> {x↑TS;②;x↑TS;②}

27. <boolean prim> ::= <logical value> | <var> {x↑TS; **if** Comp1(x)='R' **or** 'I' **then** E28 **else if** Comp1(x)='FP' **then** 'Rtcheck(Comp4(x),Comp5(x),'B',Comp2(x), Comp3(x))'} |
<function des> {x↑TS; **if** Comp1(x)='R' **or** 'I' **then** E29 **else if** Comp1(x)='FP' **then** ('Rtcheck(Comp4(x),

Comp5(x),'B',Comp2(x),Comp3(x))';
'Rtparlistcheck(Comp4(x),Comp5(x),Comp6(x))')} |
<relation> |(<bool exp>)

3. <left part> ::= <subs var> := {LPS↑TS} |
<id> := {$\mathcal{S}$(ST,<id>, **if** Comp3='V' ∧ (Comp4='A' **or** 'V') **then** (Comp2,0,0,0,0,0,0)↓LPS **else if** Comp3='F' ∧ Comp4='A' **then** ((Comp2,0,0,0,0,0,0)↓LPS;$\mathcal{S}$(FASL,<id>,'1'→Comp2, E32)) **else if** Comp3='V' ∧ Comp4='F' **then** ((Comp2,0,0,0,0,0,0)↓LPS; 'Rtcheck(Comp6,Comp7,Comp2,'V',0)') **else if** Comp3='FP' **then** ('FP','V',0,Comp6,Comp7,0,0)↓LPS **else** E20, E19)}

4. <left part list> ::= <left part> |
<left part list> <left part>
{x↑LPS;y↑LPS; **if** Comp1(y)='R' **then** (y↓LPS; **if** Comp1(x)='I' **then** E24 **else if** Comp1(x)='FP' **then** 'Rtcheck(Comp4(x),Comp5(x),'R',Comp2(x),Comp3(x))' **else if** Comp1(x)='FPS' **then** (Comp7(x)↓L;∀L:(y↑L; 'Rtcheck(Comp3(y),Comp4(y),'R',Comp1(y), Comp2(y))'))) **else if** Comp1(x)='R' **then** (x↓LPS; **if** Comp1(y)='I' **then** E24 **else if** Comp1(y)='FP' **then** 'Rtcheck(Comp4(y),Comp5(y),'R',Comp2(y),Comp3(y))' **else if** Comp1(y)='FPS' **then** (Comp7(y)↓L;∀L:(x↑L; 'Rtcheck(Comp3(x),Comp4(x),'R',Comp1(x), Comp2(x))'))) **else if** Comp1(y)='I' **then** (y↓LPS; **if** Comp1(x)='FP' **then** 'Rtcheck(Comp4(x),Comp5(x),'I', Comp2(x),Comp3(x))' **else if** Comp1(x)='FPS' **then** (Comp7(x)↓L;∀L:(y↑L;'Rtcheck(Comp3(y),Comp4(y),'I', Comp1(y),Comp2(y))'))) **else if** Comp1(x)='I' **then** (x↓LPS; **if** Comp1(y)='FP' **then** 'Rtcheck(Comp4(y), Comp5(y),'I',Comp2(y),Comp3(y))' **else if** Comp1(y)='FPS' **then** (Comp7(y)↓L;∀L:(x↑L;'Rtcheck (Comp3(x), Comp4(x),'I',Comp1(x),Comp2(x))'))) **else** (**if** Comp1(y)='FP' **then** (Comp2(y),Comp3(y),Comp4(y), Comp5(y),Comp6(y))↓L **else** Comp7(y) ⇓ L; (Comp2(x), Comp3(x),Comp4(x),Comp5(x),Comp6(x))↓L;x ⇑ L; ('FPS',0,0,0,0,0,x)↓LPS)}

55. <ass st> ::= <left part list> <arith exp> {x↑TS;②; x↑LPS; **if** Comp1(x)='B' **then** E25; **if** Comp1(x)='FP' **then** 'Rtcheck(Comp4(x),Comp5(x),'RI',Comp2(x), Comp3(x))' **else if** Comp1(x)='FPS' **then** 'Rtchecklist (Comp7(x))'} |
<left part list> <bool exp> {x↑LPS; **if** Comp1(x)='R' **or** 'I' **then** E26; **if** Comp1(x)='FP' **then** ('Rtcheck (Comp4(x),Comp5(x),'B',Comp2(x),Comp3(x))'; **if** Comp2(x)='F' **then** 'Rtparlistcheck(Comp4(x), Comp5(x),Comp6(x))') **else if** Comp1(x)='FPS' **then** (Comp6(x)↓L;∀L:(y↑L;'Rtcheck(Comp3(y),Comp4(y), 'B',Comp1(y),Comp2(y))'; **if** Comp1(y)='F' **then** 'Rtparlistcheck(Comp3(y),Comp4(y),Comp5(y))')}

60. <for list element> ::= <arith exp> {x↑TS;②} |
<arith exp> **step** <arith exp> **until** <arith exp> {x↑TS; ②;x↑TS;②;x↑TS;②} |
<arith exp> **while** <bool exp> {x↑TS;②}

62. <for clause> ::= **for** <var> := <for list> **do** {x↑TS; **if** Comp1(x)='B' **then** E27 **else if** Comp1(x)='FP' **then** (**if** Comp2(x)='VE' **then** 'V'→Comp2(x); 'Rtcheck(Comp4(x),Comp5(x),'R',Comp2(x),Comp3(x))') **else if** Comp2(x)='VE' **then** 'Rtcheck(Comp4(x), Comp5(x),Comp1(x),'V',Comp3(x))'}

70. <lower bound> ::= <arith exp> {x↑TS;②}

71. <upper bound> ::= <arith exp> {x↑TS;②}

4. <subs exp> ::= <arith exp> {x↑TS;②}

10. <proc id> ::= <id> {DUMP ⇑ APL}

11. <function id> ::= <id> {DUMP ⇑ APL}

12. <actual par> ::= <string> {(0,'STR',0,0,0,0)↓APL} |
<expression> {x↑TS; **if** Comp1(x)='FP' **then** (0, 'VE', (Comp4(x),Comp5(x),Comp2(x),Comp3(x)),0,0,0)↓APL **else if** Comp1(x)='FPS' **then** (0,'VE',Comp7(x),0,0,0)↓APL **else if** Comp2(x)≠'L' **then** (Comp1(x),'VE',0,0,0,0) ↓APL **else** (0,'L',0,0,0,0)↓APL} |
<id> {$\mathcal{S}$(ST,<id>, **if** Comp3='F' ∧ Comp4='A' **then** (Comp2,Comp3,Comp5,0,0,<id>)↓APL **else** (Comp2,Comp3,Comp5,Comp6,Comp7,0)↓APL, E18)}

17. <function des> ::= <function id> <actual par part> {$\mathcal{N}$(APL)→y;x ⇑ APL;$\mathcal{S}$(ST,<function id>, **if** Comp3='FP' **then** ('FP','F',y,Comp6,Comp7,x,0)↓TS **else if** Comp3≠'F' **then** E17 **else** ((Comp2,0,0,0,0,0,0) ↓TS; **if** Comp4='A' **then** (**if** Comp5≠y **then** E31; 'Rtparlistcheck(<function id>,0,x)') **else** ('Rtcheck(Comp6,Comp7,Comp2,'F',y)'; 'Rtparlistcheck(Comp6,Comp7,x)')),E12);DUMP ⇓ APL; λ↑DUMP}

64. <proc st> ::= <proc id> <actual par part> {$\mathcal{S}$(ST, <proc id>, **if** Comp3≠'FP' **and** 'P' **then** E17; **if** Comp4='A' **then** (**if** Comp5≠$\mathcal{N}$(APL) **then** E31; 'Rtparlistcheck(<proc id>,0,APL)') **else** ('Rtcheck (Comp6,Comp7,0,'P',$\mathcal{N}$(APL))'; 'Rtparlistcheck(Comp6,Comp7,APL)'), E12);λ ⇑ APL; DUMP ⇓ APL;λ↑DUMP}

80. <formal par> ::= <id> {$\mathcal{S}$(ST,<id>, **if** Comp4='V **then** 'V'→x **else** 0→x; (Comp2,Comp3,x)↓FPL,−)}

88. <proc heading> ::= <proc name> <formal par part>; <value part> <spec part> {$\mathcal{S}$(ST,<proc name>, **if** Comp3='F' **then** (<proc name>,0)↓FASL,−);y ⇑ FPL; (<proc name>,y)↓RTFPS}

90. <proc decl> ::=
procedure <proc heading> <proc body> {λ↑ST} |
<type> **procedure** <proc heading> <proc body> {λ↑ST;x↑FASL; **if** Comp2(x)=0 **then** E33}

35. <label ref> ::= <id> {$\mathcal{S}$(ST,<id>, **if** Comp3≠'L' **and** 'FP' **then** E13, E8)} |
<unsigned integer> {$\mathcal{S}$(ST, <unsigned integer>, **if** Comp3≠'L' **then** E13, E8)}

37. <switch ref> ::= <id> {$\mathcal{S}$(ST,<id>, **if** Comp3≠'S' **and** 'FP' **then** E16, E11)}

46. <begin symbol> ::= **begin** {ST↑SS}

49. <unlabelled block> ::= <block head>;<compound tail> {λ↑ST}

76. <array specifier> ::= **array** {STDUMP↑ST} |
<local or own type> **array** {**ditto**}

77. <array decl> ::= <array specifier> <array list> {ST↑STDUMP}

where

① = x↑TS; y↑TS; **if** Comp1(y)='R' **then** (y↓TS; ②) **else if** Comp1(x)='R' **then** (x↓TS; y→x;②) **else if** Comp1(y)='I' **then** x↓TS **else if** Comp1(x)='I' **then** y↓TS **else if** Comp1(y)='FP' **then** ((Comp2(y),Comp3(y), Comp4(y),Comp5(y),Comp6(y))↓L; **if** Comp1(x)='FP' **then** (Comp2(x),Comp3(x), Comp4(x),Comp5(x),Comp6(x))↓L **else** Comp7(x) ⇓ L;x ⇑ L; ('FPS',0,0,0,0,0,x)↓TS) **else if** Comp1(x)='FP' **then** ((Comp2(x),Comp3(x), Comp4(x),Comp5(x),Comp6(x))↓L; Comp7(y) ⇓ L;x ⇑ L;('FPS',0,0,0,0,0,x)↓TS) **else** (Comp7(x) ⇓ L;Comp7(y) ⇓ L;x ⇑ L;('FPS',0,0,0,0,0,x) ↓TS)

② = **if** Comp1(x)='FP' **then** ('Rtcheck(Comp4(x),Comp5(x), 'RI',Comp2(x),Comp3(x))'; **if** Comp2(x)='F' **then** 'Rtparlistcheck(Comp4(x), Comp5(x),Comp6(x))') **else if** Comp1(x)='FPS' **then** (Comp7(x)↓L; ∀L: (y↑L; 'Rtcheck(Comp3(y),Comp4(y),'RI',Comp1(y), Comp2(y))'; **if** Comp1(y)='F' **then** ' Rtparlistcheck (Comp3(y),Comp4(y),Comp5(y))'))

③ = **if** Comp1(x)='FP' **then** ('Rtcheck(Comp4(x),Comp5(x),

'I',Comp2(x),Comp3(x))'; **if** Comp2(x)='F' **then** 'Rtparlistcheck(Comp4(x),Comp5(x),Comp6(x))') **else if** Comp1(x)='FPS' **then** (Comp7(x)↓L;∀L:(y↑L; 'Rtcheck(Comp3(y),Comp4(y),'I',Comp1(y), Comp2(y))'; **if** Comp1(y)='F' **then** 'Rtparlistcheck (Comp3(y),Comp4(y),Comp5(y))'))

3.4 *Run time routines*

The format of these is not defined by the notation; however, the action performed by them can be expressed by means of the notation. Two stacks are used: RTFPS (Run Time Formal Parameter Stack) which is set up in Pass II, and RTAPS (Run Time Actual Parameter Stack) which is maintained by the run time routines.

Rtparlistcheck (i, j, APL) =
{**if** j=0 **then** i→k **else** (𝒮(RTAPS,j,Comp2 ⇓ Z,−); 𝒮(Z,i,Comp5→k,−);λ ⇑ Z);𝒮(RTFPS,k,Comp2↓X,−); ∀X:(x↑X;y↑APL; **if** Comp2(y)='FP' **then** 𝒮(RTAPS, Comp5(y),Comp2 ⇓ Z,−);𝒮(Z,Comp4(y),Comp2→ Comp1(x);Comp3→Comp2(x);Comp4→Comp3(x); Comp5→Comp6(x),−);λ ⇑ Z); **if** Comp2(x)='FP' **then** (𝒩(X),Comp1(y),Comp2(y),Comp3(y),Comp6(y))↓PARS **else if** Comp2(y)≠'VE' **then** (**if** Comp1(y)=Comp1(x) ∧ Comp2(y)=Comp2(x) **then** (𝒩(X),Comp1(y),Comp2(y), Comp3(y),Comp6(y))↓PARS **else** E34) **else** (**if** Comp1(y)=0 **then** (Comp3(y) ⇓ Z;∀Z:(z↑Z; 'Rtcheck(Comp1(z),Comp2(z), Comp1(x),Comp3(z),Comp4(z))'); **if** Comp2(x)≠'V' **then** E35) **else if** Comp1(y)≠Comp1(x) ∨ Comp2(x)≠'V' **then** E35; **if** Comp3(x)='V' **then** (𝒩(X),Comp1(x),'V',0,0)↓PARS **else** (𝒩(X),Comp1(x),'VE',0,0)↓PARS)); x ⇑ PARS; (k,x)↓RTAPS}

Rtcheck(x,y,i,j,k)=
{𝒮(RTAPS,y,Comp2↓S,−);𝒮(S,x, **if** Comp3≠'VE' ∨ Comp2≠0 **then** (**if** (i≠'RI' ∨ Comp2≠'R' **and** 'I') ∧ Comp2≠i **then** E36 **else if** Comp3≠j ∨ Comp4≠k **then** E37) **else** (**if** j≠'VE' **then** E38; Comp4 ⇓ TEMP; ∀TEMP: (t↑TEMP; 'Rtcheck(Comp1(t),Comp2(t),i, Comp3(t), Comp4(t))')),−);λ ⇑ S}

Rtchecklist (i) =
{𝒯(i)→x;𝒮(RTAPS,Comp4(x),Comp2(x)↓S,−); 𝒮(S,Comp3(x),Comp2→y,−); **if** y≠'R' **and** 'I' **then** E20; λ ⇑ S;∀i:(x↑i;𝒮(RTAPS,Comp4(x),Comp2↓S,−); 𝒮(S,Comp3(x), **if** Comp3='VE' ∧ Comp2=0 **then** E 38 **else if** Comp2≠y **then** E39 **else if** Comp3≠Comp1(x) ∨ Comp4≠Comp2(x) **then** E40,−);λ ⇑ S)}

3.5 *Error numbers*

1. Identifier is declared more than once in the same block.
2. Symbol used as label has already been declared in the same block.
3. Formal parameter identifier appears more than once in the same formal parameter list.
4. Formal parameter appears more than once in the same *value* list.
5. Formal parameter appears more than once in the specification part of a procedure declaration.
6. Identifier appears in the specification part of a procedure declaration which does not occur in the formal parameter list.
7. One or more identifiers in the value list of a procedure declaration do not correspond to formal parameters specified to be simple variables, arrays or labels.
8. Label referred to in a designationial expression does not exist.
9. Simple variable referred to in an expression has not been declared.
10. Array name referred to has not been declared.
11. Switch identifier referred to has not been declared.
12. Procedure identifier referred to has not been declared.
13. Label referred to in a designationial expression has been declared to have structure different from label.
14. Simple variable referred to in an expression has been declared to have structure different from simple variable.
15. Array name referred to has been declared to have structure different from array.
16. Switch identifier referred to has been declared to have structure different from switch.
17. Procedure identifier referred to has been declared to have structure different from procedure.
18. Identifier used as an actual parameter has not been declared.
19. Identifier appearing as a left part has not been declared.
20. Identifier appearing as a left part has incorrect structure.
21. Boolean variable used as a primary in an arithmetic expression.
22. Boolean procedure called in an arithmetic expression.
23. At least one of the operands of the operator ÷ is not of type integer.
24. Types of identifiers appearing in the left part list of an assignment statement are different.
25. An arithmetic expression assigned to a Boolean variable.
26. A Boolean expression assigned to an arithmetic variable.
27. A Boolean variable used as the controlled variable of a For statement.
28. An arithmetic variable used as a Boolean primary in a Boolean expression.
29. An arithmetic procedure used as a Boolean primary in a Boolean expression.
30. The number of subscripts in a subscripted variable does not match the number of dimensions of the array in the array declaration.
31. The number of actual parameters in a procedure call does not match the number of formal parameters in the procedure declaration.
32. Assignment to a function identifier outside the body of the function.
33. Function does not contain an assignment statement with the function identifier as left-part variable.
34. The type/structure of an identifier used as an actual parameter does not match the type/structure of the corresponding formal parameter as declared in the specification part.
35. The type/structure of an expression used as an actual parameter does not match the type/structure of the corresponding formal parameter as declared in the specification part.
36. The type of an identifier used as a formal parameter does not match the type required by its use.
37. The structure/number of dimensions or parameters of an actual parameter does not match the structure/number of dimensions or parameters of the formal parameter as it is used.
38. An expression is used as an actual parameter whereas the corresponding formal parameter expects an identifier (e.g. array id).
39. An actual parameter which is substituted for a formal parameter used as a left-part variable of an assignment statement, does not match the types of the other parameters substituted as left-part variables.
40. An actual parameter which is substituted for a formal parameter used as a left-part variable of an assginment statement, does not match the structure/number of dimensions of its use.
41. The type of a formal parameter used in an actual parameter expression is causing a mismatch with the declared type (in the specification part) of the formal parameter for which the

actual parameter is to be substituted.

12. The structure/number of dimensions or parameters of a formal parameter used in an actual parameter expression does not match its use.
13. A formal parameter used in an actual parameter expression expects an identifier to be substituted for it, but finds instead an expression.

4. Conclusion

This study demonstrates how the static semantic rules of two common programming languages can be specified formally. The static semantic rules of ALGOL60 are much more difficult to specify than those of BASIC, partly because of the ability to mix types within an expression under certain circumstances but not under others (e.g. ÷), and partly because the type and structure of a formal parameter need not be declared.

For the complete specification of ALGOL60, situations were encountered for which the correctness could not be decided from the original report. The implementation of such situations is left to the compiler writer's discretion. However, if one is to standardise programming languages so that a program written in a given high level language has the same syntax and produces the same effect (wherever possible) when run on different machines, one needs to specify all aspects of the language completely. Thus in the case of undecideable situations in ALGOL60 a decision was taken one way or another. Thus the specification of ALGOL60 given here will produce a similar effect to that produced by some ALGOL60 compilers but not by others. It is not suggested that the specification given here should necessarily be the generally accepted one; however, the intention is to show that the complex static semantic rules of ALGOL60 can be expressed in a formal notation which is meaningful to compiler writers and hence assist in the standardisation of such a language.

Besides its effect on standardisation such a formal specification may also be of assistance to the compiler writer from the point of view of the correctness of the compiler he produces by giving him a model of how these aspects of the language can be implemented at the very simplest level. Obviously the compiler writer will have his own ideas as to how best to implement certain features to provide the most efficient system. However, at a time when the correctness of the product is as important as its efficiency, by providing the compiler writer with a simply implementable specification of these aspects of the language, he has a yardstick by which to measure the correctness of his own implementation.

References

BULL, G. M., FREEMAN, W. and GARLAND, S. J. (1973). *Specification for Standard BASIC*, NCC Publications: Manchester.

LEDGARD, H. F. (1969). A Formal System for Defining the Syntax and Semantics of Computer Languages, Ph.D. Thesis, MIT.

LEE, J. A. N. (1972). The Formal Definition of the BASIC Language, *The Computer Journal*, vol. 15, pp. 37-41.

NAUR, P. *et al.* (1963). Revised report on the algorithmic language ALGOL60, *The Computer Journal*, vol. 5, pp. 349-367.

WILLIAMS, M. H. (1978). A Formal Notation for specifying Static Semantic Rules, submitted to *Computer Languages*.

Book reviews

The Complexity of Computational Problem Solving, edited by R. S. Anderssen and R. P. Brent, 1976; 262 pages. (*University of Queensland Press*, £4·70)

The title is mildly ambiguous, since this book is not so much concerned with problem solving in the usual sense (although this is certainly involved) as complexity in all its computational aspects, and understood indeed in a rather broad way to include efficiency in the formulation, programming and debugging phases of program construction. The problems are therefore primarily with respect to the methods of computing and less with the 'external' problem for which the computer is to be used.

An example of an important external problem is the last of the fifteen articles which is called 'the complexity of real-world scheduling problems'. These problems include resource allocation and assignment, timetabling, job/shop scheduling and the like. These are so complicated that they require overconstrained linear programs which are heuristic in nature. The extent to which such programs depart from optimality can also be measured.

Three papers in the book concentrate on hardware; one example is where parallel and sequential processing are compared. The rest of the book concentrates on software—these vary over a whole range of matters such as complexity, strategy and stability in algebraic and other computational methods.

The book is well organised and physically well produced, presumably by a photographic method, and has a flexiback cover. It is rather specialised in its contents, but for those in the field it is a book that certainly deserves a place on the bookshelf.

F. H. GEORGE (Uxbridge)

Digital System Design Automation—Languages, Simulation and Data Base, by Melvin A. Breuer; 1977; 417 pages. (*Pitman*, £13·95)

This book is aimed primarily at programmers whose function is to provide software for computer hardware designers, and generally at anyone working near the hardware/software interface of digital systems.

Chapter 1, System Level Simulation, deals with conventional simulation languages. Implementation details are given for two such languages. The emphasis is mainly on the simulation of computer programs, and there is a large section on graph models of programs. Chapter 2, Register Transfer Languages and Their Translation, describes a register transfer language called DDL. In Chapter 3, Register Transfer Language Simulation, the problem of translating DDL descriptions into executable hardware simulation programs is tackled. A weakness of both these chapters is that too much emphasis is placed on minor details of a specific language; this tends to obscure more fundamental issues such as the problems of simulating parallel or asynchronous operations.

The fourth chapter, Design Automation Aids to Microprogramming, contains a good description of microprogramming, and illustrates the use of a register transfer language for describing the operating of a microprogram. The final chapter, Data Structures, Data Base and File Management, contains standard material on data base management and the representation of data.

The book has several weaknesses; in particular, it is excessively verbose. However there is a great shortage of literature on this subject, and the book fills an important gap.

PETER J. MOYLAN (Newcastle, Australia)

16. M.H.WILLIAMS, Methods for specifying static semantics, Computer Languages, vol. 6, no. 1, 1-17 (1981).

Computer Languages Vol. 6, pp. 1 to 17. 1981
Printed in Great Britain. All rights reserved

0096-0551 81 010001-17$02.00/0
Copyright © 1981 Pergamon Press Ltd

METHODS FOR SPECIFYING STATIC SEMANTICS

M. H. WILLIAMS*
Department of Computer Science, Rhodes University, Grahamstown, South Africa

(*Received* 28 *March* 1980)

Abstract—The formal specification of a programming language involves the specification of three types of rules: syntax, static semantics and semantics. Various methods have been proposed for specifying the static semantic rules of programming languages, but as yet no method has received general acceptance. This paper looks at several different specification techniques and attempts to isolate the basic mechanisms used by each of them and explain the pattern of development of specification techniques for static semantics.

Static semantics Formal specification Context-sensitive

INTRODUCTION

IN DESIGNING a language, a language designer is faced with two particular problems:

(i) He must be sure that he has considered and catered for all possible problems which might arise when implementing different features of his language (cf. the case of "own arrays" in Algol60 [1]);
(ii) He must communicate his ideas in a clear and unambiguous way to the implementors and users of the language.

In the past it has been found that it is not sufficient simply to describe the features of a language in English, as this can very easily conceal problems which the designer has not foreseen and is very prone to misinterpretation by different readers. What is needed is some formal notation for specifying languages which will reflect the designer's ideas clearly and completely. This will provide the designer with the framework for specifying every aspect of the language exactly and will provide the medium whereby the designer's ideas can be communicated unambigously to both implementor and user.

The description of any computer language can be divided into three parts—the syntax, the static semantics and the semantics.

Syntax rules are rules which tell one the structure of a sentence or command in a language. There are two basic approaches to the specification of syntax rules, viz.

(a) *The top-down approach.* This is the approach used in grammars and in BNF, in which *nonterminals* or *metalinguistic variables* or *syntactic categories* are successively broken down into final terminal symbols.
(b) *The bottom-up approach.* This is the approach used in canonic systems in which strings are tested to see which predicates they satisfy and are successively built up into a program.

Semantic rules are rules which describe the effect which a particular syntactic construction will have. There are various approaches to the formal specification of semantics, for example, the Vienna Definition Language [2], Hoare's axiomatic approach [3], Scott and Strachey's Mathematical semantics [4], etc.

Static semantic rules (or context-sensitive rules, as they are sometimes called) lie somewhere between syntax rules and semantic rules, and are concerned primarily with such things as *relating* the occurrence of an identifier at one point in a program (in a declaration, say) with other occurrences of the same identifier elsewhere in the program (e.g. in statements), and *placing restrictions* on these occurrences (for example, if an identifier is declared as a 2-D array one would not expect it to be used with a single subscript unless the language permits this under certain circumstances). Some typical static semantic rules are given in Table 1.

* Present address: Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ, Scotland.

Table 1

BASIC

(1) *ARRAYS* The same letter may be used both as a simple variable and as a 1- or 2-D array. An array is declared either explicitly in a DIM statement or implicitly by its first appearance in the program. In the case of explicit declaration all references to the array (subscripted variables) must have the same number of subscripts as there are dimensions in the declaration; if declaration is implicit all references to the array must have the same number of subscripts as the first reference to the array. A DIM statement can occur anywhere in a program.
(2) *SCOPE OF ARRAY NAMES* The same array name cannot be used to apply to more than one array in a program.
(3) *PARAMETERS* If a function is called in a program, the actual parameters supplied must correspond in number, type and order to the formal parameters in the function definition.
(4) *SCOPE OF FUNCTION NAME* A function name may be used as a destination for a LET statement only within the definition block for that function. If a user function is called in a program, the function must be defined somewhere in the program.

ALGOL60

(1) *UNIQUENESS OF IDENTIFIERS* An identifier may not be declared more than once in any given block, although it may be redeclared to be something different within an inner block. Thus the same identifier may be used to refer to different objects in different parts of the program.
(2) *PARAMETERS* In a function call the types of the actual parameters must match the types of the formal parameters (if they have been declared).
(3) *VALUE DECLARATION* Any identifiers appearing in a *value* clause must also appear in the formal parameter list immediately preceding the value clause.

The formal specification of static semantic rules may be linked to a formal specification of syntax rules or to a formal specification of semantics. This paper looks at the prerequisites of a formal notation for specifying static semantics and then considers several different formal notations and the ways in which they satisfy these prerequisites. From this one may draw certain conclusions about the trends in the development towards a generally acceptable formal notation.

PREREQUISITES OF A FORMAL NOTATION FOR SPECIFYING STATIC SEMANTICS

What does one require of a formal notation for specifying static sematics? The following might be regarded as essential components for any such notation:

(1) The basic elements which the notation must cater for are as follows:
  (a) It must allow reference to the strings associated with nonterminals or metalinguistic variables (e.g. the string associated with ⟨**IDENTIFIER**⟩).
  (b) It must permit some form of tag or category information, such as modes (e.g. *SUM* is an INTEGER VARIABLE, *X* is a REAL VARIABLE, *Y* is a FORMAL PARAMETER OF TYPE REAL, etc.).
  (c) It must have a form of counting mechanism and counters (e.g. to count the number of dimensions of an array or subscripts of a subscripted variable, etc.).
(2) The notation must provide some mechanism whereby these basic elements can be combined into lists, e.g. one requires a list of all identifiers declared.
(3) It must provide a mechanism for making comparisons between individual basic elements and lists, e.g. to check whether a particular identifier occurs in a list or whether a count does not exceed some constant value.
(4) Finally, the notation may provide some mechanism of generating *intermediate language code* so that the syntax and static semantic specification can be regarded as a unit producing intermediate language code for a code generator or an interpreter.

The different formal notations which will be considered here, all have different ways of expressing these elements. However, it is not the rigorous formats of the notations which is of concern here but the underlying techniques which are used to specify this type of rule.

```
1.   <PROGRAM>::=<DEC SEQUENCE><STM SEQUENCE>
2.   <DEC SEQUENCE>::=<MODE><IDLIST>;
                      |<MODE><IDLIST>;<DEC SEQUENCE>
3.   <IDLIST>::=<ID>|<ID>,<IDLIST>
4.   <STM SEQUENCE>::=<STM>|<STM>;<STM SEQUENCE>
5.   <STM>::=<ASSIGN STM>
6.   <ASSIGN STM>::=<ID>=<EXP>
7.   <EXP>::=<ID>|<ID>+<EXP>|<ID>-<EXP>
8.   <ID>::=<LETTER>|<LETTER><ID>
9.   <LETTER>::=A|B|C|....|Z
10.  <MODE>::=REAL|INTEGER|BOOLEAN
```

Fig. 1. BNF definition of a simple grammar.

In order to compare the different notations, consider a simple language in which sentences consist of a sequence of real, integer or Boolean declarations followed by a sequence of assignment statements. A grammar for this language is given in Fig. 1. The following is a typical sentence of the language:

REAL A,B; BOOLEAN BOOL,B1,B2; A = B + A.

To this language we will add the following two static semantic restrictions (which are typical of most static semantic rules):

(1) All identifiers used in assignment statements must be declared in declarations;
(2) Identifiers may consist of at most six characters.

The formal notations for specifying static semantics will be classified into two categories: the earlier notations in which the nonterminals are generalized, and the more recent notations in which actions are appended to productions without affecting the basic nonterminals.

NOTATIONS WITH GENERALIZED NONTERMINALS

(A) *Ledgard's canonic systems*

The first notation to cater for static semantic rules was that of canonic systems, developed by Donovan and Ledgard [5–7] based on work of Post and of Smullyan.

Whereas in a grammar nonterminals are used to represent classes of strings which are generated by the application of a number of productions, in canonic systems *variables* are used to represent strings, and these variables together with terminal symbols are classified according to the *predicates* which they satisfy.

Instead of having a set of productions, a canonic system has a set of *canons*. A canon is a logical rule which states that a certain set of premises (zero or more) imply a certain conclusion, written as

$$a; b; \ldots ; n \vdash z$$

where a,b, ... n are premises and z is a conclusion. Each premise and conclusion consists of a string of concatenated variables and terminal symbols followed by a predicate symbol (the name of a class of objects).

For example, in a grammar with productions

⟨**DIGIT**⟩ :: = 0|1
⟨**BINARYNO**⟩ :: = ⟨**DIGIT**⟩|⟨**BINARYNO**⟩⟨**DIGIT**⟩

the nonterminal ⟨**DIGIT**⟩ may represent either the string "0" or the string "1" while the nonterminal ⟨**BINARYNO**⟩ represents all possible strings which consist of a series of one or more zeroes and ones.

In a canonic system the same effect is achieved by the following set of canons

$\vdash$ 0 *digit* which reads "The symbol 0 satisfies the predicate *digit*" (or more simply "0 is a *digit*").

$\vdash$ 1 *digit* "The symbol 1 satisfies the predicate *digit*".

x *digit* $\vdash$ x *binaryno* "If the variable x represents a string which satisfies the predicate *digit* (premise) then the same string satisfies the predicate *binaryno* (conclusion)".

x *digit*; y *binaryno* $\vdash$ xy *binaryno* "If the variable x represents a string which satisfies the predicate *digit* and variable y represents a string which satisfies *binaryno*, then the string represented by x followed by the string represented by y must satisfy the predicate *binaryno* (or: 'If x is a digit and y a binaryno then xy is a binaryno')".

From this it is clear that only the character strings "0" and "1" satisfy the predicate *digit* while any string of one or more zeros and ones will satisfy the predicate *binaryno*.

Just as in BNF productions with the same left-hand side are combined into a single production, e.g.

$$\langle \textbf{DIGIT} \rangle ::= 0 \mid 1$$

in canonic systems, canons with the same set of premises and conclusion predicate may be abbreviated as follows

$$\left.\begin{array}{l} P_1; P_2; \ldots; P_n \vdash \alpha_1 \mathbf{p} \\ P_1; P_2; \ldots; P_n \vdash \alpha_2 \mathbf{p} \\ — \\ — \\ — \\ P_1; P_2; \ldots; P_n \vdash \alpha_m \mathbf{p} \end{array}\right\} \text{ is abbreviated as } \quad P_1; P_2; \ldots; P_n \vdash \alpha_1 + \alpha_2 + \cdots + \alpha_m \mathbf{p}.$$

Thus in the above example one could have written

$$\vdash 0 + 1 \textit{ digit}$$

which reads: "0 or 1 is a digit". To illustrate the canonic system approach, the grammar given in Fig. 1 has been written as a set of canons in Fig. 2.

In order to cater for static semantic rules the idea of a variable which satisfies a predicate is extended to that of an *n*-tuple which satisfies a predicate. The notation $\langle x_1 < x_2 < \ldots < x_n \rangle$ is used to denote such an *n*-tuple.

1.		$\vdash A + B + \ldots + Z$ letter
2.		$\vdash$ REAL + INTEGER + BOOLEAN mode
3.	x letter	$\vdash$ x id
4.	x letter; y id	$\vdash$ xy id
5.	x id	$\vdash$ x exp
6.	x id; y exp	$\vdash$ x+y + x-y exp
7.	x id; y exp	$\vdash$ x=y assignstm
8.	x assign stm	$\vdash$ x stm
9.	x stm	$\vdash$ x stmsequence
10.	x stm; y stmsequence	$\vdash$ x; y stmsequence
11.	x id	$\vdash$ x idlist
12.	x id; y idlist	$\vdash$ x,y idlist
13.	x idlist; y mode	$\vdash$ yx; decsequence
14.	x idlist; y decsequence; z mode	$\vdash$ zx;y decsequence
15.	x decsequence; y stmsequence	$\vdash$ xy program

Fig. 2. Canonic system equivalent of grammar in Fig. 1.

One component of such an *n*-tuple represents the terminal string recognized or produced thus far; the remaining components consist of lists of identifiers to be used in static semantic checks.

In this example one might collect a list of identifiers declared as one constructs the declaration sequence by changing the canons as follows:

11. x *id* $\vdash$ ⟨ x < x ⟩ *idlistandsymbolsdeclared*
12. x *id*; ⟨ y < i ⟩ *idlistandsymbolsdeclared* $\vdash$ ⟨ x, y < x, i ⟩ *idlistandsymbolsdeclared*
13. ⟨ x < i ⟩ *idlistandsymbolsdeclared*; y *mode* $\vdash$ ⟨ yx; < i ⟩ *decseqandsymdecl*
14. ⟨ x < i ⟩ *idlistandsymbolsdeclared*; ⟨ y < j ⟩ *decseqandsymdecl*; z *mode* $\vdash$ ⟨ zx; y < i, j ⟩ *decseqandsymdecl.*

Thus rule 11 now reads: if x is an identifier then a pair can be formed consisting of x an idlist and x a symboldeclared. Rule 12 states that if one has an identifier x and a pair consisting of an idlist y and a list of symbols declared i, then x can be combined with y (x, y) to form a new idlist while x is added to i to form an augmented list of declared symbols. In rule 13 an idlist (x) can be combined with a mode (y) to produce a declaration sequence (yx;) with the same list of declared symbols. In rule 14 further declarations are added to the declaration sequence and their accompanying lists appended to the list of declared symbols.

In like fashion one might construct a list of the identifiers which are used in the statement sequence as follows:

5. x *id* $\vdash$ ⟨ x < x ⟩ *expandsymbols*
6. x *id*; ⟨ y < i ⟩ *expandsymbols* $\vdash$ ⟨ x + y < x, i ⟩ + ⟨ x − y < x, i ⟩ *expandsymbols*
7. x *id*; ⟨ y < i ⟩ *expandsymbols* $\vdash$ ⟨ x = y < x, i ⟩ *assignstmandsymbols*
8. ⟨ x < i ⟩ *assignstmandsymbols* $\vdash$ ⟨ x < i ⟩ *stmandsymbols*
9. ⟨ x < i ⟩ *stmandsymbols* $\vdash$ ⟨ x < i ⟩ *stmsequenceandsymbols*
10. ⟨ x < i ⟩ *stmandsymbols*; ⟨ y < j ⟩ *stmsequenceandsymbols* $\vdash$ ⟨ x; y < i, j ⟩ *stmsequenceandsymbols.*

Rule 5 states that if x is an identifier then it can be taken to be an expression and a one symbol list of the identifiers used in the expression. In rule 6 if x is an identifier, y an expression with i the list of symbols used in y, then x + y or x − y forms an expression with x,i as the list of identifiers used in this extended expression. Likewise if x is an identifier and y an expression with i the list of symbols used in y, then x = y is an assignment statement and x,i is the list of identifiers used in it (rule 7). This list of identifiers is passed to *stmsequence* (rules 8 and 9) and added to other lists (rule 10) to create a statement sequence with lists of symbols used therein.

Finally the predicate *in* is used to test for containment as follows:

15. ⟨ x < i ⟩ *decseqandsymdecl*; ⟨ y < j ⟩ *stmsequenceandsymbols*; ⟨ j < i ⟩ *in* $\vdash$ xy *program.*

This states that if x is a declaration sequence with a list i of symbols declared and y is a statement sequence with list j of symbols used in the statement sequence and if each element of the list j is contained in the list i then x followed by y constitutes a valid program. One can extend this to cater for different types of declarations by having different lists in the *n*-tuple.

Counting can be performed in a crude manner by concatenation. Thus one way of handling the second static semantic restriction might be as follows:

3. x *letter* $\vdash$ ⟨ x < 1 ⟩ *partidwithcount*
4. x *letter*; ⟨ y < i ⟩ *partidwithcount* $\vdash$ ⟨ xy < li ⟩ *partidwithcount*
4A. ⟨ x < i ⟩ *partidwithcount*; ⟨ i < 1, 11, 111, 1111, 11111, 111111 ⟩ *in* $\vdash$ x *id.*

(B) *Two-level grammars*

The second notation to appear did so soon after canonic systems. This was van Wijngaarden's two-level grammars or W-grammars [8–10].

In an ordinary (one-level) grammar, one has a fixed set of terminal symbols, a fixed set of nonterminal symbols and a fixed set of productions. However, van Wijngaarden suggested the idea that the nonterminal symbols need not necessarily be fixed but may themselves be variable strings, which are in turn controlled by a set of productions at a higher level (metarules). For example in a one-level grammar one might have:

⟨assignment stm⟩::=⟨numeric variable⟩=⟨numeric expression⟩
|⟨boolean variable⟩=⟨boolean expression⟩

In a two-level grammar one might write this as

Metasyntax
TYPE:: boolean; numeric.
Syntax
assignment stm: TYPE variable, equals symbol, TYPE expression.

In van Wijngaarden's notation for metasyntax, the nonterminals of the metasyntax (called metanotions) are written in upper case letters, the terminals of the metasyntax which form part of the nonterminals of the syntax are written in lower case letters, "::" is used in place of "::=", ";" in place of vertical bar "|" and "." is used to mark the end of a production. The notation for syntax rules is similar using ":" instead of "::=", and "," and "." as before. In BNF if a nonterminal or terminal is to be followed by another, they are simply juxtaposed—the same is true in the metasyntax of two-level grammars. However, at the syntax level the symbol "," is used as a separator between nonterminals and/or terminals.

Upper case symbols occurring in syntax rules must be expanded using metarules to obtain the complete set of productions. Thus in the case of the above example, the syntax would be expanded to produce

assignment stm: boolean variable, equals symbol, boolean expression.
assignment stm: numeric variable, equals symbol, numeric expression.

Note that in substituting for the metanotion TYPE in the syntax rule, the same string may be substituted wherever TYPE occurs (uniform replacement rule). Expansion of the syntax rules may sometimes lead to an infinity of productions, for example

Metasyntax
TYPE:: ARITHMETIC; boolean.
ARITHMETIC:: numeric; long ARITHMETIC.
Syntax
assignment stm: TYPE variable, equals symbol, TYPE expression.

On expansion this will yield:

assignment stm: boolean variable, equals symbol, boolean expression;
numeric variable, equals symbol, numeric expression;
long numeric variable, equals symbol, long numeric expression;
—
—,

Applying this to the example which we have been considering, the two-level grammar equivalent of the grammar in Fig. 1 is given in Fig. 3.

The idea of having variable nonterminal names can be used to implement static sematic rules by letting the nonterminals carry around with them lists of identifiers in the same way as n-tuples do in canonic systems. Consider how this applies in the case of our example. First one must modify the specification to accumulate a list of all identifiers which are declared. This may be done as follows:

```
METASYNTAX

M1.  LETTER::a;b;...;z.

M2.  MODE::real;integer;boolean.

SYNTAX

S1.  id : LETTER symbol;LETTER symbol,id.

S2.  idlist : id; id,comma symbol, idlist.

S3.  dec sequence : MODE symbol,idlist,semicolon symbol;
                    MODE symbol,idlist,semicolon symbol,dec sequence.

S4.  exp : id;id,plus symbol,exp;id, minus symbol,exp.

S5.  assign stm : id,equals symbol,exp.

S6.  stm : assign stm.

S7.  stm sequence : stm;stm,semicolon symbol,stm sequence.

S8.  program : dec sequence,stm sequence.
```

Fig. 3. Two-level grammar equivalent of grammar in Fig. 1.

Metasyntax
M3. **VAR::LETTER; LETTER VAR.**
M4. **VARLIST::VAR; VAR VARLIST.**
M5. **TABLE::VARLIST; VARLIST TABLE.**

Syntax
S1. **VAR id:LETTER symbol; LETTER symbol, VAR id.**
S2. **VARLIST idlist:VAR id; VAR id, comma symbol, VARLIST idlist.**
S3. **dec sequence forming TABLE:MODE symbol, VARLIST idlist, semicolon symbol;**
MODE symbol, VARLIST idlist, semicolon symbol, dec sequence forming TABLE.
S8. **program:dec sequence forming TABLE, stm sequence.**

In this case the name of each identifier becomes part of the nonterminal for that identifier (for example, an **a id** produces the identifier A, an **xyz id** produces the terminal string XYZ, etc.). These names are accumulated into lists in S2, and the lists are merged into a table in S3. To illustrate this consider the parse tree obtained when the string

REAL A,B,XYZ;

is derived. This is shown in Fig. 4.

Thus at each stage the nonterminals carry around with them the list of identifiers required. This list is gradually built up as one moves up the parse tree towards the root. The list can then be passed across to the nonterminal **stm sequence** and from there pass down the parse tree to where it is required. The productions for this are as follows:

Syntax
S8. **program:dec sequence forming TABLE, stm sequence using TABLE.**
S7. **stm sequence using TABLE:stm using TABLE;**
stm using TABLE, semicolon symbol, stm sequence using TABLE.
S6. **stm using TABLE:assign stm using TABLE.**
S5. **assign stm using TABLE:VAR id, equals symbol, exp using TABLE.**
S4. **exp using TABLE:VAR id; VAR id, plus symbol, exp using TABLE;**
VAR id, minus symbol, exp using TABLE.

A parse of the string

REAL A,B,XYZ; A = B + XYZ

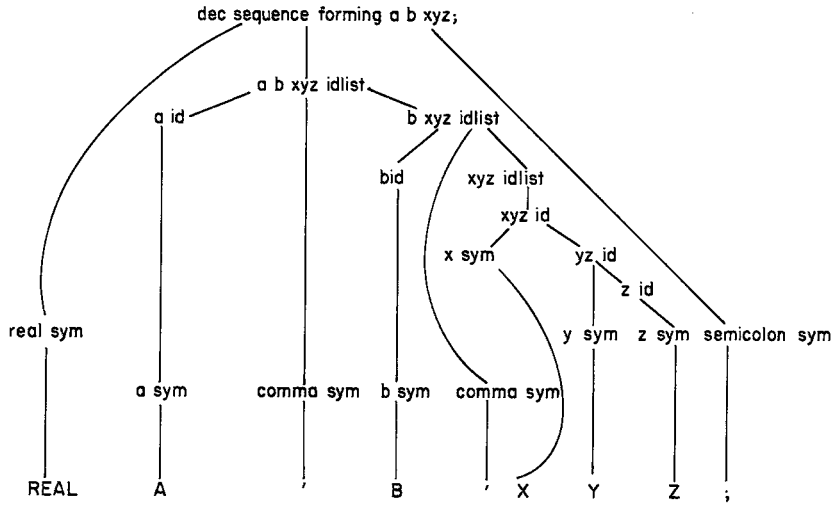


Fig. 4. Portion of parse tree for the string REAL A,B,XYZ.

using this revised set of productions, is given in Fig. 5. This illustrates how the list of identifiers may pass up and down the parse tree as required. Finally one needs to add the static semantic checks. This is done by introducing additional nonterminals into the productions. These nonterminals which represent the static semantic restrictions reduce to the null string if the conditions are satisfied and hence disappear from the parse. In this case one might have:

Metasyntax

M6. **VARSETY::VARLIST; EMPTY.**
M7. **VARSETY1::VARSETY.**
M8. **EMPTY::.**

Syntax

S5. **assign stm using TABLE:VAR id, equals symbol, exp using TABLE, where VAR is contained in TABLE.**
S4. **exp using TABLE:VAR id, where VAR is contained in TABLE; VAR id, plus symbol, exp using TABLE, where VAR is contained in TABLE; VAR id, minus symbol, exp using TABLE, where VAR is contained in TABLE.**
S9. **where VAR is contained in VARSETY VAR VARSETY1:EMPTY.**

The nonterminal **where VAR is contained in TABLE** is responsible for checking that the identifier is contained in the table of declared identifiers passed down the parse tree. If the identifier (represented by **VAR**) is in the table (i.e. the table consists of 0 or more

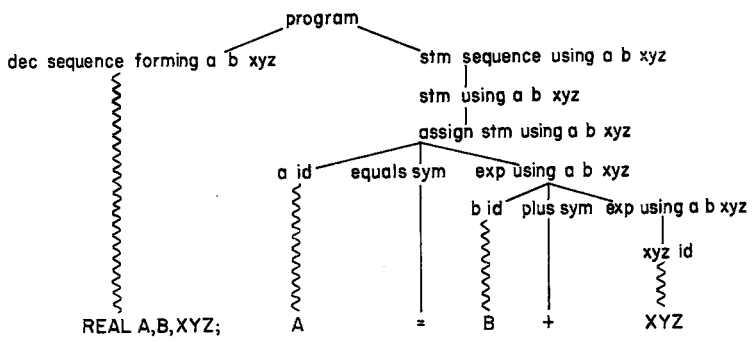


Fig. 5. Parse tree for the strong REAL, A,B,XYZ; A = B + XYZ.

variables followed by **VAR** followed by 0 or more other variables) then this nonterminal reduces to the null string. If the static semantic restriction is not satisfied, the nonterminal does not disappear and hence the parse cannot be completed.

The second static semantic restriction requiring identifiers to be $\leqslant 6$ characters in length might be implemented as follows:

Metasyntax

M9. **MAXLEN::LETTER; LETTER LETTER; LETTER LETTER LETTER;**
LETTER LETTER LETTER LETTER;
LETTER LETTER LETTER LETTER LETTER;
LETTER LETTER LETTER LETTER LETTER LETTER.

Syntax

S1. **VAR partid:LETTER symbol; LETTER symbol, VAR partid.**
S1A. **VAR id:VAR partid, where VAR is contained in MAXLEN.**
S1B. **where VAR is contained in VAR:EMPTY.**

NOTATIONS IN WHICH ACTIONS ARE APPENDED TO PRODUCTIONS

In these notations the basic productions for the context free syntax have the static semantic restrictions appended to them in a fashion which does not tamper with the form of the basic productions as is the case for the notations of the previous section. They also contain more facilities resulting in shorter and clearer definitions.

(A) *Attribute grammars*

Attribute grammars were originally proposed by Knuth [11]. Since then they have been described by various workers each with their own slight variations in notation [12–16]. An attribute grammar is based on a BNF specification of the syntax. However, with each occurrence of a nonterminal on the left hand side or right hand side of a production is associated a set of attributes. The values of these attributes may pass down the parse tree from parent node to child, in which case they are called *inherited attributes* or they may pass up the tree from immediate descendants to their parents, in which case they are referred to as *synthesized attributes*. These attributes include such things as modes, lists of identifiers, etc.

The passage of attributes up and down the parse tree is not unlike the passage of similar attributes from nonterminal to nonterminal in a two-level grammar. The chief difference is that in two-level grammars it is not clear under which circumstances attributes should move up or down the parse tree whereas in an attribute grammar, synthesized attributes are prefixed by an arrow pointing upward, inherited attributes by an arrow pointing downward.

In an attribute grammar each symbol on the right hand side of a production will be written on a new line and followed by a sequence of zero or more attributes (each preceded by $\uparrow$ or $\downarrow$). On intervening or succeeding lines one may have *conditions*, corresponding to the syntactic restrictions, and *calls to functions*, to construct lists, etc. For example

⟨**PROGRAM**⟩::= ⟨**DEC SEQUENCE**⟩$\downarrow$ *empty-symbol-table*$\uparrow$**symbol-table**
⟨**STM SEQUENCE**⟩$\downarrow$**symbol-table**

states that the constant *empty-symbol-table* (corresponding to an empty list) must be passed down to the descendants of ⟨**DEC SEQUENCE**⟩, which will in turn pass back the attribute symbol-table. This attribute is carried forward to the nonterminal ⟨**STM SEQUENCE**⟩ and passed down to its descendants.

Conditions take the form of any boolean expression which may include relations involving numeric comparisons (=, >, <, etc.) or the set inclusion operator ($\in$), while function calls define the actions to be taken. These include the construction of sets or

```
<PROGRAM>::=<DEC SEQUENCE>↓empty-symbol-table ↑ symbol-table
            <STM SEQUENCE>↓symbol-table
<DEC SEQUENCE>↓symtable1↑symtable2::=<MODE>
              <IDLIST>↓symtable1↑symtable2
              ;
              |<MODE>
              <IDLIST>↓symtable1↑symtable3
              ;
              <DEC SEQUENCE>↓symtable3↑symtable2
<IDLIST>↓symtable1↑symtable2::=
              <DECLARED ID>↓symtable1↑symtable2
              |<DECLARED ID>↓symtable1↑symtable3
              ,
              <IDLIST>↓symtable3↑symtable2
<DECLARED ID>↓symtable1↑symtable2::=<ID>↑name
              insert declaration↓symtable1↓name↑symtable2
<ID>↑name1::=<LETTER>↑name1
              |<ID>↑name2
              <LETTER>↑name3
              concatenate↓name2↓name3↑name1
<LETTER>↑name::=A
              give value to attribute↓A↑name
              |B
              .
              .
              .
              |Z
              give value to attribute↓Z↑name
<MODE>↑::=REAL
              |INTEGER
              |BOOLEAN
insert declaration↓symtable1↓name↑symtable2=>symtable2=symtable1∪{(name)}
concatenate↓name1↓name2↑name3=>name3is the concatenation of name1with
              name2
```

Fig. 6. Part of attribute grammar setting up symbol-table.

lists. etc. However, the notation for specifying functions is not rigorously defined and sometimes a verbal description of the effect of a function is given in place of a formal specification.

The productions necessary for creating the list of declared identifiers are given in Fig. 6. The nonterminal ⟨**DEC SEQUENCE**⟩ has passed to it an initial symbol table and passes back an updated symbol table. In the first production, the initial symbol table ($symtable_1$) is passed to ⟨**IDLIST**⟩ which passes it down to its descendants and returns with the updated symbol table ($symtable_2$). In the second production, the initial symbol table is passed to ⟨**IDLIST**⟩ which returns an intermediate symbol table ($symtable_3$); this in turn is passed to ⟨**DEC SEQUENCE**⟩ which returns the final updated symbol table ($symtable_2$). In the same way ⟨**IDLIST**⟩ passes an initial symbol table to ⟨**DECLARED ID**⟩ and receives an updated symbol table in return. ⟨**DECLARED ID**⟩ calls the function ***insert declaration*** to insert a new name into the symbol table. The name itself is constructed by concatenating letters in ⟨**ID**⟩. The primitive function ***give value to attribute*** is used to assign terminal values to attributes.

Having created a list of identifiers these are passed via the nonterminal ⟨**STM SEQUENCE**⟩ to its descendants and used to verify that every identifier which is used has been declared, as shown in Fig. 7.

To cater for counting a simple arithmetic assignment statement is used. For example, if one adds the restriction that identifiers must be less than seven characters in length to this specification, the only changes are shown in Fig. 8.

```
<STM SEQUENCE>↓symtable::=<STM>↓symtable
                |<STM>↓symtable
                ;
                <STM SEQUENCE>↓symtable
<STM>↓symtable::=<ASSIGN STM>↓symtable
<ASSIGN STM>↓symtable::=<USED ID>↓symtable
                :=
                <EXP>↓symtable
<EXP>↓symtable::=<USED ID>↓symtable
                |<USED ID>↓symtable
                +
                <EXP>↓symtable
                |<USED ID>↓symtable
                -
                <EXP>↓symtable
<USED ID>↓symtable::=<ID>↑name
                condition:(name)∈symtable
```

Fig. 7. Part of attribute grammar checking that all symbols used are declared.

(B) *Production systems*

Ledgard, who with Donovan was responsible for adapting canonic systems to cater for static semantic rules, continued work on formal specification of static semantic rules, resulting in his Production Systems [16, 17].

A Production System is a generative grammar similar to BNF. The basic productions of the grammar without static semantic rules have a format in which each nonterminal is given an abbreviated form (in small letters) for use on the right hand side of productions and a full title which describes the nonterminal (in capital letters) in the production for that nonterminal. The grammar given in Fig. 1 is written as a Production System without static semantic rules in Fig. 9.

In order to add static semantic restrictions to this, some form of list is required. In this notation there is a single list called the environment. This is a symbol table containing all identifiers declared thus far together with their types. For example

REAL A,B; BOOLEAN XYZ;

```
<ID>↑name::=<IDENTIFIER>↑name↑noletters
            condition:noletters<7
<IDENTIFIER>↑name1↑noletters1::=<LETTER>↑name1
            give value to attribute↓1↑noletters1
            |<IDENTIFIER>↑name2↑noletters2
            <LETTER>↑name3
            concatenate↓name2↓name3↑name1
            add one letter↓noletters2↑noletters1
addoneletter↓noletters1↑noletters2=> noletters2=noletters1+ 1
```

Fig. 8. Part of attribute grammar checking that the length of an identifier does not exceed 6 characters.

```
[Main Productions]
[1]  prog  PROGRAM  <ds ss>
[2]  ds    DECLARATION SEQ   <dec1;...;decn;>
[3]  dec   DECLARATION <m i1,...,in>
[4]  ss    STATEMENT SEQ <s1;...;sn>
[5]  s     STATEMENT   <as>
[6]  as    ASSIGNMENT STATEMENT   <id=exp>
[7]  exp   EXPRESSION  <id |id + exp|id-exp>
[8]  id    IDENTIFIER   <l1...ln>
[9]  i     DECLARED IDENTIFIER  <l1...ln>
[10] l     LETTER  <A|B|C|...|Z>
[11] m     MODE  <REAL|INTEGER|BOOLEAN>
```

Fig. 9. Production System equivalent of Grammar in Fig. 1.

might correspond to the environment

$$\rho \equiv \{A \rightarrow \text{REAL},\ B \rightarrow \text{REAL},\ XYZ \rightarrow \text{BOOLEAN}\}.$$

Now instead of associating lists of identifiers with nonterminals and passing them from node to node of the parse tree, we have a single global list ρ which may be referenced by all productions.

The static semantic restrictions are added in the form of a sequence of steps separated from each other by the symbol "&" and from the syntax rule by "←", i.e.

$$\text{syntax rule} \leftarrow \text{step}_1 \ \& \ \text{step}_2 \ \& \ \ldots \ \& \ \text{step}_n.$$

Each step may be one of:

(a) An assignment (of form: destination ≡ expression).
(b) A numeric comparison (e.g. $n_1 \geqslant n_2$).
(c) A test for containment (of form: $x \in y$).
(d) A function call.

Any functions called must be defined using further function calls, assignments, comparisons and tests for containment.

Using this one may adapt the example in Fig. 9 to set up the environment as follows:

[*Main Productions*]
[1] **prog PROGRAM** ⟨**ds ss**⟩ ← $\rho \equiv$ ***ENVCREATEDBY*(ds)**

[*Auxiliary Functions*]
[12] ***ENVCREATEDBY*(dec$_1$; ... ; dec$_n$)**
$\equiv \{$***ENVCREATEDBY*(dec$_1$), ..., *ENVCREATEDBY*(dec$_n$)$\}$**
[13] ***ENVCREATEDBY*(m i$_1$, ..., i$_n$)**
$\equiv$ **i$_1$ → dm, ..., i$_n$ → dm ← dm ≡ *DERIVEDMODEOF*(m)**
[14] ***DERIVEDMODEOF*** (REAL) ≡ **REAL**
[15] ***DERIVEDMODEOF*** (INTEGER) ≡ **INTEGER**
[16] ***DERIVEDMODEOF*** (BOOLEAN) ≡ **BOOLEAN.**

Rule 12 states that the environment created by the sequence of declarations **dec$_1$; ... ; dec$_n$** consists of a list of environments created by the individual declarations. Rule 13 states that the environment created by a single declaration of form

mode identifier$_1$, ..., identifier$_n$

consists of a list of items of form

identifier$_i$ → dm $\quad i = 1, \ldots, n$

where **dm** is the ***DERIVED MODE OF*** (**m**). In rules 14–16 the metalinguistic variable for mode is mapped onto one of the constant strings **REAL, INTEGER** or **BOOLEAN.** Hence in rule 1 the environment ρ is set up to be the list of identifiers declared together with their associated modes.

If the additional restriction is to be applied requiring that all identifiers used in the statement sequence should be declared in the declaration sequence, the productions are augmented as follows:

[*Main Productions*]
[8] **id IDENTIFIER** $\langle \mathbf{l}_1 \ldots \mathbf{l}_n \rangle \leftarrow \mathbf{l}_1 \ldots \mathbf{l}_n \in$ ***DOMAIN***(ρ)
[*Auxiliary Functions*]
[17] ***DOMAIN*** $(\{\mathbf{id}_1 \rightarrow \mathbf{dm}_1, \ldots, \mathbf{id}_n \rightarrow \mathbf{dm}_n\}) \equiv \mathbf{id}_1, \ldots, \mathbf{id}_n$.

The check in rule 8 is a test that the identifier $\mathbf{l}_1 \ldots \mathbf{l}_n$ is contained in the list produced by the function ***DOMAIN***. The ***DOMAIN*** of an environment ρ is the list of identifiers occurring in ρ.

Counting is simplified in this notation by virtue of the change in the form of productions in which, instead of writing

$$\mathbf{id\ IDENTIFIER}\ \langle \mathbf{l} | \mathbf{l\,id} \rangle$$

one can write

$$\mathbf{id\ IDENTIFIER}\ \langle \mathbf{l}_1 \ldots \mathbf{l}_n \rangle$$

i.e. an identifier is a sequence of n letters. Thus the count of number of letters is implicit in the production. The addition of the restriction that an identifier must consist of at most 6 characters, results in the following modification to rule 8:

[8] **id IDENTIFIER** $\langle \mathbf{l}_1 \ldots \mathbf{l}_n \rangle \leftarrow \mathbf{l}_1 \ldots \mathbf{l}_n \in$ ***DOMAIN***(ρ) & $n \leqslant 6$.

(C) *Extended BNF*

The final notation considered here is that of Extended BNF [18, 19]. Once again BNF is used as the basis of this notation and static semantic actions are appended to the BNF productions in the form of a syntax directed translation schema (SDTS). Each static semantic action consists of a sequence of one or more steps separated by semicolons and enclosed within braces, e.g.

$$\langle \mathbf{PROGRAM} \rangle ::= \langle \mathbf{DEC\ SEQUENCE} \rangle \langle \mathbf{STM\ SEQUENCE} \rangle \{\mathbf{S}_1; \mathbf{S}_2; \ldots; \mathbf{S}_n\}.$$

The data structures used in the notation are:
(a) Strings—these can be either literal strings (e.g. "INT"), metalinguistic variables or the null string (λ)
(b) Stacks—each stack is represented by an upper case letter name. An element on a stack may have one or more components, each of which may be a string or another stack.
(c) System Variables—each system variable contains a string or an element of a stack and is represented by a lower case letter name.

The basic operations which can be performed include:

(1) An item may be added to a stack, denoted by: **item** $\downarrow$ **stack**
(2) An item may be removed from the top of a stack, denoted by: **destination** $\uparrow$ **stack**
(3) A stack may be searched for a given item, denoted by:

$$\mathscr{S}\mathbf{(stack, item, success\ action, fail\ action)}$$

If the item is found in the stack the success action is performed; otherwise the fail action is performed.
(4) Conditional actions have the form

if **condition** ***then*** **action** ***fi***
or ***if*** **condition** ***then*** **action** ***else*** **action** ***fi.***

Conditions are simple relations—string comparisons—possibly connected by Boolean connectives.

(5) A system variable or component of a system variable or stack item may be altered (denoted by: **new contents → sys var** or **new contents → component**).

(6) A whole stack may be unpacked and transferred to some destination, denoted by: **destination ⇑ stack**

(7) Errors are denoted by E*n* where *n* is the number of the error message.

(8) The null action: —

(9) A function call.

The notation also includes several stack functions, including $\mathcal{N}$(⟨**STACK NAME**⟩) which denotes the number of items in the given stack.

Using this notation the specification of Fig. 1 can be altered to cater for the accumulation of identifiers into a symbol table ST as follows:

3. ⟨**ID LIST**⟩ :: = ⟨**ID**⟩{⟨**ID**⟩↓**ST**}
|⟨**ID**⟩, ⟨**ID LIST**⟩{⟨**ID**⟩↓**ST**}.

To check that all identifiers used are declared:

6. ⟨**ASSIGN STM**⟩ :: = ⟨**ID**⟩ = ⟨**EXP**⟩{***checkiddefined***}
7. ⟨**EXP**⟩ :: = ⟨**ID**⟩{***checkiddefined***}|⟨**ID**⟩ + ⟨**EXP**⟩{***checkiddefined***}|
⟨**ID**⟩ − ⟨**EXP**⟩{***checkiddefined***}

where

$$checkiddefined = \mathcal{S}(\mathbf{ST}, \langle \mathbf{ID} \rangle, -, \mathbf{E1}).$$

```
1.  <PROGRAM>::=<DEC SEQUENCE><STM SEQUENCE>
2.  <DEC SEQUENCE>::=<MODE><IDLIST>;
                    |<MODE><IDLIST>;<DEC SEQUENCE>
3.  <IDLIST>::=<ID>{<ID>↓ST}|<ID>,<IDLIST>{<ID>↓ST}
4.  <STM SEQUENCE>::=<STM>|<STM>;<STM SEQUENCE>
5.  <STM>::=<ASSIGN STM>
6.  <ASSIGN STM>::=<ID>=<EXP>{checkiddefined}
7.  <EXP>::=<ID>{checkiddefined}|<ID>+<EXP> {checkiddefined}
          |<ID>-<EXP>{checkiddefined}
8.  <ID>::=<PARTID>{if N(IDCOUNT)>6 then E2 fi ;λ ⇑IDCOUNT}
8A. <PARTID>::=<LETTER>{addonetocount}
            |<LETTER><PARTID>{addonetocount}
9.  <LETTER>::=A|B|C|...|Z
10. <MODE>::=REAL|INTEGER|BOOLEAN

checkiddefined = S(ST,<ID>,-,E1)
addonetocount  = "1"↓IDCOUNT

E1 = Identifier not declared
E2 = Identifier too long
```

Fig. 10. Extended BNF equivalent of grammar in Fig. 1 with static semantic restrictions added.

This function searches the symbol table ST for an occurrence of ⟨ID⟩. If it finds such an occurrence it performs the success action which is null; if not, it performs the fail action and signals an error (error no. 1).

Finally to check that an identifier consists of less than seven characters, one may have:

8. ⟨**ID**⟩ : : = ⟨**PARTID**⟩{*if* $\mathcal{N}$(**IDCOUNT**) > **6** *then* **E2** *fi*; $\lambda \Uparrow$ **IDCOUNT**}
8A. ⟨**PARTID**⟩ : : = ⟨**LETTER**⟩{"**1**" ↓ **IDCOUNT**}
| ⟨**LETTER**⟩⟨**PARTID**⟩{"**1**" ↓ **IDCOUNT**}.

On adding each letter to the incomplete identifier (8A), a marker is pushed onto a stack. When the identifier is complete (8) the number of elements on the stack is checked to see that it is less than seven, and the stack IDCOUNT is cleared.

The final specification with static semantic restrictions is given in Fig. 10.

SUMMARY

The first two notations, canonic systems and two-level grammars, are very similar in their approach to specifying static semantic rules. Both opted for a very simple basic mechanism in which

(a) an extended form of nonterminal allows for lists of identifiers or other symbols to be associated with the nonterminals and passed from node to node of the parse tree, and
(b) a test for containment enables one to check whether a particular symbol or string is contained within a list of symbols or strings.

This simple mechanism is crude and does not provide a sufficient range of facilities to enable the static semantic rules to be described concisely. The approach which they followed can be likened to the approach used in the λ-calculus where a few very simple concepts are used to simulate a variety of different properties of machines. Consequently specifications produced using either of these two notations can be very difficult to follow, and are not suitable for a compiler writer to understand, and translate into action.

A further problem with these two notations is that the lists which are constructed and associated with the nonterminals of the language can be passed from node to node up or down the parse tree. However, it is not clear from the productions (and it is sometimes a little difficult to ascertain) whether a particular list or attribute should be passed up or down at a particular point in the parse tree.

The approach taken in attribute grammars has some significant advantages over the previous two notations. In particular:

(a) The basic nonterminals are not altered in adding the static semantic restrictions. Thus in any specification the structure of the underlying syntax is clearly visible. This makes it easier for the reader to separate syntax from static semantics and see their interrelationship.
(b) The use of BNF as an underlying notation is important since virtually all programming languages have had their syntax specified in BNF, e.g. see De Remer *et al.* [20].
(c) The direction of propagation of each attribute at each node in the parse tree is clearly laid down in the specification.
(d) Additional facilities have been added to enable static semantic restrictions to be expressed in a clear and concise manner.

Ledgard's Production Systems and the Extended BNF notation both reduce the complex passage of lists from node to node of the parse tree, from nonterminal to nonterminal in the set of productions, by making use of global lists—a single global environment in the case of Production Systems and a set of global stacks in the case of Extended BNF. This is a further move away from the "fundamental approach" and towards the way in which a compiler writer would think.

Production Systems have a further advantage in that the notation for writing the productions describing the syntax has an implicit counting mechanism built into it which reduces the effort required in specifying the static semantic restrictions. However, it also has the disadvantage of avoiding BNF which has become the standard for specifying syntax rules.

Both systems provide facilities which are geared towards implementing the basic properties of static semantic rules (as mentioned in the section on Prerequisites of a Formal Notation) and hence both are able to express these rules neatly and concisely. One important difference between the two notations is that Extended BNF attempts to specify the static semantic actions as completely as possible at the level of the compiler writer so that he can implement them directly from the specification, while in Production Systems the notation is not geared as closely to the way in which the static semantic actions are to be implemented.

A further criterion of the usefulness of these notations is the fact that automatic compiler or parser generators have been implemented for both the attribute grammar [21, 22] and Extended BNF notations [23].

CONCLUSIONS

If one were to construct an ideal notation at this stage, it would be based on BNF because of the general acceptance of this notation for specifying syntax. However, the advantages of a built-in counting mechanism would favour an extension to the BNF notation similar to that used by Lee [24] in which portions of productions may be enclosed in braces with a superscript representing the count of the number of times the contents are repeated. For example

$$\langle \textbf{IDENTIFIER} \rangle ::= \langle \textbf{LETTER} \rangle \{ \langle \textbf{LETTER} \rangle \}^{i} \qquad 0 \leqslant i \leqslant 5$$

represents an identifier consisting of i + 1 letters where i + 1 ⩽ 6.

To these productions are added static semantic actions enclosed in some form of parentheses. These actions may reference a set of global stacks, metalinguistic variables, system variables, or the superscript variables (such as i in the above example).

The ideal notation might lie somewhere between the detailed level of Extended BNF and the higher (though sometimes less easily implementable) level of Production Systems. It should also provide an interface with a notation for specifying the semantics of a language so that the complete specification can be obtained. At the same time careful attention must be paid to error recovery.

With serious work being done on compiler and parser generators which utilize these techniques, it is important that the development work on these formal notations should continue until a generally acceptable and easily implementable notation can be found.

REFERENCES

1. P. Naur *et al.*, Revised report on the algorithmic language ALGOL6O, *Comput. J.* **5**, 349–367 (1963).
2. P. Wegner, The Vienna definition language, *Comput. Surv.* **4**, 5–63 (1972).
3. C. A. R. Hoare and N. Wirth, An axiomatic definition of the programming language Pascal, *Acta Informat.* **2**, 335–355 (1973).
4. D. Scott and C. Strachey, Towards a mathematical semantics for computer languages, PRG.6, Programming Research Group, Oxford University, Oxford, England (1971).
5. J. J. Donovan and H. F. Ledgard, A Formal system for the specification of syntax and translation of computer languages, in *Proc. AFIPS FJCC* **31**, 553–569 (1967).
6. H. F. Ledgard, A Formal System for Defining the Syntax and Semantics of Computer Languages, Ph.D. Thesis, M.I.T. (1969).
7. J. J. Donovan, *Systems Programming*. McGraw-Hill, New York (1972).
8. A. van Wijngaarden *et al.*, Report on the algorithmic language ALGOL68, *Numer. Math.* **14**, 79–218 (1969).
9. A. van Wijngaarden *et al.*, Revised report on the algorithmic language ALGOL68, Mathematisch Centrum, Amsterdam.
10. C. H. A. Koster, Two-level Grammars. In *Compiler Construction An Advanced Course*, pp. 146–156. Springer-Verlag, Berlin (1974).
11. D. E. Knuth, Semantics of context-free languages, *Math. Systems Theory* **2**, 127–145 (1968).
12. C. H. A. Koster, Affix grammars. In *Algol68 Implementation*. North-Holland, Amsterdam (1971).
13. G. V. Bochmann, Semantic evaluation from left to right, *Comm. ACM* **19**, 55–62 (1976).

14. D. A. Watt, The parsing problem for affix grammars, *Acta Informat.* **8**, 1–20 (1977).
15. D. A. Watt, An extended attribute grammar for Pascal, *Sigplan Notices* **14**(2), 60–74 (1979).
16. M. Marcotty, H. F. Ledgard and G. V. Bochmann, A sampler of formal definitions, *Comput. Surv.* **8**, 191–276 (1976).
17. H. F. Ledgard, Production systems: Or can we do better than BNF?, *Comm ACM* **17**, 94–102 (1974).
18. M. H. Williams, A formal notation for specifying static semantic rules, *Comput. Lang.* **5**, 37–55 (1980).
19. M. H. Williams, Static semantic features of Algol60 and BASIC, *Comput. J.* **21**, 234–242 (1978).
20. F. De Remer, P. Levy, S. Hanson, P. Jackson, R. Jullig and T. Pittman, Summary of the characteristics of several "modern" programming languages, *Sigplan Notices* **14**(5), 28–45 (1979).
21. D. R. Milton, L. W. Kirchoff and B. R. Rowland, An ALL (1) compiler generator, *Sigplan Notices* **14**(8), 152–157 (1979).
22. G. Papakonstantinou, A poor man's realization of attribute grammars, *Software-Practice and Experience*, **9**, 719–728 (1979).
23. H. L. Ossher and M. H. Williams, A parser generator which includes static semantic rules. In preparation.
24. J. A. N. Lee, The formal definition of the BASIC language, *Comput. J.* **15**, 37–41 (1972).

About the Author—MORGAN HOWARD WILLIAMS received the B.Sc. degree in Physics and Mathematics from Rhodes University, and then proceeded to an Honours degree and a Ph.D. degree in ionospheric physics. When undergraduate courses in Computer Science were started at Rhodes University in 1970, he was appointed as a lecturer in Computer Science. In 1977 he was appointed Professor of Computer Science and subsequently head of department of the joint department of Computer Science and Applied Mathematics, and in 1980 took up an appointment as Professor and Head of Department of Computer Science at Heriot-Watt University in Edinburgh.

17. M.H.WILLIAMS and A.R.BULMER, The use of a formal notation for static semantics in compiler design, Software Practice and Experience,vol. 8, 579-584 (1978).

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 8, 579–584 (1978)

Use of a Formal Notation for Static Semantics in Compiler Design

M. H. WILLIAMS AND A. R. BULMER

Computer Science Department, Rhodes University, Grahamstown, South Africa

SUMMARY

Formal notations for the specification of the syntax and the dynamic semantics of languages exist and are of great benefit to the compiler writer. However, formal notations for the static semantics of languages have tended to be tools of the language designer and of little practical significance to the compiler writer. This paper describes how a particular notation was used to assist in the implementation of a Cobol compiler and of an interpreter for a simulation language.

KEY WORDS Formal specification Static semantics Compiler writer

INTRODUCTION

Before designing a compiler for any particular computer language, there are three aspects of the language which must be known:

(a) the Syntax rules (or Syntax Form rules—those rules which specify the *form* of valid strings of the language);
(b) the Static Semantic rules (or Syntax Interpretation rules—those rules which must be complied with by strings which have the correct form before such strings can be accepted as syntactically correct; for example, 'any identifier used in a block must be declared');
(c) the Dynamic Semantic rules (those rules which specify the effect on execution of valid programming constructs).

In constructing a compiler for a simple programming language, the first step is to establish a formal specification of the syntax rules (usually in BNF[1]) and from this to construct a parser or syntax analyser. This process has been thoroughly studied and developed and can very often be done mechanically these days.

The next stage is to build into the parser a set of *ad hoc* routines for handling the static semantic rules (or at least those which one can think of) and the dynamic semantics (or code generation). In the latter case (dynamic semantics) formal notations have been developed and one would generally use some specification (formal or informal) of the semantics when constructing the code generator. However, relatively little work has been done on formal specification of static semantics for the compiler writer.

There have been two major attempts at specifying the complete syntax and semantics of large languages. In the case of PL/I, the syntax rules were specified using BNF and the static and dynamic semantics were specified using VDL.[2] Unfortunately the VDL approach

0038–6644/78/0508–0579$01.00
© 1978 by John Wiley & Sons, Ltd.
Received 14 March 1977
Revised 7 April 1978

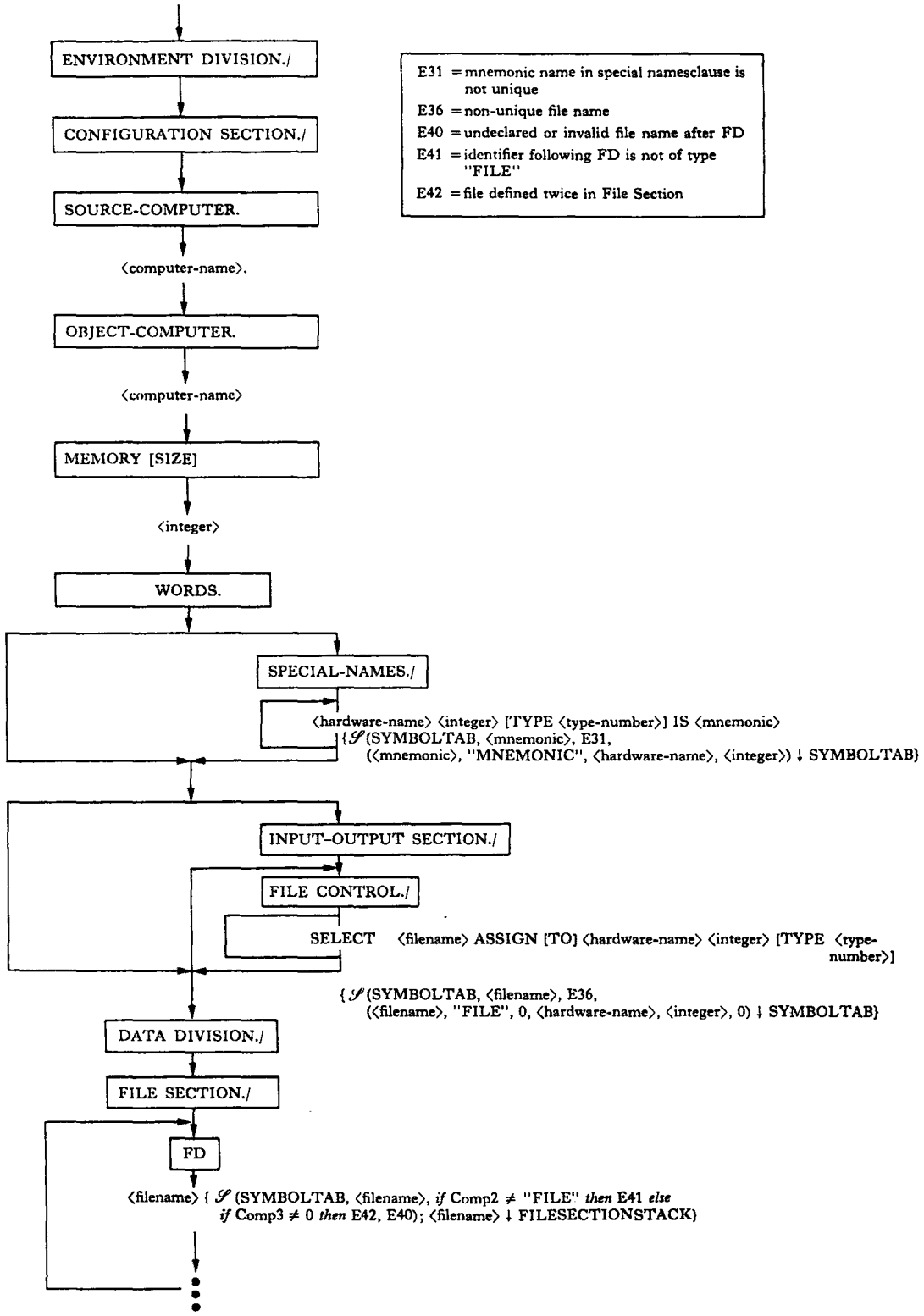


Figure 1. Illustration of part of the formal specification of the syntax of Cobol

is an interpretive one and is suitable if static semantic checking is to be performed at run time;[3] however, if one wants to construct a PL/I compiler, one would have to extract the static semantic rules from the dynamic semantics and handle them separately and somewhat differently during the compilation process.

The other major attempt at complete specification is the two-level grammar specification of Algol68.[4] However, again this does not lend itself to the construction of compilers directly from the specification.

It is possible to specify the static semantic rules in an informal notation (e.g. Griffiths[5]). However, it is not clear to what extent the same notation can be used to handle different languages. Furthermore, we know of no published specification of a language which includes the static semantic rules in this form.

One formal notation for specifying static semantic rules which is capable of handling different types of languages is Extended BNF[6,7]. Since this notation reflects the way in which the compiler might handle static semantic rules, it can be implemented directly by the compiler writer.

Recently we undertook the task of implementing a reasonably large subset of Cobol to produce a fast compiler for student compilations. Cobol is a language for which the syntax rules are relatively straightforward but the static semantic rules are not. Part of the problem lies in the difficulty in extracting the relevant rules into a suitable list since no specification of these rules exists, and part lies in knowing how to implement them.

Since Cobol is not usually specified in BNF, a variation on the Extended BNF approach based on a state diagram approach was used. However, the static semantic actions are the same as in Extended BNF.

Using this notation (part of which is described briefly in the next section) a complete specification of the syntax and static semantics was drawn up before coding the syntax analyser. The result was highly successful and is discussed further in the final section.

NOTATION FOR SPECIFYING SYNTAX OF COBOL

As mentioned above the notation used was a variation of Extended BNF, using a state diagram instead of a BNF description. Each basic symbol or terminal which is to be recognized is drawn in a box (see Figure 1) while pseudo-terminals (identifiers, integers, etc.) are not enclosed in boxes (the two exceptions being ⟨hardware name⟩ IS ⟨mnemonic⟩ and SELECT-ASSIGN TO).*

The static semantic actions are contained in braces and sited at the points where these actions are to be carried out. Thus if a SELECT-ASSIGN TO clause is successfully recognized the action

{$\mathscr{S}$(SYMBOLTAB, ⟨filename⟩, ...)}

is executed.

The actions themselves are concerned mainly with manipulating four types of items:

(a) the items of information from the source program, referred to by their non-terminal names (e.g. ⟨mnemonic⟩);
(b) literal strings (e.g. "FILE");
(c) stacks—these are data structures within the compiler and each is referred to by a name consisting of a string of upper-case letters;

*The symbol / is used to indicate that the immediately preceding keywords must appear on a line on their own.

(d) variables—these are single variables within the compiler and each is referred to by a name consisting of a string of lower-case letters.

The types of actions which may be performed include the following.

(a) x ↓ S where S is the name of a stack and x is an item to be stored on that stack
The effect of this is to place item x on stack S. Here x may be a simple item (non-terminal or literal string), e.g. ⟨file name⟩, or it may be an n-tuple of simple items, e.g. (⟨mnemonic⟩, "MNEMONIC", ⟨hardware name⟩, ⟨integer⟩). Thus if one were analysing the string

PRINTER 1 IS ERROR-PRINTER

and one encounters

(⟨mnemonic⟩, "MNEMONIC", ⟨hardware-name⟩, ⟨integer⟩) ↓ SYMBOLTAB

(see Figure 1) then the quadruple

("ERROR-PRINTER", "MNEMONIC", "PRINTER", 1)

would be added to the stack SYMBOLTAB (or rather some suitable coded version of this would be stored on SYMBOLTAB).

(b) En (n an integer)
This indicates an error, the number being the index of some suitable error message.

(c) $\mathscr{S}$(table, item, success action, fail action)
This causes the 'table' mentioned to be searched for an entry whose first component (in the case of an n-tuple) matches 'item'. If such an entry is found the success action is performed, otherwise the fail action is executed. For example,

{$\mathscr{S}$(SYMBOLTAB, ⟨mnemonic⟩, E31,
(⟨mnemonic⟩, "MNEMONIC", ⟨hardware-name⟩, ⟨integer⟩) ↓ SYMBOLTAB}

searches SYMBOLTAB for an item which matches ⟨mnemonic⟩. If one is found, this is an error (mnemonic name is not unique); if one is not found, an entry is made in SYMBOLTAB consisting of the quadruple

(⟨mnemonic⟩, "MNEMONIC", ⟨hardware-name⟩, ⟨integer⟩)

Having found an entry in a table the term Compn (n an integer) may be used to refer to various components of this particular entry.

(d) *if* condition *then* unconditional action *else* action or
if condition *then* unconditional action
For example,

$\mathscr{S}$(SYMBOLTAB, ⟨filename⟩, *if* Comp2 ≠"FILE" *then* E41
else if Comp3 ≠0 *then* E42, E40)

searches SYMBOLTAB for an item which matches ⟨filename⟩. If this is found then if its second component is not "FILE", this is an error (non-unique file name); alternatively, if its third component is not zero, this is an error (when the file is encountered in a SELECT-ASSIGN TO, this component is set to zero; when it is encountered in an FD, this component is altered. Thus if a non-zero third component is encountered, then the file has already been defined in an FD—hence an error). If ⟨filename⟩ is not found in SYMBOLTAB, this is also treated as an error (E40), since no SELECT-ASSIGN TO clause exists for this file.

(e) Actions may be compounded provided each pair is separated by a semicolon, e.g.

{$\mathscr{S}$(. . . .); ⟨filename⟩ ↓ FILESECTIONSTACK}

Once the complete specification of the subset of Cobol had been drawn up, the task of constructing a syntax analyser was relatively straightforward.

CONCLUSIONS

One of the most important advantages of any formal specification is that it enables one to lay down clearly and unambiguously the rules of the language. In designing a compiler for a language one needs a very clear picture of the rules of the language before embarking on any implementation of them.

The syntax rules are usually handled by a notation such as BNF. The static semantic rules are not so easy, but using the above mentioned notation these too can be specified in a form which is useful to the compiler writer.

The advantages of this approach are:

(a) the use of a formal specification of the static semantics enables one to separate the language problems from programming problems and hence simplify both tasks;
(b) having obtained a formal specification, production of a syntax analyser can be achieved in a relatively short space of time (particularly if one uses a systems programming language rather than an assembly language);
(c) the resulting syntax analyser has a better chance of being correct if based on a correct formal specification for both syntax and static semantics than if it is only based on a formal specification of syntax;
(d) if one wants to implement a similar compiler for a different machine, the same formal specification can be used, thus reducing implementation time considerably;
(e) if one needs to make alterations to the design these can be done with a minimum of effort by making the alterations to the formal specification first.

This approach has proved most useful in implementing a subset of Cobol. Had a less systematic approach been followed, it is quite possible that the data structures (and hence the whole program) might have needed to be altered at several points in the process of development. By using this more systematic approach the design of the syntax analyser as a whole was completed before embarking on implementation.

The same approach has also been used in developing a syntax analyser for another language, used by a simulation system. In this case a recursive descent syntax analyser was implemented using PASCAL. Since many of the static semantic rules could be coded directly into the syntax analyser, the final program looked very similar to the formal specification. The result of using this approach was a noticeable reduction in development time —the actual design and writing (excluding debugging) of the approximately 1,600-line PASCAL syntax analyser took about 4 days as opposed to an estimated 2–3 weeks.

From the experience gained in these two instances there is little doubt that this approach can be of benefit to compiler writers in developing compilers in the future.

ACKNOWLEDGEMENT

One of the authors, Mr. A. R. Bulmer, was in receipt of a bursary from the Council for Scientific and Industrial Research when this work was carried out.

REFERENCES

1. J. W. Backus, 'The syntax and semantics of the proposed International Algebraic Language of the Zurich ACM-GAMM Conference', *Proc. Int. Conf. on Information Processing, UNESCO (1959)*, 125–132 (1959).
2. P. Lucas, P. Lauer and H. Stigleitner, '*Method and notation for the formal definition of programming languages*', *IBM Tech. Rep. 25.087*, IBM Laboratory, Vienna (1968).
3. H. F. Ledgard, 'Production systems or can we do better than BNF ?', *Comm. ACM*, **17**, 94–102 (1974).

4. A. Van Wijngaarden *et al.*, 'Revised report on the algorithmic language Algol68', *Acta Informatica*, 5, 1–236 (1975).
5. M. Griffiths, 'Relationship between definition and implementation of a language', in *Software Engineering, An Advanced Course*, Springer-Verlag, New York, 1975, pp. 76–110.
6. M. H. Williams, 'A formal notation for specifying static semantic rules' (submitted to *Computer Language*.
7. M. H. Williams, 'Static semantic features of Algol60 and BASIC', *Comput. J.* (to appear).

18. M.H.WILLIAMS, Structuring two-level grammar specifications, The Computer Journal, vol. 28, no. 3, 250-256 (1985).

Structuring two-level grammar specifications

M.H. WILLIAMS
Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ

The two-level grammar notation is a powerful tool for specifying the syntax, static semantics and even dynamic semantics of programming languages. However, it can also be a very difficult notation to follow. For this reason an approach to writing two-level grammars is advocated which explicitly indicates the 'direction of propagation' of metanotions. This suggests the requirement that the direction of propagation of each metanotion within a given hypernotion should be consistent and enables one to test a two-level grammar specification for circularity.

1. INTRODUCTION

The two-level grammar or W-grammar notation[1] is a powerful tool for describing languages. By introducing parameters (metanotions) into nonterminal names and by using a separate set of productions (metaproduction rules) to define these parameters, it is a simple matter to produce grammars with an infinite number of production rules, and hence generate not only context-free languages but also context-sensitive and unrestricted (Type 0) languages[2]. This notation has the advantage that static semantic restrictions (and even dynamic semantics) can be expressed along with the syntax of a language in a single definition – but the disadvantage that for anyone who is not expert in the notation, it is considerably more difficult to follow than a simple notation such as BNF or an extended notation such as attribute grammars.

With a conventional single-level grammar it is very easy to construct a parse tree to assist one to visualize the way in which productions are applied to generate sentences of the language but with a two-level grammar this is much more difficult to accomplish owing to the complexity of the productions. In an attribute grammar the productions describing the syntax can be used to construct a parse tree, after which the passage of the attributes about the parse tree can be traced using the directions laid down in the specification. However, in a two-level grammar specification metanotions cannot be dealt with in isolation nor is there any indication of the ' direction' to be followed by individual metanotions. The Algol 68 report[3,4] is evidence in itself both of the power of the two-level grammar notation and its imperspicuity to all but a few.

The main problem of the two-level grammar notation is not dissimilar to the problem associated with the goto statement in programming languages. If goto statements are avoided and the control structures used in any program are restricted to certain simple types, it is possible to look at parts of a program in isolation and follow the flow of control through these sections. On the other hand by permitting uncontrolled usage of goto statements one may have a more general and flexible language but one forfeits the ability to look at a segment of code in isolation without first obtaining a picture of the overall control structure surrounding the segment in question.

The two-level grammar notation suffers from precisely this defect. In general it is not possible to look at part of a specification in isolation and master it without a reasonable understanding of the interface with the remainder of the specification. Again the definition of Algol 68 is evidence of this fact.

This problem could be overcome if, where metanotions are used to carry information about the parse tree, a 'direction of propagation' is associated with these metanotions in the same way as directions are associated with attributes in an attribute grammar. This will certainly assist anyone who is not expert in the notation to obtain a clearer understanding of any particular two-level grammar in a shorter period of time. At the same time it will also benefit the language designer in that portions of a specification can be dealt with in isolation.

2. BRIEF OVERVIEW OF TWO-LEVEL GRAMMAR APPROACH

Whereas in a conventional one-level grammar one has a fixed and finite set of nonterminal symbols, the power of the two-level grammar approach lies in the idea that nonterminal symbols need not necessarily be fixed but may be treated as variable strings which are in turn controlled by a set of productions at a higher level. Thus a two-level grammar is composed of two finite sets of rules called *metaproduction rules* (or *metarules*) and *hyper-rules*. From these two sets the set of production rules is derived.

For example, in a one-level grammar one might have

< assignment stm > : : =
  < numeric variable > = < numeric exp >
 | < boolean variable > = < boolean exp >

In a two-level grammar this might be written as

Metaproduction rules

TYPE:: boolean; numeric.

Hyper-rules

assignment stm: TYPE variable, equals symbol, TYPE exp.

In van Wijngaarden's notation for metasyntax, the nonterminals (called metanotions) are written in upper case letters, the terminals (which form part of the nonterminals of the syntax) are written in lower case letters, ' ::' is used to stand for 'is defined as' (in place

of '::='), ';' is used to separate options (in place of vertical bar) and '.' is used to mark the end of each metaproduction rule.

The notation for hyper-rules is similar using ':' instead of '::=' and ';' and '.' as before. In the metaproduction rules if a nonterminal or terminal is to be followed by another, they are simply juxtaposed (as in BNF); in the hyper-rules the symbol ',' is used as a separator indicating concatenation.

If the metaproduction rules are substituted into the hyper-rules in the above example, the following productions are obtained:

assignment stm: boolean variable, equals symbol, boolean exp.
assignment stm: numeric variable, equals symbol, numeric exp.

In these three types of rules, three types of symbols are used:

(i) A *protonotion* is a possibly empty sequence of lower case letters and spaces (eg assignment stm) which is the nonterminal/terminal of the production rules and the terminal of the metaproduction rules. In a production rule a protonotion which ends in 'symbol' is a terminal, one which does not, is a nonterminal.

(ii) A *metanotion* is a non-empty sequence of upper case letters (eg TYPE) and is the nonterminal of the metaproduction rules.

(iii) A *hypernotion* is a possibly empty sequence of metanotions and/or protonotions (eg TYPE variable) and is the nonterminal/terminal of the hyper-rules.

These are summarized in Table 1.

The term 'terminal metaproduction' of a metanotion refers to any protonotion which can be derived from that metanotion by applying the metaproduction rules. To change a hyper-rule to a production rule:

(1) Each metanotion found only once in the hyper-rule is replaced with a terminal metaproduction for that metanotion.

(2) If a metanotion occurs more than once in the hyper-rule, each occurrence of that metanotion must be replaced with the same terminal metaproduction. This is referred to as the uniform replacement rule or consistent substitution rule.

For more details see Cleaveland and Uzgalis[5] or Marcotty et al[6].

3. DIRECTION OF PROPAGATION

In order to understand a particular grammar and see how it is used to parse a string, it is helpful to have some idea of when the values represented by metanotions are being passed up the parse tree, when they are being passed across from one node to another at the same level (within a production) and when they are being transmitted down the tree. In other words it is useful to be able to follow the ' direction of propagation' of metanotions in the same way that one can follow the direction of propagation of attributes in an attribute grammar.

For this purpose a metanotion *m* occurring within some hypernotion will be said to be synthesized (or upward propagating) if the protonotion which that metanotion represents is determined in the process of expanding the hypernotion to obtain a terminal string. Such a metanotion will be marked with an upward arrow. Likewise an occurrence of metanotion *m* is said to be inherited (or downward propagating) if the protonotion which the metanotion represents is determined elsewhere and is 'passed down' to the hypernotion concerned from an ancestral node in the parse tree. In this case *m* will be marked with a downward arrow. These arrows can be regarded as an addition to the subscripting mechanism and are ignored in applying the consistent substitution rule.

To illustrate this convention, consider the example given by Cleaveland and Uzgalis[5] of a two-level grammar which generates the set of strings of letters in which no letter is repeated.

Metaproduction rules

1. ALPHA::a;b;c;d;e;f;g;h;i;j;k;l;m;
   n;o;p;q;r;s;t;u;v;w;x;y;z.
2. LETTER:: letter ALPHA.
3. TAG::LETTER; LETTER TAG.
4. EMPTY::.
5. NOTION::ALPHA;NOTION ALPHA.
6. NOTETY::NOTION;EMPTY.

Hyper-rules

7. s:TAG unique.
8. LETTER TAG unique: LETTER symbol,
   TAG unique, where LETTER is not in TAG.
9. LETTER unique: LETTER symbol.
10. where LETTER1 is not in LETTER2 TAG:
    where LETTER 1 isnt LETTER2,
    where LETTER1 is not in TAG.
11. where LETTER1 is not in LETTER2: where LETTER1 isnt LETTER2.
12. where letter ALPHA1 isnt letter ALPHA2:
    where ALPHA1 precedes ALPHA2 in abcdefghijklmnopqrstuvwxyz;
    where ALPHA2 precedes ALPHA1 in abcdefghijklmnopqrstuvwxyz.
13. where ALPHA1 precedes ALPHA2 in NOTETY1 ALPHA1 NOTETY2 ALPHA2 NOTETY3: EMPTY.

Table 1. Nomenclature used for nonterminal/terminal symbols in the three types of rules

Type of rule	nonterminals	terminals
Metaproduction rules	metanotions	protonotions
hyper-rules	hypernotions	hypernotions
production rules	protonotions not ending in 'symbol'	protonotions ending in 'symbol'

This specification can be made easier to understand if the hyper-rules are rewritten with the direction of propagation of the metanotions indicated as follows:

7. s: TAG↑ unique.
8. LETTER↑TAG↑ unique: LETTER↑symbol, TAG↑unique, where LETTER↓ is not in TAG↓.
9. LETTER↑unique: LETTER↑symbol.
10. where LETTER1↓ is not in LETTER2↓ TAG↓: where LETTER1↓ isnt LETTER2↓, where LETTER1↓ is not in TAG↓.
11. where LETTER1↓ is not in LETTER2↓: where LETTER1↓ isnt LETTER2↓.
12. where letter ALPHA1↓ isnt letter ALPHA2↓: where ALPHA1↓ precedes ALPHA2↓ in abcdefghijklmnopqrstuvwxyz; where ALPHA2↓ precedes ALPHA1↓ in abcdefghijklmnopqrstuvwxyz.
13. where ALPHA1↓ precedes ALPHA2↓ in NOTETY1↓ ALPHA1↓ NOTETY2↓ ALPHA2↓NOTETY3↓: EMPTY.

An example of a parse tree for a sentence of this language is given in Fig.1. This shows where the protonotions are associated with the metanotions and how they propagate about the parse tree.

It is not intended that this tracing of the direction of propagation of a metanotion should be regarded as something fundamental to two-level grammars as it is in the case of attribute grammars, but is introduced as an aid to the user in seeing how productions are selected. As will be seen in the next section some specifications have to be rewritten in order to be able to do this.

Nevertheless, where a direction of propagation can be established, it does make it easier for the reader, particularly in the case of complex grammars with a large number of hyper-rules.

4. TYPES OF HYPERNOTION

As a first step to determining the direction of propagation of metanotions, consider the different ways in which hypernotions may be used. Suppose that one has a hypernotion h containing one or more metanotions m_i ($1 \leqslant i \leqslant n$). This hypernotion may be classified according to the way in which it is used, into at least one of the following categories:

(a) *Predicate with no return value* (*PNR*). h can produce only the empty string or protonotions which cannot be reduced to terminal strings (blind alleys), and all metanotions m_i are passed down to h from above (all m_i are inherited). Examples of this include:

where LETTER1↓ is not in LETTER2↓ TAG↓,
where letter ALPHA1↓ isnt letter ALPHA2↓,

from the example in the previous section.

(b) *Predicate with returned value* (*PR*). Once again h can produce only the empty string or blind alleys. However, in this case some of the m_i are inherited and some are synthesised. The simplest example of such a hypernotion is one which passes down a symbol table and an identifier and returns the mode of the identifier, or passes down an array name and a symbol table and returns the number of dimensions of the array.

(c) *Terminal with no inherited metanotions.* Here h can

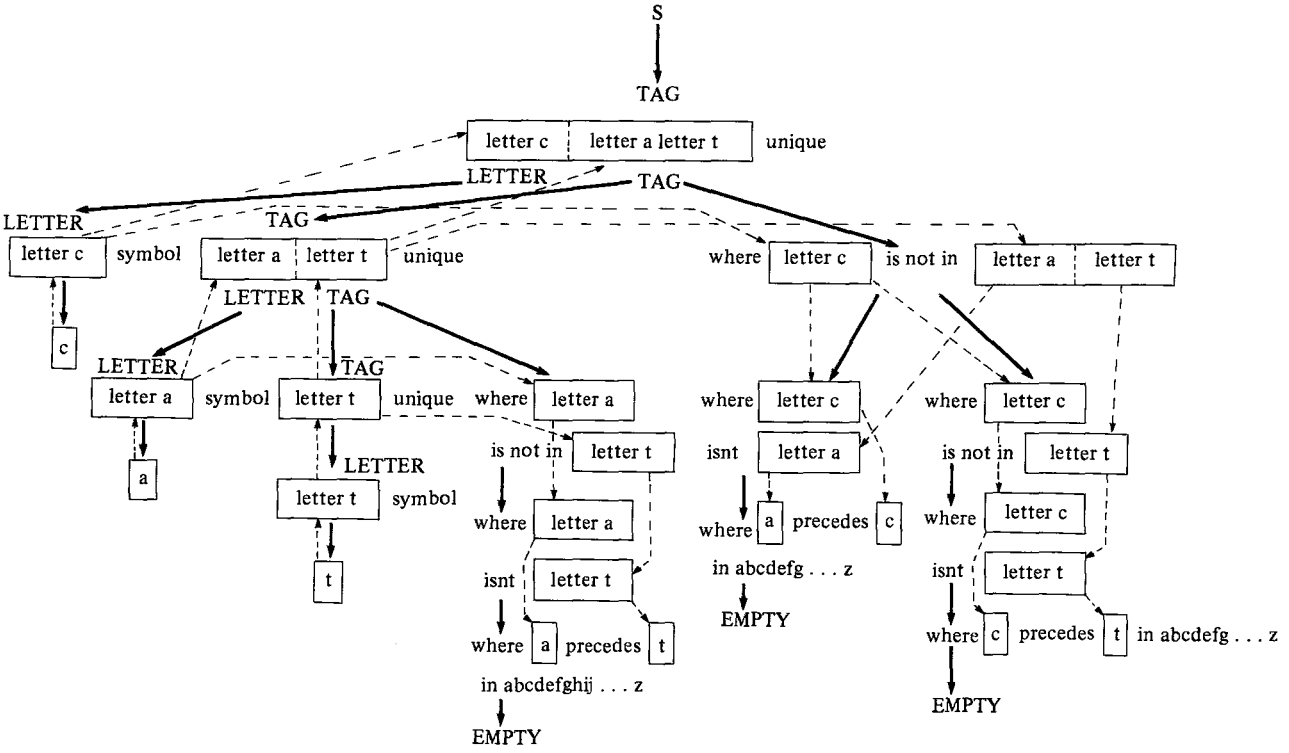


Fig. 1. Parse tree for a sentence of the language given in first example

produce only a terminal string (or the empty string) and none of the m_i are inherited. An instance of this type of hypernotion taken from the example in the previous section is:

LETTER↑ unique

(d) *Terminal with inherited metanotions.* Again h can produce only a terminal string (or the empty string) but in this case at least one of the m_i is inherited. An illustration of this type of hypernotion might occur in Algol68 where either begin-end or ( ) may be used to encompass a construction provided that the pair of delimiters used is of the same type (ie one cannot have begin...)). For example

Metaproduction rules

DELIM::end;close bracket.

Hyper-rules

stmt group: begin symbol, end group;
open bracket symbol, close bracket group.
DELIM group: DELIM stmt;
stmt, separator symbol, DELIM group.
DELIM stmt: stmt, DELIM symbol.

(e) *Mixed hypernotion.* This is a hypernotion which is capable of producing both terminal strings (including the empty string) and protonotions which cannot be reduced to terminal symbols (blind alleys). Referring again to the example in the previous section, an instance of this type of hypernotion is

LETTER↑ TAG↑ unique

For the inexperienced reader it is confusing when different directions of propagation are associated with a particular occurrence of a metanotion within a given hypernotion depending on where the hypernotion is used. This can even result in a hypernotion falling into two different categories.

To illustrate this consider the example (taken again from Cleaveland and Uzgalis) of a grammar which generates all strings of the form $a^n = b^n + c^n$ where a,b,c and n are positive integers, $n > 2$.

Note that Fermat's last theorem (which has remained unproven for generations) states that there do not exist any positive integers a,b,c,n satisfying the equation $a^n + b^n = c^n$ for $n > 2$; thus any string generated by this grammar will be a counterexample which disproves the theorem.

Metaproduction rules

1. N::one; N one.
2. EMPTY::.
3. NETY::N; EMPTY.
4. RADIX:: one one one one one one one one one one.
5. EXP:: one one N.

Hyper-rules

6. fermat: EXP poweresult N1, equal symbol,
   EXP poweresult N2, plus symbol,
   EXP poweresult N3, where N1 is N2 N3.
7. N1 poweresult N2: N3 number, power symbol,
   N1 number, where N2 is N3 to the N1 power.
8. where N1 is N2 to the NETY one power:
   where N3 is N2 to the NETY power,
   where N1 is N3 times N2;
   where N1 is N2, where NETY is EMPTY.
9. where NETY1 N is N times NETY2 one:
   where NETY1 is N times NETY2;
   where NETY1 NETY2 is EMPTY.
10. where NETY is NETY: EMPTY.
11. N1 number: N1 token; N2 number, NETY token,
    where N3 is N2 times RADIX,
    where N1 is NETY N3.
12. EMPTY token: zero symbol.
13. one token: one symbol.
14. one one token: two symbol.
15. one one one token: three symbol.
16. one one one one token: four symbol.
17. one one one one one token: five symbol.
18. one one one one one one token: six symbol.
19. one one one one one one one token: seven symbol.
20. one one one one one one one one token: eight symbol.
21. one one one one one one one one one token: nine symbol.

If one attempts to classify hypernotions in this specification one finds that in hyper-rule 6, 'where N1 is N2 N3' is a PNR hypernotion since the values of N1, N2 and N3 are passed down to it and it can produce only the empty string or protonotions which cannot be reduced to terminal strings; however, in hyper-rule 11, 'where N1 is NETY N3' is a PR hypernotion in which the values of NETY and N3 are passed down to it and the value of N1 is returned. A similar conflict occurs in several of the other rules.

It is obvious that if a hypernotion is being used in different ways at different points in the specification, resulting in confusion about the direction of propagation of its component metanotions, this hypernotion could be replaced by two or more different hypernotions each of which performs a single function. Alternatively it may be better to rewrite the hyper-rules in such a way that this confusion is removed. In the case of the above example, rules 6-11 can be rewritten and directions assigned to the metanotions as follows:

Hyper-rules

6. fermat: EXP↑ poweresult N1↑, equal symbol,
   EXP↑ poweresult N2↑, plus symbol,
   EXP↑ poweresult N3↑, where N1↓ is N2↓ N3↓.
7. N1↑ poweresult N2↑: N3↑ number, power symbol,
   N1↑ number, where N2↑ is N3↓ to the N1↓ power.
8. where N1↑ is N2↓ to the N4↓ one power:
   where N3↑ is N2↓ to the N4↓ power,
   where N1↑ is N3↓ times N2↓.

8A. where N2↑ is N2↓ to the one power: EMPTY.

9. where NETY1↑N↑ is N↓ times N2↓ one:
   where NETY1↑ is N↓ times N2↓.

9A. where N↑ is N↓ times one: EMPTY.

10. where NETY↓ is NETY↓:EMPTY.

11. N1↑ number: N1↑ token;
N2↑ number, NETY↑ token,
where N3↑ is N2↑ times RADIX↓,
where NETY↓ N3↓ is rewritten as N1↑.
11A. where NETY↓ is rewritten as NETY↑: EMPTY.

5. RULES FOR ESTABLISHING DIRECTION OF PROPAGATION

The next step is to formulate a set of rules which will enable one to establish the direction of propagation of metanotions within a given grammar in some instances. Four fairly obvious rules are given below. These rules are straightforward and place no restrictions on the specification. However, they only establish a direction of propagation when certain obvious conditions hold.

(1) *Absence from right-hand side.* If a hypernotion on the left hand side of a hyper-rule contains a metanotion *m* and at least one of the alternatives on the right hand side of the hyper-rule contains neither *m* nor any hypernotion derived from *m*, then the occurrence of *m* is inherited. For example, in a slightly simplified form of one of the hyper-rules from the definition of Algol 68 (Revised Report)

VIRACT declarer: VIRACT declarator;
TALLY applied mode indication with TAB.

VIRACT must be inherited since there is no occurrence of VIRACT or a hypernotion derived from VIRACT (virtual, actual or formal) in the hypernotion 'TALLY applied mode indication with TAB'. Similarly in the case of:

formal row NEST rower: up to token option.

there is no hypernotion derived from NEST on the right hand side, and hence NEST must be inherited.

(2) *Absence from left hand side.* If one or more alternatives on the right hand side of a hyper-rule each contains a single occurrence of a meta-notion *m* and the hypernotion on the left hand side contains neither *m* nor any hypernotion from which *m* might be derived, and if *m* can generate more than one possible protonotion, then the occurrences of *m* are synthesised. For example, in the simplified form of hyper-rule from the definition of Algol68 considered in the previous rule:

VIRACT declarer: VIRACT declarator;
TALLY applied mode indication with TAB.

both TALLY and TAB must be synthesised since no ancestor of either occurs on the left hand side and both generate more than one protonotion. Hence one can write the rule as:

VIRACT↓declarer: VIRACT declarator;
TALLY↑ applied mode indication with TAB↑.

(3) *Consistency within a hypernotion.* If one occurrence of a hypernotion *h* contains a metanotion *m* marked as synthesised (inherited) then any other occurrence of *h* in the grammar containing a corresponding occurrence of *m* (or a derivative or ancestor of *m*) must have *m* marked as synthesised (inherited). For instance, in the example of Fermat's last theorem, the hypernotion

EXP↑ poweresult N1↑

occurring in rule 6 will require all other occurrences of this hypernotion such as

EXP↑ poweresult N2↑ in rule 6
N1↑ poweresult N2↑ in rule 7

to have synthesised metanotions in the first and last positions.

(4) *Consistency within a hyper-rule.* If metanotion *m* occurs in a hypernotion *h* and there are corresponding occurrences of *m* in the alternatives on the right hand side of the hyper-rule defining *h*, and if each appropriate occurrence of *m* on the right hand side is synthesised (inherited) then the corresponding occurrence of *m* in *h* is synthesised (inherited). Similarly if there is only one occurrence of *m* on the right hand side of the hyper-rule defining *h* then if either occurrence of *m* is synthesised (inherited) then the other must be synthesised (inherited) too. Thus the hyper-rule discussed in rules (1) and (2) would become:

VIRACT↓declarer: VIRACT↓declarator;
TALLY↑applied mode indication with TAB↑.

As mentioned these rules only establish a direction of propagation for certain limited cases. Unfortunately they are not sufficient in themselves to determine the direction of propagation of most of the metanotions in any specification. Indeed it is usually possible to determine the direction of propagation after a careful study of the context, but at this stage no general set of rules has been established which will do this automatically.

6. THE PROBLEM OF CIRCULARITY

Just as one of the problems in one-level grammar specifications is that of cycles, and for any practical purposes a grammar must be cycle free, likewise one of the problems which can arise in an attribute grammar specification is that of circular definition of attributes. Knuth[7] drew attention to this problem and gave an algorithm for detecting such circularity.

The introduction of directions of propagation into a two-level grammar specification naturally raises the possibility that circularity might occur in such a specification. To illustrate the problem consider an adaptation of the example considered by Knuth.

Suppose that one wants to extend the definition of a binary integer, viz

<bit> ::= 0|1
<string> ::= <bit>|<string><bit>
<integer> ::= <string>

to incorporate the value of the integer. Following the lines of Knuth's solution using an attribute grammar, a

two-level grammar specification (with direction of flow indicated) for this might be:

Metaproduction rules

1. VALUE:: NUMBER.
2. SCALE:: NUMBER.
3. LENGTH:: NUMBER.
4. NUMBER:: one NUMBER; EMPTY.
5. EMPTY::.

Hyper-rules

6. bit with VALUE↑ and SCALE↓:
   zero symbol, where VALUE↑ is EMPTY↓;
   one symbol, where VALUE↑ is SCALE↓.
7. list with VALUE↑ and LENGTH↑ and SCALE↓:
   bit with VALUE↑ and SCALE↓, where LENGTH↑ is one;
   list with VALUE1↑ and LENGTH1↑ and SCALE1↓,
   bit with VALUE2↑ and SCALE↓,
   where VALUE↑ is VALUE1↓ VALUE2↓,
   where SCALE1↑ is SCALE↓ SCALE↓,
   where LENGTH↑ is one LENGTH1↓.
8. integer with VALUE↑: list with VALUE↑ and LENGTH↑ and SCALE↓,
   where SCALE↑ is one.
9. where NUMBER↑ is NUMBER↓: EMPTY.

If one had unintentionally written hyper-rule 8 as:

8. integer with VALUE↑: list with VALUE↑ and LENGTH↑ and SCALE↓,
   where SCALE↑ is VALUE↓.

the definition can easily be shown to be circular. Fig.2 illustrates this with the parse of the string '1'. Clearly the metanotion VALUE in the hypernotion 'integer with VALUE' is not uniquely determined by the parse. If the above specification with the incorrect version of hyper-rule 8 had been written without any indication of the direction of flow, the circularity (if it can still be referred to as such) would not be easily detectable. Nevertheless, it still exists and the grammar will be ambiguous in this case.

7. CONCLUSION

While the two-level grammar notation is both powerful and flexible, it does suffer from the drawback that specifications in this notation can be very difficult to follow. In view of this a case has been presented for indicating the 'direction of propagation' of metanotions wherever possible in a two-level grammar specification.

For the inexperienced reader tracing the direction of flow enables him to view sections of a specification in isolation and to obtain a better understanding of the specification as a whole. However, determining the direction of propagation of a metanotion within a given hypernotion is, in general, not an easy task. Two different approaches have been suggested which may assist the reader in this respect, namely

(a) Classification of hypernotions. This is based on an understanding of what the hypernotion does.
(b) General rules about propagation. These are based purely on the form of the hyper-rules.

Language designers who use the two-level grammar notation to specify their languages should determine the direction of propagation of metanotions in their specification in order to ensure that this is consistent. The advantages of doing so are:

(a) It will make the specification easier for the reader to understand.

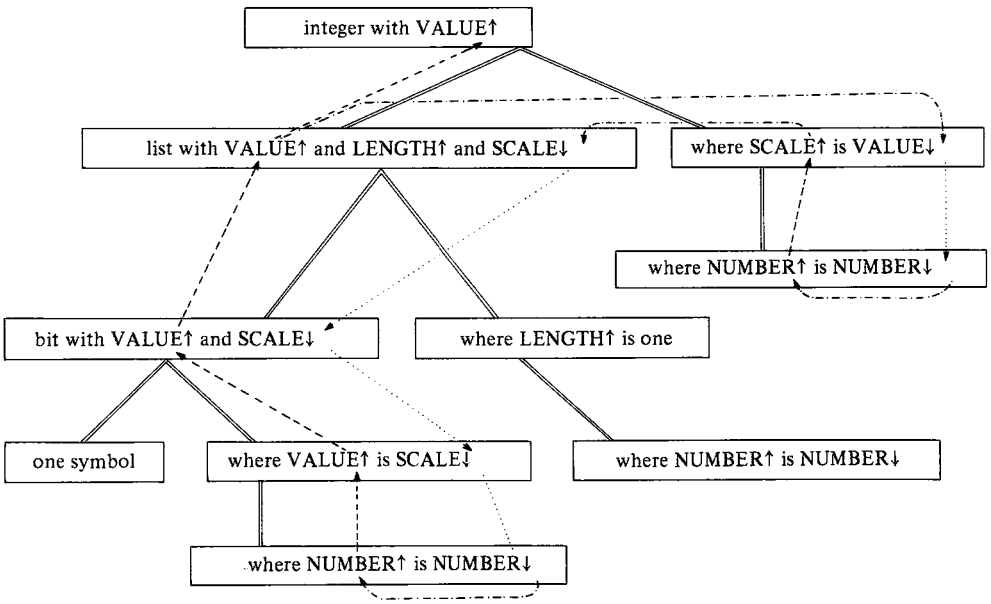


Fig. 2. Parse tree for the string '1' using the incorrect two-level grammar definition. The passage of the values represented by the metanotions VALUE and SCALE about the tree are traced with dotted lines indicating inherited values, dashed lines indicating synthesised values and a sequence of dots and dashes indicating horizontal transfer.

(b) It enables part of a specification to be considered in isolation from the remainder of the specification thereby making it less prone to error.
(c) It enables the designer to check for circularity in a specification.

The net effect of a practice such as this is to improve the readability and understandability of a specification and reduce the probability of errors, without any significant sacrifice to the power of the notation. In some ways this approach to writing two-level grammars can be compared with the discipline of structured programming in a conventional programming language. Thus just as indiscriminate use of the goto statement in computer programs is undesirable, a non-unique direction of propagation of a metanotion could be considered 'harmful'.

Acknowledgement

The author is indebted to the referee for helpful comments leading to the revision of this paper.

REFERENCES

1. C. H. A. Koster, Two-level Grammars, in *Compiler Construction An Advanced Course*, Ed. by G. Goos and J. Hartmanis, pp.146–156. Springer-Verlag, Berlin (1974).
2. M. Sintzoff, Existence of a van Wijngaarden syntax for every recursively enumerable set, *Ann. Soc. Sci. Bruxelles* **81**, 115–118 (1967).
3. A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker, *Revised report on the algorithmic language ALGOL*68, Springer-Verlag, New York (1976).
4. A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck and C. H. A. Koster, Report on the algorithmic language ALGOL68, *Numer. Math.* **14**, 84–218 (1969).
5. J. C. Cleaveland and R. C. Uzgalis, *Grammars for Programming Languages*, Elsevier, New York (1977).
6. M. Marcotty, H. F. Ledgard and G. V. Bochmann, A sampler of formal definitions, *ACM Computer Surveys* **8**, 191–276 (1976).
7. D. E. Knuth, Semantics of context free languages, *Math. Syst. Theory* **2**, 127–145 (1968).

19. M.H.WILLIAMS, A flexible notation for syntactic definitions, ACM Transactions on Programming Languages and Systems, vol. 4, no. 1, 113-119 (1982).

A Flexible Notation for Syntactic Definitions

M. HOWARD WILLIAMS
Rhodes University

In view of the proliferation of notations for defining the syntax of programming languages, it has been suggested that a simple notation should be adopted as a standard. However, any notation adopted as a standard should also be as versatile as possible. For this reason, a notation is presented here which is both simple and versatile and which has additional benefits when specifying the static semantic rules of a language.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*syntax*; F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems

General Terms: Languages

Additional Key Words and Phrases: BNF

1. INTRODUCTION

When ALGOL 60 was developed, its syntax was formally specified using the notation known as BNF [12]. Since then a wide range of different notations has been used by different authors for specifying the syntax of programming languages (e.g., the notation used for defining the syntax of COBOL [10], the two-level grammar approach used in the definition of ALGOL 68 [14, 15], the syntax diagrams used in the specification of PASCAL [4], extended BNF [1, 9], or the canonic system notation [2, 3, 8]). Although in some cases the reasons for the differences in notation can be easily understood (e.g., [9]), in others variations appear to have been introduced merely to satisfy the personal tastes and ego of the author rather than to further any clear objective.

This rapidly expanding plethora of notations is unnecessary, bewildering for the novice, and annoying for the more experienced. In 1977 Wirth [18] attacked the situation and put forward a simple notation suitable for adoption as a standard. The salient features of his notation are that

(1) nonterminals are written as identifiers without enclosing angle brackets;
(2) terminals are contained within quotation marks;
(3) braces—{ }—are used to denote "zero or more repetitions of";
(4) square brackets—[ ]—are used to denote "zero or one occurrence of"; and
(5) parentheses—( )—are used for grouping in the usual way.

This paper presents an alternative notation which is almost as simple but which has some additional advantages.

Author's present address: Computer Science Department, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ, Scotland.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1982 ACM 0164-0925/82/0100-0113 $00.75

2. THE NOTATION

At the heart of any formal notation for describing the syntax of programming languages must lie a mechanism for specifying repetition. This mechanism may be simple (catering only for the three basic types of repetition or choice which are most common in the specification of programming languages, namely, "zero or one occurrence of," "zero or more occurrences of," and "one or more occurrences of"), or it may be general, taking account of these three special cases and all other cases. In the notation presented here two mechanisms are provided: a simple one to handle the commonly occurring special cases and a more general one to take care of the exceptions.

In this notation terminals are written as character strings within quotation marks. To maintain generality, single or double quotes may be used provided that the use is consistent, for example, "(" or '('. If a quotation mark appears within a literal which is enclosed by the same type of quotation marks, the quotation mark must be written twice, for example, """".

Again for the purpose of generality, nonterminals may be written as identifiers with or without enclosing angle brackets, provided that, if enclosing angle brackets are used, they are used consistently throughout. An identifier consists of a sequence of upper- or lowercase letters, digits, underlines, and hyphens (starting with a letter and ending with a letter or digit), for example,

assign-statement or ⟨ASSIGN-STATEMENT⟩.

Since there is some confusion over the use of the term "production," the term "string equation" is used here to refer to the composite definition of a nonterminal, while the term "production" is restricted to an instance of that definition which might be substituted directly in a parse. A string equation is written in the following form: nonterminal followed by "::=" followed by a string expression, optionally terminated by a period. The symbol "::=" has been adhered to because the properties of string equations do differ slightly from those of numeric equations ("=") and considerably from those of assignment statements (":="). A string expression consists of a sequence of options separated by vertical bars, for example,

add-operator ::= "+" | "−".

An option consists of a sequence of terms concatenated together. If nonterminal identifiers are not enclosed in angle brackets, then two successive nonterminals concatenated together must be separated by at least one space. Square brackets are used to denote "zero or one occurrence of" whatever is contained within them; for example,

["+" | "−"]

represents a plus sign, or a minus sign, or nothing. The postfix operators "*" and "+" are used to denote, respectively, "zero to ∞ occurrences of" and "one to ∞ occurrences of" the item which each follows. Round brackets are used for grouping; for example,

definition ::= option (" | " option)*.

```
     Syntax ::= production+.
 production ::= nonterminal "::=" definition [";" relations] ["."].
 definition ::= option ("|" option)*.
     option ::= term+.
       term ::= "[" definition "]" | item ["+" | "*" | counter].
       item ::= "(" definition ")" | nonterminal | terminal.
nonterminal ::= "⟨" identifier "⟩" | identifier.
   terminal ::= (""""" character+ """")+ | ("'" character+ "'")+.
    counter ::= identifier.
  relations ::= relation ("," relation)*.
   relation ::= limit "≤" counter "≤" limit.
      limit ::= limterm (("+" | "−") limterm)* | "∞".
   limterm ::= integer | identifier.
    integer ::= digit+.
 identifier ::= letter ((hyphen | underline)* (letter | digit)+)*.
```

Figure 1

states that a definition consists of an option followed by a sequence of zero or more groups, each consisting of a vertical bar symbol followed by an option.

The more general mechanism for repetition involves the notion of a counter. A counter is a variable used to indicate repetition and is represented by an identifier without angle brackets. Repetition is denoted by writing a counter as a superscript after the item to be repeated and adding a relation defining the limits of the counter at the end of the string equation. A semicolon is used to separate the definition part from the relations, for example,

$$\text{syntax} ::= \text{production}^{i}; \qquad 1 \leq i \leq \infty.$$

In this definition counters are treated as being local to the productions in which they are used. There may, however, be some point in treating them as global variables, although at this stage no reason can be seen for doing so. Also, where counters are nested, every instance of the inner counter must satisfy the relations given at the end of the production.

The full definition of the notation using itself is given in Figure 1. Spaces are unimportant except within quotation marks.

3. FLEXIBILITY

This notation provides for flexibility with respect to both the lower and upper limits. Thus, besides the three standard cases, the notation also handles cases such as the following:

(1) *Upper Limit* $\neq 1$ *or* ∞. A simple example is the definition of an identifier in FORTRAN, where the length of an identifier is constrained to a maximum of six characters:

$$\text{identifier} ::= \text{letter (letter} \mid \text{digit)}^{\text{len}}; \qquad 0 \leq \text{len} \leq 5.$$

(2) *Lower Limit* $\neq 0$ *or* 1. Although this does not occur very often, there are cases where it might be useful to have a lower limit greater than 1. An example

is the switchon command in BCPL [13], which might be defined as follows:

switchon ::= "SWITCHON" expression "INTO $(" case^i "$)"; $2 \leq i \leq \infty$.

case ::= ("CASE" constant ":")+ (command "newline")+.

(3) *More Complex Situations.* A data-name in COBOL [10] is a contiguous sequence of up to 30 characters (letters, digits, or hyphens), provided that a hyphen does not occur in the first or last position of the sequence and at least one of the characters in the sequence is a letter. To define this formally, one may write

data-name ::= (digit (digit | hyphen)^i)^j letter
((digit | hyphen | letter)^m (letter | digit))^k;

$0 \leq i \leq 28, 0 \leq j \leq 1, 0 \leq k \leq 1, 0 \leq m \leq 28 - i - j$.

Another complex situation arises in the case of the multiple assignment statement in BCPL, in which the destinations are written as a list to the left of the assignation symbol (:=) while the values occur in a list to the right of it, for example,

VAL, LEFT, RIGHT := K, 0, I + 1

This causes the value K to be assigned to VAL, 0 to be assigned to LEFT, and I + 1 to RIGHT. One could certainly define this construction as follows:

assign-stm ::= destination "," assign-stm "," value
| destination "=" value.

However, the parse tree which this would produce wrongly associates the value K with the variable RIGHT and the value I + 1 with the variable VAL. A better way of defining this construction would be as follows:

assign-stm ::= destination ("," destination)^i ":=" value ("," value)^i;
$0 \leq i \leq \infty$.

4. EXTENSION TO STATIC SEMANTIC RULES

Besides the syntax there are two other aspects of programming languages which need formal specification: the static semantics and the semantics. The static semantic rules are closely related to the syntax rules, and the formal specification of the static semantics of a language is usually an extension of the formal specification of the syntax.

Any notation for formally specifying the static semantic rules of a language must provide some mechanism for counting [17]. This is necessary in order to check the length of a formal parameter list against the length of each list of actual parameters or to ensure that a subscripted variable has the correct number of subscripts.

However, the counting mechanism in some static semantic notations is fairly crude and tends to complicate the specification. For example, in the two-level grammar notation [6, 14, 15] counting is performed by successively concatenating one or more symbols to a nonterminal name. A simple illustration is the require-

ment that an identifier be no more than six characters in length. In this case the symbols of the identifier are accumulated within the nonterminal name, for example,

Metasyntax

LETTER :: a; b; . . . ; z.

VAR :: LETTER; LETTER VAR.

Syntax

VAR partid : LETTER symbol; LETTER symbol, VAR partid.

and the check on the length of an identifier would be specified by enumerating the possible differing-length sequences.

Using a variation of the notation proposed here, this specification might be written as

Metasyntax

LETTER ::= a | b | · · · | z.

VAR ::= LETTER+.

Syntax

VAR-id ::= LETTER-symbol^{len}; $1 \leq len \leq 6$.

This approach has also been used to count the number of dimensions in an array definition and check this against the number of subscripts in a subscripted variable in the definition for ALGOL 68. The resulting specification requires fewer string equations. But what is more important is that the counting function has been disentangled from the rest of the specification and set apart in an easily recognizable form, thereby making it more readable.

Another important notation for specifying static semantics is attribute grammars [5, 7, 11, 16]. There, counting is performed by explicitly performing an action which causes 1 to be added to a variable. For example, the previous illustration of an identifier of not more than six characters in length may be written as an attribute grammar in which (1) *condition* precedes a predicate on attributes, (2) ↑ prefixes attribute names passed up the parse tree and ↓ prefixes attributes passed down the tree, (3) attributes can be subscripted to distinguish occurrences, and (4) attribute evaluation rules have the form of function calls where ↓ precedes the arguments and ↑ precedes the newly defined attribute.

⟨ID⟩ ↑name ::= ⟨IDENTIFIER⟩ ↑name ↑noletters
condition: noletters < 7

⟨IDENTIFIER⟩ ↑name$_1$ ↑noletters$_1$ ::= ⟨LETTER⟩ ↑name$_1$
give value to attribute ↓1 ↑noletters$_1$
| ⟨IDENTIFIER⟩ ↑name$_2$ ↑noletters$_2$
⟨LETTER⟩ ↑name$_3$
concatenate ↓name$_2$ ↓name$_3$ ↑name$_1$
add one letter ↓noletters$_2$ ↑noletters$_1$

add one letter ↓noletters$_1$ ↑noletters$_2$ ⇒ noletters$_2$ = noletters$_1$ + 1

This example may be rewritten in a variation of the notation proposed here as follows:

$$\langle \text{ID} \rangle \uparrow \text{name}_1 ::= \langle \text{LETTER} \rangle \uparrow \text{name}_1 \; (\langle \text{LETTER} \rangle \uparrow \text{name}_2 \; \textit{concatenate} \downarrow \text{name}_1 \downarrow \text{name}_2 \uparrow \text{name}_1)^i; \qquad 0 \leq i \leq 5.$$

Once again, separation of the counting mechanism can simplify the notation, thereby improving readability.

5. CONCLUSIONS

A notation for defining the syntax of programming languages which is both simple and flexible has been presented. It could also have certain advantages when extending a syntax specification to include static semantic rules.

From the point of view of teaching, the notation is an extension of the notation for regular expressions which makes it easier for students to see the relationship between string equations and regular expressions. It is important that these concepts, which form part of a continuum, are not regarded as completely separate entities, each with its own notation.

In this notation as it has been presented, counters have been written as superscripts in productions to aid readability. However, if one wanted to enter a definition in this notation into a computer system, one would need a few minor modifications. First, the symbol "↑" (or even "**") could be used to precede a counter to indicate a superscript (as it is used in the case of exponentiation in computer languages). Second, "≤" should be written as "<=". Finally, if the upper limit in a relation is infinity, the limit and its preceding "≤" can be omitted. None of these alterations causes any ambiguity. The definition in Figure 1 need only be modified by substituting for the string equations for term, relation, and limit the equations

```
          term ::= "[" definition "]" | item ["+" | "*" | superscript-op counter].
superscript-op ::= "↑" | "**".
      relation ::= limit "<=" counter ["<=" limit].
         limit ::= limterm (("+" | "−") limterm)*.
```

It is hoped that this paper will not be viewed simply as a presentation of yet another notation for syntactic definitions. The main purpose of the paper has been to look closely at the advantages of the notation proposed, and it is hoped that in the future, before adopting any syntactic notation, readers will give careful consideration to the advantages of such a notation and avoid the introduction of new notations or variations on existing ones unless the advantages can be clearly spelled out.

REFERENCES

1. BULL, G.M., FREEMAN, W., AND GARLAND, S. *Specification for Standard BASIC.* NCC Publications, Manchester, England, 1973.
2. DONOVAN, J.J. *Systems Programming.* McGraw-Hill, New York, 1972.

3. DONOVAN, J.J., AND LEDGARD, H.F. A formal system for the specification of the syntax and translation of computer languages. In Proc. 1967 Fall Jt. Computer Conf., vol. 31. AFIPS Press, Arlington, Va., 1967, pp. 553–569.
4. JENSEN, K., AND WIRTH, N. *Pascal User Manual and Report.* Springer-Verlag, New York, 1976.
5. KNUTH, D.E. Semantics of context free languages. *Math. Syst. Theory 2* (1968), 127–145.
6. KOSTER, C.H.A. Two-level grammars. In *Compiler Construction, An Advanced Course*, G. Goos and J. Hartmanis (Eds.). Springer-Verlag, New York, 1974, pp. 146–156.
7. KOSTER, C.H.A. Affix grammars. In *Algol68 Implementation*, J.E.L. Peck (Ed.). Elsevier North-Holland, New York, 1971, p. 95.
8. LEDGARD, H.F. A Formal System for Defining the Syntax and Semantics of Computer Languages. Ph.D. dissertation, M.I.T., Cambridge, Mass., 1969.
9. LEE, J.A.N. The formal definition of the BASIC language. *Comput. J. 15*, 1 (Feb. 1972), 37–41.
10. MAGINNIS, J.B. *Fundamental ANSI COBOL Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1972.
11. MARCOTTY, M., LEDGARD, H.F., AND BOCHMANN, G.V. A sampler of formal definitions. *Comput. Surv.* (ACM) *8*, 2 (June 1976), 191–276.
12. NAUR, P., ET AL. Revised report on the algorithmic language ALGOL60. *Comput. J. 5*, 4 (Jan. 1963), 349–367.
13. RICHARDS, M. The BCPL programming manual. Univ. Cambridge Computer Laboratory, Cambridge, England, 1973.
14. VAN WIJNGAARDEN, A., MAILLOUX, B.J., PECK, J.E.L., KOSTER, C.H.A., SINTZOFF, M., LINDSEY, C.H., MEERTENS, L.G.L.T., AND FISKER, R.G. Revised report on the algorithmic language ALGOL68. Springer-Verlag, New York, 1976.
15. VAN WIJNGAARDEN, A., MAILLOUX, B.J., PECK, J.E.L., KOSTER, C.H.A., SINTZOFF, M., LINDSEY, C.H., MEERTENS, L.G.L.T., AND FISKER, R.G. Report on the algorithmic language ALGOL68. *Numer. Math. 14* (1969), 79–218.
16. WATT, D.A. An extended attribute grammar for Pascal. *SIGPLAN Notices* (ACM) *14*, 2 (Feb. 1979), 60–74.
17. WILLIAMS, M.H. Methods for specifying static semantics. *Comput. Lang. 6*, 1 (1981), 1–17.
18. WIRTH, N. What can we do about the unnecessary diversity of notation for syntactic definitions? *Commun. ACM 20*, 11 (Nov. 1977), 822–823.

Received December 1980; revised July 1981; accepted August 1981

20. M.H.WILLIAMS, Complete operator precedence conditions, Information Processing Letters, vol. 6, no. 2, 60-62 (1977).

COMPLETE OPERATOR PRECEDENCE CONDITIONS

M.H. WILLIAMS
Department of Computer Science, Rhodes University, Grahamstown, South Africa

Received 26 August 1976

Grammars, operator precedence, precedence parsing

1. The problem with operator precedence

In a recent paper in this journal Levy [1] pointed out that operator precedence grammars suffer one serious drawback, namely that while a simple shift-reduce algorithm can be used to recognise all sentences of an operator precedence grammar, there is no guarantee that such an algorithm will not also accept strings which are not valid sentences of the grammar unless ad hoc routines are added to the parser. For the purposes of this present paper the term simple parser will be used to refer to an operator precedence parser without such ad hoc routines.

To illustrate this problem Levy uses an ill-chosen example which is unfortunately not an operator precedence grammar as it has non-unique precedence relations and his argument consequently does not apply. A better example would have been the following set of productions

$S \to \vdash T \dashv$

$T \to E + E$

$E \to i$

In this case, if one parses the sentence $\vdash i \dashv$, one finds that $\vdash \lessdot i$ and $i \gtrdot \dashv$ so that the sentence is reduced to $\vdash E \dashv$. This sentential form now has $\vdash \doteq \dashv$ and if no check is performed to ensure that both terminals and nonterminals match the right-hand side of some production, $\vdash i \dashv$ will be accepted as a valid string of the language.

2. The cause of the problem

There are two reasons why invalid strings may be mistakenly recognised by an operator precedence parser. These are:

(1) If at any stage in analysing a sentential form, a nonterminal is missing, this may not be noticed by the parser. For example, if the sentence $\vdash i + \dashv$ were parsed using the grammar of Section 1, this would be reduced to $\vdash E + \dashv$, and hence since $\vdash \lessdot +$ and $+ \gtrdot \dashv$, the next reduction would be to $\vdash T \dashv$. Thus an incomplete sentence may be accepted by the parser.

(2) In stepping from one symbol to the next in a sentential form, the parser may "lose its place" in the productions. This is the case illustrated by the example in Section 1. This grammar generates only one sentence, viz. $\vdash i + i \dashv$. The relation $\vdash \lessdot i$ applies only to the first i while $i \gtrdot \dashv$ applies only to the second i. Thus in parsing $\vdash i \dashv$, the parser first checks the precedence relation between $\vdash$ and i (assuming as it does so that i is the first i – which indeed it is), and then checks the relation between i and $\dashv$ (assuming in this case that i is the second i – which it is not).

3. Extended operator precedence

In order to overcome this problem of invalid strings being recognised, Levy proposed an extension to the definition of operator precedence relations, which provides for two types of precedence relation between

terminals. These two types are as follows:

(1) The first type defines the precedence relation between two terminals if they can occur adjacent to one another in a sentential form, i.e.

$a \doteq {}_1b$ iff $U \to \alpha ab\beta \in P$

$a \lessdot {}_1b$ iff $U \to \alpha aV\beta \in P$ and $V \overset{+}{\Rightarrow} b\gamma$

$a \gtrdot {}_1b$ iff $U \to \alpha Vb\beta \in P$ and $V \overset{+}{\Rightarrow} \gamma a$

(2) The second type defines the precedence relation between two terminals if they can occur in a sentential form with a nonterminal between them, i.e.

$a \doteq {}_2b$ iff $U \to \alpha aWb\beta \in P$

$a \lessdot {}_2b$ iff $U \to \alpha aV\beta \in P$ and $V \overset{+}{\Rightarrow} Wb\gamma$

$a \gtrdot {}_2b$ iff $U \to \alpha Vb\beta \in P$ and $V \overset{+}{\Rightarrow} \gamma aW$

This doubling up of the precedence relations eliminates one of the sources of error with operator precedence parsing, namely the possibility that the absence of a nonterminal from a sential form will not be noticed. The other source of error, viz. the problem of losing one's place, is taken care of by placing conditions on the grammar which can be handled by this method.

In his original paper [1] Levy defines the condition od as the necessary and sufficient condition to ensure that an operator precedence grammar which satisfies it, can be parsed in a complete way by a simple parser, i.e. that no strings which are not in the language will be accepted by such a parser. In a subsequent paper [2] Levy adds to these conditions. However, neither of these two sets of conditions is correct nor is the full treatment given in [3], as can be illustrated by a few simple examples.

4. Examples

using the set of conditions described in [2], consider the following two examples:

(1) $T \to \vdash S \dashv$

$S \to aBbCc$

$S \to aDbCd$

$B \to x$

$C \to y$

$D \to z$

This grammar satisfies the set of conditions in [2], yet if given the string $\vdash axbyd \dashv$, a simple extended operator precedence parser will accept this as a valid sentence of the language. The problem here arises from the fact that $x \gtrdot {}_1b$ and $b \doteq {}_2d$ are both valid relationships but they refer to different b's.

(2) $S \to \vdash T \dashv$

$T \to (E)$

$T \to x(E)$

$E \to i$

This grammar does not satisfy the set of conditions in [2], yet it in fact is a perfectly well-behaved grammar which will not cause an extended operator precedence parser to accept invalid strings.

5. Revised conditions

A revised set of conditions is stated briefly for completeness. A more complete treatment of these is given in [4].

Definition. If a symbol X occurs in two separate places on the right-hand side of a set of productions and if the portion of the production to the left on the first occurrence is the same as the portion of the production to the left of the second occurrence, i.e.

$A \to \alpha X\beta$,

$A \to \alpha X\gamma$,

then the two productions are *left identical with respect to X*. Similarly if the portion of the production to the right of the first occurrence of X is the same as the portion of the production to the right of the second occurrence of X i.e.

$A \to \alpha X\beta$,

$A \to \gamma X\beta$,

then the two productions are *right identical with respect to X*.

Definition. If a grammar contains two occurrences of a symbol on the right-hand side of the set of productions and if

(a) the symbol on the left-hand side of the two

productions is the same and the two occurrences are neither left identical nor right identical with respect to the symbol, or

(b) the symbols on the left-hand side of the two productions are different and a minimal invalid subparse * can be found which includes a swop from the one occurrence to the other and a return to a common production at a lower level (i.e. closer the root) of the parse tree, then the grammar is *confusable* over the two occurrences.

Definition. If a grammar contains two occurrences of a terminal symbol x on the right-hand side of the set of productions and if the grammar is confusable over these two occurrences, then the grammar is *terminal confusable over x.*

Definition. If a grammar is not terminal confusable but contains two occurrences of a nonterminal X on the right-hand side of the set of productions ($A \to \alpha X \beta$ and $B \to \gamma X \delta$) which are confusable, and if either

(i) $\alpha = \alpha' x, \delta = y\delta'$ and $x \doteq_2 y$

(ii) $\alpha = \delta = \epsilon$

(iii) $\alpha = \alpha' x, \delta = \epsilon$ and there exists z such that $G \overset{+}{\Rightarrow} \sigma A z \tau$ and $x \gtrdot_2 z$,

or (iv) $\alpha = \epsilon, \delta = y\delta'$ and there exists z such that $G \overset{+}{\Rightarrow} \sigma z A \tau$ and $z \lessdot_2 y$,

then the grammar is *nonterminal confusable over X.*

In [4] it is shown that the extended operator precedence parsing method will accept only valid sentences if the grammar is an operator precedence grammar which is uniquely invertible with respect to terminals and if it is neither terminal confusable nor nonterminal confusable.

6. Conclusion

The concept of extended operator precedence is a considerable improvement over the normal operator precedence system. However, in order to use it, conditions must be imposed on the grammar to eliminate the problem of losing one's place in a parse. These conditions are stated briefly in Section 5.

References

[1] M.R. Levy, Complete operator precedence, Information Processing Lett. 4 (2) (1975) 38–40.
[2] M.R. Levy, A further note on complete operator precedence, to appear in Information Processing Lett. (1976).
[3] D.S. Henderson and M.R. Levy, An extended operator precedence parsing algorithm, to appear in Comput. J. (1976).
[4] M.H. Williams, Conditions for extended operator precedence parsing, submitted to Comput. J.

* This concept is discussed fully in [4].

21. M.H.WILLIAMS, Conditions for extended operator precedence parsing, The Computer Journal, vol. 22, no. 2, 164-168 (1979).

Conditions for extended operator precedence parsing

M. H. Williams

Department of Computer Science, Rhodes University, Grahamstown 6140, South Africa

The extended operator precedence method of parsing described by Henderson and Levy (1976) will only work correctly provided the grammar used satisfies certain conditions. The conditions which they propose are, however, incorrect and a revised set of conditions are put forward here.

(Received September 1976; revised July 1978)

. Introduction

A recent paper in this journal (Henderson and Levy, 1976) lescribes an extended operator precedence scheme for parsing entences of a grammar. However, this paper has several rrors, the most serious of which concerns the conditions nder which their algorithm holds. Since the method itself s an interesting one which may well prove a useful compiling echnique in the future, the question of suitable conditions or their algorithm deserves some attention.

The principle underlying their precedence scheme is the use f two types of precedence relation to distinguish between erminals which can occur adjacent to one another in a sen- ential form, and terminals which can occur with a nonterminal etween them in a sentential form. This idea is incorporated nto a parsing algorithm which does not require additional *d hoc* routines in order to ensure that only valid sentences re accepted.

The notation used here is chiefly that used by Gries (1971) nd by Henderson and Levy. As in their paper, attention will e restricted to context-free grammars, and in particular to perator precedence grammars. Unless otherwise stated, , $B, C, \ldots Z \in V_N$; $a, b, c, \ldots z \in V_T$; $\alpha, \beta, \gamma, \ldots \omega \in V^*$ here $V = V_T \cup V_N$, $Z \in V_N$ is the goal symbol and ε is the ull symbol.

The problem with operator precedence

'ith an ordinary operator precedence parser, unless one ores and scans a set of productions or resorts to *ad hoc* outines, the parser is capable of accepting invalid strings. here are two reasons for this:

) A nonterminal may be absent from a sentential form (or a phrase derived from a nonterminal may be absent from a sentence). For example, in the case of

$$Z \to \vdash T \dashv$$
$$T \to iE$$
$$E \to j$$

$\vdash i \dashv$ might be accepted as a valid string since $\vdash \lessdot i$ and $i \gtrdot \dashv$. The absence of the phrase derived from E would pass undetected.

) The parser may 'lose its place' when it reaches some symbol X on the righthand side of one production and accidentally swop to some other occurrence of the symbol X on the righthand side of the same or another production, e.g. for the grammar

$$Z \to \vdash T \dashv$$
$$T \to iEi$$
$$E \to +$$

the string $\vdash i \dashv$ may be accepted as a valid string since $\vdash \lessdot i$ (from the first i in the production $T \to iEi$) and $i \gtrdot \dashv$ (from the second i in the same production).

The method proposed by Henderson and Levy is aimed at eliminating the first of these sources of error. The second source of error is removed by imposing limitations on the grammar which can be handled by this method. However the conditions, which Henderson and Levy advocate, debar cases such as

$$Z \to \vdash T \dashv$$
$$T \to (E)$$
$$T \to x(E)$$
$$E \to i$$

which, in fact, are perfectly well behaved and not prone to errors of this type, while their conditions allow grammars such as

$$Z \to \vdash T \dashv$$
$$T \to AxBc$$
$$T \to CxDd$$
$$A \to i$$
$$B \to j$$
$$C \to k$$
$$D \to m$$

for which their algorithm will accept the invalid string $\vdash ixmd \dashv$.

3. Revised set of conditions

The set of conditions imposed on the operator grammar must eliminate the problem of losing one's place. This problem arises because the shift-reduce algorithm always considers only two terminal symbols at a time. Consider the string $x_1\, x_2 \ldots x_{i-1}\, x_i\, x_{i+1} \ldots x_n$ and the productions

$$X \to \alpha x_i \beta$$
$$Y \to \gamma x_i \delta$$

In analysing the string, x_i is approached from the left and the relation between x_{i-1} and x_i is determined. However, once x_i is placed on the stack, the relation between x_i and x_{i+1} is checked without reference to the portion preceding x_i. Thus forms such as $\alpha x_i \delta$ or $\gamma x_i \beta$ may be recognised as valid.

1. *Terminals*

It is therefore obvious that wherever a terminal occurs more than once on the righthand side of a set of productions, there is a chance that this problem may arise. In particular the conditions under which two occurrences of a terminal symbol x will cause invalid strings to be accepted by the parser, can be divided into two cases:

(*a*) where the nonterminal on the lefthand side of the two productions is the same, and

(*b*) where the nonterminals on the lefthand side are different.

Consider the first of these categories.

If the portion of the production to the left of the first occurrence of x is the same as the portion of the production to the

left of the second occurrence of x, i.e.

$$X \rightarrow \alpha x \beta$$
$$X \rightarrow \alpha x \gamma$$

then the two productions are *left identical with respect to x.*

Similarly, if the portion of the production to the right of the first occurrence of x is the same as the portion of the production to the right of the second occurrence of x, i.e.

$$X \rightarrow \alpha x \beta$$
$$X \rightarrow \gamma x \beta$$

then the two productions are *right identical with respect to x.*

Lemma 1

If two occurrences of a terminal x have the same nonterminal on the lefthand side of the productions, i.e.

$$X \rightarrow \alpha x \beta$$
$$X \rightarrow \gamma x \delta$$

then

(*a*) if the two productions are left identical or right identical with respect to x, this cannot lead to confusion for even if the parser does lose its place, the result will be the same as if it had not done so, and

(*b*) if the two productions are neither left identical nor right identical then confusion can occur and provided that a sentential form containing either $\alpha x \delta$ or $\gamma x \beta$ cannot be generated from any other production in the grammar, it is always possible to find an invalid sentence which a simple parser will accept.

Proof

If the two productions are left identical, i.e.

$$X \rightarrow \alpha x \beta$$
$$X \rightarrow \alpha x \delta$$

then even if confusion does occur, the only strings which the parser will accept are those derived from sentential forms containing either the phrase

$$\ldots . \alpha x \beta \ldots .$$

or

$$\ldots . \alpha x \delta \ldots .$$

which are both valid sentential forms.

Similarly if the two productions are right identical, i.e.

$$X \rightarrow \alpha x \beta$$
$$X \rightarrow \gamma x \beta$$

then even if confusion does occur, only valid sentential forms can be produced.

However, if the two productions are neither left identical nor right identical, i.e.

$$X \rightarrow \alpha x \beta$$
$$X \rightarrow \gamma x \delta$$

assume that the string α(which may consist of terminals and nonterminals) may produce (amongst others) some terminal string $\alpha_1\ \alpha_2 \ldots \alpha_i$, and similarly that δ may produce some string $\delta_1 \ldots \delta_j$, β may produce $\beta_1 \ldots \beta_k$, etc. Assume further that there exist terminal symbols a and b such that

$$a \lessdot_1 \alpha_1 \text{ if } \alpha \neq \varepsilon \qquad \text{or } a \lessdot_1 x \text{ if } \alpha = \varepsilon$$

and

$$\delta_j \gtrdot_1 b \text{ if } \delta \neq \varepsilon \qquad \text{or } x \gtrdot_1 b \text{ if } \delta = \varepsilon$$

(for if such terminals a and b did not exist they could always be created by adding a production such as

$$Z' \rightarrow aZb$$

to the grammar). Then since $a \lessdot_1 \alpha_1$, $\alpha \doteq_1 x$, $x \doteq_1 \delta$, $\delta_j \gtrdot_1 b$ (or some similar condition if α, δ or both are null), it is possible for a sentence derived from a sentential form containing the phrase

$$\ldots \alpha x \delta \ldots$$

to be accepted as valid. Likewise it is possible for a sentential form containing the phrase $\ldots \gamma x \beta \ldots$ to be accepted by the parser. □

On the other hand, if two occurrences of a terminal x have different nonterminals on the lefthand side of the production then the situation is more complex. In this case it is necessary to define the idea of a minimal invalid subparse.

Definition

If it is possible to confuse one occurrence of a terminal or nonterminal X on the righthand side of a set of productions with another such occurrence of the same symbol, then the two occurrences are said to be *potentially confusable.*

Example.

$$X \rightarrow Axb$$
$$Y \rightarrow Cxd$$

The two occurrences of x are potentially confusable, i.e. it might be possible to accept Axd as a valid sentential form.

However, if two occurrences are potentially confusable, this need not mean that confusion will necessarily occur. It is only possible to confuse the two occurrences and swop from the one production to the other, if it is possible to find a 'reverse swop' or a common termination so that when the parser reaches the end of the string concerned it will have reduced it to the goal symbol.

For example, in the case of

$$Z \rightarrow \vdash T \dashv$$
$$T \rightarrow iSj$$
$$S \rightarrow Axb$$
$$S \rightarrow Rf$$
$$R \rightarrow Cxd$$
$$A \rightarrow a$$
$$C \rightarrow c$$

it is possible to recognise the invalid string $\vdash iaxdfj \dashv$. The reason for this is shown in **Fig. 1.**

Definition

A *minimal invalid subparse* is that portion of a parse tree (acceptable to an extended operator precedence parser) which starts with a swop from one occurrence of a symbol X on the righthand side of a production to another such occurrence, and ends with a reverse swop or a common termination. The phrase produced by this subtree will be referred to as a *minimal invalid phrase.*

Three cases can now be distinguished:

1. Where a terminal y exists such that the final symbols of the two productions causing confusion,

$$X \rightarrow \alpha x \beta$$
$$Y \rightarrow \gamma x \delta$$

have suitable precedence relations with respect to y, i.e. the last symbol of $x\beta \gtrdot_1 y$ and the last symbol of $x\delta \gtrdot_1 y$. For example,

$$Z \rightarrow \vdash T \dashv$$
$$T \rightarrow aXbY$$
$$X \rightarrow iYcj$$
$$Y \rightarrow ij$$

for which the invalid string $\vdash aijbij \dashv$ would be accepted as valid by a simple parser. In this case the minimal invalid phrase is ij.

However, if the two occurrences of x are right identical (i.e. $\beta = \delta$)

$$X \rightarrow \alpha x \beta$$
$$Y \rightarrow \gamma x \beta$$

then as before, confusion of this type cannot lead to invalid

strings being accepted by the parser.

2. Where some nonterminal E (the root of the minimal invalid subparse) exists such that

$$E = >^* \sigma X \psi$$

and

$$E = >^* \sigma Y \chi$$

For example,

$$Z \rightarrow \vdash S \dashv$$
$$S \rightarrow Xy$$
$$X \rightarrow sBf$$
$$X \rightarrow sAe$$
$$A \rightarrow axb$$
$$B \rightarrow cxd$$

for which the invalid sentence $\vdash saxdfy \dashv$ would be accepted by a simple parser. Here the minimal invalid phrase is *saxdf*.

Note that it is not possible to have a case similar to the $\beta = \delta$ case in 1, if the grammar is uniquely invertible with respect to terminals. However, if the latter condition is not required then one might have a grammar such as

$$Z \rightarrow \vdash S \dashv$$
$$S \rightarrow Xy$$
$$X \rightarrow sBe$$
$$X \rightarrow sAe$$
$$A \rightarrow axb$$
$$B \rightarrow cxb$$

where even if confusion does arise between the two occurrences of x, the only possible strings which can be produced are $\vdash saxbey \dashv$ and $\vdash scxbey \dashv$ both of which are valid sentences of the grammar.

3. Where the minimal invalid phrase terminates with a further swop between two occurrences of another symbol. For example

$$Z \rightarrow \vdash S \dashv$$
$$S \rightarrow Vxj$$
$$S \rightarrow Az$$
$$A \rightarrow Wx$$
$$V \rightarrow Xp$$
$$W \rightarrow Yq$$
$$X \rightarrow aib$$
$$Y \rightarrow cid$$

In this case the invalid string $\vdash aidqxj \dashv$ would be accepted by a simple parser since confusion occurs first over i and then over x. This obviously could be extended to cater for three or more swops between productions within a single sentence.

Once again if one does not require that a grammar is uniquely invertible with respect to terminals, one may have a $\beta = \delta$ type situation in which the subparse is no different from a valid subparse. Such a subparse is not a minimal invalid subparse.

Lemma 2

If two occurrences of a terminal x have different nonterminals on the lefthand side of the productions then

(*a*) if a minimal invalid subparse cannot be found, confusion between these two occurrences of x will not lead to an invalid sentence which is acceptable to a simple parser, and

(*b*) if a minimal invalid subparse exists then confusion can occur and provided that the minimal invalid phrase cannot be produced legally by other productions of the grammar at that point in the string, an invalid sentence can be found which is acceptable to a simple parser.

Proof

The first condition follows from the idea of a minimal invalid subparse. The proof of part (*b*) is somewhat lengthy and follows the same lines as the proof for part (*b*) of Lemma 1. □

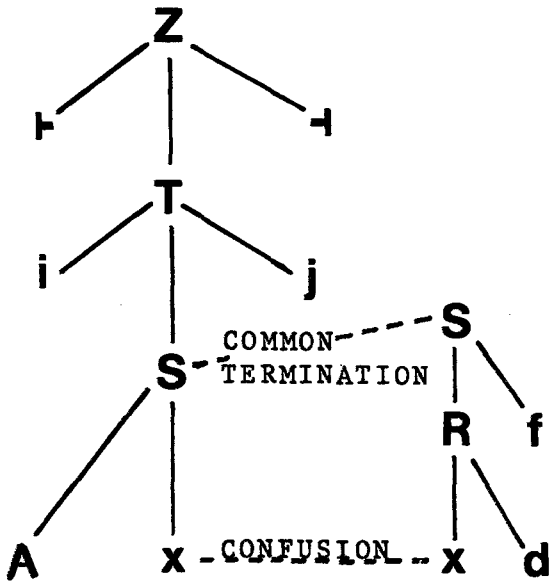


Fig. 1 On parsing the string $\vdash iaxdfj \dashv$ the parser performs a correct parse until it reaches x. It then confuses this with a second occurrence of x and proceeds along the minimal invalid subparse. When it reaches f, it will assume that it has successfully recognised S and will accept the remainder of the string as if nothing was wrong.

Thus one might define:

Definition

If a grammar contains two occurrences of a terminal symbol on the righthand side of the set of productions and if

(*a*) the symbol on the lefthand side of the two productions is the same and the two occurrences are neither left identical nor right identical with respect to the symbol, or

(*b*) the symbols on the lefthand side of the two productions are different and a minimal invalid subparse can be found which includes a swop from the one occurrence to the other and a return to a common production at a lower level (i.e. closer to the root) of the parse tree,

then the grammar is *terminal confusable over x.*

2. *Nonterminals*

Just as confusion can occur between terminals, it can also occur between two occurrences of a nonterminal. For example,

$$Z \rightarrow \vdash S \dashv$$
$$S \rightarrow ExE$$
$$E \rightarrow i$$

for which the sentence $\vdash i \dashv$ would be acceptable.

The conditions for confusion in this case are similar to those for terminals but with an additional condition to take account of the fact that nonterminals are not placed on the stack and therefore cannot shield what precedes the nonterminal from what follows it. Thus a suitable precedence relation must exist between the symbol to the left of the nonterminal in the one production and the symbol to the right of it in the second.

Definition

If a grammar is not terminal confusable but contains two occurrences of a nonterminal symbol X on the righthand side of the set of productions ($A \rightarrow \alpha X \beta$ and $B \rightarrow \gamma X \delta$) and if

1. either

(*a*) the symbol on the lefthand side of the two productions is

the same and the two occurrences are neither left identical nor right identical,

or

(*b*) the symbols on the lefthand side of the two productions are different and a minimal invalid subparse can be found which includes a swop from the one occurrence to the other and a return to a common production at a lower level of the parse tree, and

2. either

(*a*) $\alpha = \alpha' x$, $\delta = y\delta'$ and $x \doteq_2 y$,

(*b*) $\alpha = \delta = \varepsilon$,

(*c*) $\alpha = \alpha' x$, $\delta = \varepsilon$ and there exists z such that $Z = >^* \sigma A z \tau$ and $x \geqslant_2 z$,

or

(*d*) $\alpha = \varepsilon$, $\delta = y\delta'$ and there exists z such that $Z = >^* \sigma z A \tau$ and $z \leqslant_2 y$,

then the grammar is *nonterminal confusable over X*.

Theorem

If a grammar which is not terminal confusable contains two occurrences of a nonterminal X on the righthand side of the set of productions, then confusion between these two occurrences can lead to an invalid string which is acceptable to a simple parser if and only if the grammar is nonterminal confusable over X for these two productions.

Proof

Assuming that a symbol occurs more than once on the righthand side of a set of productions, and that confusion could occur between one occurrence of a symbol in one production and a separate occurrence of the same symbol in the same or a different production, then the set of conditions 1 determine whether a suitable reverse swop could be made so that an invalid string might be accepted by the parser. In the case of terminal symbols once a terminal symbol is reached, it is placed on the stack shielding the terminals preceding it which have already been placed on the stack. Hence a swop can occur from one occurrence of a terminal to another elsewhere in the productions. In order for the parser to accept this string as valid, the reverse swop conditions must be met. Thus the conditions 1. are sufficient for confusability of terminal symbols.

However, in the case of nonterminals, these are not stored on the stack; hence the need for the additional condition 2. which enables a swop to occur. The reverse swop conditions 1. are identical to those for terminal symbols. These have been proved for the case of terminal symbols in Lemmas 1 and 2, and a similar proof can be constructed for nonterminal symbols. It remains to prove that conditions 2. are sufficient for a swop to occur.

Consider each condition in turn.

(*a*) $\alpha = \alpha' x$, $\delta = y\delta'$. The additional condition in this case is

$$x \doteq_2 y$$

In this case one either has

$$\begin{array}{ll} Z = >^* \sigma A \tau & \\ A \rightarrow \alpha' x X \beta & \beta \neq y\delta' \\ A \rightarrow \gamma X y \delta' & \gamma = \alpha' x \end{array}$$

in which case it is possible for the parser to accept a sentence derived from the invalid sentential form

$$\sigma\alpha' x X y \delta' \tau$$

or one has

$$\begin{array}{l} A \rightarrow \alpha' x X \beta \\ B \rightarrow \gamma X y \delta' \end{array}$$

and a minimal invalid subparse which contains the phrase

$$\alpha' x X y \delta'$$

If the condition $x \doteq_2 y$ is removed then $\alpha' x X$ will be accepted by the parser but if the terminal y is encountered next, then either

(i) $x \leqslant_2 y$ which implies that there exist productions

$$\begin{array}{l} X \rightarrow \sigma x Y \tau \\ Y \rightarrow W y \eta \end{array}$$

or

(ii) $x \geqslant_2 y$ which implies that there exist productions

$$\begin{array}{l} X \rightarrow \sigma Y y \tau \\ Y \rightarrow \eta x W \end{array}$$

In both cases if a sentence derived from a sentential form containing the phrase $\alpha' xXy$ is accepted by a simple parser then a swop must have occurred to one of these latter sets of productions and a minimal invalid subparse must exist for the swop. Under these conditions the grammar must also be terminal confusable over x—which contradicts the original assumption.

Or

(iii) no suitable precedence relation exists between x and y. Hence it is not possible to swop from the string $\alpha' x X \beta$ to the string $\gamma X y \delta'$ and hence accept an invalid sentence.

(*b*) $\alpha = \delta = \varepsilon$

As above it can be shown that this condition leads to invalid strings being accepted. No additional condition is necessary in this case since the part of the production to the left of the first occurrence of X and the part of the production to the right of the second occurrence of X are both null.

(*c*) $\alpha = \alpha' x$, $\delta = \varepsilon$. The additional condition in this case is There exists a terminal z such that $Z = >^* \sigma A z \tau$ and $x \geqslant_2 z$. If both nonterminals on the lefthand side are the same, i.e.

$$\begin{array}{ll} A \rightarrow \alpha' x X \beta & \beta \neq \varepsilon \\ A \rightarrow \gamma X & \gamma \neq \alpha' x \end{array}$$

then it is obvious that a simple parser will accept a sentence derived from the invalid sentential form

$$\sigma\alpha' x X z \tau \qquad z\tau \neq \beta$$

If the additional condition were dropped, and if the parser accepted a string derived from . . . $\alpha' xX$ and the next symbol in the sentence is z, then either

(i) $x \doteq_2 z$ or $x \leqslant_2 z$ in which case there exist productions such as

$$\begin{array}{ll} B \rightarrow \sigma x C z \tau & \text{or } B \rightarrow \sigma x C \tau \\ & C \rightarrow D z \psi \end{array}$$

If a sentence derived from a sentential form containing the phrase $\alpha' xXz$ is accepted by the parser, then the grammar must also be terminal confusable over z which violates the original assumption; or

(ii) $x \geqslant_2 z$ but $Z \neq >^* \sigma A z \tau$ in which case there must exist productions such as

$$\begin{array}{l} B \rightarrow \sigma C z \tau \\ C \rightarrow \psi x D \end{array}$$

Again if a sentence derived from a sentential form containing the phrase $\alpha' xXz$ is accepted by the parser then the grammar must also be terminal confusable over x.

Hence it follows that there must exist a terminal such that

$$Z = >^* \sigma A z \tau \text{ and } x \geqslant_2 z$$

The same argument can be applied to the case where the nonterminals on the lefthand side are different, i.

$$A \rightarrow \alpha' xX\beta$$
$$B \rightarrow \gamma X$$

(d) $\alpha = \varepsilon$, $\delta = y\delta'$. The additional condition is:
There exists a terminal z such that $Z = >^* \sigma zA\tau$ and $z \lessdot_2 y$. In this case one either has

$$Z = >^* \sigma zA\,\tau$$
$$A \rightarrow X\beta \qquad \beta \neq y\delta'$$
$$A \rightarrow \gamma Xy\delta' \qquad \gamma \neq \varepsilon$$

in which case it is possible to accept a sentence derived from the invalid sentential form

$$\sigma zXy\delta'\,\tau$$

or one has

$$A \rightarrow X\beta$$
$$B \rightarrow \gamma Xy\delta'$$

and a minimal invalid subparse which contains the phrase

$$\sigma Xy\delta'\,\tau \qquad \sigma \neq \gamma$$

Again if the additional condition were dropped, then assume that there exists some terminal a which can precede A in a sentential form, i.e.

$$Z = >^* \sigma aA\tau$$

Then it is possible for the parser to recognise . . . aX; however, if a swop is to occur, the next symbol must be y. There are three possible precedence relations between a and y:

(i) $a \doteq_2 y$ which implies that there exists a production of form

$$F \rightarrow \psi aWy\chi$$

and that the parser would actually swop to this production.

(ii) $a \gtrdot_2 y$ which implies that there exist productions of the form

$$F \rightarrow \psi Hy\chi$$
$$H \rightarrow \eta aI$$

and that the parser would swop to this second occurrence of y.

In either case if the parser would accept such a swop as valid then the grammar must also be terminal confusable over a, thus violating the original assumption.

(iii) $a \lessdot_2 y$ In this case the parser could quite readily mistake its position and assume that it has reached the point $\gamma X \uparrow y\delta'$. Hence it would be possible for the parser to accept a sentence derived from

$$\ldots aXy\delta'$$

and hence from condition 1. of the definition of nonterminal confusability an invalid sentence will be accepted by a simple parser.

Thus the additional condition of (d) is proved, viz. that there exists a terminal a (or w) such that $Z = >^* \sigma aA\tau$ and $a \lessdot_2 y$.

4. Conclusion

The extended operator precedence parsing method will accept only valid sentences if the grammar is an operator precedence grammar which is uniquely invertible with respect to terminals and if it is neither terminal confusable nor nonterminal confusable.

The chief problem with determining whether a grammar is terminal or nonterminal confusable lies in the difficulty in determining a minimal invalid subparse. However, if this condition is dropped, i.e. condition 2. of the definition of terminal confusability and condition 1(a) of nonterminal confusability become

'The symbols on the lefthand side of the productions are different',

a grammar can easily be checked for this more general condition and only if it fails does one need to resort to determining minimal invalid subparses.

Since the extended operator precedence parsing method is a useful method, the conditions necessary for it to operate successfully are important and further consideration might be given to this problem of determining minimal invalid subparses.

References

GRIES, D. (1971). *Compiler Construction for Digital Computers*, John Wiley, New York.

HENDERSON, D. S., and LEVY, M. R. (1976). An Extended Operator Precedence Parsing Algorithm, *The Computer Journal*, Vol. 19 No. 3, pp. 229-233.

New approaches to systems analysis and design

A one-day seminar held as a tribute to Eric Mutch

On Wednesday 20 June 1979 at the Kensington Close Hotel, London W8, *The Computer Journal* is promoting a one-day seminar, designed especially to introduce to practising systems analysts in administration, commerce and industry the changes which are impacting their job function as a consequence of

(a) changing technology, and

(b) changes in organisational structures, social awareness, and legislation.

The emphasis is to be on practical examples taken from industry and the public services in the United Kingdom but the programme also includes an account of corresponding developments in Europe. The fee for the day is £55 (£45 to BCS members).

To register, or to obtain further details, contact Mrs. C. Cox, Meetings Executive, BCS headquarters, 13 Mansfield Street, London WIM 0BP.

22. M.H.WILLIAMS, A systematic test for extended operator precedence, Information Processing Letters, vol. 13, nos. 4-5, 187-190 (1981).

A SYSTEMATIC TEST FOR EXTENDED OPERATOR PRECEDENCE

M.H. WILLIAMS
Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ, U.K.

Received 14 July 1981

Syntax analysis, parsing, extended operator precedence

1. Introduction

There are many variations on the precedence technique for parsing sentences of a grammar. One such variation, called extended operator precedence, was described by Levy [2], and Henderson and Levy [1]. This is a modification of operator precedence which uses two types of precedence relations. The first set of relations ($<\cdot_1$, $\cdot>_1$ and $\doteq_1$) are used to indicate relationships between two terminals which can occur adjacent to one another in a sentential form; the second set ($<\cdot_2$, $\cdot>_2$ and $\doteq_2$) denote relationships between terminals which occur with a nonterminal between them in a sentential form.

With an ordinary operator precedence parser one must either store a set of productions and scan these whenever a handle is recognised or resort to ad hoc routines in order to avoid the problem of accepting invalid strings. The extended operator precedence parsing algorithm requires neither of these. Provided that the grammar is suitable, the two types of precedence relations are sufficient to ensure that only valid sentences are accepted by this type of parser.

However, the original criteria which Henderson and Levy proposed for determining whether this parsing technique can be applied for a particular grammar, are incorrect and a correct set of conditions were put forward in [3]. Unfortunately these conditions are difficult to apply in this form. For this reason an algorithm is described here for testing whether or not a grammar satisfies the conditions for extended operator precedence.

2. Some definitions

Consider two productions which have a common symbol x (nonterminal or terminal) occurring on their right hand sides, i.e.

$$A \to \alpha x \beta, \qquad B \to \gamma x \delta.$$

If the portion of the production to the left of x in the one production is the same as the portion of the production to the left of x in the other, i.e. $\alpha = \gamma$, then the two productions are *left identical with respect to x.*

Similarly, if the portion of the production to the right of x in the one production is the same as the portion of the production to the right of x in the other, i.e. $\beta = \delta$, then the two productions are *right identical with respect to x.*

Now consider two productions which have a common nonterminal symbol X occurring on their right hand sides, i.e.

$$A \to \alpha X \beta, \qquad B \to \gamma X \delta.$$

If one of the following conditions holds [3]:

(a) $\alpha = \alpha' x$, $\delta = y\delta'$, $x,y \in V_T$ and $x \doteq_2 y$,

(b) $\alpha = \delta = \epsilon$,

(c) $\alpha = \alpha' x$, $\delta = \epsilon$, and there exists z $(x,z \in V_T)$ such that $Z \Rightarrow *\sigma A z \tau$ and $x \cdot>_2 z$, or

(d) $\alpha = \epsilon$, $\delta = y\delta'$, and there exists z $(y,z \in V_T)$ such that $Z \Rightarrow *\sigma z A \tau$ and $z <\cdot_2 y$,

where Z is the goal symbol, then the two occurrences of X are said to be *potentially mistakable.*

The algorithm described in the next section makes

0020-0190/81/0000–0000/$02.75 © 1981 North-Holland

use of a set S of all terminal and potentially mistakable nonterminal symbols which occur on the right hand side of more than one production. Obviously if no terminal or potentially mistakable nonterminal occurs on the right hand side of more than one production, i.e. the set S is empty, then there is no problem. However, if S is not empty then it is possible that confusion may occur. Thus given some production the right hand side of which contains a symbol X from S,

$$A \to \alpha X \beta, \quad \alpha,\beta \in (V_N \cup V_T)^* \quad X \in S,$$

if some other production can be found whose right hand side contains the same symbol,

$$B \to \sigma X \tau,$$

then B is said to be *directly mistakable for A*. Likewise if the part of this latter production which follows X, viz. τ, contains another element of S, Y, i.e.

$$B \to \sigma X \gamma Y \delta,$$

and some other production can be found whose right hand side also contains Y,

$$C \to \mu Y \nu,$$

then C is also *directly mistakable for A*. In other words if one set out attempting to recognize the nonterminal A and in the process switched to the production for B or for C then B and C are directly mistakable for A (see Fig. 1).

If some string ω can be produced as a result of such a mistake (or sequence of mistakes), starting from some symbol A, then this will be written as: $A \Rightarrow^M \omega$ (i.e. A mistakenly produces ω). For example from the previous productions, $A \Rightarrow^M \alpha X \gamma Y \delta$ or $A \Rightarrow^M \alpha X \gamma Y \nu$.

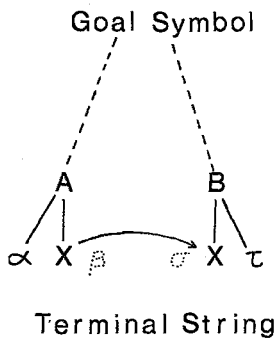


Fig. 1. While attempting to recognise the nonterminal A the parser accidentally switches productions in mid-stream and finishes up with a production for B (B is directly mistakable for A).

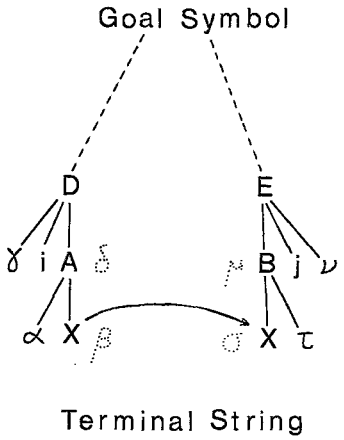


Fig. 2. Another example of switching productions in midstream (E is mistakable for A).

If some nonterminal B is directly mistakable for A then the confusion may extend to lower levels in the parse tree (i.e. closer to the goal symbol) – to nonterminals which can produce strings containing A and B. Thus B is said to be *mistakable for A* if one of the following is true:

(a) B is directly mistakable for A;

(b) a string derived from A contains some nonterminal C and B is directly mistakable for C;

(c) a string derived from B contains some nonterminal C which is directly mistakable for A; or

(d) a string derived from B contains some nonterminal C and a string derived from A contains some nonterminal D and C is directly mistakable for D.

For example, if B is directly mistakable for A and

$$D \to \gamma i A \delta, \quad E \to \mu B j \nu,$$

are productions of the grammar and $i \doteq_2 j$ then E is mistakable for D (see Fig. 2).

3. The algorithm

In order to verify whether or not a grammar is suitable for this parsing technique, one must first construct the precedence tables T_1 and T_2 for the grammar. If there are any nonunique precedence relations in either table, the grammar is obviously

unsuitable and one need continue no further.

Once the tables have been constructed, search the right hand sides of the productions for any potentially mistakable nonterminals and terminals which occur more than once and place any such symbols in the set S. If there are no repeated terminal or potentially mistakable nonterminal symbols (i.e. S is empty) then the grammar is an extended operator precedence grammar. If S is not empty, proceed as follows:

(1) Consider in turn each pair of productions containing occurrences of the same terminal or potentially mistakable nonterminal symbol. Suppose these are:

$A \to \alpha x \beta, \quad B \to \gamma x \delta.$

(a) If the nonterminals on the left hand sides of the two productions are identical (A = B) then: if the pair of productions is neither right identical with respect to x nor left identical with respect to x then the grammar is unsuitable otherwise mark the two occurrences of x.

(b) If the nonterminals on the left hand sides of the productions are different then if the pair of productions is not right identical with respect to x, search tables T_1 and T_2 for a terminal symbol y such that the last terminal symbols of each of the two productions concerned take precedence over y. For example, if the two productions end with terminals u and v respectively and there exists a terminal y such that $u \cdot\!>_1 y$ and $v \cdot\!>_1 y$ then y is the terminal being sought. If such a terminal symbol is found then the grammar is not suitable for this parsing technique.

(2) For each nonterminal $A \in V_N$ construct the set of nonterminals which are mistakable for A as follows:

(a) For each nonterminal $A \in V_N$, let MISTAKABLE(A) $\leftarrow \emptyset$.

(b) For each occurrence of an element of S on the right hand side of a production, say

$A \to \alpha X \beta, \quad X \in S,$

find all nonterminals which are directly mistakable for A and add them to MISTAKABLE(A), except that if X is a marked symbol then do not include any instance of A arising as a result of direct confusion with another marked occurrence of X.

(c) For each nonterminal $A \in V_N$ such that MISTAKABLE(A) $\neq \emptyset$ perform the following:

For each element $X \in$ MISTAKABLE(A) do: find each pair of nonterminals B, D such that one of the following three points holds:

$B \to \alpha a C_\beta, \quad C \overset{*}{\Rightarrow} A\sigma,$

$D \to \gamma E b \delta, \quad E \overset{*}{\Rightarrow} \tau X \quad \text{and} \quad a \doteq_2 b.$

Point (1) ends here.

$B \to \alpha_0 a_0 C_0 \beta_0,$

$C_j \overset{+}{\Rightarrow} \alpha_{j+1} a_{j+1} C_{j+1} \beta_{j+1} \quad 0 \leqslant j \leqslant n-1,$

$C_n \overset{*}{\Rightarrow} A\sigma, \quad D \to \gamma_0 E_0 b_0 \delta_0,$

$E_j \overset{+}{\Rightarrow} \gamma_{j+1} E_{j+1} b_{j+1} \delta_{j+1} \quad 0 \leqslant j \leqslant m-1,$

$E_m \overset{*}{\Rightarrow} \tau X,$

and each $a_i <\cdot_2$ or $\cdot>_2$ the appropriate b_j, $a_0 \doteq_2 b_0$. Point (2) ends here.

$B \to \alpha_0 a_0 C_0 \beta_0, \quad C_n \overset{*}{\Rightarrow} A\sigma,$

$C_j \overset{+}{\Rightarrow} \alpha_{j+1} a_{j+1} C_{j+1} \beta_{j+1} \quad \text{or} \quad C_j \overset{M}{\Rightarrow} \alpha_{j+1} a_{j+1} C_{j+1} \beta_{j+1}$

$0 \leqslant j \leqslant n-1,$

$D \to \gamma_0 E_0 b_0 \delta_0, \quad E_m \overset{*}{\Rightarrow} \tau X,$

$E_j \overset{+}{\Rightarrow} \gamma_{j+1} E_{j+1} b_{j+1} \delta_{j+1} \quad \text{or} \quad E_j \overset{M}{\Rightarrow} \gamma_{j+1} E_{j+1} b_{j+1} \delta_{j+1}$

$0 \leqslant j \leqslant m-1,$

and each $a_i <\cdot_2$ or $\cdot>_2$ the appropriate b_j, $a_0 \doteq_2 b_0$. Point 3 ends here.

If D is not contained in MISTAKABLE(A) then add D to MISTAKABLE(A).

If either D or X (or both) is not contained in MISTAKABLE(B) then add it (them) to MISTAKABLE(B).

Repeat this step until there is no further change to the sets MISTAKABLE.

(d) If for any nonterminal $A \in V_N$, MISTAKABLE(A) contains A then the grammar is unsuitable.

If the grammar is not found to be unsuitable in steps 1 and 2 then the grammar is an extended operator precedence grammar.

4. Examples

To illustrate some aspects of the algorithm outlined in the previous section, consider a few sample grammars, $G = (V_N, V_T, Z, P)$:

Example 1

$V_N = \{S, A\}, \quad V_T = \{a, b, c, d, e\},$

$$Z = S, \quad P = \left\{ \begin{array}{l} S \to Aa \\ A \to bcd \\ A \to dce \end{array} \right\},$$

is not suitable (by step 1(a)) but if

$$P = \left\{ \begin{array}{l} S \to Aa \\ A \to bcd \\ A \to bce \end{array} \right\} \quad \text{or} \quad P = \left\{ \begin{array}{l} S \to Aa \\ A \to bd \\ A \to ce \end{array} \right\},$$

the grammar will be an extended operator precedence grammar.

Example 2

$V_N = \{S, A, B, C\}, \quad V_T = \{a, b, c, d, e, f, g, h, i, j\},$

$$Z = S, \quad P = \left\{ \begin{array}{l} S \to gAai \\ S \to hBaj \\ A \to Cbc \\ B \to AbeA \\ C \to f \end{array} \right\}$$

is unsuitable because confusion can occur between the two occurrences of b since $c \cdot\!>_1 a$ and $e \cdot\!>_2 a$ (step 1(b)) (i.e. the parser would accept the string gfbefbcai as valid).

Example 3

$V_N = \{X, A, B, C, D, E, F, G\},$

$V_T = \{a, b, c, d, e, f, g, h, i, j, k, m, n, x, y\}, \quad Z = X,$

$$P = \left\{ \begin{array}{ll} X \to xAy & D \to fEh \\ A \to aBd & E \to eFf \\ A \to bDd & F \to gG \\ B \to Cc & G \to mjn \\ C \to ijk & \end{array} \right\}.$$

The set S = {d, f, j}, and

	AFTER 2(b)		AFTER 2(c)
MISTAKABLE(X) =	$\emptyset$	=	$\emptyset$
MISTAKABLE(A) =	$\emptyset$	=	{A, G}
MISTAKABLE(B) =	$\emptyset$	=	$\emptyset$
MISTAKABLE(C) =	{G}	=	{G, A}
MISTAKABLE(D) =	{E}	=	{E}
MISTAKABLE(E) =	{D}	=	{D}
MISTAKABLE(F) =	$\emptyset$	=	$\emptyset$
MISTAKABLE(G) =	{C}	=	{C}

and hence (by step 2) since MISTAKABLE(A) contains A, the grammar is unsuitable.

Conclusion

The algorithm outlined in this paper will determine whether or not a grammar is an extended operator precedence grammar and hence whether or not the extended operator precedence parsing technique can be used to parse sentences of the associated language. Although this parsing technique has some appeal, the class of grammars which is suitable is unfortunately rather limited.

References

[1] D.S. Henderson and M.R. Levy, An extended operator precedence parsing algorithm, Comput. J. 19(3) (1976) 229–233.

[2] M.R. Levy, Complete operator precedence, Information Processing Lett. 4(2) (1975) 38–40.

[3] M.H. Williams, Conditions for extended operator precedence parsing, Comput. J. 22(2) (1979) 164–168.

23. M.H.WILLIAMS, Long/short address optimisation in assemblers, Software Practice and Experience, vol. 9, no. 3, 227-235 (1979).

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 9, 227–235 (1979)

Long/Short Address Optimization in Assemblers

M. H. WILLIAMS

Computer Science Department, Rhodes University, Grahamstown, South Africa

Summary

In a number of computer systems an instruction in symbolic assembly language may be translated as either a long form or a short form instruction in machine code depending on certain conditions. Assemblers which optimize the size of a program by using short form instructions wherever possible, sometimes require three or more passes to achieve this. This paper describes an approach to solving this type of problem which can be implemented in a two-pass or even a one-pass assembler.

KEY WORDS Optimization Assemblers Variable-length address Span-dependent instructions

INTRODUCTION

Ignoring macros, one generally thinks of assemblers as performing a simple one-to-one translation between some symbolic form of machine code and its equivalent in binary. But this process can become more complicated if a particular symbolic opcode may be translated in more than one way resulting in different binary versions. For example, one machine architecture[1] incorporates two forms of branch instruction:

1. 4-byte branch instructions—these have a full address field and are capable of branching anywhere in memory;
2. 2-byte branch instructions—these have a very small address field and can be used for relative branching to addresses up to 30 bytes away from the current instruction.

An assembler for such a machine could rely on the user to specify whether a long or short branch instruction is required by providing different mnemonics for the two. Alternatively, the assembler could perform optimization and generate the short form wherever possible and the long form only when necessary.

The problems with an optimizing assembler which handles long and short branches in an optimum manner, are:

1. When one encounters a branch to a symbol, unless one is sure that the label is definitely more than or definitely less than or equal to 30 bytes away, it is not clear whether the instruction concerned should be a 2-byte or a 4-byte instruction.
2. When a label is encountered in the label field (call this a label definition) after one or more instructions whose lengths are uncertain, the exact location of that label is uncertain.

In order to handle this, an optimizing assembler may resort to (and some do) more than two passes of the source statements. Szymanski[2] has just published a method in which only two passes are required. The first pass constructs a symbol table and stores information about the span-dependent instructions. Between passes 1 and 2 a dependency graph is constructed and from this addresses are finalized before proceeding to the second pass.

0038–0644/79/0309–0227 $01.00 *Received 31 July 1978*
© 1979 by John Wiley & Sons, Ltd.

This paper describes an alternative approach in which addresses are finalized in the first pass as the assembler proceeds through the source code. By the end of the first pass all addresses have been finalized. This method can even be used in a one-pass assembler. The method will be described in terms of the 2-byte/4-byte example aforementioned.

A TWO PASS ASSEMBLER FOR LONG/SHORT BRANCHES

Only the first pass of a two pass assembler is affected by this. The first pass scans the source material and sets up the Symbol Table. However, in order to cater for the uncertainty about whether an instruction should be translated as a short branch or a long branch we introduce:

1. An additional table, the Branch Pending Table, which contains information about branch instructions and label definitions which have not yet been finalized.
2. A pair of location counters, LC_{min} and LC_{max}, which specify the extreme values which the location counter may have at any point.

Initially the pair of location counters contain the same value LC_0. If there is no uncertainty about the current value of the location counter (i.e. $LC_{min} = LC_{max}$) when a label definition is encountered, the appropriate entry is inserted in the Symbol Table with the *ADDRESS* field set to the value of LC_{min} or LC_{max}. Similarly, if an instruction is encountered for which there is no doubt about its length (it is a non-branch instruction or a backward branch which branches to an address which is definitely more than or definitely less than or equal to 30 bytes away) then this length is added to both LC_{min} and LC_{max}.

However, if one encounters an instruction for which there is some doubt about its length (i.e. a forward branch instruction and one does not know how far ahead the label definition occurs or a backward branch which may or may not lie within range), a 'reference' type entry is added to the Branch Pending Table and LC_{min} is incremented by 2 and LC_{max} by 4. If one is in a state in which $LC_{min} \neq LC_{max}$ and a label definition is encountered (i.e. there is uncertainty about the location of this label) a 'definition' type entry is added to the Branch Pending Table.

Entries are removed from the Branch Pending Table at two points in the assembly process:

1. Whenever the location counter pair LC_{min}, LC_{max} are updated, the Branch Pending Table is checked to see whether any reference type entry, for which the label definition has not yet been encountered, is definitely more than 30 bytes from the current location. If so, this is a long branch instruction and the entry can be removed from the Branch Pending Table and the other entries updated accordingly. If this is done, the table is checked for any further simplifications.
2. When a label definition is encountered, the Branch Pending Table is checked to see whether it contains a reference to that symbol. If it does and the branch instruction is obviously within range for a short branch instruction, then a short branch instruction is used, the entry removed from the Branch Pending Table and the remaining entries updated accordingly. Again the table is checked for further simplification which may arise as a result of this.

Except possibly when the Branch Pending Table is being simplified, if the table is non-empty then the bottom entry must be a reference type entry. The current values of LC_{min} and LC_{max} will be different if and only if the Branch Pending Table is non-empty, and the difference between them must be twice the number of reference type entries in this table.

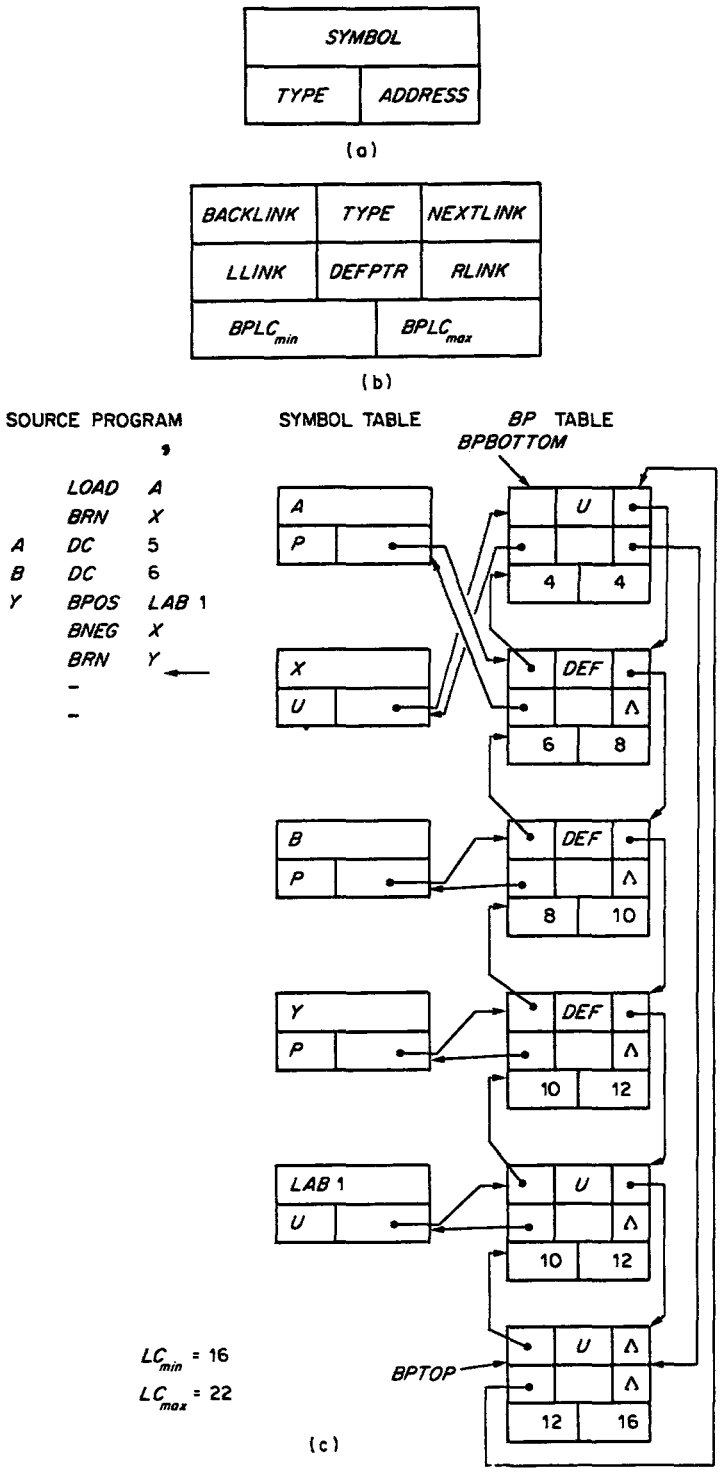


Fig. 1. (a) Format of Symbol Table entry. (b) Format of Branch Pending Table entry. (c) The Symbol Table and Branch Pending Table as set up by the first pass of the assembler when it has reached the point indicated in the source program on the left

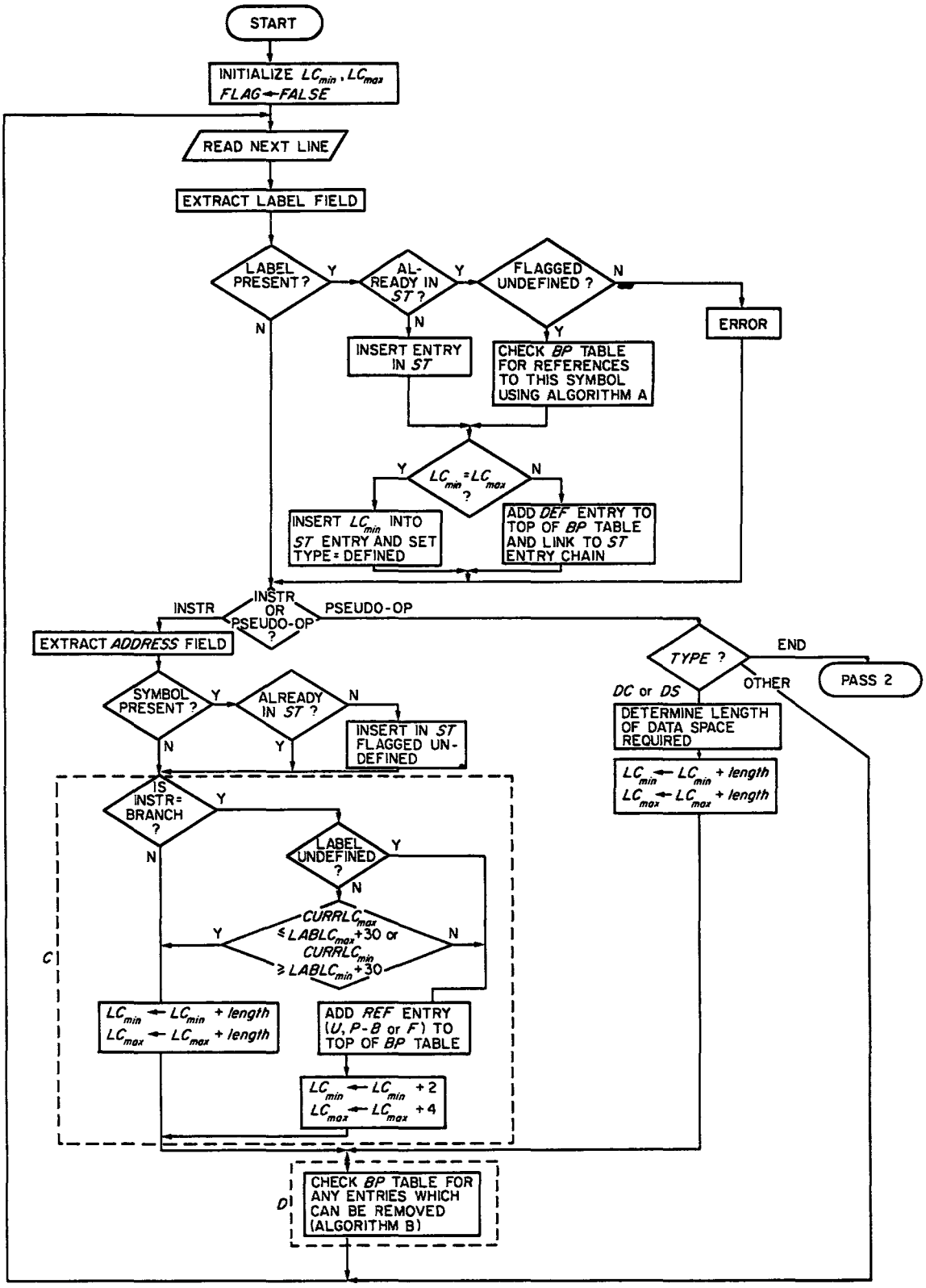


Figure 2. Flow diagram of the first pass of an assembler which performs long/short branch optimization

In order to show how the algorithm works it has been incorporated into a very simple assembler. The format of Symbol Table entries for this simple assembler is given in Figure 1(a) while the format of Branch Pending Table entries is given in Figure 1(b). The *TYPE* field of *ST* entries may contain *U* (undefined), *P* (pending—i.e. label definition encountered but it is presently on the Branch Pending Table) and *D* (defined). The *ADDRESS* field of *ST* entries either contains the final address of the symbol (if *TYPE* = *D*) or a pointer to a chain of entries in the Branch Pending Table which refer to it. The *TYPE* field of Branch Pending Table entries may contain *F* (fixed—i.e. reference to symbol whose address has been finalized), *P–F* (forward branch to a label which is pending), *P–B* (backward branch to a label which is pending), *U* (forward branch to a label as yet undefined) and *DEF* (a label definition). Figure 1(c) illustrates the position of the Symbol Table and the Branch Pending Table when assembling the source program on the left-hand side of the figure.

The first pass of a two pass assembler which caters for long/short branch optimization is shown in Figure 2 while algorithms A and B follow. These two algorithms make use of pointer variables, *P*, *Q* and *R*, and a list *TEMPLIST* which is used to store addresses when checking for further optimization.

In brief the method works as follows.

1. When a branch to an undefined label is encountered, an entry is inserted into the Branch Pending Table. If subsequently LC_{min} moves up more than 30 bytes without encountering the label definition, the branch must be a long branch and the entry is removed and dealt with accordingly. On the other hand, if the label definition is found before LC_{max} has increased by more than 30 bytes then the branch is a short branch and the entry is removed and dealt with.
2. When a branch to a defined label is encountered (i.e. a backward branch to a pending or fixed label), a long branch must be used if LC_{min} has increased by more than 30 bytes, a short branch must be used if LC_{max} has increased by not more than 30 bytes and otherwise an entry is added to the Branch Pending Table.
3. When one determines that a pending branch should be a short branch and removes the entry from the Branch Pending Table, other simplifications may become possible—these will all be in the direction of short branches (e.g. if in Figure 3 the values of

A	—	$LC_{min} = 100$, $LC_{max} = 100$
	—	
	BZ B	$LC_{min} = 128$, $LC_{max} = 128$
	BNZ A	$LC_{min} = 130$, $LC_{max} = 132$
	—	
B	—	$LC_{min} = 158$, $LC_{max} = 162$

Figure 3. A deadlock situation

LC_{min}, LC_{max}, had been 148, 152 at the label definition *B*). Likewise if one determines that a pending branch should be a long branch, further long branch simplifications may become possible (e.g. if in Figure 3, LC_{min}, LC_{max} had values 168, 172 at the label definition *B*).
4. There are deadlock situations such as that shown in Figure 3 where no further information will help to decide whether short or long branches are to be used. In this case if one can isolate a label definition such that all pending branches prior to this label refer either to this label or to other labels prior to this label, then these pending branches may be replaced by short branches.

Assuming that there are no undefined labels in a program, then by the time one reaches the final label definition of the program, the Branch Pending Table must be empty (if only by virtue of paragraph 4 above). If undefined labels are not treated as errors then they could be translated as long branches.

ANOTHER FORM OF LONG/SHORT ADDRESSING

The 6800 microcomputers[3] have two forms of long/short addressing. Besides long/short branches (-125 to $+129$ bytes) certain instructions, e.g. *LDA* (load accumulator), *STA* (store accumulator), etc., may be translated either as 2-byte instructions or 3-byte instructions depending on whether the address to which they refer is less than 256 bytes (direct addressing) or greater than or equal to this address (extended addressing).

For this purpose one could use a variation of the flowchart in Figure 2 (with the portions labelled C and D replaced by the corresponding subdiagrams in Figure 4. Algorithms A and B need to be modified appropriately.

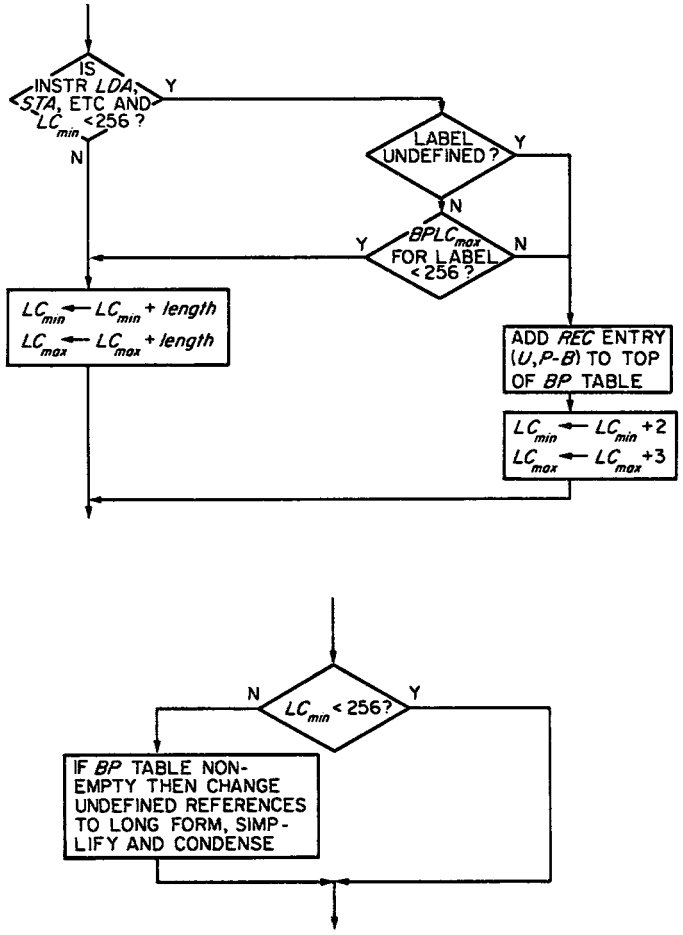


Figure 4. The subdiagrams which should replace C *and* D *in Figure 2 to cater for long/short addressing on the 6800*

ALGORITHM A

Algorithm A (follows down chain emanating from *ST* entry looking for branches which can definitely be established as short branches; if it finds any, it checks for further simplifications).

1. Set $P \leftarrow ADDRESS$ field of *ST* entry, $FLAG2 \leftarrow FALSE$
2. **while** $P \neq \Lambda$ **do**
   if $BPLC_{max}(P) \geqslant LC_{max} - 30$ **then** make-short-branch **else**
   mark-dref
3. **if** *FLAG2* is *TRUE* **then** check-for-further-short-branches

make-short-branch =

1. Set $Q \leftarrow NEXTLINK(P)$
2. **while** $Q \neq \Lambda$ **do** $\{BPLC_{max}(Q) \leftarrow BPLC_{max}(Q) - 2;\ Q \leftarrow NEXTLINK(Q)\}$
3. $Q \leftarrow P;\ P \leftarrow RLINK(P)$
4. Unlink and delete node pointed to by Q
5. Set $FLAG \leftarrow TRUE,\ FLAG2 \leftarrow TRUE$

mark-dref =

1. Change $TYPE(P)$ to *P–F*
2. Insert pointer to definition entry into $DEFPTR(P)$
3. $P \leftarrow RLINK(P)$

check-for-further-short-branches =

1. Set $P \leftarrow BPBOTTOM,\ FLAG2 \leftarrow FALSE$
2. **while** $P \neq \Lambda$ **do**
   if $TYPE(P) = DEF$ **and** $P = BPBOTTOM$ **then**
   remove-defn-from-bottom **else**
   if $TYPE(P) = P\text{–}F$ **or** *P–B* **then** check-dref-short **else**
   if $TYPE(P) = F$ **then** check-fref-short **else** $P \leftarrow NEXTLINK(P)$
3. **if** *FLAG2* is *TRUE* **then goto** 1

check-dref-short =

1. $R \leftarrow DEFPTR(P)$
2. **if** $TYPE(P) = P\text{–}B$
   then if $BPLC_{max}(P) \leqslant BPLC_{max}(R) + 30$ **then** make-short-branch-2
   else $P \leftarrow NEXTLINK(P)$
   else if $BPLC_{max}(P) \geqslant BPLC_{max}(R) - 30$ **then** make-short-branch-2
   else $P \leftarrow NEXTLINK(P)$

check-fref-short =

if $BPLC_{max}(P) \leqslant ADDRESS$ from *ST* entry $+ 30$ **then** make-short-branch-2
else $P \leftarrow NEXTLINK(P)$

remove-defn-from bottom =

1. $Q \leftarrow RLINK(P);\ R \leftarrow LLINK(P)$ [*R* points to *ST* entry]
2. Set *TYPE* field of *ST* entry to *DEFINED* and *ADDRESS* field of *ST* entry to $BPLC_{max}(P)$
3. Follow down *BP* chain starting at Q and change all *TYPE* fields to *F*, *DEFPTR* to value in *R*
4. $Q \leftarrow NEXTLINK(P)$ and delete entry pointed to by P; $P \leftarrow Q$

make-short-branch-2 =

1. Set $Q \leftarrow NEXTLINK(P)$
2. **while** $Q \neq \Lambda$ **do** $\{BPLC_{max}(Q) \leftarrow BPLC_{max}(Q) - 2;\ Q \leftarrow NEXTLINK(Q)\}$
3. $Q \leftarrow P;\ P \leftarrow NEXTLINK(P)$
4. Unlink and delete node pointed to by Q
5. Set $FLAG \leftarrow TRUE,\ FLAG2 \leftarrow TRUE$

ALGORITHM B

Algorithm B (checks through *BP* Table for any branches which can definitely be established as long branches; if it finds any, it checks for further simplifications).

1. Set $P \leftarrow BPBOTTOM,\ FLAG2 \leftarrow FALSE$
2. **while** $P \neq \Lambda$ **and** $BPLC_{min}(P) < LC_{min} - 30$ **do**
   if $TYPE(P) = U$ **then** make-long-branch **else** $P \leftarrow NEXTLINK(P)$
3. **if** *FLAG2* is *TRUE* **then** check-for-further-long-branches
4. **if** *FLAG* is *TRUE* **then** check-for-further-simplification

make-long-branch =

1. Set $Q \leftarrow NEXTLINK(P)$
2. **while** $Q \neq \Lambda$ **do** $\{BPLC_{min}(Q) \leftarrow BPLC_{min}(Q) + 2;\ Q \leftarrow NEXTLINK(Q)\}$
3. $Q \leftarrow P;\ P \leftarrow NEXTLINK(P)$
4. Unlink and delete node pointed to by Q
5. Set $FLAG \leftarrow TRUE,\ FLAG2 \leftarrow TRUE$

check-for-further-long-branches =

1. Set $P \leftarrow BPBOTTOM,\ FLAG2 \leftarrow FALSE$
2. **while** $P \neq \Lambda$ **do**
   if $TYPE(P) = DEF$ **and** $P = BPBOTTOM$ **then** remove-defn-from-bottom **else**
   if $TYPE(P) = P\text{–}F$ **or** $P\text{–}B$ **then** check-dref-long **else**
   if $TYPE(P) = F$ **then** check-fref-long **else** $P \leftarrow NEXTLINK(P)$
3. **if** *FLAG2* is *TRUE* **then goto** 1

check-dref-long =

1. $R \leftarrow DEFPTR(P)$
2. **if** $TYPE(P) = P\text{–}B$
   then if $BPLC_{min}(P) > BPLC_{min}(R) + 30$ **then** make-long-branch
   else $P \leftarrow NEXTLINK(P)$
   else if $BPLC_{min}(P) < BPLC_{min}(R) - 30$ **then** make-long-branch
   else $P \leftarrow NEXTLINK(P)$

check-fref-long =

if $BPLC_{min}(P) >$ *ADDRESS* from *ST* entry $+ 30$ **then** make-long-branch **else**
$P \leftarrow NEXTLINK(P)$

check-for-further-simplifications = (for each *DEF* type entry check whether all preceding entries are either *DEF* type entries, *P–F* or *P–B* entries whose definitions also occur in this set of preceding entries of *F* type entries)

1. $R \leftarrow BPBOTTOM$
2. **while** $R \neq \Lambda$ and $TYPE(R) \neq U$ **do**
   if $TYPE(R) = P\text{–}F$ **then** {add *DEFPTR* to *TEMPLIST*; $R \leftarrow NEXTLINK(R)$}
   else

if *TYPE*(*R*) = *DEF* **then** unpack-and-condense **else** *R*←*NEXTLINK*(*R*)
3. *FLAG*←*FALSE*

unpack-and-condense =

1. Remove every entry from *TEMPLIST* whose value equals that in *R*
2. **if** *TEMPLIST* is empty **then** condense

condense =

1. Set *P*←*BPBOTTOM*
2. **while** *P*≠*R* **do**
   if *TYPE*(*P*) = *DEF* **then** remove-defn-from-bottom **else** make-short-branch
3. remove-defn-from-bottom
4. *R*←*P*

REFERENCES

1. *Interdata 16-bit Processor User's Manual*, Perkin–Elmer, Oceanport, New Jersey (1976).
2. T. G. Szymanski, 'Assembling code for machines with span-dependent instructions', *Comm. ACM*, *21*, 300–308 (1978).
3. *M6800 Programming Reference Manual M68PRM(D)*, Motorola Inc., Phoenix, Arizona (1976).

24. M.H.WILLIAMS and J.M.MULLINS, A Basic compiler for undergraduate programs, Systems/Stelsels, vol. 4, no. 2, 8-16 (1974).

A basic compiler

for undergraduate programs

by M. H. Williams and J. M. Mullins*
Computer Science Department,
Rhodes University.

Introduction

In recent issues of SYSTEMS, two articles have appeared explaining the fundamental concepts behind compilers. The intention of this article is to describe an actual compiling system which was written to handle certain categories of student programs. The programming language chosen was BASIC, a conversational language which is very suitable for undergraduate programming. This article describes how it was implemented on a small single-programming machine.

The problem of generality

At Rhodes University we have a small single-programming computer which is used chiefly for research and teaching purposes. One of its disadvantages, however, is that, because the compiling systems provided for it are very general, the computer time required to compile and load a program is very long. For example, the average time taken to compile and load a Fortran program of 30-40 statements is of the order of two minutes.

When teaching a large class of students how to program, one reaches a stage where everyone is ready to write their first program. One may have, for example, a class of 100 Chemistry students each of whom is required to write one small program, and reckoning on the fact that each student will require three or four (or possibly more) compilation runs, then about a dozen hours of computer time must be set aside for the exercise. Imagine what effect this can have on one's normal daily schedule. For this type of situation one could happily sacrifice a little generality in favour of a more efficient compiling system.[1]

A compiler for teaching purposes?

Besides efficiency, what other features are desirable in a compiling system which is to be used for teaching undergraduate students in departments such as Chemistry or Psychology? Firstly, the language should be simple to teach and simple to learn. Since undergraduates in these departments are not interested in the detailed peculiarities of different programming languages, there is no point in teaching them a complicated programming language if a simpler one can be used which will serve their purposes equally well.

In the second place, the compiler should provide good error diagnostics. That is, the compiler should make it as easy as possible for the programmer to determine his error. Error situations should be classified carefully and error messages assigned which are meaningful even to the inexperienced programmer.

Finally, the system should provide adequate facilities for debugging programs. It is not sufficient to teach a student how to write programs; one must also teach him how to find and correct his mistakes. This task is made much simpler if one has reasonably powerful debugging aids to assist the beginner in locating his errors.

For these reasons, the language BASIC was selected and this paper describes an attempt to produce an efficient compiler with good error diagnostics for this language. In fact, there was another reason for wanting a BASIC compiler — we wanted to be able to demonstrate a conversational programming language to Computer Science students to give them an idea of what a multiaccess system looks like from the user's point of view.

Structure of the language

The commands available in our BASIC system can be divided roughly into three categories, viz.:

**J. M. Mullins is now at University of Natal, Durban.*

(i) Statements — these are the actual source program statements which are compiled into an object program, e.g. assignment statements, loops, branches, etc.
(ii) system commands — these are directives which are chiefly concerned with the storing, recalling, editing and executing of programs,
(iii) non-standard commands — commands which we have added to cater for our own particular needs.

Since BASIC is a conversational language, the normal input-output device used is the typewriter; however, the only typewriter attached to our machine is the console typewriter. Thus the compiler was designed to allow input/output through the console typewriter for demonstration purposes, or through the card reader/line printer for normal batch work, depending on the setting of a switch, controlled by a non-standard command.

The system was designed as an in-core compiling system. It can be regarded as being composed of three main segments controlled by a small controlling segment. The three main segments are:

(i) the COMPILER itself, which compiles source language statements into object code,
(ii) the LINKER, which links together the blocks of object code prior to execution, and
(iii) the SYSTEM COMMAND INTERPRETER, which takes care of system commands.

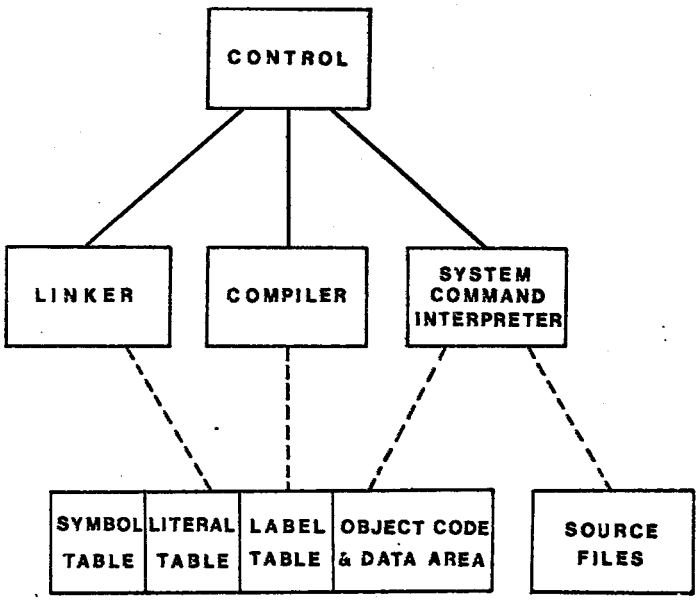


Figure 1. *A block diagram of the compiling system.*

These share a common data area which includes the symbol table, literal table, label table, object code and data area, plus the source files on disk.

Basic statements

A distinguishing feature of BASIC compared with languages such as Fortran or Algol is that it caters for both string and numeric quantities. Thus constants can be numeric (e.g. —6,3 0,87E—13 etc.) or string (any sequence of characters enclosed in quotes, e.g. "HE ▽ SAID"). Likewise there are two types of single variable in BASIC — arithmetic variables and string variables. Furthermore there is no distinction between integer and real variables — arithmetic variables are all of one type and are stored in floating point form. An arithmetic variable has a name of the form:

<letter> or <letter> <digit>

for example, A, X5, etc.

A string variable is used to store a string of characters. It has a name of the form:

<letter>$ or <letter> <digit>$

for example, A$, Z3$, etc. Similarly there are two types of arrays in BASIC — arithmetic arrays and string arrays. These may have one or two dimensions.

Arithmetic expressions consist of arithmetic variables, arithmetic array elements, numeric constants and standard functions (such as LOG, EXP, SQR, etc.) combined in the normal way by means of the operators +, —, *, /, ** (and). String expressions consist of string variables and string constants combined by the operator & (denoting concatenation).

TABLE 1.

The Main Program Commands in Basic

ASSIGNMENT:		LET v = e
BRANCH:	**Unconditional:**	GO TO ℓ
	Conditional:	IF e_1 r e_2 THEN ℓ
	Multiway:	ON e GO TO list of labels
	Subroutine Call:	GOSUB ℓ
	Subroutine Exit:	RETURN
LOOP:		FOR v = e_1 TO e_2 STEP e_3
		NEXT v
INPUT/OUTPUT:		READ list of variables
		DATA list of values
		INPUT list of variables
		PRINT list
		RESTORE or RESET
ARRAY DECLARATION:		DIM
COMMENTS:		REM or REMARK
END and STOP:		STOP
		END
FUNCTION DEFINITION:		DEF
		FNEND

where v stands for variable
e stands for expression
ℓ stands for label
r stands for relation, e.g. =, 〈, 〉, etc.

Table 1 gives a list of the main BASIC statements. From this, the assignment statement can be seen to have the form:

LET variable = expression
for example, LET A1 = B + SQR(C—D) * E**3
or LET A$ = B$ & "AND" & C$

Branch statements are straightforward; the unconditional branch statement has the form:

GO TO label
for example, GO TO 101

The conditional branch statement has the form:

IF expression-1 relation expression-2 THEN label

for example, IF A < B + C THEN 50

which will transfer control to the statement labelled 50 if the condition is true (i.e. A is less than B+C), otherwise it will pass on to the next statement. And so on.

Table 1 is not a complete list of statements but it does contain the most important ones (see Kemeny and Kurtz[2], Waite and Mather[3]).

Direct and indirect mode

One interesting aspect of BASIC is the fact that statements can be used in either direct or indirect mode. If the statement is given in indirect mode, it must be preceded by a statement number (or label). It is then compiled and stored in a form suitable for the LINKER to link together prior to execution. If the statement is given in direct mode, it is not preceded by a statement number; in this case the compiler will compile the statement and execute it immediately. See example in Figure 2.

Obviously only certain types of statement can be used in direct mode. For example,

```
LET A = 14*P/180
PRINT A, B, C,
GO TO 50
```

are acceptable but a statement such as

```
FOR I = 1 TO 5 STEP 2
```

is meaningless when used directly and will be flagged as an error.

```
NEW DEMO EXAMPLE

  1 FOR I = 3 TO 0 STEP -1
  2 READ A$, A, B, C
  3 IF A = 999 THEN 80
  4 GOSUB 50
  5 NEXT I
 50   LET X = (-B+SQR(B*B-4*A*C))/(2*A)
 51   PRINT A$; "="; X
 52   RETURN                                  } - PROGRAM - INDIRECT STATEMENTS
 80   STOP
 70 DATA "FIRST ANS", 2, 5, 1
 80 DATA "SECOND ANS", 3, 3
 90 DATA "THIRD ANS",3,2,1
100 DATA 999
200 END

RUN                                          - EXECUTE PROGRAM

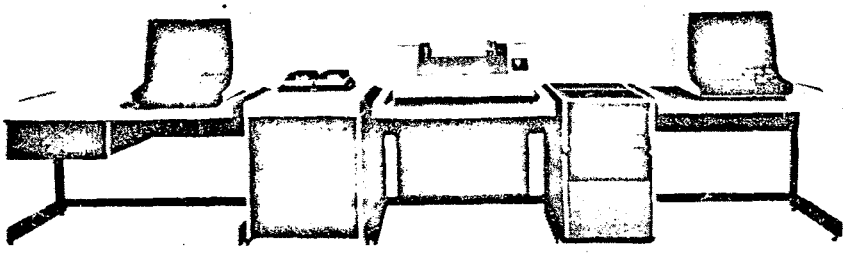
DEMO EXAMPLE    04/12/73
FIRST ANS=-0.219223
E  80 - NUMERIC ITEM <> NUMERIC DATA.
HALTED AT 2
        PRINT A                              - DIRECT STATEMENT, EXECUTE IMMEDIATELY
 5
        PRINT B
 3
        PRINT C
 1                                           - SHOWS THAT C IS INCORRECT
        LET C = 2                            - DIRECT STATEMENT, RESETTING C
        GO TO 50                             - DIRECT STATEMENT, RE-ENTERING PROGRAM
E  93 - SQR OF NEGATIVE NUMBER.
HALTED AT 50
        PRINT B*B-4*A*C                      - DIRECT STATEMENT
-31                                          - SHOWS VALUES ARE INCORRECT
        LET A = 3                            - ALTER A AND B
        LET B = 5
        PRINT B*B-4*A*C                      - CHECK ROOTS ARE REAL
 1
        GO TO 50                             - DIRECT STATEMENT, RE-ENTERING PROGRAM
SECOND ANS=-0.666667
E  48 - RETURNS EXCEED GOSUBS.               } - ERROR CAUSED BY INCORRECT GO TO 50 EARLIER
HALTED AT 52
        GO TO 5                              - RE-ENTER PROGRAM AND CONTINUE
E  93 - SQR OF NEGATIVE NUMBER.              } - ERROR CAUSED BY INCORRECT DATA
HALTED AT 50
        RETURN                               - DIRECT STATEMENT CAUSING RETURN FROM SUBROUT
E  81 - STRING ITEM <> STRING DATA.
HALTED AT 2
        GOODBYE                              - GIVES UP IN DISGUST
```

Figure 2. *An example illustrating the difference between direct and indirect mode.*

HAMAC

HERMES ACCOUNTING MACHINES AND COMPUTERS

- 40 Years in South Africa
- 20 000 Hermes users
- The total office equipment company from typewriters to real time computer systems



The Multibus real time system. Latest addition to the HAMAC office equipment range.

Here's a system to solve your stock problems — it's custom programmed for your requirements.

- Stock balances up to date **ALL THE TIME**
- Invoices prepared in less than **HALF THE TIME**
- Management information and stats on tap **AT ANY TIME**

67

That's **REAL TIME** computing with **HAMAC**. Switch the system on and you're ready to operate. That's right — it's so easy, anybody can!

TALK TO HAMAC *BRANCHES IN ALL MAJOR CENTRES*

Benoni	Bloemfontein	Cape Town	Durban	Johannesburg	Pietermaritzburg	Port Elizabeth	Pretoria
540721	80261	430533	65787	7243511	22082	43287	25382

We make our systems work

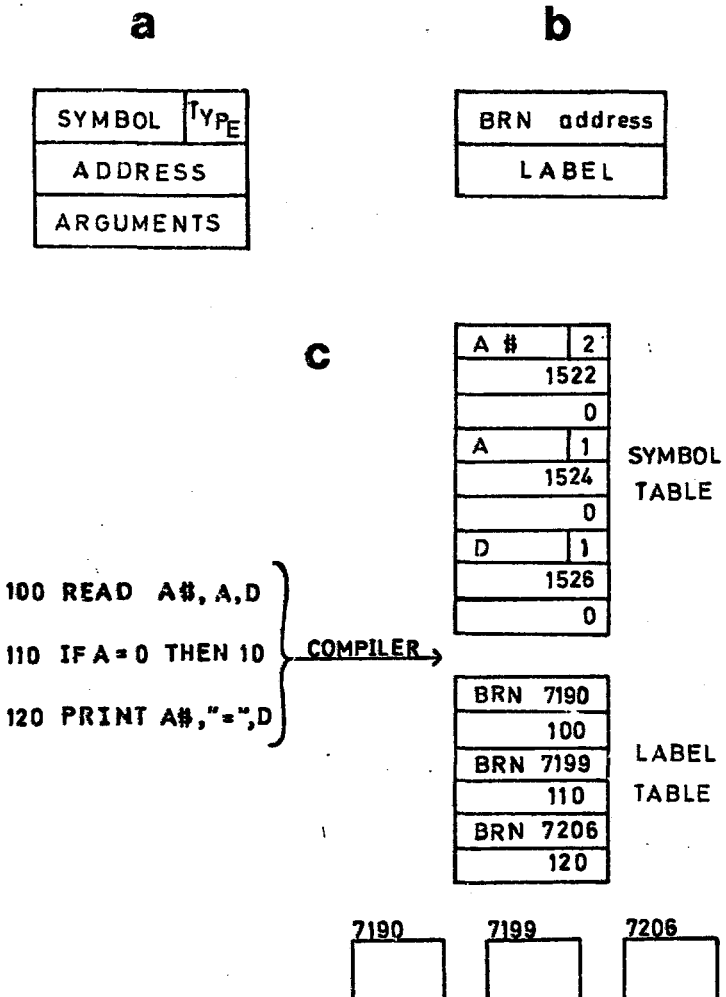


Figure 3. (a) *The format of a symbol table entry. TYPE indicates whether the symbol refers to a numeric variable, string variable, numeric array, string array or a function. ARGUMENTS contains the number of arguments in the case of a function or the dimensions in the case of an array. ADDRESS is simply the address where the variable is stored, or the address of the start of the array or function.* (b) *The format of a label table entry.* (c) *An example to demonstrate the action of the compiler in setting up entries in the symbol table and label table and generating blocks of object code.*

Implementation of the compiler

The CONTROL routine will read a string of input and if it is a statement, will hand control to the COMPILER. The latter

(a) checks that the statement is valid,
(b) generates the equivalent machine code,
(c) updates symbol and literal tables (if necessary), and
(d) updates the label table (if necessary).

Fortunately, the way in which BASIC has been designed makes it very simple for the COMPILER to recognise the type of a statement. This is uniquely determined by the first few letters of the statement, e.g. LET, PRInt, READ, RETurn, etc. One has none of the problems encountered in Fortran in distinguishing between assignment statemen and other types of statements, for example the classic

DO 1 I = 1,5 — Do statement
DO 1 I = 1 — Arithmetic statement assigning the value 1 to the variable DO1I.

Having identified the type of a statement, control is tran ferred to the appropriate subroutine to check that the re of the statement is valid and to generate code. An operat precedence method[4] is used to recognize the handle of expression at each stage of a parse and from this the co generation routine can generate code.

If any new variables are encountered when code is generate entries are automatically added to the symbol tabl declaring the new variables. Each entry in the symbol tab occupies three words (see Fig. 3(a)), and information stor in each entry includes the symbol itself, its type, addre where stored, and the number of arguments in the case of function, or dimensions in the case of an array.

Once a program statement has been identified, checked ar code generated, the label attached to the program stateme (if it is being used indirectly) is entered into the label tabl The format of entries in the label table is shown in Fig. 3(b Entries occupy two words. The first word contains a bran instruction, the address part of which refers to the addre of the block of code for the statement; the second wo contains the label that was attached to the statement.

If the mode is indirect, the COMPILER will return contr to the CONTROL routine, which reads in the next ca and the process is repeated. However, if the mode is direc then the COMPILER will enter the block of code which has just generated, execute it and then delete it.

The linker

One of the problems peculiar to terminal languages is t fact that statements can be read in any ~~other~~ order (s Fig. 4).

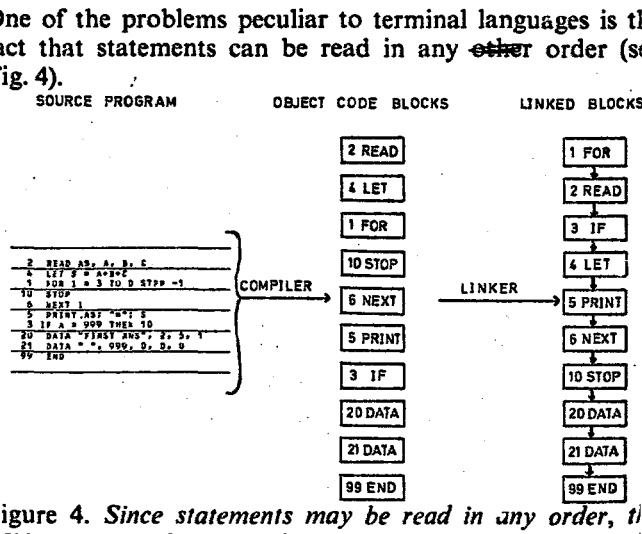


Figure 4. *Since statements may be read in any order, t LINKER must link the object code blocks together in t correct order before the program can be executed.*

Thus when the COMPILER encounters the statemer END, it transfers control to the LINKER. This has t task of linking these various blocks of code together order of increasing statement number. ~~At the same tin order of increasing statement numbe~~r. Futhermore it mu link the DATA statement blocks into a separate list.

TABLE 2.
Typical System Commands in Basic

COMMAND	FUNCTION
NEW name	introduces a new program
OLD name	recalls an existing program from disk
SAVE	stores copy of source program on disk
REPLACE	replaces existing copy of source program on disk by updated version
LIST name	lists the source program specified
SCRATCH	deletes program from disk file
RENAME	alters name of program on disk
RUN	executes program currently in core.

The system commands

The main system commands are summarized in Table 2. Since BASIC is intended as a terminal language, it must cater for storing and recalling lines of source program. Thus at the head of a BASIC program which is to be fed into the system for the first time, one must have a NEW command (see Fig. 2). This specifies the name of the subfile which is to be used to store this program. This is followed by the lines of the source program. After this one can have either a RUN command which causes the program to be executed, or a SAVE command which causes the source version of the program to be stored in this subfile on disk.

The command OLD will cause the compiler to look for the source version of the program specified amongst those stored on disk, and to compile it into main memory. If this program is edited and one wants to retain the edited version of the program on disk, one uses the command REPLACE to replace the original version with the updated one. And so on.

Implementation of system commands

Normally the source representations of programs stored on disk are kept in separate subfiles of a file called BASIC SOURCE. The NEW command specifies the name of a new subfile. After this, as the compiler compiles each indirect program statement, a block of object code is set up in core store, while a copy of the source statement is written away to a scratch file on disk. After the END statement has been encountered, the command RUN can be used to execute the object code. The command SAVE will cause the source statements to be read back from the scratch file and to be written to the specified subfile of BASIC SOURCE in order of increasing statement number. The command OLD will cause the source program from the specified subfile to be read back into core store and compiled.

Debugging aids in Basic

Two very useful commands, implemented originally by

```
      NEW DIFFICULT EX

       10 READ A, B, C
       20 GOSUB 90
       30 READ B
       40 IF B = 99 THEN 70
       50 GOSUB 90
       60 IF C <= A THEN 30
       70 PRINT "A =";A, "C ="; C
       80 STOP
       90 LET B = B+C
      100 LET A = A*SQR(B-A/2)+C*SQR(B-C/2)
      110 LET C = SQR(A+C)-SQR(A-C)
      120 RETURN
      130 DATA 10, -3, 8, 2, 8, 12, 99
      140 END

      RUN

DIFFICULT EX    07/11/73

E  93 - SQR OF NEGATIVE NUMBER.
HALTED AT 100

          PRINT A, B, C
 19.3137      8.91515      0.915154

          TRACE
          RESTORE
          RUN

DIFFICULT EX    07/11/73

* 10
* 20
* 90
* 100
* 110
* 120
* 30
* 40
* 50
* 90
* 100
* 110
* 120
* 60
* 30
* 40
* 50
* 90
* 100

E  93 - SQR OF NEGATIVE NUMBER.

HALTED AT 100

          EDIT
          130 DATA 10, -3, 8, 2, 9, 12, 99
          140 END

          UNTRACE

          RESTORE

          RUN

DIFFICULT EX    07/11/73

A = 31.6866  C = .457762E-1
```

Figure 5. *An example illustrating the use of the comman TRACE. On execution of the program, an error is encoun tered. The direct command PRINT does not give much hel in locating the error, so TRACE is used. This shows that th error has occurred after reading the second 8 in the DAT statement. After correcting the DATA statement and un setting the TRACE bits, the program runs successfully.*

Bull[5], are TRACE and BREAK. If one has an execution error in one's program, one can use the command

TRACE	which traces all statements,
or — TRACE n-m	which traces all statements with statement numbers in the range n to m.

This will cause the system to print the statement number of each statement (or only those with numbers in the range n to m) as it is encountered during execution (see Fig. 5).

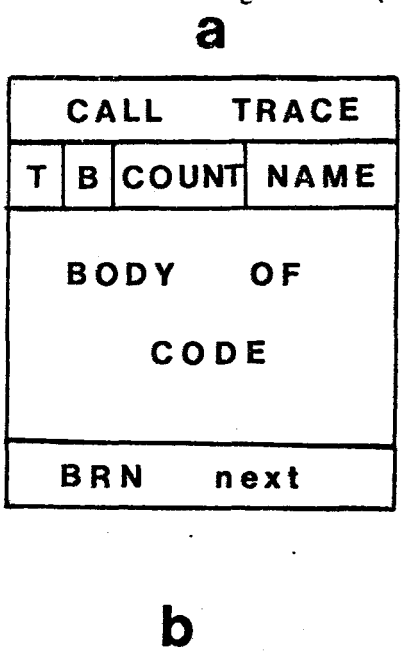


Figure 6. (a) *The format of blocks of object code.*
(b) *A typical object code block.*

Similarly one may use the command BREAK (or BREAK n-m), which causes the program to stop before execution of each program statement. The technique used to implement these commands depends on the format of the blocks of object code. This format is shown in Fig. 6.
Thus, when any program statement is executed, the first thing which happens is that the TRACE routine is called. This checks whether the TRACE or BREAK bits (T and B in Fig. 6) for this object code block are set. If neither is set, control is returned to the object code block; if either is set, it takes the appropriate action, either printing out the statement number, n, or printing the message

BREAK AT n

and halting, or possibly both. The TRACE routine obtains the statement number, n, from the field NAME.
When the TRACE command is encountered by the compiling system, the system simply sets the TRACE bits in the appropriate code blocks. The command UNTRACE unsets these bits. The commands BREAK and UNBREAK are treated in exactly the same way, setting and unsetting the BREAK bits in the appropriate blocks of code.
Another debugging feature which we have implemented, is the command DUMP. This can be used as a direct or an indirect command. When it is encountered, the system prints the contents of each of the single variables used in the program (see Fig. 7).

```
NEW DUMP EXAMPLE

  1 DATA 1, 5, 7, 12, 999
100 READ X
110 IF X < 0 THEN 300
120 IF X = 999 THEN 400
200 LET S = S+X
210 LET Q = Q+X*X
220 LET N = N+1
230 GO TO 100
300 LET M = M+1
310 GO TO 100
400 LET E = S/N
500 GO TO 100
999 END

RUN

DUMP EXAMPLE     02/11/73

E  82 - OUT OF DATA.

HALTED AT 100

        DUMP

E     =    6.25
M     =    0
N     =    4
Q     =    219
S     =    25
X     =    999
PI    =    3.14159
```

Figure 7. *An example to illustrate the command DUMP.*

In order to implement this facility, a copy of the symbol table must be retained at execution time. This is also

TABLE 3.

COMPARISON OF COMPILATION AND EXECUTION TIMES FOR SOME SMALL PROGRAMS WRITTEN IN FORTRAN AND BASIC

Program	No. of Statements		Compile Time from Cards		Execution Time		Edit and Compile Time from Disk	
	Fortran	Basic	Fortran	Basic	Fortran	Basic	Fortran	Basic
1. Simple Regression	43	39	120 secs.	16 secs.	8 secs.	7 secs.	150 secs.	8 secs.
2. Sort Program	26	24	110	10	7	6	140	5
3. Graph	54	62	130	21	35	45	160	10
4. Differential Equation	24	21	110	10	6	5	140	5
5. Turing Machine Emulation	59	75	120	22	24	17	150	11

required for use by direct statements.

A further feature which we have been able to include as a result of this object code block format, is the effect of a real-time clock. Since our computer is a small single-programming machine, there is no built-in real-time clock and therefore no check against a program looping. However, since each statement calls the trace routine, a counter is incremented on each entry to the trace routine, and when this counter exceeds a predetermined limit, the program run is halted with an error message. The counter is reset to zero for each run. The limit is set by default to some suitable value but may be altered by a non-standard command if required.

Non-standard commands

To cater for various options which we wanted to introduce to suit our own requirements, a verb SET was introduced which may be followed by a variety of arguments.

As mentioned earlier, one may select either the console typewriter or the card reader as an input device, and similarly either the the console typewriter or the line printer as an output device. In order to change from one input medium to another, the command

SET INPUT (n) where n = CR or TY

was introduced. Similarly to change output medium the command

SET OUTPUT (n) where n = LP or TY

is used. Other options are:

SET FILE (name) (gen. no.) — to use any file other than BASIC SOURCE for storing source programs,

SET LIMIT (n) — to alter the number of instructions the simulated real-time clock will execute before interupting execution, etc.

Efficiency of the System

Some indication of the efficiency of the system has been obtained by timing several programs written in BASIC and comparing the compilation times against those for the equivalent Fortran programs. These times are shown in Table 3.

Since BASIC programs are normally run in batch mode the compilation time given in Table 3 is the time required by the system to compile and link a program. There is an additional overhead of approximately 20 seconds required to compile the job description and call the BASIC compiler. However, if one batches several programs together, this overhead will be shared by the different programs and will not increase the compilation time significantly. The "edit-compile time from disk" is the time required to compile a program stored on disk while at the same time editing one or two of the source statements.

When Fortran programs are compiled from cards, each job is handled separately. Thus the compilation times here include the times required to compile the job description, to call the Fortran compiler, to compile the program, to call the consolidator, to consolidate the program and finally to load it into core store ready for execution.

If one stores a Fortran program in source form on disk "edit and compile time from disk" represents the time required to edit one statement and compile the program if the program is not batched. Batching will obviously reduce this figure but will not reduce it much below the figure for "compile time from cards."

Thus the BASIC compiling system can handle approximately five to ten times as many small programs as can the Fortran compiler in the same amount of time. A class of 100 students (each submitting one program to be run) will therefore require about 30 minutes of computer time instead of three hours. And because of the simpler language and better error diagnostics, students should require fewer compilations on average to get a program working, thus improving the overall efficiency considerably.

REFERENCES:

1. van den Heever, R. J. *Vertalerteorie in die Praktyk*, *Systems*. Vol. 3, No. 1 (Jan. 1973), pp. 37-40.
2. Kemeny, J. G., and Kurtz, T. E. *BASIC Programming*. John Wiley, 1971.
3. Waite, S. V. F., and Mather, D. G. *BASIC*. University Press of New England, 1971.
4. Hopgood, F. R. A. *Compiling Techniques*. Elsevier, 1969.
5. Bull, G. M. *Dynamic Debugging in BASIC*. Comp. Journal, Vol. 15, No. 1 (Jan. 1972), pp. 21-24.

25. M.H.WILLIAMS and A.R.BULMER, A transportable code generator system, Information Processing Letters, vol. 9, no. 3, 122-125 (1979).

A TRANSPORTABLE CODE GENERATOR GENERATOR SYSTEM

M.H. WILLIAMS and A.R. BULMER
Department of Computer Science and Applied Mathematics, Rhodes University, Grahamstown, South Africa

Received March 1979

Code generator, transportability, compiler generator

1. Introduction

A code generator generator system is a system which enables one to generate a code generator for any particular language and machine. This paper discusses the structure of one such system with particular reference to its transportability.

Although there is no general agreement concerning the structure of the interface between a parser and a code generator, one of two approaches is usually followed:

(a) calls to the code generator routines are built into the parser so that at the appropriate point in the parsing process, the correct routine is called and code is generated, or

(b) the parser produces a sequence of pseudo-codes which are stored either in main memory or in a file on a backing store device. On completion of its task, control is passed to the code generator which scans the pseudo-codes and generates the final object code.

Consider a simplified view of a code generator of the latter type. A diagrammatic representation of it is given in Fig. 1. Basically the code generator accepts as input a stream of pseudo-codes; it has built into it a set of code skeletons which it assembles on the basis of the pseudo-codes to produce an object program.

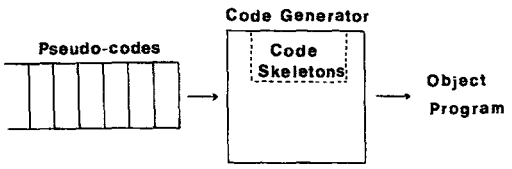


Fig. 1.

In constructing a code generator generator system which will generate code generators for different languages, one of the first decisions which needs to be taken, concerns the structure of a code generator produced by such a system. In the system to be described it was decided to use a table-driven approach; in other words, each code generator produced is a table driven program.

A code generator constructed by the system is made up of three major components, viz.

(a) *The Pseudo-code Action Table*: This contains information on what action is to be performed on encountering each pseudo-code. It includes code skeleton information and information on how to assemble the code skeletons. This information is stored in the table in a suitably encoded form.

(b) *The interpreter*: This is a program which accepts as input the pseudo-codes from the parser and performs the corresponding actions either directly or by reference to the Pseudo-code Action Table.

(c) *Local routines*: These are machine dependent and language dependent routines used by the interpreter.

The actions coded into the Pseudo-code Action Table include fetching addresses, assembling instructions, storing code in buffers, etc. The local routines might include routines to read buffers from backing store, write buffers to backing store, allocate general purpose registers, etc.

In order to generate a code generator for a particular language, the user must

(a) create the specific Pseudo-code Action Table relevant to the language and machine;

(b) write whatever local routines which may be necessary; and

(c) create an interpreter in the machine code of the particular machine and incorporate the Pseudo-code Action Table and local routines in it.

The first part of this process, the creation of a Pseudo-code Action table, for a specified mapping, is performed by a program written in a high-level language, which can be transported as required. The local routines (of which in general there will be very few) will need to be written afresh for each new machine (and possibly for different languages). However, it is the creation of the interpreter which is the concern of this current paper.

2. The interpreter

The interpreter could have been implemented by writing it in a suitable systems programming language. However, this approach was decided against for several reasons; in particular

(a) the systems programming language would have to be chosen very carefully so that it is of a sufficiently high level (e.g. BCPL [2]) and sufficiently far removed from machine dependent characteristics to make it transportable, and yet sufficiently low-level to provide the facilities of a code generator with reasonable efficiency, and

(b) a compiler for the language must exist on the target machine.

Another alternative would be to write the interpreter in a high-level language, but a similar argument applies. Instead it was decided to use the system to reproduce itself. In order to accomplish this a high level notation geared specifically to the writing of an interpreter, was designed. Then a set of intermediate pseudo-codes (X) for this language was postulated. In other words at this stage the front end of a compiler for this notation was completely defined and could have been implemented if necessary.

The interpreter was then written in this high level notation. In the absence of a compiler for this language, the interpreter was translated by hand into

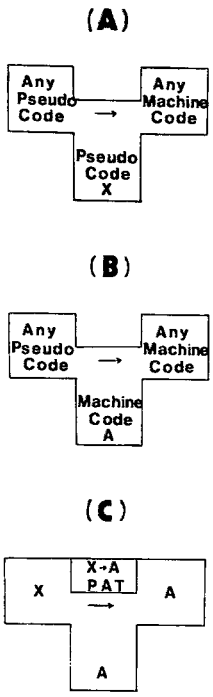


Fig. 2.

pseudo-code form. At this point we had a version of the interpreter capable of generating code for any particular machine and language (given the appropriate Pseudo-code Action Table and local routines), but expressed in a theoretical pseudo-code format. This is represented diagrammatically as an incomplete T-diagram [1] as shown in Fig. 2(A). The 'bite' out of the T-diagram represents the hole into which the Pseudo-code Action Table and local routines must be inserted.

Finally the interpreter was bootstrapped onto the system as follows:

(a) a Pseudo-code Action Table for converting the pseudo-codes X into machine code A was generated;

(b) the essential functions of the interpreter were translated by hand to produce a version in a high-level language;

(c) using the crude interpreter from (b) with the Pseudo-code Action Table from (a), the pseudo-code version of the interpreter (as shown in Fig. 2(A)) was converted to machine code A (illustrated in Fig. 2(B)).

In order to use this interpreter a Pseudo-code

Action Table and local routines must be inserted into it. Suppose that a Pseudo-code Action Table for the conversion of pseudo-code X to machine code A is generated and inserted into this gap. The result is shown in Fig. 2(C). The resulting code generator is capable of accepting any program in pseudo-code X and converting it to machine code A.

To generate a code generator for some other language with pseudo-codes Y, which converts these codes to machine code A, one simply adjusts the initial parameters of the pseudo-code version of the interpreter so as to select those facilities required in the Y → A conversion and set up space for the Pseudo-code Action Table and local routines, feeds it through the code generator of Fig. 2(C) and then inserts the Pseudo-code Action Table for the Y → A conversion, and the local routines. The result is a code generator in machine code A which translates pseudo-codes Y to machine code A.

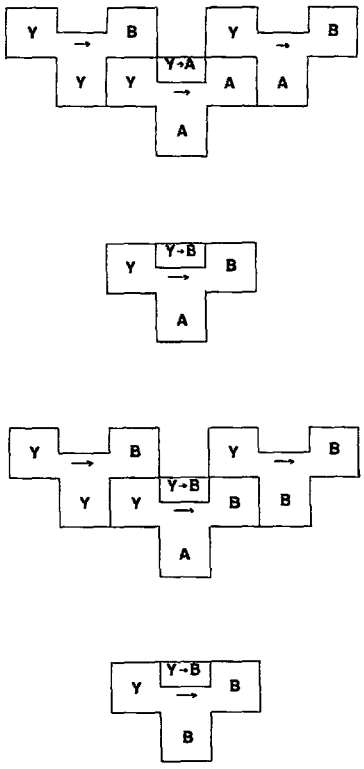


Fig. 3.

In order to transport the code generator system to some other machine with machine code B, once again one adjusts the initial parameters of the pseudo-code version of the interpreter to take account of the properties of machine code B and one creates a Pseudo-code Action Table for the X → B conversion. Then one proceeds as follows:

(a) convert the pseudo-code version of the X → B interpreter to machine code A;

(b) insert the X → B Pseudo-code Action Table and local routines for A;

(c) using the code generator produced by step (b), convert the pseudo-code version of the X → B code generator to machine code B;

(d) insert the X → B Pseudo-code Action Table and local routines for B.

This process is illustrated in Fig. 3. At this stage one has a complete code generator for machine B.

3. Practicality of the system

Setting up a code generator for some pseudo-code language L on some machine M requires a clear understanding of the semantics of the language L, and the amount of time taken to create a working code generator is largely dependent on the user's understanding of the language L and the machine code of M.

On the other hand transporting the code generator system to some other machine is a straightforward process which should not take more than a few days for a programmer faimilar with the machine code of the target machine.

The chief advantage of the system is that once the mapping of the pseudo-code language to machine code has been clearly understood and specified, it is a simple matter to obtain a code generator for that particular mapping. The specification of the mapping is sufficient documentation for the systems programmer or the user to understand what the code generator is doing. If there are errors in this specification it is a simple matter to regenerate a code generator for the corrected specification. Thus it is hoped that such a system should be of assistance in locating and removing errors from compilers, and hence in producing more reliable system software.

Acknowledgment

One of the authors, Mr. A.R. Bulmer, was in receipt of a CSIR bursary at the time that this work was carried out.

References

[1] J. Earley and H. Sturgis, A formalism for translator interactions, Comm. ACM 13 (1970) 607–617.
[2] M. Richards, BCPL – A tool for compiler writing and systems programming, AFIPS SJCC (1969).

26. J.C.NEVES, S.O.ANDERSON and M.H.WILLIAMS, A Prolog implementation of Query-by-Example, Proceedings of the Seventh International Computing Symposium, ed. H.J.Schneider, B.G.Teubner, Stuttgart, 318-332 (1983).

A PROLOG IMPLEMENTATION OF QUERY-BY-EXAMPLE

J.C. Neves, S.O. Anderson and M.H. Williams, Edinburgh

Summary: Prolog is a programming language which is particularly well suited to data base applications. However, since formulating queries for Prolog data bases requires a knowledge of Prolog syntax, various workers have been involved in developing better interfaces to Prolog data bases. Query-by-Example (QBE) is a data base management language which provides a simple interface to relational data bases and which has a syntax which is very similar to Prolog goals. This paper examines the correspondence between QBE and Prolog from the viewpoint of a particular implementation, and concludes that QBE is a natural choice as an interface to Prolog data bases.

1. Introduction

Kowalski[5] has shown that the language of Horn clauses, a restricted form of predicate calculus, is suitable for the representation of relational data bases (Codd[2]), and indeed extends the power of relational schemes by permitting the expression of general rules in the data base. Prolog[9,11,14] is a qualified implementation of Horn clauses which has become important as a vehicle for Artificial Intelligence applications such as natural language processing[3,8], data base systems[4,12,13], expert systems[1], and so on. The importance of Prolog has been given a considerable boost by its choice by the Japanese for their fifth generation computer systems[7].

In several instances where Prolog has been used as the vehicle to implement a data base system (e.g. Kowalski), queries to the data base must be formulated as Horn clauses. Since many potential data base users may have difficulty with Horn clause syntax, it is desirable to have a simple interface to such systems.

One data base management language which suggests itself for this purpose is Query-by-Example[15] since this is a simple interface designed for relational data bases and since

it has already been noted by van Emden[10] that a close correspondence exists between the syntax of Prolog goals and Query-by-Example.

An investigation has been conducted into the translation of QBE to Prolog goals, and an implementation of QBE in Prolog realised. On the whole the translation process is straightforward, although the treatment of integrity constraints proved interesting.

The next two sections describe briefly the main features of QBE and Prolog. This is followed by a discussion of how aspects of QBE might be implemented in Prolog.

2. Query-by-Example

Query-by-Example (QBE) is a tabular data base management language which was developed by Zloof[15] for relational data bases. It is designed for interactive use and makes use of a special-purpose screen-editor to enable users to compose queries.

Initially the user is presented with the skeleton of a table as follows:

(1)	(2)
(3)	(4)

The four areas delimited by the skeleton are:

(1) Table name field,
(2) Column name field,
(3) Tuple command field,
(4) Tuple entry field.

The screen editor may be used to position the cursor in any of these four areas to insert a command and/or a variable or constant element. The four basic commands which are available are "P.", "I.", "D." and "U." representing respectively PRINT, INSERT, DELETE and UPDATE.

A fundamental concept in Query-by-Example (from which it

derives its name) is the notion of example elements. To illustrate this idea consider a simple example. Suppose that one has the data base given in the Appendix. This contains three relations - one containing the details of each type of part, another containing the details of each supplier and the third relating the suppliers to the parts which they supply. Suppose further that one wants to list the names of all suppliers who have status equal to 20. One starts by entering "suppliers" in the table name field of the skeleton. Since the relation already exists in the data base one can ask the system to generate the column headings (attributes of suppliers), viz.

suppliers	sno	sname	status	city

One may now enter "P.X" in the supplier name field (sname) and 20 in the status field as follows:

suppliers	sno	sname	status	city
		P.X	20	

Here "20" is a constant element representing a specific value in the status field, while "X" is an example element - a character sequence which begins with a lower case letter is taken to be a constant while one beginning with an upper case letter or an underline symbol "_" is taken to be an example element. Preceding the example element with the command "P." instructs the system to print the value of X. Thus this command is interpreted as "find every supplier entry with status equal to 20 and print the value of the supplier name field".

The use of constant and example elements provides a flexible and powerful means of interrogation. For example, one can request the join of two relations simply by using the same example element in two tuples entries in the skeletons. The example elements may also involve arithmetic expressions signifying some arithmetic constraint on the query to be

satisfied.

The operator "P." may either precede the example elements to be printed or may be written in the tuple command field if it applies to the whole tuple. Thus to print the number, name, status and city of any supplier whose status exceeds 10, one may either have:

suppliers	sno	sname	status	city
	P.N	P.X	P.S > 10	P.C

or

suppliers	sno	sname	status	city
P.	N	X	S > 10	C

Further examples will be discussed later.

3. Prolog

A Prolog program consists of a set of Horn clauses each of which express an implication. A typical Prolog program describing a particular family tree might consist of the following clauses:

```
descendant(X, Y) :- child(X, Y).
descendant(X, Y) :- child(X, Z), descendant(Z, Y).
child(james, john).
child(mary, john).
child(john, alfred).
child(john, susan).
child(susan, mark).
```

In this example each clause is either of the form:

p :- q1, q2, ... qn.

in which case it is interpreted as "p is true if all of q1, q2 ... qn are true", or the form:

p.

in which case p is unconditionally true and is referred to as a fact. In the example given the last five clauses are facts, the first of which should be read as "james is a child of john". The first two clauses define the relation descendant and should be read as "X is a descendant of Y if X is a child

of Y" and "X is a descendant of Y if X is a child of some Z and Z is a descendant of Y". Coincidentally, Prolog uses the same convention as QBE to distinguish between constants and variables: "john" is a constant because it begins with a lower case letter but "X" is a variable (the equivalent of an example element) because it begins with a capital.

Once a program such as the one above has been created, one can query it, for instance by typing:

```
? - descendant(john, mark).
```

in which case the system would respond with the answer "yes", or

```
? - descendant(X, mark).
```

to which the system response would be to list one by one the descendants of "mark".

The mechanism which Prolog uses to evaluate queries is important to the understanding of Prolog programs; the last example query will serve as an illustration of this mechanism. Initially the system attempts to satisfy the goal descendant(X, mark). Using a pattern matching process called unification this is recognized to be an instance of descendant(X, Y), the head of the first clause in the program, with Y instantiated to "mark". The goal descendant(X, mark) is therefore replaced by the goal child(X, mark). Unification is again used to recognize that child(X, mark) with X instantiated to "susan" is the fact child(susan, mark). Thus the system responds with:

```
X = susan
```

After being prompted to continue, the system backtracks to the beginning and replaces the goal descendant(X, mark) by the two goals child(X, Z) and descendant(Z, mark). An attempt is now made to satisfy these two goals, but in the strict order child(X, Z) followed by descendant(Z, mark). Since the former is satisfied in five ways, Prolog proceeds to attempt to satisfy descendant(Z, mark) in five ways - with Z instantiated to "john", "john", "susan", "alfred" and "mark". Satisfying

descendant(john, mark), say, promulgates attempts to satisfy child(john, mark) and child(john, Z) followed by descendant (Z, mark). Only when all subgoals have terminated with success or failure does the search process stop.

From this brief discussion the close correspondence between the basic query mechanisms of QBE and Prolog should be evident.

4. The Screen Editor

The implementation of a screen editor which enables the user to access the elements of the skeleton and perform insert/delete operations on characters and words, is a straightforward task if a conventional programming language is used; Prolog is unconventional but it does provide facilities for programming in a conventional way.

In the previous section the logical interpretation of a clause,

```
p :- q1, q2, ... qn.
```

was given. An alternative interpretation which holds good for the screen editor, is that it is a procedure p with body q1, q2, ... qn. Thus, for example, the first clause in our implementation of the screen editor (effectively, the "main program") reads:

```
qbe :-
      !,
      initialization,
      type_out(welcome),
      repeat,
            input_command(Option),
            respond_to_command(Option),
            clear_screen,
            display('GOOD_BYE').
```

This clause defines a QBE session to consist in turn of an initialization process, the typing out of a welcome to the user, a repeated process of input and response to a command, and finally clearing the screen and displaying the message "GOOD_BYE". Two features which require explanation are (i) the so-called cut symbol "!" and (ii) "repeat".

The cut symbol (!) is a crude device used in Prolog to prevent backtracking. It is vital to the input/output routines to prevent the interpreter from beginning all over again when, say, one character in the user's input is invalid. The use in this example is strictly unnecessary and is included only for the sake of efficiency.

The built-in predicate repeat always succeeds when an attempt is made to satisfy or resatisfy it. The effect here is that if either of the predicates input_command or respond_to_command fails, an attempt will again be made to satisfy them; should they both succeed the repetition terminates and the program proceeds to clear the screen and output "GOOD_BYE".

5. Implementing Simple Queries

The user's data base and the QBE system together form one large Prolog program and, so far as Prolog is concerned, they are indistinguishable. The technique used here to distinguish the system clauses from the user's data base is to have input routines which put "zz" on the front of incoming data.

A typical user's data base describing the suppliers to an organization might consist, among others, of the following clauses:

```
suppliers(1, smith, 20, vienna).
suppliers(2, jones, 15, london).
suppliers(3, blake, 30, paris).
suppliers(4, clark, 20, vienna).
suppliers(5, adams, 30, athens).
suppliers(_, _, _, _, _).
scheme(suppliers, 5, 4, [sno, sname, status, city]).
```

The first five of these clauses provide the number, name, status and city of each of the suppliers. The sixth clause is a so-called "catchall" clause. Note that it has five parameters, one more than the preceding five clauses. Its function is to enable constraints to be imposed and it will be explained in more detail later. The final clause provides information necessary to the system - the name of the table (relation), the number of entries in the table, the number of attributes of the relation and a list of the names of each of

the attributes of the relation.

Because of the direct correspondence between QBE and Prolog queries, conversion from one to another is easy. For example, the query:

suppliers	sno	sname	status	city
P.	A	B	C	D

is transformed directly into the clause:

```
?- suppliers(A, B, C, D),
   fail.
```

where fail is an artificial mechanism in Prolog which results in backtracking.

Conditions are handled with equal ease; for example:

suppliers	sno	sname	status	city
P.	A	B	C > 10	D

becomes:

```
?- suppliers(A, B, C, D),
   C > 10,
   fail.
```

Conditions specified within a single row are considered to be joined together by "AND" operations. It is also possible to "AND" together two conditions on the same column. As an example suppose that one wishes to determine the number of suppliers who supply both part numbers 1 and 4 (in relational algebra notation this corresponds to a select/project operation). The QBE query will be:

supplier_parts	sno	pno	qty
	P.A	1	
	A	4	

being converted into the Prolog clause:

```
?- supplier_parts(A, 1, _),
   supplier_parts(A, 4, _),
   fail.
```

To specify alternative conditions ("OR") it is necessary to specify them in separate rows. For example the skeleton entry:

suppliers	sno	sname	status	city
P.	X		10	London

is transformed into the Prolog clause:

```
?- (suppliers(A, B, 10, C);
    suppliers(X, Y, Z, london)),
    fail.
```

which requests the union of all suppliers tuples for suppliers whose status is 10 or who are based in london. Here the ";" operator specifies a disjunction of goals (suppliers(A, B, 10, C) or suppliers(X, Y, Z, london)).

Queries involving more than one relation are handled in the same way. For example, if one wishes to find the names and cities of all suppliers of the part numbered 1 who have status greater than 10 (in relational algebra notation this corresponds to a join/select/project operation), one might express this in QBE as:

suppliers	sno	sname	status	city
	N	P.X	Y > 10	P.Z

supplier_parts	sno	pno	qty
	N	1	

which would be translated into Prolog as:

```
?- suppliers(N, X, Y, Z),
   supplier_parts(N, 1, _),
   Y > 10,
   fail.
```

6. Insertions, Deletions, Updates and Constraints

In the absence of constraints, insertions and deletions of entries in the data base can be handled in a simple manner using the two built-in predicates assert and retract which respectively add clauses to and retract clauses from a program. Both of these predicates are utilized in the update operation. As an example of this, consider the following QBE command which instructs the system to update all entries with a status of 10 in the suppliers table to a status of 15.

suppliers	sno	sname	status	city
U.	X	Y	Z	10
	X	Y	Z	15

This can be converted into the Prolog clause:

```
?- retract(suppliers(X, Y, Z, 10)),
   asserta(suppliers(X, Y, Z, 15)),
   fail.
```

which will achieve the desired effect.

However, if one wants to include integrity constraints into the system, a slightly different approach must be adopted. The format for an integrity constraint in QBE as put forward by Zloof[15] consists of two components: the one is the constraint itself, the other an indication of when it is to be checked. There are three basic commands concerned with constraints:

```
I.CONSTR - insert a new constraint,
P.CONSTR - print constraints,
D.CONSTR - delete a constraint.
```

I.CONSTR and D.CONSTR must be followed by a condition list in parentheses specifying when the constraint must be applied. For example, to place a constraint on the status of suppliers such that no new supplier may have a status greater than 99, one may enter:

suppliers	sno	sname	status	city
I.CONSTR(I.,U.).I.			X < 100	

This will check that the value in the status field for each insertion and update command is less than 100.

In order to cater for this, insertions, deletions and updates must be handled in a slightly different fashion. Consider the data base of suppliers illustrated in the previous section. Now, whenever a command is issued to the system to alter the data base in any way, it is first expressed as a query in which one extra parameter signifying the operation to be executed is added to the attributes of the relation in question. Thus to insert a new tuple into the suppliers relation, one might enter the QBE command:

suppliers	sno	sname	status	city
I.	6	mactavish	40	edinburgh

This will be translated into the query:

```
?- suppliers(insert, 6, mactavish, 40, edinburgh),
   asserta(suppliers(6, mactavish, 40, edinburgh)).
```

that is, an assert condition is added at the end of the sequence of preconditions of the goal. If the data base is in the form given in the previous section the catchall clause suppliers(_, _, _, _, _) will always succeed and the system will then proceed to insert the clause by executing:

```
asserta(suppliers(6, mactavish, 40, edinburgh)).
```

If constraints are specified, these can be dealt with by including additional clauses before the catchall clause. For example, suppose that one wishes to impose the constraint on suppliers that no addition may be made to it for which the supplier number exceeds 50. The expression of this constraint in QBE is as follows:

suppliers	sno	sname	status	city
I.CONSTR(I.).I.	A <= 50			

The effect of this on the user's data base is to add an extra clause as shown below:

```
suppliers(1, smith, 20, vienna).
suppliers(2, jones, 15, london).
suppliers(3, blake, 30, paris).
suppliers(4, clark, 20, vienna).
suppliers(5, adams, 30, athens).

suppliers(Y, A, B, C, D) :-
      (Y == insert),
      !,
      A <= 50.
suppliers(_, _, _, _, _).
```

Note that this clause has five parameters like the catchall clause. Now if an attempt is made to insert a new tuple into the data base, e.g.

```
suppliers(6, mactavish, 40, edinburgh).
```

the query:

```
?- suppliers(insert, 6, mactavish, 40, edinburgh).
```

presented to the system will first match the constraint clause. The cut symbol (!) in that clause prevents any backtracking and so the query succeeds only if the value of the supplier number (6) is less than or equal to 50. If it does succeed, the system continues as outlined before. In this way the data base can be protected from changes which do not conform to the

specified constraints.

7. Conclusions

The similarity between the syntax of QBE and goals in Prolog is an indication of the suitability of QBE as a natural interface to Prolog data bases. An implementation of QBE in Prolog for use as an interface to Prolog data bases has been successfully realized on both a PDP11/34 and a DEC 10 system.

The chief problem with Prolog at the current time is its inefficiency, which in turn is due to the inflexibility of current Prolog implementations. Although some optimization can be achieved by tinkering with clause order[12], this solution is only appropriate in a limited range of situations. Work is actually being carried on to provide far more flexible evaluation order and this work is motivated by an attempt to improve computational efficiency rather than retrieval efficiency.

However, the choice of Prolog by the Japanese as the core language of their fifth generation computer project is likely to remove the problem of efficiency completely. Their goal of producing personal computers capable of 100,000 to 1 million logical inferences per second (and larger machines capable of up to 1,000 million logical inferences per second) is so ambitious that even if they fall short of it by a factor of ten, the machines they produce will be very powerful Prolog machines.

In view of this it is important that suitable interfaces to Prolog data bases such as QBE be found. The advantages of QBE over other languages such as SEQUEL and SQUARE have been clearly demonstrated by Thomas and Gould[17] who showed that subjects using QBE required about one-third the training time, were about twice as accurate in writing queries and were somewhat faster. Yet QBE supplies only a syntax for goals in Prolog. One of the distinctive features of logic data bases is the inclusion of general rules with specific data. To enable the user of QBE to express such rules the syntax of QBE

must be extended to provide a correspondence between QBE and any Prolog clause. Extensions of QBE along these lines will certainly prove fruitful.

The implemented system has demonstrated that considering QBE as a two dimensional syntax for Prolog and Prolog as a logical interpretation of QBE queries operates to the mutual advantage of each approach resulting in a powerful and flexible data base system.

Acknowledgement

The work of the first author, J.C. Neves, was supported by the Calouste Gulbenkian Foundation under Grant 14/82.

References

[1] Clark, K.L.; McCabe, F.G.: PROLOG: A language for implementing expert systems, in Machine Intelligence 10 (Hayes, J. and Michie, D., Eds) Ellis and Horwood, 1982.

[2] Codd, E.F.: A relational model for large shared data banks, Comm. ACM 13 (1970) 377-387.

[3] Colmerauer, A.: Metamorphosis Grammars, in Natural language communication with computers (Bloc L., Ed.), Springer Verlag, 1978.

[4] Futo, I.; Szeredi, P.; Darvas, F.: Some implemented and planned PROLOG applications, Workshop on logic and data bases, Toulouse, November, 1977.

[5] Kowalski, R.A.: Logic for Problem Solving, Memo. No. 75, Department of Computational Logic, University of Edinburgh, 1974.

[6] Kowalski, R.A.: Algorithm = Logic + Control, Research report, Department of Computing and Control, Imperial College, London, 1976.

[7] Moto-Oka, T. et al: Challenge for Knowledge Information Processing Systems, Fifth Generation Computer Systems, North-Holland, Amsterdam (1982) 3-89.

[8] Pereira, F.C.N.; Warren, D.H.D.: Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks, Artificial Intelligence 13 (1980) 231-278.

[9] Roussel, P.: PROLOG: Manuel de reference et d'utilisation, Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Universite d'Aix-Marseille II, 1975.

[10] van Emden, M.H.: Computation and deductive information retrieval, in Formal description of programming concepts (Neuhold, E., Ed.), North-Holland, Amsterdam, 1978.

[11] Warren, D.H.D.: Implementing Prolog - Compiling predicate logic programs, Research reports No. 39 and 40, Dept. of Artificial Intelligence, University of Edinburgh, 1977.

[12] Warren, D.H.D.: Efficient processing of interactive relational data base queries expressed in logic, Department of Artificial Intelligence, University of Edinburgh, 1981.

[13] Warren, D.H.D.; Pereira, F.C.N.: An efficient easily adaptable system for interpreting natural language queries, Research Report, Department of Artificial Intelligence, University of Edinburgh, 1981.

[14] Warren, D.H.D.; Pereira, L.M.; Pereira, F.C.N.: PROLOG - The language and its implementation compared with Lisp, Proc. SIGART/SIGPLAN conference on Programming Languages, Rochester, New York, 1977.

[15] Zloof, M.M.: Query-by-Example: A data base language, IBM Systems Journal 16 (1977) 324-343.

[16] Zloof, M.M.: Security and integrity within the Query-by-Example data base management language, IBM Research Center, 6982, Yorktown Hts, N.Y., 1978.

[17] Thomas, T.C.; Gould J.D.: A psychological study of query by example, Proc. National Computer Conference (1975) 439-445.

Appendix

Consider a simple business data base containing the following three relations.

parts	pno	pname	colour	weight
	1	nut	red	12
	2	bolt	green	17
	3	screw	blue	17
	4	screw	red	14
	5	cam	blue	12
	6	cog	red	19

Table 1.1 - The parts relation

suppliers	sno	sname	status	city
	1	smith	20	vienna
	2	jones	15	london
	3	blake	30	paris
	4	clark	20	vienna
	5	adams	10	athens

Table 1.2 - The suppliers relation

supplier_parts	sno	pno	qty
	1	1	300
	1	2	200
	1	3	400
	1	4	200
	1	5	100
	1	6	100
	2	1	300
	2	2	400
	3	2	200
	4	2	200
	4	4	300
	4	5	400

Table 1.3 - The supplier_parts relation

Authors' full address: Computer Science Department,
Heriot-Watt University, Edinburgh EH1 2HJ, Scotland.

27. M.H.WILLIAMS, J.C.NEVES and S.O.ANDERSON, Security and integrity in logic data bases using Query-by-Example, Proceedings of Logic Programming Workshop '83, ed. L.M.Pereira, Universidade Nova de Lisboa, Lisbon, 303-340 (1983).

SECURITY AND INTEGRITY IN LOGIC DATA BASES USING QUERY-BY-EXAMPLE

M H WILLIAMS, J C NEVES and S O ANDERSON

Department of Computer Science

Heriot-Watt University

Edinburgh

Scotland

KEYWORDS: Security, Integrity, Query-by-Example, Prolog, Logic data base.

Abstract

Security and integrity are two important and inter-related aspects of data base systems, and data base management languages must make provision for the specification and enforcement of such constraints. In the case of the data base language Query-by-Example a style for handling certain types of security and integrity constraints has been developed by Zloof.

An alternative approach to integrity in QBE is presented here which is based on the idea of consistency of the data in the data base. This approach allows for a more general type of constraint which includes the handling of functional, multivalued and embedded-multivalued dependencies, as well as the more conventional and simpler type of integrity constraints in a uniform manner.

Both security and integrity constraints have been implemented in Prolog as part of a logic data base.

1. INTRODUCTION

One of the important functions of any data base management system is to preserve the integrity of any data stored within the data base by ensuring that it is consistent with the prescribed properties of such data (integrity constraints). Integrity constraints can be classified into three types (Ullman [1], Nicolas and Yazdanian [2]) :

(a) Value-based constraints. These are conditions which the values of the domain elements must satisfy. They are usually restrictions on the range of values which a field can assume or are concerned with non-structural relationships amongst various fields. For example in the set of relations given in Appendix 1 one might wish to impose restrictions such as:

(i) The weight of a part is always less than 100 units (simple restriction on range).

(ii) An entry may only appear in the supplier_parts table if an entry for the supplier concerned exists in the supplier table (existence check).

(iii) Any supplier from Vienna or Athens must have a status which is at least 20 (non-structural relationship), etc.

(b) Structural or "Value-oblivious" constraints. These are restrictions concerned not with the value in any

particular field of a tuple but with whether certain fields of one tuple match those of another. Three specific types of structural constraints are addressed in this paper:

(i) Functional Dependencies. If X and Y are two sets of attributes from some relation scheme, then X functionally determines Y (or Y functionally depends on X), written "X -> Y", if any pair of tuples which agree in the components for all attributes in set X must likewise agree in all components corresponding to attributes in set Y.

Examples of functional dependencies in the set of relations in Appendix 1 include:

> sno -> sname (corresponding to each supplier number is a unique name),
>
> sno, pno -> qty (corresponding to each supplier/part number combination is associated an unique quantity),

and so on. It has been shown [3] that any set of functional dependencies can be transformed to an equivalent set in which all functional dependencies have the form "X -> Y" where Y is a singleton set.

(ii) Multivalued Dependencies. If X and Y are two sets of attributes from some relation scheme then X multidetermines Y (or there is a multivalued dependency of Y on X), written "X ->-> Y", if corresponding to a given set of values for the attributes of X there is a set of zero or more associated values for the attributes of Y, and this set of Y-values is independent of the values of any attributes

not contained in X U Y.

An example of a multivalued dependency taken from the relation scheme in Appendix 2 (taken from Ullman [1]) is:

course ->-> period, room, teacher

that is, associated with each "course" is a set of "period-room-teacher" triples which does not depend on any other attributes. For example, given the pair of tuples:

cs2a	3	601	jones j	adams a 42
cs2a	5	302	smith t	zebedee e 67

one would expect to be able to exchange (3, 601, jones j) with (5, 302, smith t) and obtain two valid tuples, viz:

cs2a	5	302	smith t	adams a 42
cs2a	3	601	jones j	zebedee e 67

However, it is not possible to exchange one or two fields of the triple without exchanging all of them, eg:

cs2a 5 601 smith t adams a 42

is not in the data base since "course ->-> room" does not hold.

(iii) Embedded Multivalued Dependencies. These are multivalued dependencies which do not apply in the full set of data but which become applicable when the data set is reduced by projection. Formally, given a relation scheme R, an embedded multivalued dependency is one which holds only when any relation r in R is projected onto some subset X [

R. For example, in the relation scheme presented in Appendix 2, the multivalued dependency "course ->-> prerequisite" does not hold since tuples such as:

cs2a zebedee e cs1b 1978

are not present in the data base. However, if the data in progresstable is projected onto the subset {course, student, prerequisite} giving:

cs2a	adams a	cs1a
cs2a	adams a	cs1b
cs2a	zebedee e	cs1a
cs2a	zebedee e	cs1b

then "course ->-> prerequisite" does hold, as does "course ->-> student".

(c) Transition constraints. These are restrictions on the way in which the data base may change; or, more specifically, the relationship between the states of the data base before and after any change is made. They include restrictions on the way in which:

(i) Values in a single field may change, e.g. values such as age or salary may only increase, marital status may only change in a particular way, etc.

(ii) Values in a set of fields (possibly in different relations) may change, e.g. the amount of special low-interest-rate loan may be increased only if the grade of the employee is above a certain level, etc.

Security, on the other hand, is concerned with who may access what information in the data base and what operations may be performed. The distinction between security and integrity constraints is not always clear as will be seen in later sections.

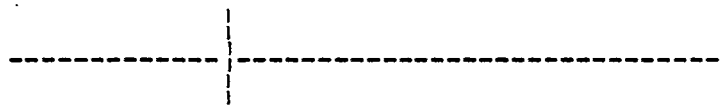
Zloof [4] has developed mechanisms for handling security and integrity constraints within the data base language Query-by-Example (QBE). The approach used for handling integrity constraints is a trivial extension of the concept of transition constraints in which constraints may be placed on insert, delete and update operations as well as on print operations. The problem with such an approach is that it is not possible to make any general statements about the data in the data base without a detailed history of the data base.

The object of this paper is to present a slightly different approach which includes all three types of constraints, and which does lend itself to statements about the properties of data in the data base.

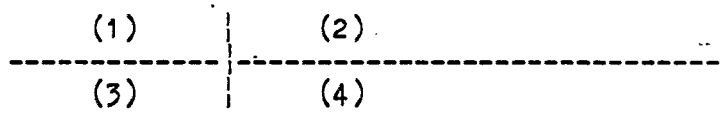
The following section gives a brief introduction to Query-by-Example, while section 3 looks briefly at the specification of security constraints (a slight variation from Zloof's approach). The remainder of the paper is devoted to the integrity constraints and implementation details.

2. QUERY-BY-EXAMPLE - THE BASIC LANGUAGE

Query-by-Example [5] is a two-dimensional language which is designed for use at a terminal and makes use of a special-purpose screen editor to compose queries. On striking a particular key, the user is presented with the skeleton of a table as follows:



The four areas delimited by this skeleton are:



(1) Table name field,

(2) Column name field,

(3) Tuple command field, and

(4) Tuple entry field.

Using the screen editor the user may position the cursor in any of these four areas in order to insert a command and/or a variable or constant element. The formulation of queries is achieved by setting up tuples containing variables, constants and conditions. An attribute which is to be displayed is indicated to the system by typing "p.", followed possibly by a variable name and possibly by a condition, in the column corresponding to that attribute. In our implementation lower-case letters have

been used in place of upper-case letters for the basic operations.

For example, to print the status of a particular supplier, say "clark", given the data base of Appendix 1, the user may enter the table name "suppliers" in the table name field, viz (the parts which the user might enter are underlined "___"):

suppliers	
----------	----------

Since the relation already exists in the data base the column headings (attributes of suppliers) can be generated by the system, i.e.:

suppliers	sno	sname	status	city
----------	----------			

One can now enter "clark" in the sname field and "p.X" in the status field as follows:

suppliers	sno	sname	status	city
----------	----------			
		clark	p.X	

Any character sequence beginning with a lower case letter, such as "clark", is taken to be a constant representing a specific value, while one beginning with an upper case letter or an underline symbol "_" is taken to be

a variable. Thus this is interpreted as a request to print the status of any supplier whose name is "clark".

Similarly to print the details of any supplier whose status exceeds 10, one may enter:

suppliers	sno	sname	status	city
	p.X	p.Y	p.A::A>10	p.C

or one may write the command "p." in the tuple command field as follows:

suppliers	sno	sname	status	city
p.	X	Y	A::A>10	C

where the infix operator "::" is used as a syntactic aid and is to be read as "such that".

A query may require more than one relation in which case appearances of the same variable name in different parts of a query represent the same value. For example, to display the names of all suppliers who supply parts which are red, one may enter:

parts	pno	pname	colour	weight
	X		red	

supplier_parts	sno	pno	qty
	Y	X	

suppliers	sno	sname	status	city
	Y	p.Z		

Complex conditions are handled by use of a separate condition box. For example, suppose that one wishes to display the names of all suppliers for whom the quantity of part number 2 lies between 100 and 300. One may enter:

suppliers	sno	sname	status	city
	X	p.Y		

supplier_parts	sno	pno	qty
	X	2	Z

CONDITIONS
Z>99 and Z<301

Besides the query operator "p." there are three other

operators: "i." (Insert), "d." (Delete) and "u." (Update). As an illustration of the use of "i.", consider the addition of a new part tuple to the relation parts:

parts	pno	pname	colour	weight
i.	7	washer	red	10

3. SECURITY IN QUERY-BY-EXAMPLE

Security constraints take the form of an authorization for a user to perform certain operations on a relation. For example, if one wishes to permit a user John to perform print, update and insert operations on the relation suppliers, this may be specified as follows:

suppliers	sno	sname	status	city
i.autr(p.,u.,i.).john	A	B	C	D

where once again lower case letters have been used and the final "i." omitted [4].

The presence of a variable in each field of the relation indicates that John has access to that field. If the variable C had been omitted and the status field left blank, this would indicate that John does not have access to the status field. Just as in other QBE statements, one may add conditions to these variables or link them to fields in other relations.

A more complex example which illustrates this imposes the constraint that John may only read details from the supplier_parts relation if the status of the supplier is less than 30 or the supplier comes from Paris. This is specified as follows:

supplier_parts	sno	pno	qty
i.autr(p.).john	A	B	C

suppliers	sno	sname	status	city
	A		E	F

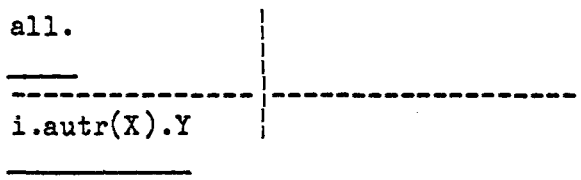
CONDITIONS
E<30 or F=paris

In each case the entry in the tuple-command-field has the form:

i.autr(<access rights lists>).<user>

The <access rights list> is a list of one or more of the four rights "p.", "i.", "u." or "d." while <user> is the name of the user to whom access is to be granted. In generalizing these two items, following the philosophy of QBE, variables may be used. Similarly if the keyword "all." is used in the table-name-field it will refer to all

relations. Thus the constraint:



will allow any user to perform any operation on any relation.

4. REALIZATION IN PROLOG

In our initial implementation of QBE in Prolog [6], each QBE request (insertion, deletion, update, print, constraints) was translated directly into Prolog and applied to the data base. However, when we changed our approach to integrity constraints and adopted the approach which will be described in the next section, a different implementation strategy was called for.

In the current system (which runs both on a PDP 11/34 and a DEC 10 machine), each QBE request is translated into a clause in a meta-language which is then interpreted using the remainder of the data base.

The following notation is used to express object-level knowledge in the meta-language:

(1) A rule clause is represented as:

p <- [q1, q2, ..., qn, {s}].

which stands for p :- q1,q2, ...,qn.
while the string s in braces {} is used
to store information for recreating the

original QBE request.

(2) A goal clause is represented by:

<- [q1,q2,...,qn].

which stands for ?- q1,q2,...,qn.

(3) A fact or assertion is represented as:

p.

which stands for p.

The usual interpretations are to be understood for rules, goals and assertions [7]. The use of the meta-language at the object-level has the great advantage of allowing one to use clauses and predicates as terms.

5. EXTENSION TO HANDLE INTEGRITY CONSTRAINTS

The general philosophy behind the approach described here is that any constraint which is currently operative must apply to all data in the data base. Thus whenever a new constraint is defined, it is immediately checked against the data in the data base. If any of the data does not satisfy the constraint, the exceptions are reported and the user is given the opportunity of either updating the data or revising the constraint. If all the data does satisfy the new constraint, it is stored and used to check all insertions and update operations conducted in the future. Three new operators are introduced for this purpose:

ic. - insert a new constraint

dc. - delete an existing constraint

pc. - print constraints

The form of a constraint definition is similar to that of a query. As a simple example, consider the insertion of the constraint that the value in the quantity field of each supplier_parts tuple should be greater than zero. To do this one may enter:

supplier_parts	sno	pno	qty
ic.			X::X>0

or one may use the condition box as follows:

supplier_parts	sno	pno	qty
ic.			X

CONDITIONS
X > 0

which is translated by the system to yield:

```
supplier_parts(_, _, X) <-
      [ X>0,
        {X>0}
      ].
```

To ensure that a tuple may only exist in the

supplier_parts relation if a tuple for the supplier concerned exists in the suppliers relation, one may have:

supplier_parts	sno	pno	qty
ic.	X		

suppliers	sno	sname	status	city
	X			

which is translated by the system to yield:

```
supplier_parts(X, _, _) <-
      [ suppliers(X, _, _, _),
        {}
      ].
```

A more complicated value-based constraint is the restriction that any supplier from Vienna or Athens must have a status which is at least 20. To specify this, one has:

suppliers	sno	sname	status	city
ic.			X	Y

CONDITIONS
(Y = vienna or Y = athens) implies (X >= 20)

which is translated by the system to yield:

```
suppliers(_, _, X, Y) <-
   [ not (Y=vienna or X>=20) and
     not (Y=athens or X>=20),
     {(Y=vienna or Y=athens) implies (X>=20)}
   ].
```

Functional dependencies are specified in the condition box using the format:

<var> -> <var>

or (<varlist>) -> <var>

For example, in the parts relation, suppose that "pno -> weight". This can be specified as a constraint as follows:

parts	pno	pname	colour	weight
ic.	X			Y

CONDITIONS
X -> Y

which is translated as follows:

```
parts(X, _, _, Y) <-
      [ parts(X, _, _, U),
        Y=U,
        {1->4}
      ].
```

This can be read as:

```
for all X, A, B, Y:
if there exists R, S, U such that
if parts(X, R, S, U) and Y=U are true
then parts(X, A, B, Y) is true.
```

When this command is given, the data base will be checked immediately to ensure that the data already present satisfies this condition. Provided it does, the constraint will be added to the data base. Thereafter whenever the user inserts or updates a tuple in the parts relation it attempts to deduce "weight" from "pno" and fill it in automatically for the user.

Multivalued dependencies are specified in a similar way using the format:

<X> ->-> <Y>

where <X> and <Y> each stand for either a single variable or a variable list enclosed in parentheses. Thus in the example from Appendix 2 one might express the constraint:

timetable	course	period	room	teacher	student	mark
ic.	W	X	Y	Z		

CONDITIONS
W ->-> (X, Y, Z)

which is formalized as follows:

```
timetable(W, X, Y, Z, R, S) <-
      [ timetable(W, A, B, C, M, N),
        timetable(W, A, B, C, R, S),
        timetable(W, X, Y, Z, M, N),
        {1->->(2,3,4)}
      ].
```

Once again when this command is given the data base is checked for any violations. If violations arise they are reported, if not the constraint is added to the data base. Thereafter whenever an insertion or update operation causes this constraint to be invoked, the system generates (and displays) the full set of tuples which need to be added to the data base in order to maintain consistency. If the user is content with the set of tuples generated, the system adds the full set to the data base, otherwise the insertion/update operation is abandoned.

Embedded multivalued dependencies are specified using the format:

<X> ->-> <Y>/<Z>

where <X>, <Y> and <Z> each stand for either a single variable or a variable list in parentheses. This is interpreted as X multidetermines Y if the set of attributes Z is removed. For example, to express the fact that "course ->-> prerequisite" if the relation "progresstable" in Appendix 2 is projected onto the subset {course, student, prerequisite}, one may enter:

progresstable	course	student	prerequisite	year
ic.	X		Y	Z

CONDITIONS
X ->-> Y/Z

which is translated by the system to yield:

```
progresstable(X, A, Y, Z) <-
      [ progresstable(X, B, C, E),
        progresstable(X, A, C, R),
        progresstable(X, B, Y, S),
        {1->->3/4}
      ].
```

When this command is given, the data base is checked for consistency. If violations arise the user is prompted to correct them or abort the constraint. Once the constraint is added to the data base, any further insertions or update operations are checked against the constraint and where required the system will generate the full set of tuples needed to fulfil any particular operation, prompting the user for the additional information (year) required to complete each tuple.

Transition constraints, which are concerned with the way in which values in the data base may change, are expressed using a pair of entries for the relation in question. The field in this relation which is to be controlled, will be represented by two different variables -

the one occurring in the line with the ic command in the tuple-command-field represents the new value of the variable, the other the old value.

For example, suppose that one wishes to place a constraint on the status of a supplier whereby it can only increase, one might enter:

suppliers	sno	sname	status	city
ic.	N		X	
	N		Y	

CONDITIONS
X>=Y

which is translated by the system to yield:

```
suppliers(N, _, X, _) <-
      [ suppliers(N, _, Y, _),
        X>=Y,
        {X>=Y}
      ].
```

Similarly one might impose a constraint on the age or salary of an employee whereby the values of these fields for a particular employee can only increase. In the case of marital status the only permissible transitions may be:

```
single    --->   married
married   --->   divorced or widowed
divorced  --->   married
widowed   --->   married
```

which may be specified as follows:

employee	empno	ename	salary	status	grade
ic.	N			X	
	—			—	
	N			single	
	—				

CONDITIONS
X=married or X=single

employee	empno	ename	salary	status	grade
ic.	N			X	
	—			—	
	N			married	
	—				

CONDITIONS
X=married or X=divorced or X=widowed

and so on. This is translated by the system to yield:

```
employee(N, _, _, X, _) <-
      [ employee(N, _, _, single, _),
        X=married or X=single,
        {X=married or X=single}
      ].

employee(N, _, _, X, _) <-
      [ employee(N, _, _, married, _),
        X=married or X=divorced or X=widowed,
        {X=married or X=divorced or X=widowed}
      ].

...
...
...
```

Alternatively the four constraints may be combined into a single one using two variables.

As an example of a more complex form of constraint, consider the restriction that the value of a loan may only increase (or decrease) if the grade of the employee is greater than 5. This might be specified as follows:

loantable	empno	loan
ic.	X	NL
—	—	—
	X	OL
	—	—

employee	empno	ename	salary	status	grade
	X				Y
	—				—

CONDITIONS
(Y<=5) implies (NL=OL)

This is translated by the system to yield:

```
loantable(X, NL) <-
        [ loantable(X, OL),
          employee(X, _, _, _, Y),
          not Y<=5 or NL=OL,
          {(Y<=5) implies (NL=OL)}
        ].
```

The complete syntax of these constraints is given in Appendix 3.

6. OVERLAP OF INTEGRITY AND SECURITY CONSTRAINTS

The transition constraints discussed in the previous section deal only with the way in which data in the data base may change (i.e. be updated). It does not cater for transitions involving insertion or deletion.

Thus suppose one wishes to impose the constraint that a loan may only be granted to an employee with grade between 5 and 8, but once an employee has been granted a loan, if his grade changes to a value outside the range 5-8, he will not lose his existing loan. This type of constraint is not a simple property of the data (i.e. one cannot conclude that any employee who has a loan, must have a grade in the range 5 to 8). However, it can be handled using a security constraint, eg.

loantable	empno	loan
i.autr(i.).X	A	B
	—	—

employee	empno	ename	salary	status	grade
	A				C
	—				—

CONDITIONS
(C>=5) and (C<=8)

Likewise the example considered by Nicolas and Yazdanian [2] in which a constraint needs to be placed on the system to prevent employees whose income is less than some value (say 5000) from being deleted, can be treated as follows:

employee	empno	ename	salary	status	grade
i.autr(d.).X	A	B	C	D	E
	—	—	—	—	—

CONDITIONS
C>=5000

7. CONCLUSIONS

The specification and enforcement of integrity constraints in a data base system is essential in order to guarantee the consistency of data within the data base. The role of security constraints is to control the types of operations which individual users may perform on the data base. The two types of constraints overlap to some extent.

This paper presents an integrated approach for specifying generalized integrity and security constraints within the data base management language Query-by-Example.

The important aspects of this approach are:

(a) It caters for all three types of integrity constraints in a generalized and consistent manner.

(b) It treats integrity constraints as properties of the data applying to all data in the data base, rather than as properties of particular operations (as proposed by Zloof [4]).

(c) It ensures that the user is aware of the

implications of any operation producing changes in the data base which affect fields involved in multivalued or embedded-multivalued dependencies.

ACKNOWLEDGEMENTS

The work of one of the authors, J C NEVES, was supported by the Calouste Gulbenkian Foundation under grant 14/82 and by an ORS award from the Committee of Vice-Chancellors and Principals of the Universities of the United Kingdom.

J C NEVES is on leave from Minho University, Largo do Paco, 4700, BRAGA, PORTUGAL.

8. REFERENCES

[1] J. D. Ullman, Principles of Database Systems, London: Pitman, 1980.

[2] J. M. Nicolas and K. Yazdanian, Integrity Checking in Deductive Data Bases, in: Logic and Data Bases, Plenum Press, 1978, 325-344.

[3] W. W. Armstrong, Dependency Structures of Database Relationships. Proc. IFIP 74, 1974, 580-583.

[4] M. M. Zloof, Security and Integrity within the Query-by-Example Database Management Language, IBM RC6982, Yorktown Heights, N. Y., 1978.

[5] M. M. Zloof, Query-by-Example: A Data Base Language, IBM Systems J., Vol. 16, No. 4, 1977, 324-343.

[6] J. C. Neves, S. O. Anderson and M. H. Williams, A Prolog Implementation of Query-by-Example, in: Proceedings of the 7th International Computing Symposium, March 22-24, 1983, Nurnberg, Germany.

[7] W. F. Clocksin, and C. S. Mellish, Programming in Prolog, Springer-Verlag, 1981.

[8] K. Bowen, and R. A. Kowalski, Amalgamating Object Language and Metalanguage in Logic Programming. To appear in Logic Programming (K. L. Clark and S. -A. Tarnlund Eds.)

Academic Press,1982.

[9] M. H. Williams, A Flexible Notation for Syntatic Definitions, ACM Trans. on Prog. Lang. and Syst., Vol. 4, No. 1, 1982, 113-119.

Appendix 1: A simple business data base

Consider a simple business data base which contains:

(i) A relation "parts" with attributes (columns): pno, pname, colour and weight.

(ii) A relation "suppliers" with attributes: sno, sname, status and city.

(iii) A relation "supplier_parts" with attributes: sno, pno and qty.

(iv) A relation "employee" with attributes: empno, ename, salary, status and grade.

(v) A relation "loantable" with attributes: empno and loan.

Suppose that the current content of each relation is:

parts	pno	pname	colour	weight
	1	nut	red	12
	2	bolt	green	17
	3	screw	blue	17
	4	screw	red	14
	5	cam	blue	12
	6	cog	red	19

Table 1.1 - The parts relation

suppliers	sno	sname	status	city
	1	smith	20	vienna
	2	jones	10	paris
	3	blake	30	paris
	4	clark	20	vienna
	5	adams	30	athens

Table 1.2 - The suppliers relation

supplier_parts	sno	pno	qty
	1	1	300
	1	2	200
	1	3	400
	1	4	200
	1	5	100
	1	6	100
	2	1	300
	2	2	400
	3	2	200
	4	2	200
	4	4	300
	4	5	400

Table 1.3 - The supplier_parts relation

employee	empno	ename	salary	status	grade
	12	morley	6500	married	10
	7	warren	7135	single	7
	15	exner	4475	single	4
	17	berry	5345	married	12
	5	john	6725	widowed	9

Table 1.4 - The employee relation

loantable	empno	loan
	7	570
	17	1500

Table 1.5 - The loantable relatio

Appendix 2: A simple departmental data base

Consider a simple departmental data base which contains:

(i) A relation "timetable" with attributes: course, period, room, teacher, student, grade.

(ii) A relation "progresstable" with attributes: course, student, prerequisite, year.

Suppose that the current content of the data base is:

timetable	course	period	room	teacher	student	grade
	cs2a	3	601	jones j	adams a	42
	cs2a	5	302	smith t	zebedee e	67

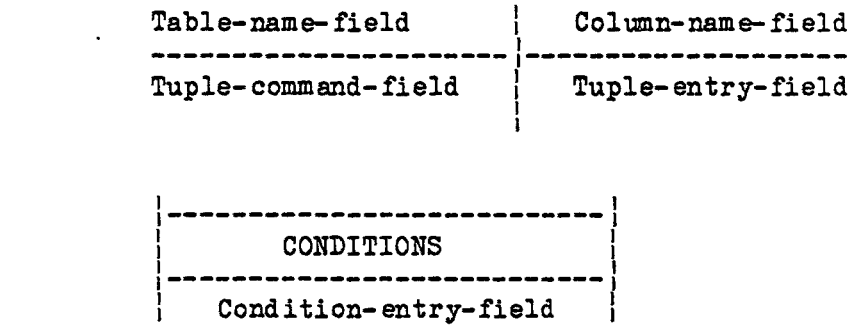
Table 2.1 - The timetable relation

progresstable	course	student	prerequisite	year
	cs2a	adams a	cs1a	1978
	cs2a	adams a	cs1b	1979
	cs2a	zebedee e	cs1a	1978
	cs2a	zebedee e	cs1b	1979

Table 2.2 - The progresstable relation

Appendix 3: Concrete syntax of the data base query language Extended-Query-by-Example

The basic Extended-Query-by-Example (EQBE) format is as follows:



where the syntax of each of these components is defined as:

```
table-name-field ::= ("i." | "u.") string-constant |
                     ["p." | "d."] [string-constant] |
                     "all."
column-name-field ::= ["p."] [string-constant]
tuple-entry-field ::= ["p."] [example-element
                      [":" relation] | p-relation]
                      | string-constant | integer
authorization ::= "autr" ["(" access-rights-list ")"] "."
                  user-list
access-rights-list ::= access-right ("," access-right)* |
                       example-element
access-right ::= "p." | "i." | "d." | "u."
user-list ::= list | example-element | string-constant
list ::= "(" string-constant ("," string-constant)* ")"
```

```
tuple-command-field ::= ["ic." | "dc." | "pc." |
                         ("i." | "d." | "u." | "p.")
                         [authorization]]
condition-entry-field ::= functional-dependency
                        | multivalued-dependency
                        | embedded-multivalued-dependency
                        | boolean-expression
functional-dependency ::= set "->" example-element
multivalued-dependency ::= set "->->" set
embedded-multivalued-dependency ::= set "->->" set "/" set
set ::= "(" example-element ("," example-element)* ")"
       | example-element
boolean-expression ::= boolean-secondary
                       ("implies" boolean-secondary)*
boolean-secondary ::= boolean-term ("or" boolean-term)*
boolean-term ::= boolean-factor ("and" boolean-factor)*
boolean-factor ::= ["not"] boolean-primary
boolean-primary ::= boolean-constant | relation
                  | "(" boolean-expression ")"
boolean-constant ::= "true" | "false"
relation ::= numeric-exp relational-op numeric-exp
           | string-exp relational-op string-exp
p-relation ::= relational-op (numeric-exp | string-exp)
numeric-exp ::= [add-op] numeric-term
                (add-op numeric-term)*
numeric-term ::= factor (multiply-op factor)*
factor ::= [function-designator] numeric-variable
```

```
          ¦ numeric-constant ¦ "(" numeric-exp ")"
multiply-op ::= "*" ¦ "/"
add-op ::= "+" ¦ "-"
function-designator ::= "max." ¦ "min." ¦ "ave."
                      ¦ "cnt." ¦ "sum."
string-exp ::= string-primary ("+" string-primary)*
string-primary ::= string-variable ¦ string-constant
integer ::= digit +
string-constant ::= """"non-quote-character*""""+ ¦
                    lower-case-letter letter-or-digit*
string-variable ::= example-element
numeric-variable ::= example-element
example-element ::= capital-letter letter-or-digit*
                  ¦ underscore letter-or-digit*
letter-or-digit ::= lower-case-letter ¦ digit
capital-letter ::= "A" ¦ "B" ¦ "C" ¦ "D" ¦ "E" ¦ "F"
                 ¦ "G" ¦ "H" ¦ "I" ¦ "J" ¦ "K" ¦ "L"
                 ¦ "M" ¦ "N" ¦ "O" ¦ "P" ¦ "Q" ¦ "R"
                 ¦ "S" ¦ "T" ¦ "U" ¦ "V" ¦ "W" ¦ "X"
                 ¦ "Y" ¦ "Z"
lower-case-letter ::= "a" ¦ "b" ¦ "c" ¦ "d" ¦ "e" ¦ "f"
                    ¦ "g" ¦ "h" ¦ "i" ¦ "j" ¦ "k" ¦ "l"
                    ¦ "m" ¦ "n" ¦ "o" ¦ "p" ¦ "q" ¦ "r"
                    ¦ "s" ¦ "t" ¦ "u" ¦ "v" ¦ "w" ¦ "x"
                    ¦ "y" ¦ "z"
digit ::= "0" ¦ "1" ¦ "2" ¦ "3" ¦ "4" ¦ "5"
        ¦ "6" ¦ "7" ¦ "8" ¦ "9"
```

where the notation used is that given by Williams [9].

28. F.MOUTA, M.H.WILLIAMS and J.C.F.M.NEVES, Implementing query languages in Prolog, in 'Prolog and Databases: Implementations and New Directions', Ed. P.M.D.Gray and R.J.Lucas, Ellis Horwood, Chichester, 13-21 (1988).

1

Implementing query languages in Prolog

Fernando Mouta
Departmento de Engenharia Electrotecnica e de Computadores, Faculdade de Engenharia da Universidade do Porto, Rua dos Bragas, 4099 Porto Codex, Portugal.
M. Howard Williams
Computer Science Department, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ, Scotland.
Jose Maia Neves
Area de Informatica e Controlo, Universidade do Minho, Largo do Paco, 4719 Codex, Portugal.

ABSTRACT

The close relationship between the query languages for relational database systems and logic programming languages facilitates translation between them. This chapter considers two different relational query languages, the tuple-oriented language SQL and the domain-oriented language QBE, and shows how queries in each of these languages can be mapped into the logic programming language Prolog. In both cases, query language processors have been constructed in Prolog to perform this translation and to execute the resulting queries

1. INTRODUCTION

First-order predicate logic plays an important role in relational database theory and can be used to express relational database formalisms and the functionalities normally associated with them [1]. Furthermore, it has been shown that Horn clause logic (a subset of first-order predicate logic) is sufficient for expressing the fundamental operations of the relational model [2]. Indeed, both first-order logic and Horn clause logic offer a deductive capability which considerably extends the relational model and provides a basis for more powerful database systems.

This chapter focuses on relational database systems and their relation to Horn clause logic in the form of the programming language Prolog. Two

different relational query languages are considered — the tuple-oriented language SQL and the domain-oriented language QBE. In each case a query language processor was developed in Prolog to accept queries in the language, map them into Prolog, and execute them.

2. REPRESENTATION OF THE DATABASE

Each relation in a relational database comprises a set of tuples. In mapping these into Prolog, each tuple is represented as a ground unit clause, e.g.

```
emp(1,'Fernando',2,30,3).
emp(2,'Carlos',3,60,3).
emp(3,'Jose',1,25,4).
emp(4,'Luis', 2,20,5).
```

In addition, for each relation there is a clause, 'scheme', which defines the structure of the relation. This contains the name of the relation, the number of tuples in the relation, the number of attributes and a list of the attribute names. Although slightly different formats were used in the two systems, the differences are very minor and only one will be used here. For the above relation, the clause would be.

```
scheme(emp,4,5,[empno,name,dno,sal,mgr]).
```

Each query language processor (for SQL and for QBE) accepts a query in the query language concerned and maps it into equivalent Prolog clauses which, when executed, produce the effect of the query. In the process, the system builds up structures with the same name and arity as the clauses that represent the tuples of the relations concerned as well as structures representing conditions on attributes. These structures are transformed to produce a Prolog goal representing the query.

For example, the SQL instruction

```
SELECT name,sal
FROM emp
WHERE dno=2
```

and the QBE query

emp	empno	name	dno	sal	mgr
		P.X	Y	P.Z	

CONDITIONS
Y=2

can be transformed to the Prolog form

```
emp(_,X1,X2,X3,_),X2=2
```

or simply

```
emp(_,X1,2,X3_)
```

3. USER INTERFACE

Each system makes use of a special-purpose screen editor to assist users to compose queries. In the case of QBE a tabular format is used. At the beginning of a query the user is presented with a skeletal table of form

(1)	(2)
(3)	(4)

where the four areas are

(1) Table name field,
(2) Column name field,
(3) Tuple command field,
(4) Tuple entry field.

The four basic commands which are available are 'P.' (standing for PRINT), 'I.' (for INSERT), 'D.' (for DELETE) and 'U.' (for UPDATE). Using the screen editor, the cursor may be moved to any of the four areas. Starting with the table name field, if one of the basic commands is entered, it applies to the set of relations, e.g. print all relation names, insert a new relation, etc. On the other hand, one may enter the name of a relation, and if it already exists in the database, the system can generate the column names automatically. If the cursor is moved to the tuple command field and the command 'P.' is entered, it applies to whole tuples (i.e. print each tuple which satisfies some condition) whereas if it occurs in the tuple entry field, it applies only to the attribute concerned. For example,

emp	empno	name	dno	sal	mgr
P.			2		

will print all tuples from the relation emp which have the value 2 in the dno field whereas

emp	empno	name	dno	sal	mgr
		P.X	2	P.Y	

will print only the name and salary attributes of tuples for which dno is 2.

The user interface for SQL is rather different. It also uses a screen editor, but in this case a more conventional screen editor for text strings. This makes

it easier for the user to enter a query, especially one which is long and complex.

Once a query has been entered, it is subjected to lexical analysis followed by syntax analysis to check that it is valid and to convert it to internal form. To assist the user further a spelling correction and abbreviation identification module is constructed. When the system analyses a query and encounters an incorrect word in the parse, this word together with information about the kind of word expected in that position of the query is passed to the correction module. The correction module attempts to match the characters of the word with those of possible correct words, checking for incorrect characters, missing characters, additional characters or permuted characters. It also checks for abbreviations of possible correct words. Thus if one were to enter the query

```
selact nmae
from emp
whre d=2
```

it would respond with the messages

```
LEXICAL ERROR — 1 letter of the word SELECT is wrong
LEXICAL ERROR — The name of the field name was written
                with 2 adjacent letters permuted
LEXICAL ERROR — 1 letter is missing in the word WHERE
LEXICAL ERROR — The name of the field dno has been
                abbreviated
```

and if the user accepted these corrections as valid, it would process the query as if the corrections had been made.

In each case the user interface was written in Prolog as part of the query language processor.

4. THE SQL SYSTEM

The overall structure of the SQL system is shown in Fig. 1. For this case a top-down parsing approach [3] was used, in which the nodes of the parse tree are created in pre-order starting from the root. The parser is implemented by building a procedure for each non-terminal node. As the BNF definition of SQL has left-recursive rules, these had to be rewritten to remove left recursion.

In the process of parsing, each attribute name that occurs in the query needs to be fully identified, checking the relation to which it belongs and its position in the list of attributes belonging to that relation. Each occurrence of an attribute name is mapped into a structure of form

```
[Relname,Lab,N,Attr]
```

where

Relname is the name of the relation to which the attribute belongs,

Lab represents a label name — if in the SQL query the relation to which

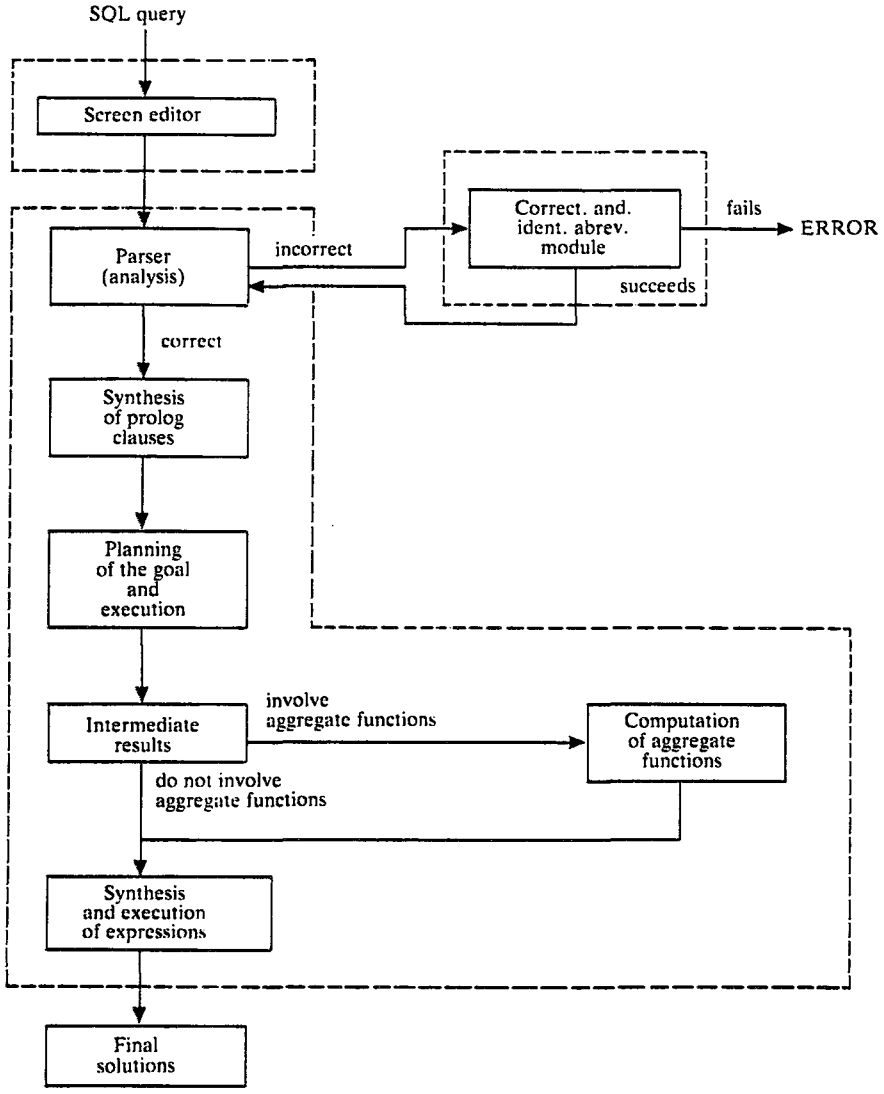


Fig. 1 — Structure of the SQL system.

the attribute belongs is referenced by a label, then Lab has that value, otherwide it is '*',
N is the position of the attribute in the list, and
Attr is the attribute name.

The aggregate functions are mapped into structures like

[Rel,Lab,N,setfn(AgrFn,Attr)]

where AgrFn is the name of an aggregate function ('count'. 'max','min',etc.).

In cases where these items appear in expressions, the evaluation of these expressions can only be performed after the appropriate tuples have been selected which satisfy the conditions in the 'WHERE' clause. In the case of aggregate functions, the computation can only be completed once all possible tuples have been selected.

From these attribute structures, a more complex structure is built, containing all the data necessary to construct a Prolog goal equivalent to the SQL query. This is a nested structure of form

str(N,Relname,Label,T,L.S,C,str(. . .))

where

N is the order number,
Relname is the relation name,
Label is the label name,
T is a term structure identical to the clauses that represent the tuples of the relation, with variables as arguments,
L is a list of the attributes of the relation,
S contains details of the selected items, and
C is a clause equivalent to the imposed conditions.

This structure will have a number of levels equal to the number of relation–label pairs. The innermost level has an anonymous variable as its last argument.

The argument S contains two lists of variables that stand for selected attributes. The first (SVar) contains variables which are used directly in the final Prolog goal to retrieve values of attributes for tuples which satisfy the query. The second is used for computing expressions and aggregate functions over the values retrieved by the goal.

The Prolog goal is constructed from the structure str, and is made up of the predicates representing the relations and those representing the conditions. Since execution of the Prolog goal can be inefficient, especially if the SQL query involves cartesian products of relations with many tuples, the planning of the Prolog goal attempts to minimize the total number of tuples considered during execution. The planning algorithm, described in [4], uses the following strategy:

(1) Divide the goal into independent parts that do not share uninstantiated variables, and add control information so that it will not be necessary to repeat them in different backtrack paths.
(2) Order the predicates (subgoals) of each independent part according to the size of the corresponding relations and the state of instantiation of the arguments. The planning algorithm selects the predicate with the least number of tuples that need to be considered, determines which variables will become instantiated as the result of its execution, and then repeats this whole strategy for the remaining subgoals of this part.

To illustrate this, consider the query which seeks the names of all

employees who earn more than their managers. This may be represented in SQL as

```
SELECT X.name,Y.name
FROM emp X,emp Y
WHERE X.mgr=Y.empno AND X.sal>Y.sal
```

The attribute structures created will be

```
[emp,X,2,name]
[emp,Y,2,name]
[emp,X,5,mgr]
[emp,Y,1,empno]
[emp,X,4,sal]
[emp,Y,4,sal]
```

From this will be constructed the Prolog goal

```
(emp(_3B69,_3B6D,_3B71,_3B75,_3B79),
 emp(_3F11M,_3F15,_3F19,_3F1D,_3F21),true),
 cond((_3B79=_3F11,_3B75>_3F1D))
```

with the selected attributes

```
[name:_3B6D,name:_3F15]
```

5. THE QBE SYSTEM

The translation from QBE to Prolog is much simpler than the translation from SQL. Each line in a QBE query maps onto a Prolog goal in a very straightforward manner.

In order to collect together the attributes to be displayed, a separate predicate, 'solution', is introducted. Thus the query

emp	empno	name	dno	sal	mgr
P.			2		

can be translated into

```
solution (A,B,C,D,E):-emp(A,B,C,D,E),C=2.
?-solution(A,B,C,D,E).
```

whereas the query

emp	empno	name	dno	sal	mgr
		P.X	2	P.Y	

would be transformed into

```
solution(X,Y):-emp(_,X,2,Y,_).
?-solution(X,Y).
```

The SQL query from the previous section could be expressed as

emp	empno	name	dno	sal	mgr
	Z	P.Y		A	
		P.X		B	Z

CONDITIONS
B>A

which could be translated directly to

```
solution(X,Y):-emp(Z,Y,_,A,_),emp(_,X,_,B,Z),B>A.
```

Insertions and deletions of entries can be handled using the Prolog predicates assert and retract which, respectively, add clauses to and retract clauses from the database. For example, the QBE command

suppliers	sno	sname	status	city
U	X	Y	Z	10
	X	Y	Z	15

which instructs the system to update all entries with a status of 10 in the suppliers table to a status of 15, can be converted to a Prolog clause

```
?-retract(suppliers(X,Y,Z,10)),
 asserta(suppliers(X,Y,Z,15)),
 fail.
```

which will achieve the desired effect.

In this case the process of analysis is much simpler than for SQL, and this is described further in [5]. More examples of QBE and translation to Prolog, including the use of the GROUP-BY operator, are given in [6].

Other aspects such as the treatment of integrity constraints and authorization are described in [7].

6. CONCLUSION

This chapter has attempted to show how Prolog can be used to handle different relational query languages. Two query languages have been chosen for this study — the tuple-oriented language SQL and the domain-oriented language QBE. In each case a special-purpose screen editor was constructed to interface with the user.

The translation process for SQL is more complex than that for QBE since the system has to identify attributes occurring in the query, check these against the schema, and gradually build up a structure representing the query. This is then mapped into a Prolog query. In the case of QBE, the names and positions of attributes in a relation are handled automatically by the screen editor so that the translation process reduces to a much more straightforward mapping of the elements of the query into Prolog.

REFERENCES

[1] Gallaire, H., Minker, J., and Nicolas, J. M., Logic and Databases: a Deductive Approach, *ACM Computing Surveys,* **16**(2), 153–185, 1984.

[2] Kowalski, R. A., Logic as a Database Language, Research Report, Dept. of Computer Science, Imperial College, 1981

[3] Aho, A. V. and Ullman, J. D., *Principles of Compiler Design,* Addison-Wesley, Reading, MA, 1977.

[4] Warren, D. H. D., Efficient Processing of Interactive Relational Database Queries Expressed in Logic, *Proc. 7th Int. Conf. on VLDB,* 1981, 272–281.

[5] Neves, J. C. F. M., Anderson, S. O., and Williams, M. H., A Prolog Implementation of Query-by-Example, *Proc. 7th Int. Computing Symposium,* Teubner, Stuttgart, 1983, 318–332.

[6] Gray, P. M. D., *Logic, Algebra and Databases,* Ellis Horwood, Chichester, 1984, 145–153, 258.

[7] Williams, M. H., Neves, J. C. F. M., and Anderson, S. O., Security and Integrity in Logic Databases using Query-by-Example, *Proc. Logic Programming Workshop '83,* Albufeira, Ed. L. M. Pereira, 1983, 304–340.

29. J.C.F.M.NEVES and M.H.WILLIAMS, Towards a co-operative data base management system, Proceedings of Logic Programming Workshop '83, Ed. L.M.Pereira, Universidade Nova de Lisboa, Lisbon, 341-370 (1983).

TOWARDS A CO-OPERATIVE DATA BASE MANAGEMENT SYSTEM

J C NEVES and M H WILLIAMS

DEPARTMENT OF COMPUTER SCIENCE

HERIOT-WATT UNIVERSITY

EDINBURGH

SCOTLAND

KEYWORDS: Logic programming, Horn clauses, Prolog, Logic data base, Query-by-Example, Incomplete queries, Co-operativeness.

Abstract

A desirable feature of any high-level data base query system is that it should be user-friendly. This should extend beyond the provision of a query syntax which is easy to use, to some attempt at intelligent helpfulness or co-operativeness. In particular additional knowledge about the structure of the data in a data base or the incomplete data contained in a query may be used to benefit the user. In this respect, despite its simplicity and ease of use, the data base management language Query-by-Example is relatively inflexible.

This paper looks at several ways in which the co-operativeness of Query-by-Example can be improved. These are concerned with incomplete queries (i.e. queries in which certain information has been omitted), incomplete updates and queries which fail as a result possibly of misconceptions on the part of the user. Consideration is also given to how these are implemented in Prolog.

1. Introduction

The application of first order logic and resolution based theorem proving to machine intelligence problems started during the early 1970's. Recently, logic programming has received a considerable boost due to its choice as the basis of the core programming languages for the Japanese Fifth Generation Computer Systems [1].

Prolog [2] is a qualified implementation of Horn clauses which has become important as a vehicle for Artificial Intelligence applications. In particular there is growing interest in its use for data base applications [3]. Since Prolog itself is not very convenient as a query language, various researchers have sought to develop other user interfaces to Prolog data bases. These include natural language interfaces [4] and Query-by-Example [5].

Query-by-Example (QBE) is a non-procedural data base query language developed by Zloof [6] in which queries are expressed by filling in skeleton tables with examples of the result required. In a human factors experiment conducted by Thomas and Gould [7] to determine the ease of use of data base query languages, the advantages of QBE over SQUARE and SEQUEL were clearly demonstrated. In particular they found that subjects using QBE required about one-third the training time, were somewhat faster in expressing queries and were about twice as accurate [7].

In view of this and the similarity between the syntax

of Prolog goals and QBE [8], an implementation of QBE interfacing with a logic data base has been realized in Prolog. Details of the implementation are given in [5].

Despite its simplicity and ease of use, QBE is relatively inflexible and makes no attempt at intelligent helpfulness or co-operativeness. This paper consider some ways in which the co-operativeness of QBE can be improved.

2. Incomplete queries

In QBE all queries must be expressed in full in a manner which reflects the way in which the data has been stored in the data base. However, the inexperienced or casual user may have difficulty in remembering the internal structure of the data and the way in which any particular query must be framed in order to reflect this. On the other hand the experienced user may find the process a little clumsy and look for short cuts. The idea of an incomplete query may appeal to either type of user.

In QBE any simple query which involves the join of two relations makes use of a common variable which occurs in one field of each of the two relations.

For example, given the data base in Appendix 1, suppose that the user wants to find the names of all suppliers who supply part number 2. The parts which he/she might enter are underlined "___". The entry might be:

suppliers	sno	sname	status	city
	S	p.N		

supplier_parts	sno	pno	qty
	S	2	

The common variable here which serves to join the two relations is S. Such a variable will be referred to as a link variable, and the fields of the two relations which are linked together (sno of suppliers and sno of supplier_parts) will be referred to as link fields.

In general there is no choice in the pair of link fields which can be used to join two relations together. For example, in the case of the relations suppliers and supplier_parts, the field sno of relation suppliers and field sno of relation supplier_parts are the only possible pair of fields which can be used to join these two relations.

In some cases it may not be possible to join relations directly and a join may only be effected via one or more intermediate relations.

For example, suppose that the user wishes to retrieve the names of all suppliers who supply at least one red part. The essential information in this query is:

suppliers	sno	sname	status	city
		p.N		

parts	pno	pname	colour	weight
			red	

although the complete query is:

suppliers	sno	sname	status	city
	S	p.N		

supplier_parts	sno	pno	qty
	S	X	

parts	pno	pname	colour	weight
	X		red	

where S and X are both link variables and supplier_parts is an intermediate relation.

Since in general link variables are not an essential part of a query but rather a result of the way in which data are stored in the system, it should be possible for the user to omit link variables from any query (together with any empty intermediate tables which may result). Any query in which one or more of the link variables have been omitted will be referred to as an incomplete query.

However, there is one snag with the omission of the link variables. Consider the query:

suppliers	sno	sname	status	city
		p.N		

supplier_parts	sno	pno	qty
		2	p.X

If this is treated as an incomplete query the system would attempt to link together these two requests. The result might be:

suppliers	sno	sname	status	city
	S	p.N		

supplier_parts	sno	pno	qty
	S	2	p.X

which would be interpreted as "print the name of each supplier who supplies part number 2 and the quantity supplied". On the other hand, the original query is sufficient in its own right being interpreted as "print the names of all suppliers and the quantities of part number 2 as supplied by different suppliers". The latter is a form of OR-query.

In general an incomplete query will have the same form as an OR-query and the system will be unable to distinguish

between the two. Thus it must be assumed that an incomplete query will not involve an OR-condition and that the user will indicate when an incomplete query has been issued.

In the next section the underlying data structures and the general approach to implementation of incomplete queries are discussed.

3. Implementation of incomplete queries

If a user wishes to issue an incomplete query, the query is entered in exactly the same way as any other query except that a different key (for example, a special function key in the keyboard) is used to signal the end of the query.

When the system is presented with an incomplete query, it attempts to link together the separate parts of the request. If it succeeds in finding appropriate links, the resulting query will be displayed in full to the user. If this resulting query satisfies the user, he/she indicates acceptance of the query by pressing the key normally used at the end of a complete query; if it is not what the user wants, a different key is used to indicate to the system to continue its search. If no suitable links can be found, the system reports this to the user.

To illustrate this, consider a request for the names of any suppliers who supply widgets and to whom one does not owe money at the present moment. IQ and CQ are used to denote the keys corresponding to Incomplete Query and

Complete Query respectively. The dialogue might be as follows (commentary is in /* ... */ brackets):

suppliers	sno	sname	status	city
	p.N			

parts	pno	pname	colour	weight
	widget			

supplier_balance	sno	amountowed
	X::X=<0	

```
IQ  /* signals the end of an incomplete query */
```

The infix operator "::", is used for syntactic convenience only and is to be read as "such that".

In response to this the system might display:

suppliers	sno	sname	status	city
	A	p.N		

supplier_parts	sno	pno	qty
	A	B	

parts	pno	pname	colour	weight
	B	widget		

supplier_balance	sno	amountowed
	A	X::X=<0

3.1. Formal specification of links

The data structure used to represent the data base relations and the connections between the relations is an undirected graph.

Fig 1 is a diagrammatic representation of the graph representing the links of the data base in Appendix 1. Every data base relation is represented by a vertex or node, called a relation node, and for every two nodes, if the same attribute occurs in both relations, an edge will connect the pair. This edge is labelled with the pair of attribute names from the two relations.

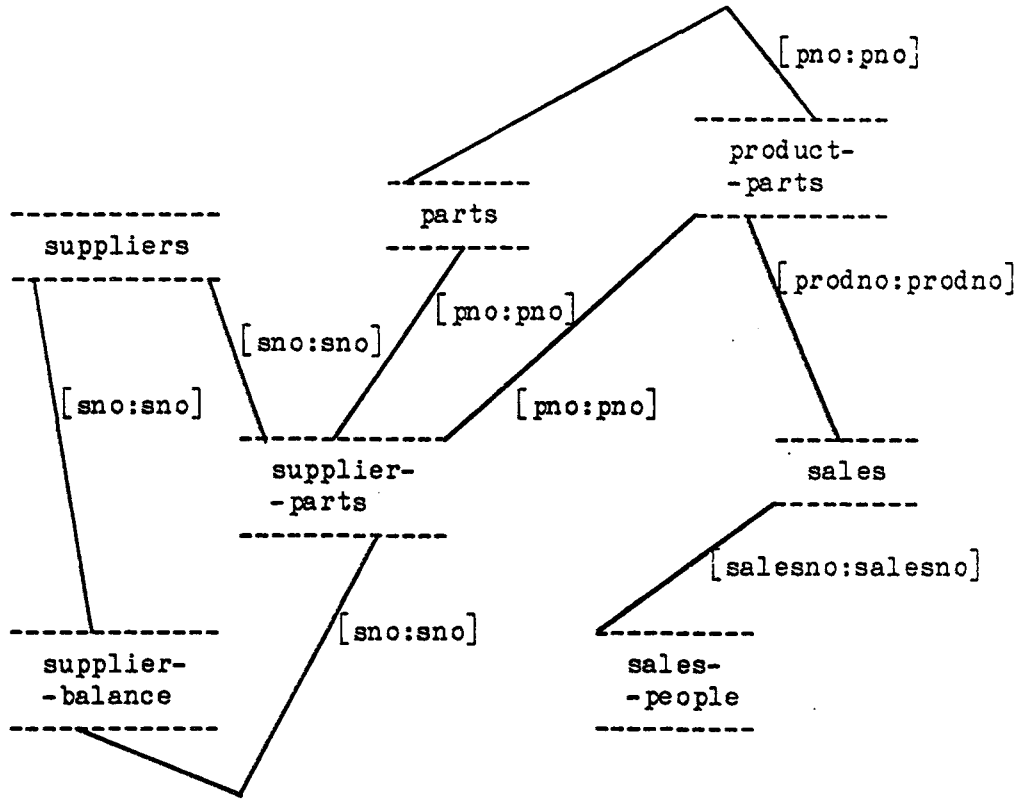


Fig 1 - The graph structure representing the link dictionary for the data base given in Appendix 1.

This information is represented within the query language system by a set of clauses of the form:

```
link(RELATION NAME 1, RELATION NAME 2,
      [
         ORDERED SEQUENCE OF LINK FIELDS OF RELATION 1:
         ORDERED SEQUENCE OF LINK FIELDS OF RELATION 2
      ]).

setofnodes(GRAPH NAME,
           [
              SET OF GRAPH NODES
           ]).
```

In Appendix 2 the link dictionary for the data base in Appendix 1 is given. Also presented is the Prolog program for searching for a path linking any pair of relations in

the data base.

The link dictionary described can be accessed by the user through the normal query mechanism, thus enabling the user to examine or update the structure of the data in the data base.

For example, suppose that the user wants to find which relations are linked with which . The entry might be:

```
p.links
-------
CQ  /* signals the end of a complete query */
--
```

The system will respond by displaying for each relation R a list of relations linked to R, e.g.

```
links    | supplier_parts
---------|----------------
         | supplier_balance
         | suppliers
         | parts
         | product_parts

         ...
         ...
         ...
```

3.2. Handling join conditions

In order to handle join conditions the system determines the number N of unlinked components of the request and then seeks to establish the paths linking them together.

For example, suppose that the user wants to find the names of any suppliers to whom no money is owed at the present moment and who supply part number 1023. The entry might be:

```
suppliers   | sno   sname   status   city
------------|---------------------------
            | X     p.N
              —     ——

supplier_balance  | sno   amountowed
------------------|-----------------
                  | X     A::A=<0
                    —     ———————

product_parts   | prodno   pno   noreqd
----------------|----------------------
                | 1023
                  ————

IQ  /* signals the end of an incomplete query */
—
```

where the user has partially specified the links by using the variable X to link relation suppliers with relation supplier_balance.

Given a query which contains join conditions, the paths linking the different components of the whole request may be established using:

> (i) - relation merging; that is, if two relations x and y which form part of the query are related to each other through the join variables A1, A2, ..., An (n>=1), merge relations x and y by performing joins between relations x and

y. Repeat this operation until no further merges are possible.

(ii) - graph generation; that is, look for paths which connect the remaining unlinked components of the graph (these must involve intermediate relations).

Let join-relation be the relation obtained by the join of relation suppliers with relation supplier_balance. Then the graph for the unlinked components of the initial request is as shown in Fig 2.

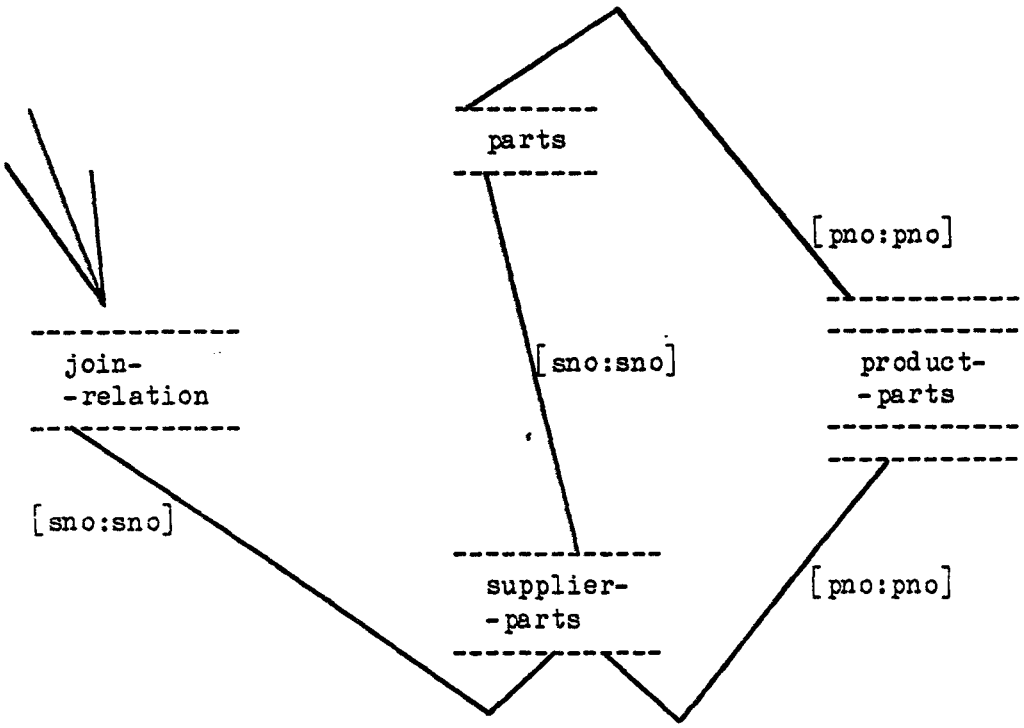


Fig 2 - Graph structure after merging.

The start node is indicated on the graph by an arrow, and double bars have been used to distinguish the final node. As a response, the system might display:

suppliers	sno	sname	status	city
	X	p.N		

supplier_balance	sno	amountowed
	X	A::A=<0

supplier_parts	sno	pno	qty
	X	Y	

product_parts	prodno	pno	noreqd
	1023	Y	

4. Incomplete updates

The ideas outlined in the previous section apply also to update operations.

For example if the user wishes to set the quantity to zero for all suppliers living in London, he/she might enter:

supplier_parts	sno	pno	qty
u			0

suppliers	sno	sname	status	city
				london

```
IQ  /* signals the end of an incomplete query */
```

to which the system will respond with:

supplier_parts	sno	pno	qty
u	Q		0

suppliers	sno	sname	status	city
	Q			london

In addition this link information may also be used in the case of update operations to ensure that when the user attempts to update a value in a link field of some relation, he/she is reminded of the possibility that the corresponding link field in some other relation may need to be updated too. In such a case the system might ask the user whether he/she wishes the same operation to be performed in the corresponding link field in the appropriate relation.

For example, if the user wishes to change the supplier number 13 to 3, the user might enter:

suppliers	sno	sname	status	city
u	3			
—	—			
	13			
	—			

```
CQ  /* signals the end of a complete query */
—
```

The system should then ask the user whether in addition he/she wishes to perform the following updates:

supplier_parts	sno	pno	qty
u	3		
	13		

supplier_balance	sno	amountowed
u	3	
	13	

In such case the user must indicate whether he wishes the additional update operations to be performed or rejected.

5. Queries which fail

In formulating a query a user inevitably makes certain presuppositions about the data present in the data base. These presuppositions are inherent in the information contained in the query and are an indication of what the user believes about the state of the information in the data base.

A data base query can be viewed either as requesting the selection of a subset (termed the response set) from a set of qualified instances in the data base, or as expressing some general belief about the data in the data base. In either case queries presented in QBE are translated into an intermediate meta language before being presented as a conjecture that a resolution-based theorem prover (e.g. Prolog) attempts to prove. This meta language is a graph structure, the nodes of which represent both data base relations and conditions imposed on the relation's

attribute(s).

The query graph is divided into connected subgraphs, each of which in itself constitutes a well-formed query in the meta language and is translated into a conjecture that can be presented to the theorem prover to be proved (i.e. each connected subgraph corresponds to a presupposition the user has made about the domain of discourse).

The next section discusses how the presuppositions inherent in these subgraphs can be used to provide a more co-operative response to users for both queries that request the selection of a subset of qualified instances in the data base and YES-NO queries.

5.1. Constructing corrective indirect answers

When dealing with queries requesting the selection of qualified instances in the data base (i.e. with queries defining a property of data base objects) consider the situation where the system fails to prove the conjecture (the initial query returns the empty set as an answer). In this case, on request from the user, the system will try to establish the user's presuppositions by translating each connected subgraph into a conjecture to be proved. This approach ensures that should a presupposition fail, an appropriate corrective indirect answer [9] will be returned to the user.

For example, suppose that the user wishes to retrieve

the numbers of all suppliers living in London who supply part number 2. The entry might be:

suppliers	sno	sname	status	city
	p.X			Y

supplier_parts	sno	pno	qty
	X	Z	

CONDITIONS
Z=2 and Y=london

```
CQ  /* signals the end of a complete query */
```

This query is based on the following presuppositions (i.e. the preconditions for the correctness of any direct answer):

(i) There are suppliers.
(ii) There are suppliers who supply parts.
(iii) There are suppliers supplying part number 2.
(iv) There are suppliers living in London.
(v) There are suppliers living in London who supply part number 2.

Should any of these presuppositions fail to be true, the system would, in general, respond with an empty list or "NULL". If, however, this query were addressed to a human being one might expect a more co-operative response which identifies the failing presupposition(s).

A complex query asking for the display of certain data items subject to a variety of retrieval conditions will be decomposed into a number of basic components in the meta language (i.e. connected subgraphs), each of which are acceptable queries in their own right. With each sub-query is associated a subset of the original set of presupposition(s). In the case of the above example, this can be represented diagramatically as:

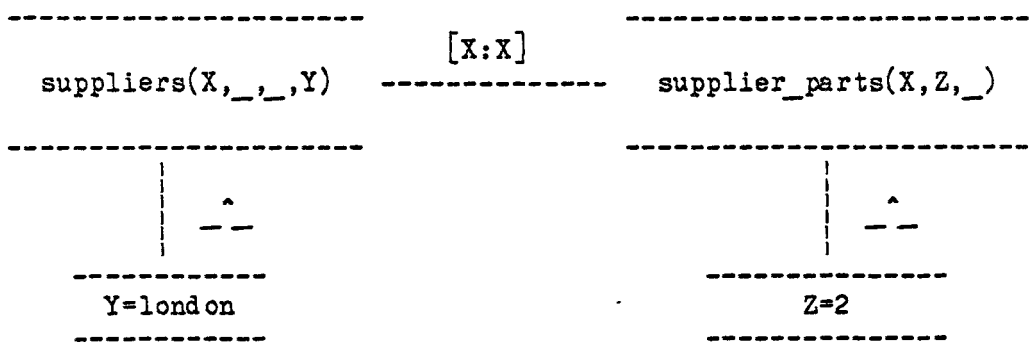


Fig 3 - The complete query.

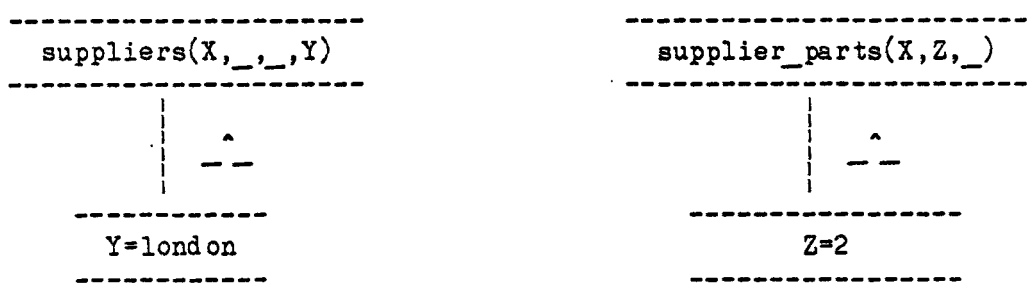


Fig 4 - Two first-level components



Fig 5 - Two second-level components

which will be translated by the system to yield:

```
<- suppliers(X, _, _, Y),
   Y=london,
   supplier_parts(X, Z, _),
   Z=2.
```

This clearly consists of two components:

```
<- suppliers(X, _, _, Y),
   Y=london.

<- supplier_parts(X, Z, _),
   Z=2.
```

each of these in turn depend on components:

```
<- suppliers(X, _, _, Y).

<- supplier_parts(X, Z, _).
```

In this case the system's response "NULL" will be produced only in the case where the top level query has failed but all sub-queries have succeeded. Otherwise the message "NULL-LOWER LEVEL QUERY FAILED" will be displayed.

On request the system will attempt to determine the cause of failure. If any sub-query fails and its failure contributes to the failure of the top level query, then:

- the failure of the sub-query will be reported back together with any other sub-query on the same level which

contributes to the failure of the top level query,

- any higher level failing sub-queries, not failing due to failure of component sub-queries, will be also reported back.

In the current implementation this is achieved by typing the keyword "WHY".

For example, if the query described above is presented to the system but the system fails to find any supplier living in London, it will respond with:

suppliers	sno	sname	status	city
	p.X			london

```
results: NULL  /* the empty set */
```

that is, the system recognizes the failure of the component:

```
<- suppliers(X, _, _, Y),
   Y=london.
```

and responds appropriately.

On the other hand, an OR-query fails if and only if all of its sub-queries fail. If one succeeds, the query as a whole succeeds, even if all the others fail. Such a situation might contribute to the misinterpretation by the user of the system's response due to the false assumptions made about the way the answer was inferred.

For example, suppose that the user wishes to retrieve the names of all suppliers who live in London or Paris. The entry might be:

suppliers	sno	sname	status	city
		p.X		london
		p.Y		paris

```
CQ  /* signals the end of a complete query */
--
```

to which the system's answer is:

results	sname
	jones
	blake

But it is known (see Appendix 1) that Jones and Blake both live in Paris, and that no one is in London. That is, the user can think of suppliers living in London and living in Paris to be correct, and carry on with a frustrating series of questions, or worse, misinterpret the system's response. To avoid such a situation, the user can, as soon as the answer has been displayed, request further information about the process used in the evaluating of the query. The result might be:

suppliers	sno	sname	status	city
		p.X		london

```
results: NULL  /* the empty set */
```

which indicates to the user that the presupposition that some suppliers were living in London is incorrect.

To the extent that update operations involve an initial request (query) aimed at locating certain qualified instances or tuples in the data base, a similar type of analysis as the one described above would apply in the case of failure.

In the case of a query which expresses some property of the data base as a whole, should the system fail to prove the conjecture, an attempt is made to prove the negation of the conjecture in order to answer "NO". Should the system fail to prove or disprove a given conjecture an answer of "DON'T KNOW" is returned to the user. This is the case when neither a "YES" nor "NO" answer is possible from the axioms in the data base.

If the answer is "NO" or "DON'T KNOW" an analysis of the presuppositions made might follow if requested.

6. Conclusions

Most query systems currently available respond to queries in a very literal manner, giving an answer to what the user actually asked for - no more and no less. Though the responses are literally correct, such rigidity can be very unhelpful at times, and a more flexible system is desirable. This flexibility in the interpretation of queries in a manner which is both natural and of benefit to the user is termed co-operativeness.

This paper outlines several ways in which the query

language QBE can be made more co-operative. These features have been added to a version of QBE implemented in Prolog, which is running under UNIX on both a PDP 11/34 and a DEC 10 system.

The main features of such a system are:

(i) A link dictionary has been implemented which contains information about the data base relations and the linkages between them. This facility was interfaced with the query facility to provide the user with the means to examine how the data in the data base are organized and how they should be accessed and used.

(ii) The system attempts to handle incomplete queries and updates by filling in link variables. This can be of use to casual users of the data base who do not have the details of the structure of the data base at their fingertips, as well as to experienced users who seek short cuts.

(iii) The system reminds users of possible side effects when updates are performed on link variables.

(iv) The system attempts to provide a helpful response when a complex query fails to give the user an indication of why it failed. The same tabular form is used to explain the reasoning it followed to arrive at the answer as that used to enter the initial request.

Acknowledgements

The work of one of the authors, J C NEVES, was supported by the Calouste Gulbenkian Foundation under grant 14/82 and by an ORS award from the Committee of Vice-Chancellors and Principals of the Universities of the United Kingdom.

J C NEVES is on leave from Minho University, Largo do Paco, 4700 BRAGA, PORTUGAL.

7. References

[1] T. Moto-Oka et al, Challenge for knowledge information processing systems, in: Fifth Generation Computer Systems (North-Holland, Amsterdam, 1982) 3-89.

[2] D. H. D. Warren, Implementing Prolog - Compiling Predicate Logic Programs. Technical Report 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

[3] H. Gallaire and J. Minker (eds), Logic and data bases, Plenum Press, New York, 1978.

[4] F. C. N. Pereira and D. H. D. Warren, Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks, Artificial Intelligence 13 (1980) 231-278.

[5] J. C. Neves, S. O. Anderson and M. H. Williams, A Prolog implementation of Query-by-Example, in: Proceedings of the 7th International Computing Symposium, March 22-24, 1983, Nurnberg, Germany.

[6] M. M. Zloof, Query-by-Example: A data base language, IBM Systems Journal 16(4) (1977) 324-343.

[7] J. C. Thomas and J. D. Gould, A psychological study of Query-by-Example, Proc. National Computer Conference (1975) 439-445.

[8] M. H. van Emden, Computation and deductive information retrieval, in: Formal Description of Programming Concepts (North-Holland, Amsterdam, 1978).

[9] S. J. Kaplan, Co-operative responses from a portable natural language query system, Artificial Intelligence 19 (1982) 165-187.

Appendix 1: A simple business data base

The examples in this paper make use of the following relations:

(i) A relation called parts with attributes (columns): pno (part number), pname (part name), colour and weight.

(ii) A relation called suppliers with attributes: sno (supplier number), sname (supplier name), status and city.

(iii) A relation called supplier_parts with attributes: sno (supplier number), pno (part number) and qty (quantity supplied).

(iv) A relation called supplier_balance with attributes: sno (supplier number) and amountowed.

(v) A relation called sales_people with attributes: salesno (sales number) and salesname.

(vi) A relation called product_parts with attributes: prodno (product number), pno (part number) and noreqd (number of parts required).

(vii) A relation called sales with attributes: salesno (sales number), prodno (product number) and qtysold (quantity sold).

Typical values of these relations are as follows:

parts	pno	pname	colour	weight
	1	nut	red	12
	2	bolt	green	17
	3	screw	blue	17
	4	screw	red	14
	5	cam	blue	12
	6	cog	red	19

Table 1.1 - The parts relation

suppliers	sno	sname	status	city
	1	smith	20	vienna
	2	jones	10	paris
	3	blake	30	paris
	4	clark	20	vienna
	5	adams	15	athens

Table 1.2 - The suppliers relation

supplier_parts	sno	pno	qty
	1	1	300
	1	2	200
	1	3	400
	1	4	200
	1	5	100
	1	6	100
	2	1	300
	2	2	400
	3	2	200
	4	2	200
	4	4	300
	4	5	400

Table 1.3 - The supplier_parts relation

supplier_balance	sno	amountowed
	1	100
	2	90
	3	0
	4	0
	5	145

Table 1.4 - The supplier_balance relation

sales_people	salesno	salesname
	1	flanagan
	2	ellis
	3	smith
	4	schafer

Table 1.5 - The sales_people relation

product_parts	prodno	pno	noreqd
	1027	1	350
	1023	1	200
	1028	1	100
	1033	3	275
	1040	4	435
	1072	5	555
	1045	2	315
	2001	6	125
	1067	5	111

Table 1.6 - The product_parts relation

sales	salesno	prodno	qtysold
	1	1023	100
	1	1027	45
	2	1028	40
	3	1033	150
	3	1040	75
	4	1072	20

Table 1.7 - The sales relation

Appendix 2: The link dictionary for the data base in Appendix 1.

The link dictionary for the data base in Appendix 1.

```
link(supplier_parts, supplier_balance, [sno]:[sno]).
link(suppliers, supplier_balance, [sno]:[sno]).
link(supplier_parts, product_parts, [pno]:[pno]).
link(parts, product_parts, [pno]:[pno]).
link(sales, product_parts, [prodno]:[prodno]).
link(sales_people, sales, [salesno]:[salesno]).
link(supplier_parts, suppliers, [sno]:[sno]).
link(supplier_parts, parts, [pno]:[pno]).

setofnodes(graph, [supplier_balance, product_parts, sales,
                   sales_people, suppliers, parts,
                   supplier_parts]).
```

The Prolog program for searching for a path linking any pair of relations in the data base:

```
                    ?- op(40, xfx, :).

                    /* declare ":" infix operator   */


clause 1            path(GRAPH, X, Y, PATH) <-
                    setofnodes(GRAPH, SET),
                    member(X, SET),
                    member(Y, SET),
                    walk(GRAPH, [X], Y, PATH).

clause 2    walk(GRAPH, [Y|L], Y, [Y|L]).
clause 3    walk(GRAPH, [X|L], Y, PATH) <-
                    (
                       link(X, Z, _);
                          link(Z, X, _)
                    ),
                    not(member(Z,L)),
                    walk(GRAPH, [Z,X|L], Y, PATH).

clause 4    member(X, [X|_]).
clause 5.   member(X, [_|Y]) <-
                    member(X, Y).
```

30. M.H.WILLIAMS and J.RIDDALL, The segregation of incompatible duties, The Computer Journal, vol. 29, no. 1, 9-11 (1986).

The Segregation of Incompatible Duties

M. H. WILLIAMS* AND J. RIDDALL
Computer Science Department, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ

One of the most important elements of an internal control system is the segregation of incompatible functions. Essential to this process is the task of identifying the sets of functions which are incompatible. Once these have been determined the access capabilities (both direct and indirect) of each individual or group in the organisation must be traced to ensure that no person has access to a set of incompatible functions. The logic programming language Prolog can be used to assist in this process.

Received January 1984

1. INTRODUCTION

Since the development of Prolog[1,2] as a practical programming language based on logic programming, it has been successfully used to solve problems in a growing range of different application areas (e.g. data bases,[3] expert systems,[4] natural language processing,[5] CAD modellers[6]). Recently it has been applied to problems in the area of security and control. In 1983 Carroll and Wu[7] reported an application in which it was used to track functional relationships through a complex data-processing system to ensure that security classifications are consistent.

This paper is concerned with a problem in the area of segregation controls. The problem involves tracing the access capabilities of individuals to ensure that no person has access to a set of incompatible functions. A system developed in Prolog can be used to assist in this process.

2. INTERNAL CONTROLS AND THE SEGREGATION OF DUTIES

One of the chief elements of an internal control system is the set of organisational controls which deal with the employee structure within the company. The most important of these is the segregation of incompatible functions.[8]

The aim of segregating incompatible functions is to prevent an employee from being allocated duties which enable him both to commit fraud and to conceal this fact. As an illustration of the problem suppose that an employee has charge of the approval of credit notes as well as the receipt of cheques. Such a person could approve a false credit note and use it to balance a debtor's account. Then when the actual payment is received into the company the employee will simply pocket it, leaving no trace of what has actually happened. Thus the approval of credit notes and the receipt of cheques are referred to as incompatible functions.

Many auditing manuals present lists of specific duties which are incompatible.[9] These include approval of (or access to) credit notes/receipt of cheques, preparation of sales invoice/receipt of cheques, etc.

Within any organisation some of these functions may be performed manually (e.g. custody of stock), some through a computer (e.g. trade creditors ledger, payment of cheques) and some not at all. In addition, each individual organisation may have its own particular incompatible functions which are either different from or variations of items in the above list.

To complicate the problem still further it is not always possible to adhere rigidly to any allocation of duties since illness and death on the one hand and natural growth and development on the other may necessitate dynamic reorganisation. Thus the problem of ensuring that the duties allocated to each employee will not enable him to commit fraud and to conceal this fact is not a simple one. Nevertheless, its importance has been underlined by a recent survey conducted by BIS Applied Systems which showed that many computer frauds have been successful because the victim companies failed to appreciate the need for proper segregation of duties.[10]

3. DETERMINATION OF INCOMPATIBLE FUNCTIONS

In order for management to set up appropriate segregation controls and for the auditor to verify that these controls are indeed adequate, the sets of incompatible functions need to be determined. In seeking a systematic approach to determining these sets, it was observed that each set of incompatible duties can be regarded as an action–reaction pair. To illustrate this, consider a simple example. Suppose that the basic functions and their interrelationships for the company under consideration are given in Fig. 1. Each box represents a particular function and the directed lines indicate the flow of information, documents, goods, etc.

To demonstrate how the network of functions and flows has been arranged, the area of Sales will be examined in detail. The event which normally initiates action within a company's Sales section is the receipt of an order for goods (box 1). This results in the production of a dispatch note (2) and the removal of the goods from the store (3). Acting on the dispatch note, an invoice is sent to the customer (4) taking into account any discounts available (5). A summary of invoices sent out is compiled within the trade debtors ledger (6).

Should a customer require regular statements, they will be compiled from information in accounts receivable and dispatched (7). If goods are returned to the company (8) they may be moved to stores where they will be dealt with (3). The customer will receive either a cheque (9) or a credit note (10). The credit note may be used by the customer as part payment for goods or may be returned

* Address for correspondence: Professor M. H. Williams, Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ.

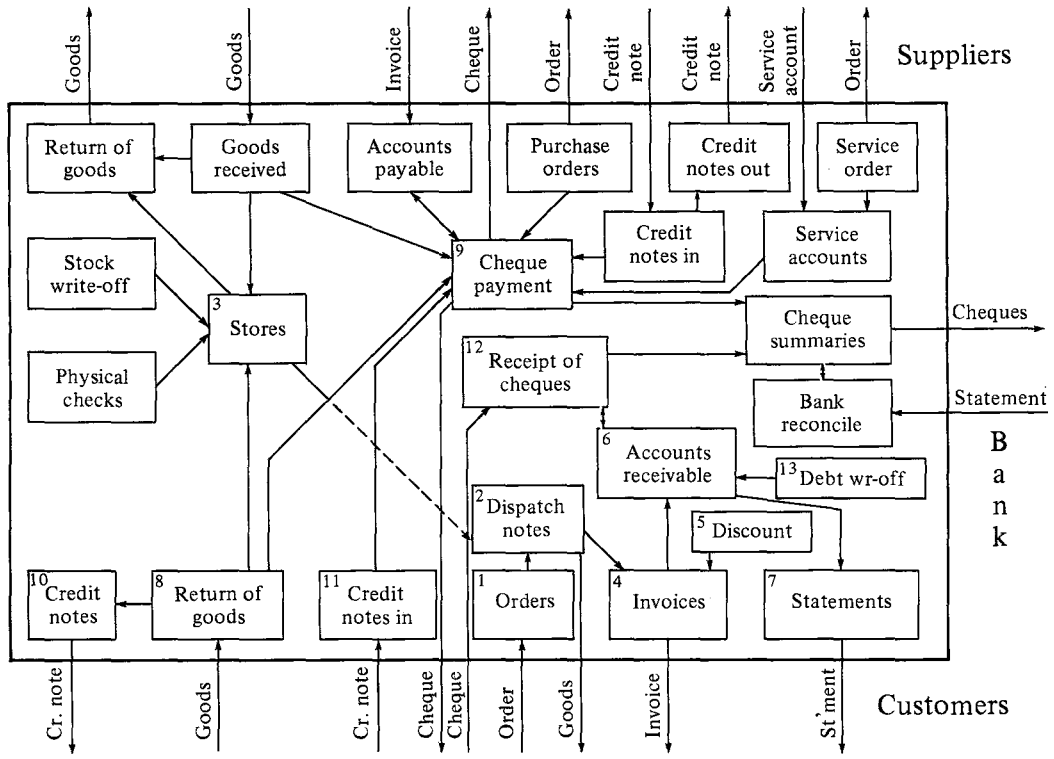


Figure 1. The basic functions and their interrelationships for a typical company.

intact to the company (11) in exchange for a cheque to the same value (9).

When payment for goods is received by the company (12), this is checked against the accounts receivable (6) to ensure that the correct amount has been received. If the customer refuses to pay for goods he has received then after a while the accounts receivable record of his debt may be removed, i.e. the bad debt is written off (13).

In attempting to relate the typical sets of incompatible functions as illustrated in the previous section with the functions in this diagram, each set can be regarded as corresponding to an action–reaction pair, e.g.

SALES
(1) Order (in)/goods and sales invoice (out).
(2) Sales invoice (out)/cheque (in).
(3) Statement (out)/cheque (in).
(4) Goods returned (in)/credit note (out).
(5) Returned credit note (in)/cheque (out).
(6) Goods returned (in)/cheque (out).

PURCHASES
(7) Purchase order (out)/goods and invoice (in).
(8) Invoice (in)/cheque (out).
(9) Services account (in)/cheque (out).
(10) Goods returned (out)/credit note (in).
(11) Credit note (out)/cheque (in).

BANK
(12) Cheque summary (out)/bank statement (in).

Two less obvious action–reaction pairs contained in Fig. 1 arise from the stock and debt write-off functions. These are
(13) Goods (out)/stock-write-off slip (in).
(14) Goods and invoice (out)/bad-debt-write-off (in).

From a simplistic point of view, if an employee's duties give him control over such an action–reaction pair, then he is able to short-circuit the pair to his advantage without leaving any trace. This is analogous to the basic concept that underlies double entry bookkeeping.

4. TRACING THE ACCESS CAPABILITIES

The discussion of the previous section was concerned with determining the incompatible functions and as such was not restricted to computer systems but applies equally to manual systems. This section considers the problem of tracing the access capabilities of each individual or group in the organisation to ensure that no person has access to a set of incompatible functions. In this section we concentrate specifically on computer systems.

The type of company which is of interest here is one which makes significant use of computing facilities. Such an organisation may have a large number of employees who have access to these facilities, either for the purposes of running application programs or for simple data entry or update. Each individual or group is assigned particular access rights which permit him to run specific programs or access particular information in files, etc.

The problem is complicated by the close interrelationships between programs and files. For example, although a person may not have direct access to a program which prints invoices, he may have access to one which enables him to update a file which is in turn used by the program which prints invoices. Thus although an individual may not have direct access to a set of incompatible functions, he may have indirect control over such a set.

A computer program (SEGREGATE) has been written in Prolog to assist an auditor in tracing the access capabilities (both direct and indirect) of each individual

in the organisation to determine whether or not the set of segregation controls for the given computer system is sufficient. The program was designed for a specific formulation of the problem used by auditors.

Within the computer system is a collection of computer programs which can be accessed by employees of the organisation. Each program has associated with it an access code, and each employee has a list of access keys (or codes to which he has access).

The program SEGREGATE asks the user (person responsible for the segregation controls) for the following information about the system.

(a) Incompatible functions: the names of incompatible functions.
(b) The code name and set of access keys for each employee.
(c) The names of the programs associated with functions which form part of incompatible function sets.
(d) For each program details of its actions in the form of fields/files which the program can update and any other fields/files on which such an update depends.
(6) The access code for each program.

Once this information has been gathered SEGREGATE searches through it to determine whether any employee has access to programs which either directly or through some chain effect enable him to control a set of incompatible functions. Any such employee is reported together with the incompatible functions to which he has access.

In its present form the program is not a general-purpose auditing tool since it has been designed to solve a particular formulation of the problem. More work is required both on the level of program description and on the range of different approaches to access rights before it could be regarded as such. Nevertheless, it has demonstrated the usefulness of this approach for solving this type of problem.

5. APPLICATION

Besides the use of these ideas in a special-purpose auditing program which can be run from time to time to enable management or an auditor to check the segregation of duties within a computer system, another important application area is that of intelligent database systems.

An intelligent database system is one which incorporates functions involving metalevel data (i.e. knowledge about the data in the database) which are used to protect the data in the database and to assist the user in various ways. The simplest and most common examples are security and integrity constraints.

If a database is used to control and record movements of items (stock, cheques, goods, services, etc.) into and out of a company it would be desirable if the segregation-of-incompatible-duties control function could be built into the intelligent functions of the database management system, so that whenever a change is made to the access pattern of the system (new users or relations added, existing ones deleted or changed), the metalevel data is updated and the system checked to ensure that reliance can still be placed on it. This will provide a continuous monitor of this auditing control function rather than the occasional snapshot provided by the special-purpose program described in the previous section.

If all changes to the database are made through a database query language such as Query-by-Example,[11] it may not be necessary to store information on individual programs in the system. In addition all information on the data which each user is entitled to access and the conditions under which he can perform various operations (e.g. existence of items elsewhere in the database) will be defined in the security constraints (authorisation statements) in the system. The only additional information which is required is the names of items associated with incompatible functions, and the relations and fields within these relations which are associated with these items. This can be handled by creating a special table for this purpose. The analysis which must be performed is similar to that outlined in the previous section.

Acknowledgements

The authors are grateful to Mr C. R. G. B. Rinaldi of the Department of Computer Science and Mr I. Whittaker of Arthur Young McClelland Moores & Co. for their advice and assistance on the work described here.

REFERENCES

1. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, Berlin (1981).
2. D. H. D. Warren, L. M. Pereira and F. C. N. Pereira, Prolog – the language and its implementation compared with Lisp, *Sigplan Notices* **12** (8), 109–115 (1977).
3. J. C. Neves, S. O. Anderson and M. H. Williams, A Prolog implementation of Query-by-Example, *Proc. 7th International Computing Symposium*, edited H. J. Schneider, pp. 318–332. B. G. Teubner, Stuttgart (1983).
4. K. L. Clark and F. G. McCabe, Prolog – a language for implementing expert systems. In *Machine Intelligence 10*, edited J. Hayes and D. Michie. Ellis and Horwood (1982).
5. F. C. N. Pereira and D. H. D. Warren, Definite clause grammars for language analysis – a survey of the formalism and a comparison with augmented transition networks, *Artificial Intelligence* **13**, 231–278 (1980).
6. J. Camacho Gonzalez, M. H. Williams and I. E. Aitchison, Evaluation of the effectiveness of Prolog for a CAD application, *IEEE Computer Graphics and Applications* **4**, 67–75 (1984).
7. J. M. Carroll and O.-L. Wu, Methodology for security analysis of data-processing systems, *Computers and Security* **2**, 24–34 (1983).
8. Arthur Young McClelland Moores & Co., Audit Manual, private communication.
9. R. J. Anderson, *External Audit* **1**. Pitman, Toronto (1977).
10. K. Wong, A revealing look at the criminal mind, *Computing*, p. 24 (19 May 1983).
11. M. M. Zloof, Query-by-Example: a data base language, *IBM Systems Journal* **16**, 324–343 (1977).

31. M.H.WILLIAMS, I.M.PATTISON and J.C.F.M.NEVES, Reorganization in a simple database system, Software Practice and Experience, vol. 16, no. 8, 719-729 (1986).

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 16(8), 719–729 (AUGUST 1986)

Reorganization in a Simple Database System

M. H. WILLIAMS, I. M. PATTISON AND J. C. NEVES†
Computer Science Department, Heriot-Watt University, Edinburgh EH1 2HJ, U.K.

SUMMARY

When a database is set up initially its structure is designed to match the expected pattern of usage. With time, changes to the database and to its pattern of usage may cause the structure to be less than ideal for the pattern of usage, and the database may need to be reorganized. A set of tools is described which will assist the database administrator in the tasks associated with reorganization. The tools have been implemented in Prolog and incorporated into a database accessed via the database language Query-by-Example.

KEY WORDS Database Prolog QBE Reorganization

INTRODUCTION

Prolog[1, 2] is a logic programming language based on Horn clauses, which was developed by Colmerauer[3] in 1972, and which has become important as a vehicle for artificial intelligence applications, such as natural language processing[4, 5] and expert systems.[6] Its importance has been given a considerable boost by its choice as the basis for the kernel languages in the Japanese fifth generation computer project.[7]

In the field of databases Prolog has been found to be a particularly appropriate language. Kowalski[8, 9] has demonstrated its suitability for the representation of relational databases and shown how it can extend the power of relational schemes by allowing the expression of general rules in the database. Van Emden[10] noted the close correspondence between the syntax of Prolog goals and the tabular database query language Query-by-Example (QBE) which was developed by Zloof,[11] an observation which has been confirmed by implementing a QBE interface to a database in Prolog.[12] In another study QBE, SQL and a relational algebra were combined into a composite query language package for access to a relational database.[13]

Not only is Prolog suitable for realizing the conventional aspects of database systems but it is also convenient for implementing extensions of various types. Neves and Williams[14] have shown how it can be used to produce a more 'co-operative' interface with the user, and Williams *et al.*[15] have used it to produce more consistent and powerful integrity and security mechanisms in QBE.

In the implementation of QBE used by Neves and Williams, tuples of a relational database are stored as Prolog clauses. Thus a tuple such as

suppliers	sno	sname	status	city
	5	smith	20	edinburgh

† Current address: Universidade do Minho, Largo do Paco, Braga, Portugal

0038–0644/86/080719–11$05.50 *Received 13 September 1984*
© 1986 by John Wiley & Sons, Ltd.

is stored as the clause

```
suppliers(5, smith, 20, edinburgh).
```

The system is described in detail in References 12 and 16, and is used as the basis for the present paper.

One important aspect of conventional database systems is the ability for the database administrator to reorganize the structure of the database should this be required. This paper considers various aspects of the problem of reorganizing databases and describes how these have been handled in the Prolog-based implementation of QBE.

ROLE OF THE DATABASE ADMINISTRATOR

With the exception of a database whose structure is frozen at the time of set-up and whose pattern of usage remains constant with time, it is generally recognized that from time to time some consideration will need to be given to possible restructuring of data in a database. Reasons for this include:

(a) The pattern of usage of a database may change with time. Functions believed to be of secondary importance or possibly not even considered at all when a database is created, may become more important with time. As a result the original division of information into relations may no longer be the most suitable one.
(b) As new applications are developed, additional information may need to be stored. This information may be added in the form of new relations so as to create as little disturbance as possible to existing applications. With time, as the newer applications become established, the structure of the data in the database may prove to be less than ideal.
(c) The division of relations into separate files enables more efficient access to individual relations. This must, however, be balanced against the requirement to join relations in order to satisfy some queries. As the database itself and its pattern of usage changes, regrouping of the information in different files may become desirable.

In order to assess the need for restructuring it is necessary to establish the pattern of usage of the database over a reasonable period of time. This may reveal that:

(a) the pattern changes little during the period of interest
(b) it fluctuates in some regular fashion which depends on season, demand or other external influence
(c) it varies in a random fashion
(d) it undergoes a definite change as a result possibly of additional information added to the database or of a change in the relative frequency of functions.

or some combination of these.

Whatever the case, the responsibility lies with the database administrator to monitor the pattern of usage and establish whether the structure of the database is best suited to the pattern of usage over any reasonable period of time. If the structure can be improved the database administrator is responsible for performing the restructuring.

RESTRUCTURING THE DATABASE

Different database management systems provide different tools for restructuring

according to the particular features of the system concerned. A comprehensive review of such tools is given by Sockut and Goldberg.[17]

However, one of the basic premises of database systems is that the users should be isolated from the physical organization of data in the database. Thus although some applications may be modified to take advantage of any restructuring, others may need to be protected from it. This means that any changes which may be made to the physical structure of the data should be able to be made without affecting the way any particular user sees the system.

This can be accomplished by the use of user views. A view[18] is a 'virtual relation' which is derived from the stored relations by appropriate database transformations. Although its normal use is to provide a logical subset of some base relations, it can equally be used to maintain a consistent interface to the user despite changes in the underlying physical structure of the data. Thus, if the database needs to be restructured, views can be created which make the restructured version appear as the original database to those users that require it.

In QBE user views are catered for by creating new tables using the VIEW command. For example, the specification

I.VIEW sp	sno	name	pno
	X	Y	Z

suppliers	sno	sname	status	city
	X	Y		

supplier-parts	sno	pno	qty
	X	Z	

will create the logical relation sp which maps into the physical tables suppliers and supplier-parts. Just as with any QBE query, conditions may be placed on any of the fields, thereby restricting the set of tuples which may be accessed via the view.

In order to implement this in Prolog a master directory is maintained which contains information about each relation in the database and the set of views which may be used to access these relations. At the beginning of a query session this master directory is loaded into the Prolog work space.

When a query is put to the system by a user, it is transformed into a Prolog question. For example the query

suppliers	sno	sname	status	city
	Y	p.X		edinburgh

supplier-parts	sno	pno	qty
	Y	1	

which will print the names of all suppliers living in Edinburgh who supply part number 1, will be transformed into the meta-level form

query([X])← [suppliers (Y,X,_,edinburgh), supplier-parts(Y,1,_)]

from which the final query is generated. In the process of this transformation each relation name will be looked up in the master directory to ensure that it represents a physical relation, and if it is found to represent a view a simple substitution is performed, replacing it by its definition. At the same time security checks are performed to ensure that the user has permission to access all of the fields concerned.

During the process of conversion the query is also optimized to remove any duplicated reference to the same relation which might have arisen as a consequence of two views mapping onto the same physical relation. For example, if

sup–1	sno	sname

and

sup–2	sno	status	city

are two views of the physical relation

suppliers	sno	sname	status	city

then the query

sup–1	sno	sname
	X	p.N

sup–2	sno	status	city
	X		london

will be replaced by

suppliers	sno	sname	status	city
	X	p.N		

suppliers	sno	sname	status	city
	X			london

and finally optimized to

suppliers	sno	sname	status	city
	X	p.N		london

To assist the database administrator in restructuring the database, a command restruct has been added to the system. When this command is given, it must be followed by the data to be restructured in the form of a query and the equivalent data in its restructured form.

For example, if the database administrator wishes to subdivide the relation suppliers, he may enter the restruct command followed by

suppliers	sno	sname	status	city	amountowed
	A	B	C	D	E

A special function key is used to indicate the end of the first set of data. This might be followed by

supl	sno	sname	status	city
	A	B	C	D

sup2	sno	amountowed
	A	E

On completion of the command, the system sets up two new relations, sup1 and sup2. It then reads each tuple in turn from the relation suppliers, sets up the corresponding tuples in the relations sup1 and sup2 and deletes the original tuple from suppliers. When this has been done, the relation suppliers will be empty and can be deleted. Finally a view called suppliers is created which maps into sup1 and sup2.

The restruct command can likewise be used to merge relations, to move attributes from one relation to another and other more complex reorganizations. In each case views are automatically created so that the users need not be aware of any change to the system.

MONITORING THE PATTERN OF USAGE

In order to make an assessment of the suitability of the database structure, summary statistics on the pattern of usage during the period under consideration are required. Typically it might be useful to have details such as

(a) What proportion of the accesses to each relation are simple queries (involving only the relation in question)?

(b) For each relation what other relations are joined to it and how frequently are these join operations performed?

and so on.

This immediately raises the question as to what level of information would be adequate for this purpose. At one extreme, if one simply collected statistics on the number of accesses to each relation, this would give an indication of the frequency of use of each relation, although it would be insufficient to make any judgement on the suitability of the database structure in general. At the other extreme one might store in its entirety every query that is put to the database system. However, this will be wasteful of storage space and could require a large amount of additional processing time to analyse the queries at a later stage.

One can distinguish two levels of detail in the statistics which would be useful for evaluating the performance of the database system, namely

(a) the structure of the queries in terms of the relations and attributes involved

(b) the values of the attributes used.

The first of these (the structure of queries) is the easier to deal with. In the second case, discovering patterns in the attribute values which are used in queries can require an enormous overhead in processing time and storage with no guarantee of success.

For instance, suppose that the tuples in a relation can be partitioned into two subsets, one of which is used frequently both for queries and for alterations and the other is used infrequently. An example of this might occur in a theatre reservation system in which a relation is used to store the number of seats sold at every performance (past and future) for the theatre for the current year. The tuples concerned with future performances may be accessed frequently (both queries and alterations), whereas those concerned with past performances are queried only rarely and never altered. The overall performance of such a system might be improved if this relation were divided into two separate relations dealing with past and future performances, respectively. Such a situation is unlikely to be recognized by any mechanical approach to query monitoring.

Since it is impractical to produce a system which will automatically gather sufficient statistics to deal with all possible eventualities, one solution is to provide two types of monitoring functions.

General monitoring functions

These functions are used to keep track of the frequency of all operations on the basis of their elementary structure. In the case of simple queries details of the relation name and the attributes involved are stored together with a count of the number of times that a query involving these attributes has been encountered during the period of observation. For example, if simple queries to the relation suppliers involving the three attributes, sname, status and city have been encountered on five occasions during the period of interest, this is recorded in a Prolog fact of the form

```
simple (suppliers,[sname,status,city],5).
```

In the case of queries involving joins, details of the relation names and attributes involved are stored together with a count of the number of times that each particular combination has been encountered, e.g.

```
join (suppliers,[sname,city],supplier-parts, [qty],27).
```

indicates that a join of the two relations suppliers and supplier-parts involving the additional attributes sname, city and qty has occurred 27 times.

If a query involves more than one join on different attributes, details of each join are recorded separately. By way of illustration suppose that a user wishes to print the names of parts supplied by either Smith or Jones. This query may be expressed as follows;

suppliers	sno	sname	status	city
	X	A		

supplier-parts	sno	pno	deptno	qty
	X	Y		

parts	pno	pname	colour	weight
	Y	p.N		

CONDITIONS
(A=jones) or (A=smith)

This query contains two joins involving suppliers and supplier-parts, and supplier-parts and parts, respectively. Each join is recorded separately in the join statistics.

If several relations are joined on the same attribute (using the same variable), e.g.

suppliers	sno	sname	status	city
	X	p.N		

supplier-parts	sno	pno	deptno	qty
	X	1		

supplier-bal	sno	amountowed
	X	p.Q

This is recorded as

```
multiplejoin ([suppliers,supplier-parts,supplier-bal],
[[sname],[pno],[amountowed]]).
```

Statistics on modifications (i.e., insertion, deletion or update operations) are recorded separately. In the case of a simple command such as

suppliers	sno	sname	status	city
i	2	wright	30	glasgow

if this were the first modification involving all the attributes to the relation suppliers, then this would be recorded in a Prolog fact of form

```
update(suppliers,[sno,sname,status,city],1).
```

More complex modifications involving joins are broken down into their constituent components and recorded accordingly.

Directed monitoring functions

As mentioned previously, any general system of monitoring which takes full account of the values of attributes used in queries and analyses these for patterns, is impractical. Nevertheless the database administrator may be aware of or suspect certain patterns in attribute values. In order to verify or disprove his suspicions a facility should be provided to enable the database administrator to monitor specific types of queries.

For this purpose an additional command m. (standing for 'monitor') has been added to the repertoire of QBE commands. It is used in a very similar way to an integrity constraint[15] except that where an integrity constraint is used to enforce a condition, the monitor command simply records statistics on it.

For example, suppose that the database administrator suspects that the relation suppliers can be subdivided in some way and wishes to verify this by monitoring all accesses to suppliers with status less than 30 which involve a join to the supplier-parts relation. This may be specified as follows:

suppliers	sno	sname	status	city
m.	X			Y<30

supplier-parts	sno	pno	qty
	X		

This is converted by the system into a monitor constraint and used to monitor all operations performed on these relations. In a way this is very similar to a trigger, as described in Reference 16.

ADVISING ON REORGANIZATION

Having determined the pattern of usage over a reasonable period of time, the database administrator must then decide whether the structure of the database is best suited to this pattern of usage or whether an alternative organization might improve the overall performance of the system. However, this is not a simple process.

Clearly it would be helpful if one could formalize what it is that a database administrator might look for and how he would use particular observations to determine the effectiveness of the structure of the database. If this knowledge could be built into the database system itself, the system might then monitor its performance perpetually and report to the database administrator at regular intervals.

Some simple examples of this might be:

(a) If one relation x is generally only accessed by joining it with some other relation y, and the two relations can be combined into a single one without any radical increase in the space required or the access times to individual fields of either relation, then it may be sensible to replace the two separate relations by a single one.

(b) If relation x is frequently involved in join operations with relation y in order to add field a of relation y to the fields of relation x, and field a of relation y is not used very frequently in any other queries, then it may improve overall performance if field a were transferred from relation y to relation x.

(c) If two relations are frequently used together in queries and these two relations are stored in separate files, it may be possible to rearrange the distribution of relations to files to bring these two relations together in the same file.

Unfortunately situations which arise in practice are seldom as simple and clear-cut as this. For example, one needs to take into consideration the effects of insertion, deletion and update operations as well as queries, the relative sizes of the relations involved and the relative frequency of access to the relation in question compared to the overall activity of the database. However, if this knowledge can be formalized, this is the type of problem for which expert systems or knowledge-based systems are well-suited.

In the system described here several expert system rules have been formulated. These have been built into a separate process called restructure. This is a free-standing process which may be executed at any time by the database administrator. To be effective this may be done after a set period of time or after a specific number of queries have been put to the system.

The process attempts to detect areas in the use of the database where a reorganization of certain relations would lead to a more efficient response to the majority of queries involving these relations. For example if a database contains two relations suppliers and supplier-bal and these relations are frequently required to be joined together, then a merger of the relations might lead to greater efficiency since join queries involving the two relations would become simple queries to the new merged relation.

In order for restructuring to be worth while it must effect a noticeable saving. Thus in searching for a possible restructure it is only necessary to inspect the more costly operations. To this end each database operation must be costed out. This is done in a fairly crude manner by restructure, and a nominal cost is calculated on the basis of the number of tuples in each relation, the type of operations performed and the number of occurrences of the particular database operation.

These calculations are performed on the statistics collected by the general monitoring process. Once this has been done, each operation which has a total cost above a given cost threshold is a candidate for restructuring. The expert rules are applied to each such operation to determine whether a restructuring of attributes would lead to improved performance.

If an alternative structure is suggested, the effect of this alternative structure is costed out against the existing pattern of usage. If a significant improvement in cost can be

achieved, the reorganization which produces the greatest saving will be reported to the database administrator.

CONCLUSION

Over a period of time the pattern of usage of a database may change and it may become desirable to restructure the database to improve performance. The responsibility for this decision normally rests with the database administrator.

The concept of a self-reorganizing or adaptive database system has been approached by Stocker[19] who presents a picture of what might be expected from a general system of this type. The present paper describes a much simpler set of tools designed to assist the database administrator in this task and implemented as a simple extension to a database system.

The system, which has been based on the query language Query-by-Example, maintains statistics on all database accesses on the basis of their general format. It also provides a facility to monitor queries of any form specified by the administrator.

When required, the administrator can execute the program *restructure* which processes the statistics and, by applying a knowledge-based approach, advises on possible reorganization. Several simple expert rules have been built into this program, although more are required and can be added as experience accumulates.

Finally, there is the controversial question of whether the system should perform the restructuring automatically or whether this should be done by the database administrator. In the system described here the responsibility for this remains in the hands of the administrator. However, a simple command has been added to QBE to enable him to make changes to the structure of the database without affecting the way in which users see the system.

A similar approach could be adopted for distributed database systems. In the case of a system in which the data is stored in a number of files on different machines, monitoring functions could be set up to monitor the traffic of data and a set of expert rules developed to advise on whether reorganization of the files between machines might lead to improvements in performance.

The system has been implemented in Prolog and is running under UNIX on an Orion microcomputer.

REFERENCES

1. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, Berlin, 1981.
2. D. H. D. Warren, L. M. Pereira and F. C. N. Pereira, 'PROLOG—the language and its implementation compared with LISP', *Proc. Symp. on AI and Programming Languages, SIGPLAN Notices,* **12**, (8), 109–115 (1977).
3. A. Colmerauer, H. Kanoui, R. Pasero and P. Roussel, 'Un Système de Communication Homme–machine en Français-Rapport'; Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy, 1972.
4. A. Comerauer, 'Metamorphosis grammars', in L. Bloc (ed.), *Natural language communication with computers,* Springer Verlag, Berlin, 1978.
5. F. C. N. Pereira and D. H. D. Warren, 'Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks', *Artificial Intelligence*, **13**, 231–278 (1980).
6. K. L. Clark and F. G. McCabe, 'PROLOG—a language for implementing expert systems', in J. Hayes and D. Michie (eds), *Machine Intelligence 10,* Ellis Horwood, Chichester, 1982.

7. T. Moto-oka *et al.* 'Challenge for knowledge information processing systems', in *Fifth Generation Computer Systems,* North-Holland, Amsterdam, 1982, pp. 3–89.
8. R. A. Kowalski, 'Logic for problem solving', *Memo, No. 75,* Department of Computational Logic, University of Edinburgh, Edinburgh, 1974.
9. R. A. Kowalski, 'Logic as a database language', *Internal Report*. Department of Computer Science, Imperial College, London, 1981.
10. M. H. van Emden, 'Computation and deductive information retrieval' in E. Neuhold (ed.) *Formal description of programming concepts,* North-Holland, Amsterdam, 1978.
11. M. M. Zloof, 'Query-by-Example: a database language', *IBM Systems Journal,* **16**, 324–343 (1977).
12. J. C. Neves, S. O. Anderson and M. H. Williams, 'A Prolog implementation of Query-by-Example', in H. J. Schneider (ed.), *Proc. 7th International Computing Symposium,* B. G. Teubner, Stuttgart, 1983, pp. 318–332.
13. D. Y. Li and F. G. Heath, 'ILEX: an intelligent relational database system', *Proc. ACM Conf. on Small Computer Systems,* San Diego, 7–9 December 1983.
14. J. C. Neves and M. H. Williams, 'Towards a co-operative database management system', *Proc. Logic Programming Workshop 83,* Universidade de Lisboa, Portugal, 1983, pp. 341–370.
15. M. H. Williams, J. C. Neves and S. O. Anderson, 'Security and integrity in logic data bases using Query-by-Example', *Proc. Logic Programming Workshop 83,* Universidade de Lisboa, Portugal, 1983, pp. 304–340.
16. J. C. Neves, 'The application of logic programming to data bases', *Ph. D. thesis,* Department of Computer Science, Heriot-Watt University, Edinburgh, 1983.
17. G. H. Sockut and R. P. Goldberg, 'Database reorganization—principles and practice', *ACM Computing Surveys,* **11**, 371–395 (1979).
18. W. Kim, 'Relational database systems', *ACM Computing Surveys,* **11**, 185–211 (1979).
19. P. M. Stocker, 'Self-organising database systems, *Report,* School of Computing Studies, University of East Anglia.

32. M.H.WILLIAMS, The design of a Prolog database machine, Proceedings of the First Workshop of the Special Interest Group in Knowledge Manipulation Engines, Alvey Directorate, L1-5 (1987).

THE DESIGN OF A PROLOG DATABASE MACHINE

M.H.Williams

Computer Science Department, Heriot-Watt University.

Although Prolog has been shown to be very suitable for implementing various aspects of database systems, current Prolog translators are not able to deal efficiently with the problem of retrieval from large sets of clauses. This paper describes briefly the status of a project whose aim is to develop a system involving special purpose hardware and software which will overcome this problem.

Address for correspondence:

Professor M.H.Williams,
Head of the Department of Computer Science,
Heriot-Watt University,
79 Grassmarket,.
Edinburgh EH1 2HJ

Tel.No. 031 225 6465 Ext.550

1. INTRODUCTION

The logic programming language Prolog[1] was developed by Colmerauer in 1972 as a tool for solving problems in artificial intelligence. Since then it has been shown to be well suited to handling problems in a variety of application areas[2-5]. In particular, in the area of databases Prolog has been shown to be very suitable for representing relational data[6], for realising relational query languages such as Query-by-Example (QBE)[7,8] and for implementing relational database management system concepts[9,10]. However, its importance for databases lies in its potential for creating more advanced database systems (deductive databases or knowledge bases) which handle rules as well as facts and which include an inference mechanism.

Unfortunately the chief drawback with using Prolog for database applications at the present time is that almost all currently available Prolog systems are incapable of dealing with large data sets of the size necessary for database applications. What are urgently needed are systems in which Prolog programs can operate on large sets of clauses with a reasonable degree of efficiency.

Different solutions to this problem have been proposed and there are various ways of classifying the architectures of such systems. Gallaire[11] has suggested that one can distinguish in the first instance between coupled systems in which both the database and the deductive component coexist and are linked together by some mechanism, and integrated systems where one component is integrated into the other. An integrated approach may start with a database system and add to it an inference mechanism to provide deductive capabilities, or it may start with a logic programming system and extend it by the addition of database storage and retrieval techniques.

The system described in this paper follows an integrated approach in which an existing Prolog system is modified and extended to make use of special purpose hardware so that the resulting system provides an efficient database capability from within Prolog.

2. OVERALL STRATEGY

The objective of this research project is to develop a system based on special purpose hardware which will provide a means of retrieving clauses efficiently from a large set of clauses on disk and to integrate this facility within a Prolog system. The system will be based on an ICL series 39 model 30 accessed via a DRS 300 microcomputer with several terminals. The version of Prolog which is being used is the Prolog-X system which runs on SUN3 microcomputer systems.

The system will be implemented as a back-end machine and as such will require some modifications to Prolog although these will be kept to a minimum. One of the features of the Prolog-X system is a module facility and this provides a convenient basis for some of the extensions which are necessary.

For the sake of efficiency clauses must be stored on disk in a format which is suitable for fast processing by the Prolog engine. To achieve this each module is compiled and stored in a separate file together with its symbol table. The system will distinguish between three types of files, viz.

(a) Small files, in which the number of clauses is sufficiently small that the whole file can be loaded as and when required and no special retrieval techniques need be employed.

(b) Medium files, in which the number of clauses is somewhat larger and for which only a subset will be required at any time in any application.

(c) Large files, in which the number of clauses is large and, in general, will not fit into main memory.

3. INITIAL PHASE: SMALL FILES

The first phase of the project was aimed at providing a facility for handling small files. This facility is an extension of the Prolog-X system in which compiled modules may be stored and dynamically loaded when required.

For this purpose an internal representation has been selected for the saved module on the basis of compactness and ability to load it quickly into main memory. The code has been written in such a way as to facilitate extension to object files of different types using the same interface. This has also necessitated other changes to the Prolog-X system.

The development of this facility is now complete and performance measurements are very encouraging. A database consisting of the mail-niftp section of the NRS database has been adapted into Prolog, giving rise to two procedures, one with 595 clauses of which 81 were rules (this corresponds to Application Relays), the other with 94 clauses of which 9 were rules (pss). Using the standard consult facility in Prolog-X, the time taken to load this module was 9 minutes 25 seconds; using the new dynamic loading facility, the load time was reduced to 6 seconds. The time taken to set up the save file was 9 seconds. Equally dramatic improvements were obtained from a test set of 10,000 unit clauses generated by a clause generator, although in general such a file would normally be classified as a medium sized one.

4. MEDIUM AND LARGE FILES

The lower bound for medium sized files, for which special techniques will be used to improve retrieval times, is expected to be in the region of 2000-5000 clauses in a single procedure. However, the decision as to whether a module should be regarded as dynamically loadable (small file) or simple disk based (medium file) will be left to the user to take.

The essential difference between medium and small sized files is the inclusion of a simple mechanism for handling the indexing facilities for retrieval of clauses. The technique to be used here is one of the adaptations of the superimposed code word technique. This is currently being implemented in software but will be realised in hardware by the end of the project.

Large files will employ an extension of the indexing facilities used for medium files. Experimentation on the best approach to be used is still under way.

The hardware implementation involves associative retrieval with on-the-fly processing of clauses. This will be based around the CAFS system on the ICL series 39 model 30.

References

1. W.F.Clocksin and C.S.Mellish, Programming in Prolog, Springer-Verlag (1981).

2. F.C.N.Pereira and D.H.D.Warren, Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks, Artificial Intelligence, Vol.13, No.3, pp.231-278 (1980).

3. K.L.Clark and F.G.McCabe, Prolog: A Language for Implementing Expert Systems, Internal Report, Dept.of Computing, Imperial College, London (1980).

4. Z.Markusz, Application of Prolog in Designing Many-Storied Dwelling Houses, Proc.Logic Programming Workshop, Debreceu, Hungary, pp.249-260 (1980).

5. J.Camacho Gonzalez, M.H.Williams and I.E.Aitchison, Evaluation of the Effectiveness of Prolog for a CAD Application, IEEE Computer Graphics and Applications, vol.4, No.3, pp.67-75 (1984).

6. R.A.Kowalski, Logic as a Database Language, Internal Report, Dept.of Computing, Imperial College, London (1981).

7. J.C.Neves, S.O.Anderson and M.H.Williams, A Prolog Implementation of Query-by-Example, Proc.7th International Computing Symposium, B.G.Teubner, Stuttgart, pp.318-332 (1983).

8. D.Y.Li and F.G.Heath, ILEX: An Intelligent Relational Database System, Proc.ACM Conference on Small Computer Systems (1983).

9. J.C.Neves and M.H.Williams, Towards a Co-operative Data Base Management System, Proc.Logic Programming Workshop, Albufeira, Portugal, pp.341-370 (1983).

10. M.H.Williams, J.C.Neves and S.O.Anderson, Security and Integrity in Logic Data Bases using Query-by-Example, Proc.Logic Programming Workshop, Albufeira, Portugal, pp.304-340 (1983).

11. H.Gallaire, Bridging the Gap between AI and Databases: Logic Approach, Proc.IFIP Conference on Knowledge and Data, Aldeia das Acoteias, Portugal, G1-G22 (1986).

33. M.H.WILLIAMS, G.CHEN, D.FERBRACHE, P.MASSEY, S.SALVINI, H.TAYLOR and K.F.WONG, Prolog and deductive databases, Knowledge-Based Systems, vol. 1, no. 3, 188-192 (1988).

Prolog and deductive databases

M H Williams, G Chen, D Ferbrache, P Massey, S Salvini, H Taylor and K F Wong

The logic programming language Prolog has been shown to be a very suitable language for implementing database concepts. However, current Prolog implementations are limited, and the database examples used have all consisted of relatively small sets of clauses. The reason for this is that existing Prolog implementations do not scale up to handle large databases. This paper describes a project whose aims are to develop the links between logic programming and databases. The prime aim is to develop a Prolog system which is capable of handling large sets of clauses. The second aim is to implement a deductive database management system in Prolog, while a third is to evaluate the effectiveness of parallel logic languages for implementing database applications.

Keywords: Prolog, Parlog, logic programming, deductive databases

Within the field of database systems several different approaches have been used, but of these the relational database has emerged as the most popular. The clean, uniform approach of the relational data model, and the power and simplicity of relational query languages have been the main reasons for the success of relational databases. These, in their turn, owe much to logic, as the relational calculus is a restricted form of first order predicate calculus.

With the growth of interest in knowledge processing, and in expert systems in particular, a need has been developing for more powerful database systems — systems which can store and retrieve knowledge; such systems are generally referred to as 'knowledge base management' systems, or 'expert database' systems. One of the formalisms used for representing knowledge is first order predicate calculus.

In view of the role of logic in underpinning relational databases and in describing knowledge, a natural successor to relational database systems, in the form of logic or deductive databases, has been proposed. Such systems are natural extensions of relational databases based on a more general form of first order logic. One particular form, known as a definite deductive database, has been studied in some detail[1]. This form of database is based on a firm theoretical foundation, providing the facility to handle rules as well as facts, and offers considerable promise as a successor to relational databases. The main focus of research at this stage is the development of efficient and safe query processing strategies.

Various approaches to achieving such systems are currently under investigation by different groups. This paper describes work being directed towards a two-level strategy which is being implemented as part of a programme of research funded by the Alvey initiative.

PROLOG AS AN INTERMEDIATE LANGUAGE

The suitability of Prolog for implementing various aspects of databases has been known for some while[2]. Not only is it ideal for analysing and translating queries into internal form, but it is also a natural candidate as the internal form into which queries are translated and then executed[3]. For this reason it was selected as the intermediate language for this project.

However, although Prolog has proved its suitability for realizing databases on a smaller scale, current Prolog implementations suffer from the drawback that they do not scale up. In order to assess the performance of current systems in this regard, a set of benchmarks has been drawn up which are specifically geared to database applications[4]. Initially, a fairly complex set of tests was devized, but in the end this had to be reduced to a more limited set because of the poor responses recorded.

When the benchmarks were applied to different Prolog systems, their shortcomings were clearly revealed. In a study involving three Prolog systems, C-Prolog[5], Prolog-X[6] and NIP[7] on a 4Mbyte Sun3 system the following conclusions were reached:

- Given a large set of simple ground facts, each with three arguments, all three systems were unable to cope when the number of clauses reached some limit between 30K and 70K. In each case, the total space required by the clauses and the associated symbol table exceeded the available space at this level.
- The time taken by each system to compile 30K to 70K clauses from source into their respective internal forms is considerable.
- The time taken by each system to load 30K to 70K

Computer Science Department, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ, UK

0950–7051/88/030188–05 $03.00 © 1988 Butterworth & Co (Publishers) Ltd

compiled clauses from a file on disc into main memory is significant.

Since a real database might be expected to handle millions of clauses, the limitations of current Prolog systems are unrealistic. To handle a million clauses by simply extending the main memory would be impractical, since the amount of memory required would be excessive.

If a database is to be accessible to more than one user, and may be updated by any of them, then it cannot be retained in the main memory. In this case the time taken to load it into the main memory for each user becomes an intolerable overhead. For these two reasons one needs to retain the database on disc, and load only those clauses which satisfy the query or subquery. Because compilation time is high, the clauses need to be stored in compiled form, and a facility to perform incremental updates is essential.

OVERALL ARCHITECTURE

There are various possible ways in which the power of logic programming languages and the storage capabilities of database systems may be brought together. The simplest of these is the idea of a loosely coupled system in which a Prolog compiler or interpreter is linked to a conventional database system. This approach has been followed by various people, e.g. Ceri *et al.*[8], but generally is very inefficient and clumsy.

A more efficient alternative is to connect a Prolog translator to a database system using a tight coupling. In this case one or both of the two components will be modified to effect a direct linkage. This approach has been followed in the EDUCE system developed at ECRC[9]. A survey of different approaches is given in Reference 10.

Although the idea of coupling two existing systems together has advantages in terms of rapid development, this is not a long term solution to the problem of deductive databases. First, the database system is not geared to storing and retrieving clauses; second, the Prolog system will still be limited in terms of the size of the set of clauses it can manipulate; and third, the interface to the system is the language Prolog itself, which is not suitable as a query language for inexperienced users.

The strategy adopted in the current project is a two level approach in which different aspects of the problem are tackled at different levels. The project has three prime objectives:

1 To develop a Prolog system based on special purpose hardware which is capable of handling large data sets efficiently. The system being developed, referred to as the Prolog Database Machine (PDBM), extends conventional Prolog with efficient clause retrieval from large data sets stored on disc.
2 To implement a Deductive Database Management System (DDBMS) in Prolog which will provide a simple query language by which a user can access clauses in the database, and facilities to assist the user in the management of his data.
3 To evaluate the effectiveness of parallel logic programming languages for implementing deductive databases. To this end, a compiler for the language Parlog will be linked to the special purpose hardware outlined in (1) and used to implement a database application.

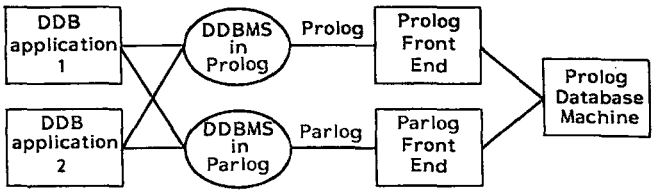


Figure 1. Overall system structure

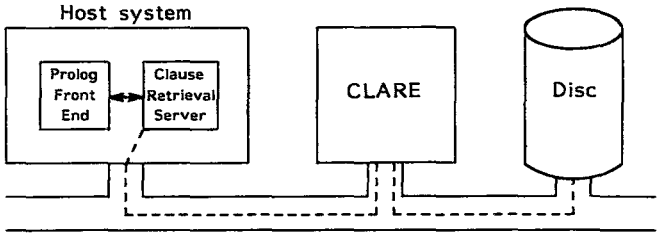


Figure 2. Prolog Database Machine (PDBM) architecture

To demonstrate the DDBMS executing on the PDBM various applications are being considered, and two in particular are being implemented. Thus, the overall structure of the system is as illustrated in Figure 1. The version of Prolog chosen for this purpose is Prolog-X. It was originally developed by Clocksin as an abstract machine code compiler and emulator with a view to microcoding the instruction set on a suitable host[6]. The PDBM is based on a C version of the Prolog–X system, and this is currently running on Sun3 systems.

THE PDBM

The Prolog database machine is basically a Prolog system which has been extended to handle large sets of clauses. The version of Prolog on which it is based has a module facility which enables clauses to be grouped together into modules. If a module is small in size, the whole module can be loaded into the main memory and used directly; if it is large, it will be disc resident, and clauses will be taken into the main memory as and when they are needed.

Thus, the Prolog system has been extended as detailed below. The front end of the system executes Prolog programs. Provided that the modules it requires are in the main memory, the front end operates as a normal Prolog system. If it requires access to a disc resident module, it invokes the Clause Retrieval Server (CRS). This is a separate sub-system which is responsible for communicating with and controlling some special purpose hardware known as the Clause Retrieval Engine (CLARE), an associative retrieval engine for clauses. The CRS is responsible for setting up and sending the appropriate parameters to CLARE, and receiving the results returned. The architecture of the system is illustrated in Figure 2.

As depicted in Figure 2, CLARE is a special purpose engine which links to the bus connecting the processor and its disc(s). When a clause or a set of clauses is to be retrieved from the disc, the appropriate set of data is passed to CLARE. It then sifts through the clauses as fast as they are delivered by the disc and passes any matching clause or clauses to the processor.

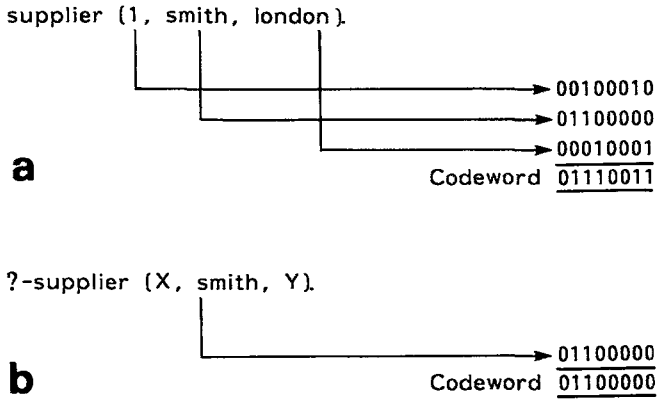


Figure 3. Generation of a simple codeword for (a) a fact, (b) a query

The clause retrieval engine itself is based on a two stage filtering process. The first stage makes use of superimposed codeword indexing[11]. This is a form of hashing, in which a special hashing function is applied to each constant argument of a clause and the resulting bit strings are combined together by a logical 'or' operation to form a codeword (see Figure 3). When a query is issued, it is subjected to the same process to produce a corresponding codeword. The query codeword is then compared with the codeword for each clause in turn following a special matching procedure in order to determine possible hits.

Any possible hits obtained by the first phase are passed to the second phase of the clause retrieval engine. This is a partial unification filter, which performs most of the checks carried out by a software unification routine. It was decided not to extend this to full unification checking, since the part that has been omitted is a time consuming process which accounts for only a very small proportion of queries. Realization of full checking would impede on–the–fly clause retrieval and complicate the hardware considerably. The clauses which satisfy this second phase of the engine are passed back to the CRS.

On receiving clauses from CLARE, the CRS returns these results to the Prolog front end where they are subjected to the normal unification process by software. An additional level of clause management is incorporated into the CRS to make the system simple to use and nearly transparent to the user.

Since Prolog–X supports the incremental compilation of clauses, it is possible to store clauses in compiled form on disc. This avoids the overhead of compiling and decompiling clauses whenever they are transferred into and out of main memory.

THE DDBMS

The deductive database management system fits hand in glove with the Prolog Database Machine. On the one hand the DDBMS needs the PDBM in order to provide adequate performance for database applications; on the other hand, the DDBMS provides a useful yardstick by which the performance of the PDBM can be judged against other Prolog systems.

The DDBMS is a natural extension of a relational database system which handles both facts and rules. A fundamental decision in the design of the DDBMS is the form of clauses to be supported by the system. At one extreme one has full clausal form, in which each clause in the database may have the form

$$P_1 \vee \ldots \vee P_n \leftarrow Q_1 \wedge \ldots \wedge Q_m \qquad n,m \geqslant 0.$$

Full clausal form has the expressive power of first order predicate calculus, but is impossible to implement efficiently and leads to problems with regard to satisfying the Closed World Assumption for negative information. At the other extreme, one has Horn clauses, which have the form

$$P \leftarrow Q_1 \wedge \ldots \wedge Q_m \qquad m \geqslant 0$$

Horn clause resolution can be implemented efficiently, and for Horn clauses the Closed World Assumption is directly equivalent to negation as failure. However, the expressive power of Horn clauses is limited.

Since work is progressing on finding suitable intermediate representations between Horn clause and full clausal form[12], it was decided to proceed with the design of a query language which can handle full clausal form. Restrictions can then be placed on the form of clauses permitted by any particular implementation depending on the internal representation used.

A DDBMS based on this assumption is currently under development. In particular, it will provide the following facilities:

- The query language on which it is based is an extension of SQL, which caters for clauses as well as tuples. In order to achieve this, instead of treating the tuple as the basic unit of the database (as is the case for relational databases), the basic unit is taken to be the clause. With this extension the tuple variable in SQL is extended to a clause variable which represents each clause in the database. The resulting query language is considerably simpler to use than the programming language Prolog, and enables a user with relatively little training to query a database containing both facts and rules. In addition to facilities to query the database, the system also provides facilities for the user to define and manipulate data and rules. These too are simple to use.
- The query evaluation sub-system must support different external models of the data. Thus, in order to enable different knowledge-based system applications to store their knowledge in the deductive database, the query evaluation system must provide the necessary transformations between the external and internal formats. By introducing the notion of a clause variable into the language, the user can access and manipulate the knowledge stored in the deductive database in a unified manner, and at the same time cope with the knowledge from different models. It can also deal with the situation where a database user wishes to treat rules as representing implicit data, and to use them to deduce the data implied by the database, whereas a database administrator may wish to treat them as a set of first order sentences from which some structural properties or integrity constraints might be derived.
- The initial implementation of the query evaluation sub-system is based on the use of Horn clauses as the means of internally representing knowledge. However, a more general internal representation is currently being considered, and it is hoped that this will

be incorporated into the final system. This does mean that, initially, all clauses which are stored as objects in the database must be converted to Horn clause form.

- A facility to handle general integrity constraints is being provided whereby the user may specify and enforce conditions imposed upon facts as well as rules. This facility can be used by other knowledge-based systems to specify and check the integrity of the knowledge base. Integrity constraints are also expressed as clauses, but are not restricted to Horn clauses. Thus they may have the general form

$$P_1 \vee \ldots \vee P_n \leftarrow Q_1 \wedge \ldots \wedge Q_m$$
$$\text{or } \leftarrow Q_1 \wedge \ldots \wedge Q_m \wedge \neg P_1 \wedge \ldots \wedge \neg P_n$$

In addition to normal static integrity constraints, provision has also been made for transition constraints[13].

APPLICATIONS

In order to demonstrate the usefulness of a database system which can store, retrieve and manipulate rules as well as facts, two specific applications are being developed. The first involves adapting several different expert system shells so that for each shell, instead of storing the knowledge in its own internal knowledge base, it stores it in the deductive database. Each shell will then be able to fetch and store rules via the query language.

The rationale behind this application is that, as the number of expert systems in an organization grows, the amount of common knowledge used by the different systems will increase. For example, in a commercial environment, common business practice may underlie a number of different expert systems; in an engineering firm, knowledge of good design practice, safety rules and basic engineering principles may be common to a number of systems; and so on. As the number of systems proliferates, and more and more knowledge is duplicated, there will be growing pressure to structure the knowledge into local and global knowledge, and to store knowledge rules in such a way that some sets of rules may be shared by two or more expert systems, whereas others are private to a specific expert system.

The second application involves the storage of a database of medical records, together with various types of rules. This is then used by an automatic learning system to extract and formulate knowledge from the raw data and store this knowledge back in the same database.

PARLOG

The parallel logic programming language Parlog is executed by a committed choice resolution strategy over guarded Horn clauses that makes it unsuitable for obtaining more than one solution to a database query. Accordingly, Parlog has been defined to have two Prolog-like extensions to handle all-solution searches over a Horn clause database stored separately from the executed Parlog program[14].

This project is developing the first serious implementation of these all-solutions constructors, along with a clause update facility that will be linked via the Clause Retrieval Server to the PDBM. A new sequential implementation of Parlog based upon a kernel guarded Horn clause emulator, will incorporate these database search engines into a full Parlog system with a Prolog–style interface. On top of this implementation a DDBMS will combine the distinctive systems programming virtues of Parlog with this facility for efficient database retrievals and updates, enabling it to be compared with a Prolog DDBMS and enabling their relative abilities for handling database applications to be assessed.

CONCLUSION

The project described in this paper has various objectives. First, it aims to produce a Prolog system which can deal with large sets of clauses efficiently: this is needed urgently as a vehicle for prototyping database applications, although many other applications will require large clause sets in the future too. It is too early to provide performance figures for the final system, but some preliminary results for the PDBM obtained using an ICL 3930 are encouraging. It must be stressed that the PDBM is not a database management system, but a Prolog system extended to deal with large sets of clauses and coupled to special hardware to enable this to be done efficiently.

The second objective of the project is to implement a deductive database management system with a suitably powerful yet simple query language. Although the initial implementation is based on Horn clauses, a more general internal representation is being studied. The resulting system will be similar to a relational database management system, except that it will have the capability of dealing with rules as well as facts. Although Prolog is being used as the language for developing the prototype, it could clearly be re-engineered in a language such as C if it proves successful.

To demonstrate the usefulness of such a system, two particular applications are being developed:

1 The connection of several different expert systems to the deductive database management system in order to store the knowledge of the expert systems in a central knowledge base.
2 The use of the deductive database management system by an automatic learning program to store both raw data and processed rules.

Finally, the potential for implementing database systems in the parallel logic language Parlog will be explored.

ACKNOWLEDGEMENTS

The authors wish to acknowledge the support received from ICL, and from the Science and Engineering Research Council under the Alvey initiative, who are currently funding the project.

REFERENCES

1 **Gallaire, H, Minker, J and Nicolas, J–M**, 'Logic and databases: a deductive approach' *ACM Comput. Surveys* Vol 16 No 2 (June 1984) pp 153–185
2 **Kowalski, R A** *Logic as a Database Language* Internal Report, Department of Computer Science, Imperial College, London, UK (1981)
3 **Neves, J C, Anderson S O and Williams, M H** 'A Prolog implementation of query-by-example' *Proc.*

7th Int. Comput. Symp. B.G. Teubner, Stuttgart, FRG (1983) pp 318–332

4 **Williams, M H, Massey, P A, Crammond, J A** *Benchmarking Prolog for Database Applications* Internal Report, Computer Science Department, Heriot–Watt University, Edinburgh, UK (1987)

5 **Warren, D, Bowen, D, Byrd, L and Pereira, L** *C-Prolog User's Manual* 1.4d.edai (September 1985)

6 **Clocksin, W F** 'Design and simulation of a sequential Prolog machine' *New Generat. Comput.* Vol 3 (1985) pp 101–120

7 **Bowen, D L, Byrd, L, Chung, P W H, Pereira, F C N, Pereira, L M, Rae, R and Warren, D H D** *Edinburgh Prolog (The New Implementation) User's Manual*, version 1.5, AIAI, Edinburgh University, (1st June, 1987)

8 **Ceri, S, Gottlob, G, and Wiederhold, G** 'Interfacing relational databases and Prolog efficiently' *Proc. 1st Int. Conf. on Expert Database Systems* Charleston, USA (1986)

9 **Bocca, J** 'EDUCE a marriage of convenience: Prolog and a relational DBMS' *Proc. 3rd Symp. of Logic Programming* Salt Lake City, Utah, USA (1983)

10 **Gallaire, H** 'Logical data bases vs deductive data bases' *Proc. Logic Programming Workshop* Universidade Nova de Lisboa, Portugal (1983) pp 608–622

11 **Ramamohanarao, K, Shepherd, J** 'A Superimposed Codeword Indexing Scheme for Very Large Prolog Databases' *Proc. 3rd Int. Conf. on Logic Programming* London, UK (1986) pp 569–576

12 **Chisholm, P, Chen, G, Ferbrache, D, Thanisch, P and Williams, M H** 'Coping with indefinite and negative data in deductive databases: a survey' *Data and Knowledge Engineering* (to appear)

13 **Gaebler, A G and Williams, M H** *An Integrated System of Static and Transition Constraints for a Deductive Database* Internal Report, Computer Science Department, Heriot-Watt University, Edinburgh, UK (1987)

14 **Gregory, S** *Parallel Logic Programming in Parlog — The language and its Implementation* International Series in Logic Programming, Addison-Wesley, London, UK (February 1987)

34. H.TAYLOR and M.H.WILLIAMS, Applying coupled resolution engines to knowledge bases, Proceedings of the UK IT 1990 Conference, IEE/BCS, Southampton, 379-386 (1990).

APPLYING COUPLED RESOLUTION ENGINES TO KNOWLEDGE BASES

H. Taylor & M.H. Williams

Heriot-Watt University, UK

INTRODUCTION

As part of Alvey project IKBS 90, a software architecture has been developed that couples a concurrent logic programming or CLP engine with a multi-threaded Prolog engine. Concurrency among mutually invoking Prolog and CLP computations is sustained partly through and-parallel execution of CLP computations on the CLP engine, and partly by executing multiple Prolog processes and a CLP engine under Unix on a multi-processor or a uni-processor. Communication among the multiple computations is realised by message passing, enabling the architecture to be multi-processed with a low degree of contention for shared memory. The coupled architecture supports multi-user knowledge based systems without compromising on the expressive power and performance of Prolog and CLP languages, and without diminishing the capacity for multi-processing at a task apposite grain of parallelism.

This approach has three advantages over an earlier proposal by Clark and Gregory (2) for a hybrid system for executing Prolog and the CLP language Parlog together. The proposal

- facilitates efficient multi-processing by not coupling the engines using shared memory
- keeps the process interpretation of CLP execution intact by not letting messages passed by variable bindings be revocable
- retains efficiency in the CLP engine by not adding the complex machinery for sustaining chronological backtracking to it.

Several interfaces between Prolog and CLP computations are defined, and programs using these interfaces are given that show how this approach to coupling resolution engines can effectively combine Prolog's ability to handle deduction over knowledge bases with a CLP language's ability to program concurrent systems. The combination of these two capabilities allows simple and expressive programming of multi-user knowledge based systems in logic.

A COUPLED ARCHITECTURE

CLP languages are apt for programming asynchronous and concurrent system interactions. Programs are represented by guarded definite clauses, resolution over them is done concurrently and involves committed clause choices, and variable bindings are irrevocably made. On the other hand Prolog is apt at supporting exhaustive solution searches of stratified knowledge bases. Programs are represented by definite clauses, resolution over them is sequential and exhibits don't know non-determinism, and bindings to variables are made on a trial basis in the search for the right satisfier. The difference of approach to the revocability of bindings to variables, to commitment in clause choice, and to the capacity for parallel execution makes for quite different styles of implementation and language.

Each language excels where the other language has marked limitations. CLP languages are poor at supporting exhaustive search, and Prolog is poor at programming concurrent systems. However, both CLP-like systems programming capabilities and Prolog-like exhaustive solution searches of knowledge bases are required for fully concurrent multi-user knowledge based systems. The coupled architecture aims to reconcile their differences in a fully concurrent

scheme for harnessing these resolution engines to each other. This would enable a multi-user knowledge based management system to be programmed in the CLP language, which could invoke concurrently multiple exhaustive searches over a knowledge base represented in Prolog, and thereby realise a multi-user knowledge based system in logic.

The coupled architecture consists of

- CLP language engine
- Prolog engine
- Unix framework

The CLP engine supports the Parlog control meta-call, allowing different CLP computations to execute within separate meta-calls with a fine grain of concurrency. The CLP engine may be realised by a single Unix process or by several tightly coupled Unix-like processes sharing memory.

The Prolog engine should be WAM based with extensions for escape exception and delayed goal handling. On a uni-processor it should be able to execute in a multi-threaded fashion, using lightweight SUN OS 4.0 or equivalent processes that access a common symbol table and code area for execution over a shared database, or using separate Unix processes for execution over non-shared databases.

Mutual invocations of each language through transient and boundary communicating styles of interface allow CLP and Prolog processes to invoke each other synchronously and concurrently. The whole configuration of Unix processes supporting logic programming computations can be multi-processed or executed on a single processor.

A prototype of much of the architecture has been developed in C to run on Sun workstations. The CLP engine executes full GHC and Parlog, can invoke multiple Prolog-X processes, and uses Prolog-X's interface to a Prolog Database Machine, Massey et al (3), for resolution over large scale clause databases. Alvey project IKBS 37 produced the Prolog Database Machine. It consists of a software server using various indexing strategies and a VME board extension to a Sun in order to speed retrieval by head unification, of large numbers of clauses stored on disc, Wong and Williams (8).

Multi-user knowledge based applications can be developed in the high level logic programming facilities of the coupled system, and the interface to the Prolog Database Machine enables the range over which the system can deduce to be extended to very large knowledge bases.

A COUPLED PROLOG-CLP SYSTEM

In the coupled architecture four interfaces connect Prolog and CLP computations. The environments of each computation are kept separate. All communication across interfaces is irrevocable, and can be realised by copying across the computation's current version of the goal. If the copies cannot unify, the invoked computation fails with normal goal failure consequences for the invoking environment. A high level scheme for implementing these interfaces is developed in [5]. Two forms of invocation, that cannot be resatisfied, allow Prolog to call a CLP computation, namely

call to CLP relation

- communication at boundaries
- synchronous

call to *clp(Goal)*

- communication transiently
- asynchronous

Goals in Prolog, that are not defined in the database and are not calls on primitives, invoke the CLP engine synchronously, passing values only at the start and end of invocation without any concurrency between the calling computation and the

computation that is called. Input matching suspension on invocation takes place if needed. Calls in Prolog to *clp/1* invoke the CLP engine asynchronously. Argument values may be passed transiently during invocation subject to the following restriction on shared variables

No Complex Term CLP Bindings Ban

> CLP execution may not bind variables shared with Prolog to complex terms during execution.

A CLP computation is only allowed to bind shared variables to simple terms when executing on behalf of a *clp/1* call. The interface is responsible for detecting the attempt to communicate to Prolog the binding of a shared variable to a complex term. This restriction on transient communication is essential because there is nowhere appropriate in a canonical Prolog engine to store transient bindings. *clp/1* goals in Prolog execute concurrently with clause body sibling goals. The parent goal of the spawned call to the CLP engine is not be able to exit until the *clp/1* call has succeeded or failed. When a *clp/1* goal fails, Prolog backtracks to resatisfy the parent goal. The functionality provided by *clp/1* goals is like the deterministic *::/2* fork primitive proposed in Clark and Gregory (2), but does not introduce a new conjunction operator.

Each call to a CLP computation is over the same guarded clause database of a single CLP engine. It executes in its own meta-call environment, and can only be linked to the environment of another CLP meta-call via a common Prolog computation.

CLP computations have two interfaces for calling Prolog, viz

call to Prolog relation

- asynchronous
- communication at boundaries

call to *prolog(Goals)*

- asynchronous
- communication transiently

Both interfaces allow the CLP computation to execute concurrently with the Prolog engine. CLP goals that are not calls on primitives and are not defined by a guarded clause relation, invoke the Prolog engine. The values of the invocation are passed at the boundaries of invocation. No suspension on invocation takes place. Calls from within a CLP computation to *prolog/1* invoke Prolog asynchronously in a fully concurrent fashion. Input and output bindings are communicated transiently except that the No Complex Term CLP Bindings Ban applies to the CLP computation executing concurrently with *prolog(Goals)*. The interface is responsible for detecting the attempt to communicate to Prolog shared variables bound to complex terms by the CLP computation. The call to *prolog/1* suspends on its argument if it is unbound.

To allow successive calls to be made to the same Prolog database state and to allow databases to be shared or not shared among concurrently executing Prolog computations, the following rule is observed

One Database per CLP Meta-call rule

> Each CLP meta-call is associated with a distinct persistent Prolog database.

This simple rule can be implemented on a uni-processor realisation of the coupled architecture by associating one Unix process executing Prolog with each CLP meta-call. Only control meta-calls like Parlog's *call/3* that recursively invoke the CLP engine and not simple meta-calls like *call/1* count for this purpose. The process is created by the first Prolog invocation within the meta-call, and although separate execution threads (lightweight processes under Sun OS 4.0) may be invoked over the one database state within the one Unix

process, only one process with one database state persists and can be accessed from the meta-call until the meta-call ends.

CLP ENGINE APPLICATIONS

The prototype of the CLP engine uses the medium of a lingua franca to support Parlog and full GHC. The lingua franca supports its own programming style as well as those of Parlog and GHC. Both full GHC and Parlog can be translated to the lingua franca and back again, Taylor (6). The lingua franca is a modeless, purely and-parallel, committed choice language, that uses input matching on head arguments, sequences head argument matching with guard evaluation and supports Parlog-style sequential search. It embodies extensions beyond both GHC and Parlog to support various forms of unification that cannot be supported in either language, and to allow Prolog style management by side effects of its clause database. A major difference is that lingua franca primitives can be multi-moded and output to arguments of primitives is always done by atomic unification with no bindings in the event of failure. A meta-interpreter of the lingua franca invoked by the goal *meta_clp(Goals)* is

```
meta_clp((A,B)):- true |
                  meta_clp(A),
                  meta_clp(B).
meta_clp(A)     :- primitive(A) |
                  call(A,yes);
meta_clp(A)     :- true |
                  clauses(A,Cls),
                  reduce(A,Cls,B),
                  meta_clp(B).


match(G,H,Guard) :- H <= G |
                    call(Guard,yes).
```

```
reduce(G,[(H:-Gd|B)|Cls],B1) :-
                  match(G,H,Gd) |
                  B = B1.
reduce(G,[(_:-_|_)|Cls],B1) :-
                  reduce(G,Cls,B) |
                  B = B1.
reduce(G,[((H:-Gd|B);_)|Cls],B1) :-
                  match(G,H,Gd) |
                  B = B1;
reduce(G,[(_;C)|Cls],B1) :-
                  reduce(G,[C|Cls],B) |
                  B = B1.
```

The first clause of *meta_clp/1* handles conjunctions, which are always executed in and-parallel. The second clause uses *primitive/1* to recognise primitives and evaluates them with the meta-call *call/2*, unifying its second argument to *yes* or *no* on the meta-call's success or failure. The third clause of *meta_clp/1* handles resolution. Clauses retrieved in the second list argument of *clauses/2* define the relation of the first argument goal, and have their selectability tested by *match/3*. The first clause of *reduce/3* examines whether the leading clause can reduce the goal. The second clause searches the next clause in parallel. The third clause examines the last clause in a group of clauses to be searched in parallel, and the fourth clause continues clause search after all previous clauses have been been found unable to reduce the goal. The clause for *match/3* ensures that head matching is performed before the guard is satisfied. The primitive <=/2 performs one-way unification, ensuring that unifying its two arguments does not bind or share right hand argument variables. The conciseness of the meta-interpreter demonstrates the lingua franca's expressive power.

A simple interface for a Prolog engine *wam/2* can be realised in the CLP engine

```
wam(yes,F)  :- write('yes\n|?-',F) |
               read(Goals,File),
               solve(Goals,R),
               wam(R,File).
wam(no,F)   :- write('no\n|?-',F) |
               read(Goals,File),
               solve(Goals,R),
               wam(R,File).

solve(G,R) :- nonvar(G) | horn(G,R).
```

The primitive *read(G,File)* unifies G with the result of reading a term asynchronously from *File*. The call to *solve/2* suspends until its first argument is bound and then uses boundary communication to invoke a Prolog relation *horn/2*

```
horn(G,yes) :- call(G).
horn(G,no).
```

Several instances of Prolog can be run concurrently on multiple terminals by opening these terminals as files via the CLP goal *open(Terminal)*, and executing in and-parallel the goal *wam(yes,Terminal)* for each *Terminal*.

This simple example of a multi-user Prolog system organised within the framework of a CLP-Prolog program illustrates how the coupled architecture can be programmed to support multiple interfaces on different terminals that invoke separate Prolog execution threads. If each Prolog execution thread realises a knowledge base inference engine, and if each CLP front-end to that thread handles the interaction with the user, the whole can realise a multi-user knowledge based system.

Boundary communication is useful where premature communication using a transient interface is undesirable. For example, the CLP call

```
|?- prolog(try(A)), check(A).
```

where *try/1* is defined in Prolog by a generate and test pair of goals

```
try(A) :- generate(A), test(A).
```

risks having a premature binding for A created by satisfying *generate(A)*. If this binding is communicated early to the CLP computation, and the Prolog goal *test(A)* subsequently fails for that binding, then A is restored and may be rebound to a new value in resatisfying *generate(A)*. Should the new value satisfy *test(A)*, the interface will detect an inconsistency when the final value for A is communicated, and has to fail the *prolog/1* call on a semantic exception. If instead the original call had been

```
|?- try(A), check(A).
```

then the boundary nature of the communication means that a binding for A would not have been communicated until *try(A)* had succeeded.

Concurrent calls can be made over the same Prolog database state using the One Database per Meta-call rule, e.g.

```
|?- horn(a(A,B)), try(A).
```

These two boundary communication calls to Prolog execute concurrently over the same database state. Calls to two different Prolog database states can be achieved by using distinct control meta-calls. For example, the call

```
call(try(A),_,_),call(try(B),_,_).
```

executes the two *try/1* calls over separate Prolog databases in separate Unix processes. When the Prolog engine is coupled to a suitable Database Machine for handling concurrent access to shared clauses, even these distinct concurrent Prolog processes can share parts of their database as demonstrated below.

PROLOG ENGINE APPLICATIONS

The Prolog engine supports multiple concurrent Prolog computations. Each Prolog computation can be used to conduct an exhaustive search over a stratified knowledge base, e.g.

```
parent(john,mary).
parent(mary,tim).

ancestor(A,B) :- parent(A,B).
ancestor(A,C) :- parent(A,B),
                 parent(B,C).
```

by using a classic all solutions predicate like *findall/3*.

```
|?- findall(X,ancestor(X,tim),L).

L = [mary,john]
```

Two significant extensions to Prolog are needed for realising the interfaces to CLP computations by the methods described in Taylor (5).
(a) Delayed goal handling is needed for handling communication with CLP computations. It can be realised by a minor variation on an efficient implementation of the *freeze/2* predicate, Carlsson (1).
(b) Escape exception handling is also needed to deal with system and user defined escape exceptions in Prolog, Taylor (4). An escape exception is propagated by special backtracking that continues past ordinary choice points to the first exceptional choice point of the raised exception's name. Such a choice point is created by the primitive *otherwise/2*. Its first argument is an atomic name or a variable which unifies with the first exception raised. Its second argument is the goal alternative to the rest of the clause body that is to be executed instead. A user can raise an escape exception using *fail(Name)*, which returns control to the last choice point for *Name*. Alternatively the system can raise a built in exception. Escape exceptions are needed for failing Prolog computations that execute concurrently with spawned *clp/1* goals that fail.

Escape exception handling is also useful for expanding the scope of Prolog's knowledge base handling capabilities beyond stratified databases to cope with recursion. It allows non-terminating recursions to be handled gracefully. The following clauses for *any/3* define a multiple solutions predicate like *findall/3* that copes with non-terminating recursions

```
any(X,G,L) :-  assert(store(M-M)),
               more(X,G,L).

more(X,G,L) :-
      otherwise(stack_overflow,p(L)),
      call(G),
      collect(X),
      fail.
more(X,G,L) :-
      retract(store(L-[])).

p(L) :- retract(store(L-_))),


collect(X) :-  retract(store(A-B)),
               B = [X|C],
               assert(store(A-C)), !.
```

The predicate *any/3* takes a solution element template X, a goal G and unifies its argument L with a list of all satisfiers of the form X for each satisfier of G. It initialises the difference list of solutions and then calls *more/3*. *more/3* generates successive solutions by backtracking using *collect/1* to add new solutions to the stored difference list.

The multiple solutions predicate *any/3* collects solutions like *findall/3* but where local stack overflow occurs, as it would on the attempt to satisfy a subgoal that searches depth-first down an infinite branch of the SLD resolution tree, Prolog raises the built in exception *stack_overflow*. This results in failure back to the last exceptional choice point for that exception and its alternative *p(L)* is executed instead. The effect of exceptional backtracking is to free much of the space claimed on the local stack and to enable execution to continue using the list of solutions computed so far. The fact that the computation has not been able to complete is indicated by the unbound end of the list of solutions in *any/3*. In this way escape exception handling allows graceful recovery from recursion loops with non-stratified databases.

Using *clp/1* Prolog can set up concurrent interfaces to CLP computations

```
par([F|Fs]) :- clp(user(F)),
               par(Fs).
par([]).
```

where *user/1* is defined in CLP as

```
user(F) :- open(F) | ask(yes,F).

ask(yes,F) :-  write('yes\n|?-',F) |
               read(G,F),
               call(G,Result),
               ask(Result,F).
ask(no,F) :-   write('no\n|?-',F) |
               read(G,F),
               call(G,Result),
               ask(Result,F).
```

Over this program the Prolog query

```
|?- par(['/dev/tty1','/dev/tty2']).
```

would set up interfaces to two CLP computations for two terminals.

COUPLED ENGINE APPLICATIONS

A meta-interpreter for the coupled system using all four interfaces can be built out of the meta-interpreter of the lingua franca given earlier and a Prolog meta-interpreter. *meta_clp/1* is redefined as

```
meta_clp((A,B)) :-
              true |
              meta_clp(A),
              meta_clp(B).
meta_clp(A) :-
              primitive(A),
              A \= prolog(_) |
              call(A,yes).
meta_clp(prolog(A)) :-
              nonvar(A) |
              prolog(meta_prolog(A));
meta_clp(A) :-
              clauses(A,[C|Cls]) |
              reduce(A,[C|Cls],B),
              meta_clp(B);
meta_clp(A) :-
              true |
              meta_prolog(A).
```

The new third clause handles the transiently communicating interface *prolog/1*. Execution of the guard suspends until A is instantiated. The new fifth clause handles the boundary communication interface. It is only invoked if the goal is not defined in the CLP engine. A simple Prolog meta-interpreter invoked by *meta_prolog(Goal)* supports two interfaces to the CLP engine

```
meta_prolog((A,B)):- meta_prolog(A),
                     meta_prolog(B).
meta_prolog(A) :-    primitive(A), !,
                     call(A).
meta_prolog(A) :-    clause(A,B),
                     body(B).
meta_prolog(A) :-    not(clause(A,_)),
                     meta_clp(A).

body((clp_call(A,X),B)) :- !,
                     clp_call(A,X),
                     meta_prolog(B).
body(A) :-           meta_prolog(A).
```

The first clause of *meta_prolog/1* handles conjunctions. The second identifies calls to primitives with *primitive/1* and executes them with *call/1*. The third clause retrieves clause unifiers using *clause/2* and executes clause bodies using *body/1*. The last clause is only invoked if A is neither a primitive nor a user defined goal. As *meta_clp/1* is only defined in the CLP system, the body goal invokes the CLP engine atomically.

Clauses using *clp/1* are translated from the first to the second form

```
check(A,B) :-  test(B),
               clp(try(A,C)),
               check(B,C).

check(A,B) :-  clp_call(try(A,C),X),
               test(B),
               check(B,C),
               data(X).
```

where the call *clp_call/2* sets the CLP computation going, and Prolog binds X when it succeeds. In the meantime *data/1* delays further execution beyond the body of the clause until X is bound.

The failure of the CLP computation causes resatisfaction of the parent goal *check/2*.

The conciseness of this meta-interpreter for the coupled system helps confirm how natural, simple and expressive the four interfaces are.

PROLOG DATABASE MACHINE APPLICATIONS

Prolog-X is a module based Prolog, specially adapted to exercise the facilities of a Prolog Database Machine for handling very large numbers of clauses, Williams et al (7). Using Prolog-X as the Prolog engine of a coupled engine system gives the system large scale resolution capabilities. Prolog-X clauses can either be held in internal modules in main memory or in external modules on disc under the control of the Prolog Database Machine. External modules appear almost like internal modules, except that all accesses and updates of external modules must be performed within a transaction. A Prolog-X call

```
|?- crs_loadmodule(db,'/tmp/db').
```

opens a module called *db* and associates the persistent file '/tmp/db' with it. If the file already exists, then the clauses already in it become visible in the current module. The call

```
|?- transaction(db_query(L)).
```

tries to satisfy *db_query(L)* over the clauses in '/tmp/db'. Should concurrent access to the same file by another user cause transaction failure, the whole transaction meta-call fails without updates to clauses in the module being committed to the external file. Successful transactions atomically commit their updates to the file on exit.

CONCLUSIONS

The examples show how the right combinations of resolution engines in an appropriate configuration and linked to the facilities of a Prolog Database Machine can be used to program large multi-user knowledge based systems in logic. The next generation of computer systems will be knowledge processing engines, and the approach outlined here, describes one way to implement them by executing logic programs in parallel.

ACKNOWLEDGEMENTS

Support from SERC and ICL for Alvey project IKBS 90 is acknowledged.

References

1. M. Carlsson, "Freeze, Indexing and Other Implementation Issues in the WAM," Logic Programming, Proceedings of the Fourth International Conference, pp. 40-58, Melbourne, Australia, (May 1987).

2. K.L. Clark and S. Gregory, "Parlog and Prolog United," Logic Programming, Proceedings of the Fourth International Conference, pp. 927-961, (May 1987).

3. P.A. Massey, D.J. Ferbrache, and M.H. Williams, "Adapting Prolog for Knowledge Based Applications," Department of Computer Science Technical Report, Heriot-Watt University, Edinburgh, (May 1989).

4. H. Taylor, "Escape Exception Handling in Prolog," Technical Report 88/11, Computer Science Dept, Heriot-Watt University, (December 1988).

5. H. Taylor, *Coupling Committed and Trial Binding Resolution Engines*, Computer Science Dept, Heriot-Watt University, (June 1989).

6. H. Taylor, *A Lingua Franca for Concurrent Logic Programming*, pp. 1-20, Computer Science Dept, Heriot-Watt University, (April 1989).

7. M.H. Williams, G. Chen, D.J. Ferbrache, P.A. Massey, S. Salvini, H. Taylor, and K.F. Wong, "Prolog and Deductive Databases," Knowledge-Based Systems, pp. 188-192, (June 1988).

8. K.F. Wong and M.H. Williams, "Design Considerations for a Prolog Database Engine," Proceedings of the Third International Conference on Data and Knowledge Bases, Jerusalem, Israel, (June 1988).

35. M.H.WILLIAMS, P.A.MASSEY and J.A.CRAMMOND, Benchmarks for Prolog from a database viewpoint, Proceedings of the Second Workshop of the Special Interest Group for Knowledge Manipulation Engines, Alvey Directorate, C1-8 (1987).

Benchmarks for Prolog from a Database Viewpoint

M.H. Williams
P.A. Massey
J.A. Crammond†

Computer Science Department,
Heriot-Watt University,
Edinburgh.

Abstract

Several researchers have put forward benchmark programs that have been designed to measure and compare the effectiveness of different Prolog systems. One area for which Prolog is particularly well-suited is that of databases. However, the benchmarks proposed for database applications are not very realistic. This paper describes briefly some of the problems and proposes a more realistic set of benchmarks.

1. Introduction

The logic programming language Prolog [1] was initially developed as a language for artificial intelligence applications, for which it has proved very effective. Its usefulness is not restricted to AI and another related application area for which it is potentially well-suited is that of databases [2, 3, 4]. However, in designing benchmarks to measure and compare the performance of different Prolog implementations, this area has generally been neglected [5, 6, 7].

The most important consideration with databases is that of *scale*. If one regards a small set of clauses as being of the order of 10 clauses, a medium sized set as being of the order of 100 to 1000, a large set as being 10,000 to 100,000, and a very large set as being in excess of 1 million clauses, then it is the very large sets which are of most interest in databases.

A recent set of benchmarks proposed by IST on behalf of Alvey [5] includes benchmarks to assess the performance of the database. On closer inspection these refer to databases which are to be held in main memory and fail to be useful for assessing the capabilities of a system with respect to very large sets of data. The reasons for this are as follows:

(a) The size of the data set produced is of the order of several thousand clauses. When this is scaled up to very large data sets, the data generation program cannot cope.

(b) The form of test data is limited in two ways, viz:

- The number of different atoms used in the set of clauses is very small and thus even if the size of the data set could be scaled up, the clauses so generated will in no way tax the symbol table handling capabilities of the system, as would be expected in a typical database application.
- The test data contains little or no structure which might occur in, for example, an expert system.

(c) No attempt has been made to assess the performance of Prolog systems in handling typical database operations such as the join operation.

---

† Department of Computing, Imperial College, London.

For these reasons this paper considers the problem of benchmarks for database applications and proposes a set of such benchmarks.

2. Benchmarking

A benchmark is a program or set of programs which allows the performance of similar system features in different implementations to be compared. The performance statistics can be used to decide on the value of particular design decisions, or to identify the strengths and weaknesses in a system. Usually, the statistics gathered focus on the time taken to perform certain operations.

When developing a set of benchmarks two general approaches are normally adopted. The first is one in which the system is decomposed into its constituent parts. Each part is then assessed separately. This is normally approached either from the bottom up, building up from single discrete operations to complete application programs, or from the top downwards, reducing the application program to its basic operations.

The second approach is to take existing programs, analyse them both statically [8] and dynamically and synthesize a **typical** or **dhrystone** program. We are only aware of a few large database application programs, including Chat-80 [9], which has around 3000 clauses, a Molecular Biology Database [10] with about 40,000 clauses and a Real-Time Expert System called RESCU [11]. This makes it unreasonable to attempt to develop a dhrystone benchmark program at present.

In developing benchmarks certain possible criticisms have to be addressed. Typical criticisms which may be applied to benchmarks in general [12] are that:

- The test material is too restrictive.
- The test material emphasises only certain features (ie. the whole system must be compared, not just one or a few features).
- The area which the benchmark is to test is not clearly defined.
- There is no means for allowing comparison between tests executed on different systems.

In this case the test material does emphasise only certain features in that the benchmarks described are focussed specifically on the database. However, it is intended that these tests should be regarded as an addition to the smaller tests put forward by others which are not specific to the database. If this is accepted, the total package is certainly not too restrictive. In the tests put forward here, the area to be tested is clearly defined. The benchmarks proposed here will be described under three headings, viz:

Low	dealing with single discrete operations (eg. a single retrieval).
Medium	dealing with combinations of the low level operations.
General	dealing with either real application programs whose performance characteristics are not necessarily understood, or a dhrystone benchmark program which is specifically written to behave as a typical program.

The benchmark tests described are concerned essentially with the low and medium levels, while some programs which might be suitable for the general level are described in [11].

3. Benchmark Tests

In designing a set of benchmarks to assess a Prolog systems database handling capabilities, it is useful first to consider what information is required. This can initially be achieved by using a set of questions, in this case directed at the database. Such questions as [6, 7]:

- What is the retrieval time for the first solution, the second solution, all solutions ?
- What is the time to fail (time to retrieve last solution and fail or time to find that no solution exists and fail) ?
- What is the time taken to assert/retract a clause ?
- What is the time to perform key primitives such as sort, bagof ?
- What is the maximum number of clauses which can be inserted into the database ?
- What is the time to consult a file ? reconsult it ?

Questions of this kind help to focus the attention on particular aspects of the system. The actual proposed benchmark tests which are to be applied to the database are given in section 5.

4. Data Generation

In order to assess the variation in performance of the system for database applications, a variable amount of test data is needed. The data should be produced by a single portable program or a set of program(s). In testing the different systems the form of the test data is important as it can have a significant effect on the results obtained. In database application programs two factors in particular can influence the performance of the system. These are:

(a) The number of clauses, and

(b) The size of the symbol table.

Various methods already exist for generating large amounts of data with a prescribed format. The IST benchmarks [5] referred to in the Introduction rely on the use of a generator program which produces unit clauses having the following format:

calendar(Day-of-week,Day-of-month,Month,Year).

The shortcomings of this generator have already been mentioned. Of most significance is the fact that although each clause is unique, the number of unique atoms generated is small. An alternative generator given in [13] produces data concerning a hypothetical conference. While this will generate a larger number of unique atoms it does not cater for structured terms (A shortcoming of the IST benchmarks too).

Both of these generators are limited in the number of clauses that they can reasonably produce and in the form of the clauses generated. The following two sections describe data generators which can be used with the benchmark tests proposed in this paper.

4.1. Space Filling Curve Data Set

One easy way of generating a very large set of facts is to set up a substantial 2 dimensional grid of points and create a set of edges connecting them. Each point may be represented by a unit clause of the form:

vertex(Vertex_Name,X,Y).

which describes the points passed through when generating the curve, while each edge is represented by a unit clause of the form:

edge(Edge_Name,Vertex_Name1,Vertex_Name2).

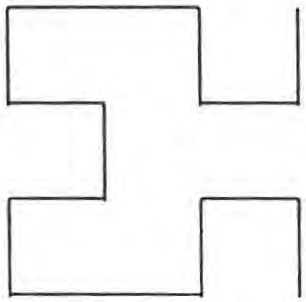
A large set of rules may be created using a variation of this method involving a form of incomplete information. Following the approach of Williams and Nicholson [14] one might add a third dimension to the set of coordinates for

the grid of points and then generate vertices in the form:

vertex(Vertex_Name,X_Coord,Y_Coord,Z) :- range(Z,Z_Upper,Z_Lower).

where range denotes the limits of the z- coordinate. A program for generating a set of rules in this form is given in [15].

A simple and fast way of generating a large number of systematically related clauses is to generate Peano, Hilbert or Murray polygons [16, 17, 18]. This data generation method guarantees uniqueness of each data item. Figure 2 shows a Hilbert Curve of level 2, which would produce two files, each having 15 clauses.



A Hilbert's Curve of Level 2

This would result in the following data set:

```
vertex('9999',99,99)     edge('99999966','9999','9966')
vertex('9966',99,66)     edge('99666666','9966','6666')
vertex('6666',66,66)     edge('66666699','6666','6699')
vertex('6699',66,99)     edge('66993399','6699','3399')
vertex('3399',33,99)     edge('3399099','3399','099')
vertex('099',0,99)       edge('099066','099','066')
vertex('066',0,66)       edge('0663366','066','3366')
vertex('3366',33,66)     edge('33663333','3366','3333')
vertex('3333',33,3       edge('3333033','3333','033')
vertex('033',0,3         edge('03300','033','00')
vertex('00',0,0)         edge('00330','00','330')
vertex('330',33,0)       edge('330660','330','660')
vertex('660',66,0)       edge('6606633','660','6633')
vertex('6633',66,33)     edge('66339933','6633','9933')
vertex('9933',99,33)     edge('9933990','9933','990')
vertex('990',99,0)
```

To generate a Hilbert curve only two outside inputs are needed - the screen size and the curve form. The program, which for reasons of efficiency, was written in C is given in [15]. The program produces two data files, one containing the vertex clauses and one containing the edge clauses.

4.2. Pseudo-Random Data Set

The Hilbert curve data generator described above, is quick to execute and produces data containing many different atoms. However, it suffers from one of the two previously outlined problems, in that the clauses which it generates contain no structured arguments. Consequently, an alternative data generator has been developed to address this problem and will produce data with a specifiable amount of structure.

This data generator takes a *format list* which is used to describe the format of the data required. The actual data is produced using a pseudo-random number generator which uses integer only arithmetic so that systems (eg. Prolog-

X) which do not have floating point arithmetic can use it.

A more complete description of the pseudo-random data generator together with the actual program is given in [15].

5. Main Tests

The benchmark test methodology used in gathering results is similar to that used by Wilk [6] and the IST benchmarks. A shell program is used to perform the actual test and to log the results.

For those tests which use the data produced by the pseudo-random data generator the data could have the following form:

```
cn(abba,96354,lls(lalla,allal),64646,[7474,43838,92921,86558]).
cn(dddd,8484,ala(lsls,slsl),4646,[4848,8282,131,43554,4361]).
```

This form was chosen so that it would contain structure of the two most common forms: lists and terms.

Each of the tests given in the following section should be performed a number of times, and the average value taken.

5.1. Low Level

The tests in this category correspond to single discrete instructions. The first three tests would be performed using the data set produced by the Hilbert curve program, while the remaining tests would use the pseudo-random data set. The following tests are concerned with elementary retrievals from the database.

(1) Retrieval on all ground arguments. A clause is selected from approximately the middle of the database. In the Hilbert curve data set the clause *vertex('00',0,0)* is well positioned for such a search.

(2) Retrieval on 1st argument, ie. for clause *vertex('00',_,_)*. A set of solutions would be searched for, even though there would be only one match.

(3) Retrieve on third argument. The search is for the clause *vertex(_,_,0)*. This test would again return a set of solutions, but would identify those systems which index the clause in their database on only the first and second arguments.

(4) Retrieval on 2nd element of a list argument. The clause *cn(_,_,_,_,[_,123|_])* could be searched for. This would test the ability of the systems where structure is concerned.

The next two tests are concerned with database updates and use the pseudo-random data generator as a means of producing unique clauses which could then be inserted into the database. These tests would have to encompass any available additional storage mechanisms (eg. the ability of the system to maintain clauses in secondary storage).

(1) The time taken to assert a set of clauses into the database.

(2) The time taken to retract a set of clauses from the database.

5.2. Medium Level

The tests in this section require, in most cases, large amounts of backtracking over the sets of clauses, and test the concepts of join and exhaustive search.

The following benchmarks use the vertex data set, and return a set of solutions (via fail). These tests would be fairly typical in that they would describe the connection between a number of related objects.

(1) single join: vertex(X,Middle_elem,_),edge(_,X,_)

(2) double join: vertex(X,Middle_elem,_),edge(_,X,Y),vertex(Y,_,_).

The following tests use the data generated by the Hilbert curve program.

(1) A clause is asserted to the end of the database identifying an extra edge. A search is then made for the vertex with three edges.

```
test(_):-solution(A,B),!.

solution(A,B):-
        edge(X,Y,Z),
        two_other_edges(X,Y,Z),
        vertex(Y,A,B).

two_other_edges(X,Y,Z):-
        edge(A,Y,B),
        not(X=A),
        edge(C,D,Y).

two_other_edges(X,Y,Z):-
        edge(A,B,Y),
        edge(C,D,Y),
        not(A=C).
```

(2) Search to find the horizontal edges along the middle row of the Hilbert curve. This test requires that it be given the X and Y Coordinate of the start point, the maximum width of the Hilbert curve and an increment value which is the length of an edge.

```
test(_,(Start_X,Middle_Y,Max_X,INC)):-
        solution(Edge_name,Start_X,Middle_Y,Max_X,INC).

solution(Edge_name,Start_X,Middle_Y,Max_X,INC):-
        x_inc(Start_X,X,Max_X,INC),                    % Get X Co-Ord
        vertex(V1,X,Middle_Y),
        rest_test(V1,X,Middle_Y,Edge_name,INC).

rest_test(V1,X,Middle_Y,Edge_name,INC):-
        Next_X is floor(X + INC),
        vertex(V2,Next_X,Middle_Y),
        rest_test2(V1,V2,Edge_name),!.

rest_test2(V1,V2,Edge_name):-
        edge(Edge_name,V1,V2),!.

rest_test2(V1,V2,Edge_name):-
        edge(Edge_name,V2,V1),!.

x_inc(Start,Val,Max_val,INC):-          % return succesive values for
        Start =< Max_val,               % X- Coordinate on failure.
        (Start=Val;
         Val1 is Start + INC,
```

```
x_inc(Val1,Val,Max_val,INC)).
```

The following tests use the data produced by the pseudo-random data generator. The tests have been designed so that the only factor influencing the time to find the solution is the size of the database and the amount of searching which has to be performed.

(1) Find all those clauses whose fourth argument is a multiple of 80. This would be a common retrieval operation ie. retrieve all those records which have a field which matches some pattern.

```
test(_,[Name,Arity,Depth]):-
        find(Name,Arity,130171,0,4,_).

find(_,_,Range,Value,_,[]):-
        Value >= Range,!.

find(Name,Arity,Range,Value,Arg_nos,[Clause|T]):-
        gen_mult(Value,Range,Val),
        functor(Clause,Name,Arity),
        arg(Arg_nos,Clause,Val),
        call(Clause),
        Val2 is Val + 80,
        Cd1 is Cd + 1,!,
        find(Name,Arity,Range,Val2,Arg_nos,T).

find(_,_,_,_,_,[]):-!.

gen_mult(Value,Range,Val):-
        Value > Range,!,fail.
gen_mult(Value,Range,Value).
gen_mult(Value,Range,V2):-
        Val is Value + 80,
        gen_mult(Val,Range,V2).
```

(2) Find all those clauses whose first argument is in the range of the first and second elements of the fifth argument list. This would also be a common operation, for example, to search for all those people who are over 65 and live in Brighton.

```
test(Clause_List):-
        find(cn,5,Clause_List).

find(Name,Arity,[Clause|T]):-
        functor(Clause,Name,Arity),
        call(Clause),
        arg(Arity,Clause,[A1,A2|_]),
        arg(1,Clause,Value),
        Value >= A1,
        Value =< A2,
        Val2 is Val + 80,
        Cd1 is Cd + 1,!,
        find(Name,Arity,T).

find(_,_,[]):-!.
```

2. Conclusions and Future Work

This paper describes a set of benchmark programs for evaluating the performance of Prolog systems in the context of database applications. A fuller paper describing the tests and the results obtained from several different Prolog systems will be submitted for publication shortly.

References

1. W. F. Clocksin, C. S. Mellish, *Programming in PROLOG*, Springer-Verlag, New York, (1981).

2. M.H. Williams, *The Design of a Prolog Database Machine*, Proc. First SIGKME Workshop, (January 1987).

3. J.C. Neves, S.O. Anderson and M.H. Williams, "A Prolog Implementation of Query-by-Example," *Proc. 7th International Computing Symposium*, pp. 318-332, B.G. Teubner, Stuttgart, (1983).

4. R.A. Kowalski, *Logic as a Database Language*, Internal Report, Department of Computer Science, Imperial College, (1981).

5. S. Dickens and A. Bustany, *Alvey Prolog Benchmarks*, Imperial Software Technology, Cambridge, (1986).

6. P. F. Wilk, "The Production and Evaluation of a Set of Prolog Benchmarks," AIAI-TR-13, (1983).

7. P. F. Wilk, "Prolog Benchmarking," AIAI-TR-14, (1986).

8. H. Matsumoto, "A Static Analysis of Prolog Programs," *Programming Systems Group, Note 24*, AI Applications Institute, Edinburgh, (January 30, 1985).

9. P.F. Wilk, *Chat-80*, Internal Report, Edinburgh University.

10. C.J. Rawlings, K. Seifert, J. Saldanha, "A Large Knowledge Based System For Molecular Biology," *Proc. First SIGKME Workshop*, (January 1987).

11. "Report on the Alvey Architecture Club's Collection of Benchmark Programs," *Proc. First SIGKME Workshop*, (1987).

12. Richard Gane, "Better Benchmarking - getting the most from your tests," *CADCAM International*, **Vol. 4 No 3.**, pp. 10-18, (March 1985).

13. R.A. Overbeek, E.L. Lusk, *Prolog Benchmark 1: Accessing Databases*, Argonne National Laboratory.

14. M.H. Williams and K.A. Nicholson, *An approach to handling Incomplete Information in Databases*. Due to appear in The Computer Journal

15. M.H Williams, P.A. Massey, J.A. Crammond, *Benchmarking Prolog for Database Applications*; Computer Science Department, Heriot-Watt University, (Edinburgh).

16. A. J. Cole, "A Note on Space Filling Curves," *Software - Practice and Experience*, **Vol. 13**, pp. 1151-1159, (1983).

17. I. H. Witten, B. Wyvill, "On The Generation and Use of Space Filling Curves," *Software - Practice and Experience*, **Vol. 6**, pp. 519-525, (1983).

18. Niklaus Wirth, *Algorithms & Data Structures*, Prentice Hall Inc., (1986).

36. M.H.WILLIAMS, P.A.MASSEY and J.A.CRAMMOND, Benchmarking Prolog for database applications, in 'Prolog and Databases: Implementations and New Directions', Ed. P.M.D.Gray and R.J.Lucas, Ellis Horwood, Chichester, 161-187 (1988).

11

Benchmarking Prolog for database applications

M. H. Williams, P. A. Massey
Computer Science Department, Heriot-Watt University, Edinburgh
J. A. Crammond
Department of Computing, Imperial College, London.

ABSTRACT

With the growth in interest in declarative programming languages and knowledge-based systems, there has arisen a need to measure and compare the performance of different systems. In the case of the logic programming language Prolog, several sets of benchmark programs have been proposed for evaluating the performance of different implementations. However, one area which has been neglected is that of Prolog's suitability for handling large sets of facts and rules, an essential prerequisite for database and large knowledge-base applications This chapter puts forward a practical set of benchmarks for evaluating this aspect of Prolog.

1. INTRODUCTION

One of the consequences of the Japanese fifth generation computer programme has been to create an interest in evaluating the performance of logic programming systems. In the preparatory work, Furukawa *et al.* [1] introduced the term *Logical Inferences Per Second* (LIPS) as the basic unit of measurement, 1 LIPS being stated to be equivalent to approximately 100–1000 instructions per second on a conventional computer [2]. The objectives of the fifth generation programme include the design of high performance systems capable of up to 1 Giga-LIPS — a figure which has given rise to some controversy. However, whether or not these performance targets are achieved, they have focused attention on the need for measuring and improving the performance of such systems.

In the case of Prolog, various simple tests have been proposed as benchmarks for measuring the processing capability of Prolog systems. In particular, benchmark programs have been proposed by Moss [3], Wilk

[4,5] and Syre [6], and more recently a set of benchmarks has been proposed by Dickens and Bustany [7] as part of the Alvey research programme in the UK. However, while these benchmarks cover database indexing as one aspect which is exercised, they do not adequately test the effectiveness of systems in handling large sets of clauses.

One of the application areas for which logic programming languages offer considerable potential is that of databases. Their suitability for implementing database concepts was first recognized by Kowalski [8] and since then much work has been done on this [9,10]. Unfortunately, although they provide a very natural medium for expressing such concepts, they also suffer from the drawback that most current Prolog systems are very inefficient when handling large data sets. Thus there is growing interest in finding ways of combining database techniques with those of logic programming to produce systems which are both efficient and powerful.

In order to assess the success of such systems, yardsticks are required to measure and evaluate their performance in the context of database applications. This chapter explores a set of benchmarks which could be useful for this purpose.

2. TYPES OF BENCHMARK

Benchmarking is normally approached either from the bottom up — dealing first with single discrete operations and building up to complete application programs — or from the top downwards — starting with the application program and reducing it to its basic operations (attempting to identify those critical operations which have the most significant effects on the performance of the program). One further approach is to take existing programs, perform some sort of analysis on them and from this synthesize a **typical** or **dhrystone** program. The overall approach to benchmarking should, however, be one of iterative development. Benchmarks can generally be categorized into three levels. These are

Low	dealing with single operations (e.g. a single retrieval).
Medium	dealing with combinations of the low-level operations.
General	dealing with either real application programs whose performance characteristics are not necessarily understood, or a **dhrystone** benchmark program which is specifically written to behave as a **typical** program.

The benchmark tests described in this chapter are concerned essentially with the low and medium levels, while some programs which might be suitable for the general level are described in [11]. The approach to benchmarking which is taken in this chapter is that of bottom up. The reason for this is that there are currently very few large database application programs implemented in Prolog. Those known to us include Chat-80 [12], which comprises around 3000 clauses, a Molecular Biology Database [13]

having approximately 40 000 clauses and a Real-Time Expert System called RESCU [11] (which is written in POPLOG). In view of this it is not really feasible to develop a **dhrystone** benchmark program at the present time.

3. BENCHMARK TESTS

In designing a set of benchmarks to assess the database handling capabilities of a Prolog system, it is useful first to consider what information is required. This can be achieved by using a set of questions, in this case directed at the database. Such questions might include [5,4]:

— What is the retrieval time for the first solution, the second solution, all solutions?
— What is the time to fail (time to retrieve the last solution and fail or time to find that no solution exists and fail)?
— What is the time taken to assert/retract a clause?
— What is the maximum number of clauses which can be inserted into the database?
— What is the time to consult a file? reconsult it?

Questions of this kind help to focus the attention on particular aspects of the system. The following sections describe the actual benchmark tests to be applied to the database.

3.1 Low level

As mentioned earlier, the tests in this section correspond to a single, simple Prolog instruction. They assess the time taken to perform the following operations.

(a) Search a set of clauses for a single solution.
(b) Search for a set of solutions.
(c) Search on:
  (1) all ground arguments
  (2) first argument
  (3) third argument
  (4) part of a structured argument value.
(d) Assert a clause.
(e) Retract a clause; retract all clauses with a given argument value.

3.2 Medium level

The tests in this section use combinations of these low-level operations. These are:

(a) combining the results of two relations in a **join** operation [14]. This is a common relational database operation;
(b) retrieving a list of solutions in a manner similar to findall, bagof, etc.;
(c) nested rule invocation.

3.3 Additional database benchmarks

This section covers additional features which should be tested.

(a) What is the maximum number of clauses which can be inserted into the database?
(b) What is the time to reconsult files of various sizes? Reconsult time can be tested without user intervention (as opposed to consult which cannot).
(c) Is there a facility to allow quick access to the clauses in compiled form, thus removing the need to consult the database clauses each time they are needed?

4. DATA GENERATION

In order to assess the variation in performance of the system with varying size of database, a variable-sized amount of data is needed. The data should be able to be produced by a single portable program or set of programs. The program should not require any outside information, except the size of data set to be produced and possibly the form of data needed. When testing the database handling capabilities of different systems, the form of the test data can have a significant effect on the results obtained.

A number of methods are available for artificially generating large amounts of data with a prescribed format. However, on close inspection the data generator programs are aimed at databases which are to be held in main memory and fail to be useful for assessing the capabilities of systems with respect to very large databases.

The benchmarks proposed recently as part of the Alvey programme by Dickens and Bustany [7] rely on the use of a generator program which generates unit clauses having the following format:

```
calendar(Day-of-week,Day-of-month,Month,Year).
```

Each clause is unique, but the number of different atoms generated is small. In [15] a data generator is given which generates data concerning a hypothetical conference. While the latter will generate a larger number of unique atoms, neither generator deals with the problem of structured terms. In addition to this limitation in the form of clauses, both data generators are limited in the number of clauses that they can reasonably generate. The following sections describe two data generators which can be used with the benchmark tests.

4.1 Space filling curve data set

One easy way of generating a very large set of facts is to set up a substantial two-dimensional grid of points and create a set of edges connecting them. Each point may be represented by a unit clause of form

```
vertex(Vertex_Name,X,Y).
```

and each edge by a clause of the form

```
edge(Edge_Name,Vertex_Name1,Vertex_Name2).
```

The Vertex_Name is created by concatenating the X and Y positions

together, and the Edge_Name is created by joining two Vertex_Names together.

One simple and fast way of generating a large number of regularly distributed edges is to generate Peano, Hilbert or Murray polygons [16–18]. This data generation method guarantees uniqueness of each data item. Fig. 1 shows a Hilbert curve of level 4, which is described by a set of 256

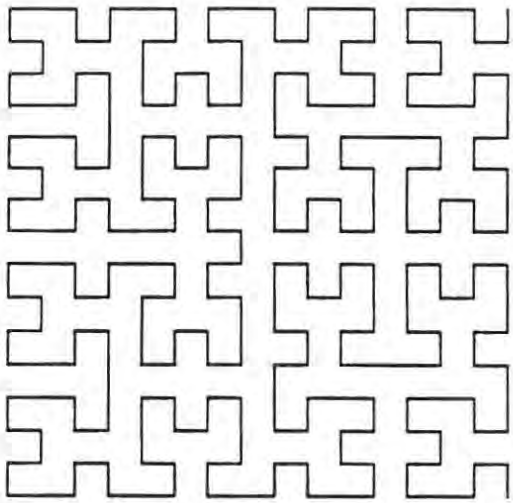


Fig. 1 — A Hilbert curve of level 4.

vertex clauses and 255 edge clauses.

To generate a Hilbert curve, only two input values are needed — the screen size and the curve form. The program which, for reasons of efficiency, was written in C is given in Appendix 1. This program produces two data files, one containing the vertex clauses and one containing the edge clauses.

To create a large set of rules, a variation of this two-dimensional data set may be employed, which incorporates a form of incomplete information. Williams and Nicolson [19] have shown how incomplete information may be represented by means of Prolog rules. Suppose therefore one were to add a third dimension to the set of coordinates for the grid of points and then generate vertices in the form

```
vertex(Vertex_Name,X_Coord,Y_Coord,Z):-
    range(Z,Z_Upper,Z_Lower).
```

where Z is a variable and the other parameters are all constants. This reads *'vertex Vertex_Name has x- and y-coordinates X_Coord and Y_Coord and as z-coordinate a value (represented by Z) which is not known exactly but*

lies somewhere in the range Z_Upper >= Z >= Z_Lower'. One can use the same method as before for generating the x- and y-coordinates of the grid of points while the limits on the ranges of the z-coordinate may be set up using a random number generator. The program to generate data of this form is the same as the one to generate simple vertices and is given in Appendix 1.

4.2 Pseudo-random data set

The Hilbert curve data generator described above is easy to use and will produce the data quickly. However, although the data generated contain many different atoms, they suffer from the second of the two previously outlined problems. They contain no structures. The data generator described below is one that has been developed to address this problem.

The data generator takes a *format list* which is used to describe the format of the data required. The actual data are produced using a pseudo-random number generator which has been integerized so that systems (e.g. Prolog-X) which do not have floating-point arithmetic will be able to use it. The use of integer-only arithmetic also avoids any rounding/truncating problems which might arise owing to differences in the various system implementations. The pseudo-random number generator will produce, depending on the range, a maximum of 131 072 unique numbers after which it will cycle. The generator is called as follows:

dg(Name,Format List,Uniqueness List,Clause).

A more complete description of the form of the arguments required by the data generator is given in Appendix 2. By repeated calling (on backtracking), files containing any number of clauses can be produced (depending on the limit of symbol table size). In generating the data set, a second file, called the **mark** file, is also generated containing information which can be used by the benchmark tests, currently the clause in the middle of database. Examples of possible output data are

```
(1) ?- dg(vertex,[int,int,int),[131072,360,360],Clause).
    vertex(123637,327,255).
    vertex(77570,352,34).
    vertex(104739,319,321).
(2) ?- dg(net,[funct(const),[int,tail]],[10,900,90],Clause).
    net(bb(cki),[64|_185]).
    net(h(iqi),[8|_175]).
    net(j(yqh),[80|_171]).
(3) ?- dg(names,[vlenlist(const,5,2,notail)],[9999],Clause).
    names([rqie,pikf,tbqj,vijd]).
    names([ljne,ffid,jhf,fljf,fdhb,1lmd]).
    names([omf,iiib,hmkb,ekoi,ljf]).
(4) ?- dg(names,[vlenfn(vlenlist(const,2,0.5,notail),4,2)],[99,999,10,10],
      Clause).
    names(ke([uig,ljb,hhf],[moi,sic])).
```

```
names(oj([mcb,rnh,rmf],[lbb,lgf],[uke,ogc],[ngc,iff])).
names(jb([ndc],[nmd,hmd],[ig,blg])).
```

A listing of the program used to generate these data is given in Appendix 3.

5. MAIN TESTS

The benchmark test methodology used in gathering the following results is similar to that used by Wilk [5] and the Dickens and Bustany benchmarks [7]. A shell program is used to perform the actual test and to log any results. In some cases, however, the tests were terminated early because the hardware system itself could not cope and started to thrash.

For those tests which use the data generated by the pseudo-random data generator, the data have the following form:

```
cn(abba,96354,lls(lalla,allal),64646,[7474,43838,92921,86558]).
cn(dddd,8484,ala(lsls,slsl),4646,[4848,8282,131,43554,4361]).
```

This form was chosen so that it would contain structures of the two most common forms: lists and terms.

The following tests were all performed on a SUN-3 microcomputer system running BSD 4.2 UNIX.† The systems tested were NIP 1.5, Prolog-X and C-Prolog 1.5a. All the tests were performed using the same machine (with the files NFS mounted on a system which had the required disk space) and the same file system. In Tables 9, 10 and 11 the *number of clauses* refers to the number of vertex/3 clauses, while the same number (minus 1) of edge/3 clauses is also used in these tests.

5.1 Setting up the database

Once the datasets have been created by the data generation programs, the first set of results is obtained by determining the time taken to reconsult each dataset. Timings obtained for the three systems concerned are given in Table 1 for the pseudo-random datasets and Table 2 for the vertex datasets.

5.2 Low level

Each test in this category corresponds to a single discrete instruction. The first three tests were performed using the data set produced by the Hilbert curve program, while the remaining tests use the pseudo-random data set.

5.2.1 Retrievals

(1) The time to retrieve on all ground arguments. This attempts to find more than one solution. The search is for the clause *vertex('00',0,0)*. See Table 3.
(2) Time to retrieve on first argument (set of solutions). The search is for the clause *vertex('00',_,_)*. See Table 4.

† UNIX is a trademark of Bell Laboratories.

Table 1 — Time taken to reconsult the various-sized pseudo-random datasets†

Number of clauses	NIP	C-Prolog	Prolog-X
4	410	50	1650
16	1180	200	5133
64	4870	716	20800
256	19670	2951	85883
1024	78880	11734	335916
4096	322430	50416	1322483
16384	1363150	223316	5304384
30000	2564160	442215	9763000
40000	3455690	596033	12905267

† The results given in this and the following tables are all in milliseconds. All the tests excluding the two reconsult tests (Tables 1 and 2) were performed more than once and the figures given in the tables are the average times obtained. However, in the case of the two reconsults, because the timings involved were so large, each was performed only once.

Table 2 — Time taken to reconsult the vertex/3 datasets

Number of clauses	NIP	C-Prolog	Prolog-X
16	500	83	1784
64	1880	400	7284
256	7420	1450	29650
1024	30220	6166	120800
4096	125630	25950	517000
16384	594820	134183	1851533
65536	3284710	1728031	—

(3) Time to retrieve on third argument (set of solutions). The search is for the clause *vertex(_,_,0)*. See Table 5.
(4) Time to retrieve on tail of list argument (fifth argument, and again a set of solutions is retrieved), i.e. *cn(_,_,_,_,[_|T])*. See Table 6.

5.2.2 Updating the database

These tests use the pseudo-random data generator as a means of generating unique clauses to be asserted. It should be noted that in the case of systems which hold the clauses externally on secondary storage, both types of database updates should be performed in such a way as to use any of the available additional storage mechanisms (e.g. clauses held on disk).

Table 3 — Time to retrieve a single clause with all ground terms

Number of clauses	NIP	C-Prolog	Prolog-X
16	1	2	0
64	4	3	0
256	2	10	1
1024	5	35	12
4096	19	143	48
16384	70	585	201
65536	—	2475	—

Table 4 — Time to retrieve a set of solutions on 1st argument (using vertex datasets)

Number of clauses	NIP	C-Prolog	Prolog-X
16	2	1	0
64	2	1	0
250	3	9	4
1024	8	36	12
4096	20	141	48
16384	68	572	204
65536	—	2510	—

Table 5 — Time to retrieve on 3rd argument (a set of solutions), using the vertex datasets

Number of clauses	NIP	C-Prolog	Prolog-X
16	6	2	3
64	9	5	9
256	28	23	39
1024	89	86	153
4096	320	339	602
16384	1217	1342	2535
65536	—	5578	—

Table 6 — Time to retrieve on the tail of a list (5th argument using pseudo-random datasets)

Number of clauses	NIP	C-Prolog	Prolog-X
4	3	1	1
16	5	3	5
64	15	12	20
256	55	49	81
1024	230	206	334
4096	863	787	1300
16384	3540	3146	5431
30000	6251	6004	10023
40000	—	8119	13161

(1) The time taken to assert clauses into the database. See Table 7.
(2) The time taken to retract clauses from the database. See Table 8.

5.3 Medium level

The tests in this section require, in most cases, large amounts of backtracking over the sets of clauses.

5.3.1 Join operation

The following benchmarks use the Hilbert curve dataset, and each returns a set of solutions (via fail). See Tables 9 and 10.

(1) vertex(X,Middle_elem,_),edge(_,X,_).
(2) vertex(X,Middle_elem,_),edge(_,X,Y),vertex(Y,_,_).

5.3.2 Exhaustive searches

The following test uses the data generated by the Hilbert curve program.

(1) Search to find the horizontal edges along the middle row of the Hilbert curve. This test requires the X and Y Coordinates (Start_X,Middle_Y) of the start point, the maximum width of the Hilbert curve (Max_X) and an increment value (INC) which is the length of an edge. The results are shown in Table 11.

```
test(_,_,[Start_X,Middle_Y,Max_X,INC]):-
    solution(Edge_name,Start_X,Middle_Y,Max_X,INC).
solution(Edge_name,Start_X,Middle_Y,Max_X,INC):-
    x_inc(Start_X,X,Max_X,INC),        %Get X Co-Ord
    vertex(V1,X,Middle_Y),
    rest_test(V1,X,Middle_Y,Edge_name,INC).
rest_test(V1,X,Middle_Y,Edge_name,INC):-
    Next_X is X+INC,
```

Table 7 — Time taken to assert a single fact into the database

NIP	C-Prolog	Prolog-X
27.69	5.1	17.10

Table 8 — Time taken to retract a clause from the database

NIP	C-Prolog	Prolog-X
6.86	2.13	8.85

Table 9 — Time for a join operation between two clauses

Number of clauses	NIP	C-Prolog	Prolog-X
16	4	7	9
64	34	42	66
256	250	301	494
1024	1930	2325	3799
4096	15196	18277	29949
16384	121740	145056	246888

Table 10 — Time to perform a join operation between three clauses

Number of clauses	NIP	C-Prolog	Prolog-X
16	5	9	10
64	36	66	75
256	269	481	552
1024	2068	3750	4229
4096	16266	29635	32955
16384	133592	236329	286430

Table 11 — Time taken to find the edges which form a line across the middle of the Hilbert curve

Number of clauses	NIP	C-Prolog	Prolog-X
16	12	21	21
64	99	140	199
256	738	950	1490
1024	6188	7775	12468
4096	49130	61207	98909
16384	404931	494664	836266

```
    vertex(V2,Next_X,Middle_Y),
    rest_test2(V1,V2,Edge_name),!.
rest_test2(V1,V2,Edge_name):-
    edge(Edge_name,V1,V2),!.
rest_test2(V1,V2,Edge_name):-
    edge(Edge_name,V2,V1),!.
x_inc(Start,Val,Max_val,INC):-      % return successive values
    Start=<Max_val,          %      on failure.
    (Start=Val;
    Val1 is Start+INC,
      x_inc(Val1,Val,Max_val,INC)).
```

The following two benchmarks give two possible ways of implementing a selective search over the whole database. The first approach is to select out the required clauses while the second tests each clause explicitly in the database against some condition (in the normal implementation manner of findall). Both use the data generated by the pseudo-random generator, and the results are given in Tables 12 and 13.

(1) find all those clauses whose fourth argument (integer) is a multiple of 500.

```
test(_,_,[Name,Arity]):-
    find(Name,Arity,130171,0,4,_),!.
find(Name,Arity,Range,Value,Arg_nos,[Clause|T]):-
    gen_mult(Value,Range,Val),
    functor(Clause,Name,Arity),
    arg(Arg_nos,Clause,Val),
    call(Clause),
    Val2 is Val+500,
    find(Name,Arity,Range,Val2,Arg_nos,T).
find(_,_,_,_,_,[]):-!.
gen_mult(Value,Range,Val):-
```

Table 12 — Time taken to find all the clauses in the database whose 4th argument is a multiple of 500

Number of clauses	NIP	C-Prolog	Prolog-X
4	630	720	459
16	976	1063	1180
64	2378	2396	4013
256	8166	7746	15386
1024	30930	29180	60463
4096	120600	114930	240176
16384	489390	442692	938543
30000	832562	817473	1733056
40000	—	1051926	I

I = Integer overflowed during test.

Table 13 — Time taken to find all the clauses in the database whose 1st argument is less than the sum of the list which is its 5th argument

Number of clauses	NIP	C-Prolog	Prolog-X
4	10	23	13
16	22	63	40
64	108	276	146
256	458	1183	599
1024	1816	4566	2343
4096	7136	17850	9190
16384	29516	71689	38743
30000	54820	133840	70603
40000	—	178723	92499

```
    Value>Range,!,fail.
gen_mult(Value,Range,Value).
gen_mult(Value,Range,V2):-
    Val is Value+500,
    gen_mult(Val,Range,V2).
```

(2) find all those clauses whose second argument is greater than the sum of the elements of the list which is its fifth argument.

```
test(_,_,[Name,Arity]):-
    find3(Name,Arity,_),
```

```
        % In a fully implemented findall an assert would be placed here.
        fail.
test(_,_,_):-!.
find3(Name,Arity,Clause):-
        functor(Clause,Name,Arity),
        call(Clause),
        arg(Arity,Clause,List),
        arg(2,Clause,Value),
        sum_list(List,Sum),
        Value>=Sum.
sum_list([],0):-!.
sum_list([H|T],Value):-
        sum_list(T,Val),
        Value is Val+H,!.
```

5.4 Rule invocation

The following tests are based on the use of the vertex/4 rule datasets. The tests are slightly modified versions of the simple retrieval operations, the only difference being that the Z co-ordinate is set to some particular value.

1. Search for a set of solutions to a query where the first three arguments are ground, i.e. vertex('00',0,0,-5). See Table 14.
2. Search for a set of solutions to a query where the first three arguments are unspecified and a solution exists, i.e. vertex(_,_,_,-5). See Table 15.
3. Search for a set of solutions to a query where the first three arguments are unspecified and no solution exists, i.e. vertex(_,_,_,-20). See Table 16. The only difference between this test and the previous test is that the goal call fails.

Table 14 — Time taken to search for a set of solutions to a query where first three arguments are ground, i.e. vertex('00',0,0,-5)

Number of clauses	NIP	C-Prolog	Prolog-X
16	2	2	0
64	4	3	1
256	2	8	4
1024	5	35	12
4096	20	144	48
16384	75	583	244
65536	—	2607	—

Table 15 — Time taken to search for a set of solutions to a query where the first three arguments are unspecified and a solution exists, i.e. vertex(_,_,_,-5)

Number of clauses	NIP	C-Prolog	Prolog-X
16	4	11	6
64	15	37	24
256	52	146	104
1024	204	579	415
4096	788	2322	1670
16384	3148	9292	7107
65536	—	39255	—

Table 16 — Time taken to search for a set of solutions to a query where the first three arguments are unspecified and no solution exists, i.e. vertex(_,_,_,-20)

Number of clauses	NIP	C-Prolog	Prolog-X
16	5	9	7
64	13	38	27
256	49	143	102
1024	197	578	417
4096	785	2326	1669
16384	3138	9290	7070
65536	—	39191	—

6. CONCLUSIONS AND FUTURE WORK

The benchmarks proposed in this chapter are aimed at assessing the database handling capabilities of a Prolog system for real applications. They focus primarily on the time taken to perform certain basic database operations, although the amount of space required does affect the results. The tests were performed on a SUN-3 microcomputer with 4 Mbytes of main memory using three different Prolog systems: NIP 1.5, Prolog-X and C-Prolog.

The sizes of the datasets for which measurements were taken in the case of reconsults and simple retrievals extend to around 65 000 clauses. The reason for this is that the different systems ran out of memory at some number between 30 000 and 70 000 clauses. This demonstrates clearly the limitations of current Prolog systems in dealing with database applications.

The times taken to reconsult large datasets were considerable and varied

from about 10 min (for C-Prolog) to about 200 min (for Prolog-X) for a set of 40 000 clauses, as shown in Table 1. If this could be extrapolated to, say, 1 million clauses, this overhead would be unacceptable.

Many Prolog systems use indexing on the first argument to improve the efficiency of database accesses. This is the reason for structuring the tests in section 5.2.1 as they are. The first and second tests both have a constant in the first argument position, and systems which index on the first argument should perform well for these tests. The third and fourth tests seek to match on other argument positions. The difference in performance is reflected clearly in the results obtained in Tables 3 and 4 compared with Tables 5 and 6. Although, some of the results obtained show a drop in performance which is greater than would be expected, it is thought that this might be due to a possible interplay between the memory management techniques of the operating system and that of the Prolog systems.

The first test involving a join operation performs a join on a variable in the second argument position of the second predicate, thereby losing the advantage of indexing on the first argument. The second test includes this but in addition has a join on a variable in the first argument position of the second predicate. Since the latter should take advantage of indexing and is only a one–one join, one would not expect it to add much to the time for the first join operation, which has to be done by sequential search. This is borne out in Tables 9 and 10 for Nip and Prolog-X. In the case of C-Prolog there is a significant increase in the times observed between Tables 9 and 10, but this is consistent with the much higher time required by C-Prolog for retrieval on the first argument (Tables 3 and 4).

The tests involving rule invocation examine the handling of rules as opposed to facts. In systems where these are indexed on the first argument position too, the results given in Table 14 should not be very different from those in Table 3. This was indeed observed to be the case for the three systems under investigation. It should also be noted that the results given in Table 16 are very similar to those obtained in Table 15 since in both cases one has to enumerate all tuples. The results would have been different if large sets were constructed, e.g. with success on more than half the vertex clauses.

Various other conclusions can be drawn from the comparative results produced here, although it is not clear whether these would necessarily apply to other Prolog systems.

Since the development of benchmark programs is necessarily iterative, it is reasonable to consider the benchmarks proposed in this chapter as a first approximation to what a future set of tests would look like. The reasons for this are that there are as yet no very large Prolog database application programs around and until such applications are developed it is not possible to give anything other than an approach based on relational type database queries. Once a larger number of very large prolog application programs become available it will then also be possible to develop a **dhrystone** benchmark program. This will require both a static analysis [20] and a dynamic analysis of the programs.

7. THE BENCHMARK PROGRAMS

The full set of benchmark programs described in this chapter are available from the Computer Science Department at Heriot-Watt University.

ACKNOWLEDGEMENTS

The authors wish to acknowledge the financial support given by the Science and Engineering Research Council and by ICL under the Alvey initiative. We would also like to acknowledge the patience and advice given by the members of the Logic Database, Prolog Database Machine and Parlog Groups at Heriot-Watt University.

REFERENCES

[1] Furukawa, K., *et al.*, Problem Solving and Inference Mechanisms, *Proc. Int. Conf. on Fifth Generation Computer Systems*, Japan Information Processing Development Centre, October 1981.

[2] Treleaven, P., Japan's Fifth Generation Computer Systems Project, *Proc. Fifth Generation Conf.*, SPL International, London, 7–9 July 1982, 0–23.

[3] Moss, C., Prolog Benchmarks, Internal Report, Imperial College, London, 1986.

[4] Wilk, P. F., Prolog Benchmarking, Report AIAI-TR-14, AI Applications Institute, Edinburgh University, 1986.

[5] Wilk, P. F., The Production and Evaluation of a Set of Prolog Benchmarks, AIAI-TR-13, 1983.

[6] Syre, J. C., Benchmark Programs for Prolog Systems, Internal Report, European Computer-Industry Research Centre, Munich, 1986.

[7] Dickens, S. and Bustany, A., *Alvey Prolog Benchmarks*, Imperial Software Technology, Cambridge, 1986.

[8] Kowalski, R. A. Logic as a Database Language, Internal Report, Department of Computer Science, Imperial College, 1981.

[9] Neves, J. C., Anderson, S. O., and Williams, M. H., A Prolog Implementation of Query-by-Example, *Proc. 7th Int. Computing Symp.* B. G. Teubner, Stuttgart, 1983, 318–332.

[10] Williams, M. H., The Design of a Prolog Database Machine, *Proc. First SIGKME Workshop*, January 1987.

[11] Report on the Alvey Architecture Club's Collection of Benchmark Programs, *Proc. First SIGKME Workshop*, S. Lavington (ed.), IEE, January 1987.

[12] Wilk, P. F., Chat-80, Internal Report, Edinburgh University, 1980.

[13] Rawlings, C. J., Seifert, K., and Saldanha, J., A Large Knowledge Based System for Molecular Biology, *Proc. First SIGKME Workshop*, S. Lavington (ed.), IEE, January 1987.

[14] Thom, J. A., Ramamohanarao, K., and Naish, L., A Superjoin Algorithm for Deductive Databases, Technical Report 86/1, Depart-

ment of Computer Science, University of Melbourne, Australia, 1986.
[15] Overbeek, R. A. and Lusk, E. L., *Prolog Benchmark 1: Accessing Databases,* Argonne National Laboratory.
[16] Cole, A. J., A Note on Space Filling Curves, *Software — Practice and Experience,* **13**, 1151–1159, 1983.
[17] Witten, I. H. and Wyvill, B., On the Generation and Use of Space Filling Curves, *Software — Practice and Experience,* **6**, 519–525, 1983.
[18] Wirth, N. *Algorithms & Data Structures*, Prentice-Hall, Englewood Cliffs, NJ, 1986.
[19] Williams, M. H. and Nicolson, K. A., An Approach to Handling Incomplete Information in Databases, to appear in *The Computer J.*
[20] Matsumoto, H., A Static Analysis of Prolog Programs, Programming Systems Group, Note 24, AI Applications Institute, Edinburgh, 20 January 1985.

APPENDIX 1

```
/*********************************************************/
/* This is a recursive C program to create a prolog data */
/*      file containing points visited in the production */
/*      of a "Hilberts Space Filling Curve".             */
/* The Algorithm is from N.Wirth "Algorithms & Data      */
/*     Structures".                                      */
/* ***************************************************** */
/* This program produces data in two forms:              */
/*                                                       */
/*   form 1: vertex/4 rules as follows-                  */
/*           vertex(VName,X,Y,Z):-                       */
/*                      range(Z,Z_upper,Z_lower).        */
/*     and:                                              */
/*           edge(EName,VName1,VName2).                  */
/*                                                       */
/*           The middle clauses '00' has its range fudged*/
/*           to allow various tests to be performed.     */
/*                                                       */
/*   form 2: vertex/3 unit clauses as follows-           */
/*           vertex(VName,X,Y).                          */
/*     and:                                              */
/*           edge(EName,VName1,VName2).                  */
/*                                                       */
/*   It is called (once it has been compiled as "hilbert")*/
/*         hilbert <data-form> <curve-number>            */
/*   where:                                              */
/*         data-form    is 1 or 2 (for forms 1 or 2).    */
/*         curve-number is the maximum sized curve       */
/*                      number (2-10).                   */
/*                                                       */
/*********************************************************/

#include <math.h>
#include <stdio.h>
#include <ctype.h>

#define  A  0                /* Only for clarities sake */
#define  B  1
#define  C  2
#define  D  3

FILE  *data1,*data2;          /* output data files       */
int   X,Y;                    /* X + Y positions         */
int   Type;                   /* Type of Data required   */
int   ins[4][7] = {{D,2,A,3,A,0,B},
                   {C,1,B,0,B,3,A},
                   {B,0,C,1,C,2,D},
                   {A,3,D,2,D,1,C}};


/*******************************************************/
/* The Following procedure actually produces the curve */
/*     deciding where to go next.                      */
/*******************************************************/
```

```
proc(Instruction,I,U)
  int   Instruction,I,U;
{
 extern int ins[][7];
 int        i;
 if (I>0)
    {for(i=0;i<=5;i=i+2)
        {proc(ins[Instruction][i],I-1,U);
         move_xy(ins[Instruction][i+1],U);}
     proc(ins[Instruction][6],I-1,U);};
}

/*************************************************/
/* Move to the next point on the "screen" + output */
/*     the prolog clauses representing points that */
/*     have been visited                           */
/*************************************************/

move_xy(Rotation,U)
  int   Rotation, U;
{
 extern int  X,Y,Type;
 extern FILE *data1,*data2;
 int         Old_x,Old_y;
 int         Z_Upper,Z_Lower;

 Old_x=X;
 Old_y=Y;
 switch(Rotation) {
   case 0: X = Old_x + U;
           break;
   case 1: Y = Old_y + U;
           break;
   case 2: X = Old_x - U;
           break;
   case 3: Y = Old_y - U;
           break;};

 if (Type == 1)
   {/* vertex(X,Y,Z). - dataset */
    fprintf(data1,"vertex('%d%d',%d,%d).\n",X,Y,X,Y);
   }
 else
   {/* Generate Z co-ordinate information */
    /* Z_Upper always higher than Z_lower */
    if ((X==0) && (Y==0))
       {/* fudge values for test !!*/
        Z_Lower = -10;
        Z_Upper = 0;}
    else
       {Z_Lower=(rand() % 1000);
        Z_Upper=(rand() % 100000)+1000;};
    fprintf(data1,"vertex('%d%d',%d,%d,Z):-range(Z,%d,%d).\n",
                                          X,Y,X,Y,Z_Upper,Z_Lower);
  };
 fprintf(data2,"edge('%d%d%d%d','%d%d','%d%d').\n",
                                     Old_x,Old_y,X,Y,Old_x,Old_y,X,Y);
}
```

```
/***                          MAIN PROGRAM                          */

main(argc,argv)
  int argc;
  int *argv[];
{
 int          U,I,Width=20000,Max_Curve;
 extern FILE  *data1,*data2;
 extern int   X,Y,Type;
 char file1[8],file2[8];

 if (argc != 3)
    {printf("Incorrect number of arguments\n");
     exit(-1);};

 sscanf(argv[1],"%d",&Type);
 if (Type != 1 && Type != 2)
    {printf("\nUnknown Data Type number\n");
     exit(-1);};
 sscanf(argv[2],"%d",&Max_Curve);
 if (Max_Curve<2 || Max_Curve>10)
    {printf("\nCurve number out of range\n");
     exit(-1);};

 printf("\nProducing data...\n");

 for(I=2;I<=Max_Curve;I++)
    {/* Produce Data files */

     sprintf(file1,"%s%d","data1.",I);
     sprintf(file2,"%s%d","data2.",I);

     data1=fopen(file1,"a");       /* Open Output datafiles */
     data2=fopen(file2,"a");

     U= (int) ((float) Width/(float) (pow((double) 2,(double) I)-1));
     X= Y= (int) ((Width / U)*U);

     /* Output Start Point */

     if (Type == 1)
        fprintf(data1,"vertex('%d%d',%d,%d).\n",X,Y,X,Y);
     else
        fprintf(data1,"vertex('%d%d',%d,%d,Z):-range(Z,%d,%d).\n",
                        X,Y,X,Y,((rand() % 100000) + 1000),(rand() % 1000));

     proc(A,I,U);                  /*        Produce Curve        */

     fclose(data1);
     fclose(data2);
    };
 printf("\nCompleted\n\n");
}
```

APPENDIX 2

The generator is called as follows:

dg(Name,Format List,Uniqueness List,Clause).

where the arguments are:

Name — this is the name/functor of clause to be generated.

Uniqueness List — this is a list of integers which gives the 'uniqueness' of the clauses arguments. This is effectively a range in which the pseudo-random number generator will guarantee to generate unique numbers (and from that unique arguments).

Clause — this is the return argument which will be unified with the clause generated.

Format List — this consists of an arbitrary length list which gives the *type* of the clause arguments, the various format types allowed by this data generator are:

- integers — this is defined by *int* appearing in the argument type list.
- constants — this is defined by *const* appearing in the argument type list.
- functors — this produced structures of the forn 'a(b)', 'a(b,c)' depending on the required format. The format is of the form
  funct(⟨type⟩,⟨type⟩,..,⟨type⟩)
- lists — this type is used to generate a list, the elements of which are of form ⟨type⟩. The format of this is
  [⟨type⟩,⟨type⟩,..,{⟨type⟩|tail}]
  If the last element type is 'tail' then an open list is generated. There is also a form of variable length list which can be generated. The defining format has the form
  vlenlist(⟨type⟩,⟨length⟩,⟨variation⟩,
  {tail|notail})
  The arguments have type ⟨type⟩. The mean arity is given by ⟨length⟩ and the variation on this arity is ⟨variation⟩. The fourth argument is *tail* for an open list and *notail* for a closed list.
- structures — this generates a structure with a variable number of arguments. The arguments are as for the variable length list format. The call format has the form
  vlenfn(⟨type⟩,⟨length⟩,⟨variation⟩)

APPENDIX 3

```
%%%
```

```
marker(_,_,_,_).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Main Clause

dg(System,Name, ArgList, RangeList, ClauseType,Clause_out) :-
      do_args(System,ArgList, RangeList, Args),
      ClHead =.. [Name|Args],
      do_body(System,ClauseType, Name, Args, ClBody),
      (ClBody == [],
         Clause_out=ClHead;
       Clause_out=(ClHead:-ClBody)),!.

do_args(System,ArgList, RangeList, Args) :-
      do_args(System,ArgList, RangeList, _, Args).

do_args(System,[], R, R, []) :- !.

do_args(System,[var|Ts], Rs, Rs1, [_|Alist]) :-          % variables
      do_args(System,Ts, Rs, Rs1, Alist), !.

do_args(System,[int|Ts], [Range|Rs], Rs1, [I|Alist]) :-  % integers
      random(System,Range,I,_),
      do_args(System,Ts, Rs, Rs1, Alist), !.

do_args(System,[const|Ts], [Range|Rs], Rs1, [C|Alist]) :- % constants
      random(System,Range,I,_),
      int_to_const(System,I,C),
      do_args(System,Ts, Rs, Rs1, Alist), !.

do_args(System,[[HTTT]|Ts], Rs, Rs1, [L|Alist]) :-       % lists
      do_args(System,[HTTT], Rs, Rs2, L),
      do_args(System,Ts, Rs2, Rs1, Alist), !.

do_args(_,[tail], Rs, Rs, _).     % special end-of-list type

do_args(System,[Struct|Ts], Rs, Rs1, [S|Alist]) :-   % structures
      Struct =.. [funct|ArgTypes],
      do_args(System,[const|ArgTypes], Rs, Rs2, Slist),
      S =.. Slist,
      do_args(System,Ts, Rs2, Rs1, Alist), !.

                        % variable length lists
do_args(Sys,[vlenlist(Type,Avelen,Var,Withtail)|Ts],[Range|Rs],Rs1,[L|Alist]) :-
      random(Sys,_,R,N),
      Len is Avelen + ((Var * N)/10),
      mklist(Type, Len, AL),
      mklist(Range, Len, RangeList),
      ( Withtail == tail, append(AL, [tail], ArgList) ;  ArgList = AL ),
      do_args(Sys,ArgList, RangeList, L),
      do_args(Sys,Ts, Rs, Rs1, Alist), !.

                        % var length structures
do_args(System,[vlenfn(Type,Avelen,Var)|Ts],[FRange,ARange|Rs],Rs1,[S|Alist]):-
      random(System,_,R,N),
      Len is Avelen + ((Var * N)/10),
      mklist(Type, Len, ArgList),
      mklist(ARange, Len, RangeList),
```

```
    do_args(System,[const|ArgList], [FRange|RangeList], Slist),
    S =.. Slist,
    do_args(System,Ts, Rs, Rs1, Alist), !.

do_body(_,empty, _, _, []) :- !.
do_body(_,literal(X), _, _, X) :- !.
do_body(_,recursive, Name, Args, ClBody) :-
    ClBody =.. [Name|Args], !.

do_body(_,_, _, _, []) :- !.

mklist(_, N, []) :-
    N =< 0.
mklist(Elt, N, [Elt|L]) :-
    N1 is N - 1, mklist(Elt, N1, L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% from(Lower,Upper,Value) Count from Lower to Upper on failure -
%                         following Wilk's paper.

from(I,I,I):-!.

from(L,N,I):-
  N1 is (L+N)//2,                % C-Prolog
% N1 is (L+N) div 2,               % Nip 1.5
  from(L,N1,I).

from(L,N,I):-
  L1 is (L+N)//2+1,              % C-Prolog
% L1 is (L+N) div 2+1,             % Nip 1.5
  from(L1,N,I).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Random Number Generator:       Uses Integer Arithmetic Only

random(System,Range,Num,_):-          % Generate Number in Range
    nonvar(Range),
    new_seed(Tmp),
    i_div(System,Tmp,16384,T1),
    T2 is Tmp mod 16384,
    i_div(System,(Range*T1),32,Num1),
    i_div(System,(Range*T2),524288,Num2),
    Num3 is Num1 + Num2,
    i_abs(Num3,Num),!.          % So that a Positive number always rtn.

random(System,Range,Num,Deviation):-  % Generate Number+Deviation*10
    var(Range),
    new_seed(Tmp),
    i_div(System,Tmp,16384,T1),
    T2 is Tmp mod 16384,
    i_div(System,(10000*T1),32,Num1),
    i_div(System,(10000*T2),524288,Num2),
    Num3 is Num1 + Num2,
    i_abs(Num3,Num),          % So that a Positive number always rtn.
    ntable(Deviation,Area),
    Area >= Num,!.

% Set Seed Value
```

```
reset_seed:-
      retract(seed(_)),
      retract(cycle_start(_)),
      asserta(seed(12345)),
      asserta(cycle_start(12345)),!.

% Generate a new seed value

seed(12345).
cycle_start(12345).          % Start of Cycle Value

new_seed(New_val) :-
      old_seed(X),
      New_val is (125 * X) mod 524288,
      assert(seed(New_val)),!.

old_seed(Seed):-
      seed(Old_seed),
      retract(cycle_start(Old_seed)),
      retract(seed(Old_seed)),
      Seed is Old_seed + 13,
      assert(cycle_start(Seed)),!.
old_seed(Seed):-
      retract(seed(Seed)),!.

% a normal distribution table.

ntable(-40,   0).
ntable(-35,   4).
ntable(-30,  13).
ntable(-25,  62).
ntable(-20, 228).
ntable(-15, 668).
ntable(-10, 1587).
ntable(-5 , 3085).
ntable( 0 , 5000).
ntable( 5 , 6915).
ntable( 10, 8413).
ntable( 15, 9332).
ntable( 20, 9772).
ntable( 25, 9938).
ntable( 30, 9987).
ntable( 35, 9996).
ntable( 40, 9999).

%%%
```

```
%%%
```

37. K.F.WONG and M.H.WILLIAMS, Limiting a resolution set in a large Prolog database using special hardware, Proceedings of the ACM SIGSMALL/PC Symposium on Personal and Small Computers, Cannes, France, 133-145 (1988).

Limiting A Resolution Set In A Large Prolog Database Using Special Hardware

K.F. Wong and M.H. Williams

Prolog Database Machine Group
Heriot-Watt University
79, Grassmarket
Edinburgh EH1 2HJ.

Abstract

There is a growing interest in the application of Prolog and related languages like Parlog to database implementation. Existing Prolog systems, however, are unable to cope with large sets of data of the size necessary for databases. A large Prolog database would consist of millions of clauses. The size of physical memory required to store the database has deemed main memory impractical in many computer systems. Therefore clauses are usually placed on secondary disc memory. Clause retrieval, however, becomes extremely slow. Furthermore, the enormous amount of time required for unification upon a large set of clauses has drastically degraded system performance. Either of these factors leads to non-timely database response which is irritating and in many cases, intolerable. To mediate these problems, a special hardware unit is proposed - the CLAuse Retrieval Engine (CLARE). The functions of the CLARE are to search discs storing database clauses on-the-fly and select potentially unifiable clauses. Two stages of filtering will be applied to the database clauses; thus, a small resolution set will be guaranteed.

1. Introduction

One of the application areas of Prolog and related languages like Parlog is database implementation. The functionality and data structures of Prolog have been demonstrated to be well suited for realising conventional database concepts[1,2] At present, there are, however, two main limiting factors which have been restricting Prolog to implementing only small databases. They are

1) slow clause retrieval rate from disc

2) poor system performance caused by the time consuming unification process.

A Prolog database system is proposed† by our research team and is currently under construction. Our system is specially designed to execute a large Prolog database efficiently. This is achieved by mediating the limiting factors using both software and hardware techniques. In this document, the hardware design of our database system is described. A detailed overview of our project can be found in [4].

† A Parlog database system has also been planned [3] Initially, the database under Parlog will be concurrently interpreted in a sequential implementation by resource sharing. Upgrades of the system to true parallelism are anticipated in the future.

1.1. Section Summary

The problems of Prolog in database applications have been identified. A brief description of the solutions adopted by our system for improving clause retrieval and unification times is given in section two. A dedicated hardware engine based on these solutions has been designed. Its architecture and functionality are explained in section three and four. Finally, in section five, possible implementations of the design are suggested.

2. Our Prolog Database - A Brief Overview

The storing of huge sets of clauses in database applications using Prolog would render primary memory impractical. The use of secondary disc memory would, inevitably, create a slow rate of clause retrieval. In order to improve the clause retrieval rate from disc, a one level indexing approach will be adopted in our database system. Each clause within our database will be encoded. Clause retrieval will consist in searching through encoded words, extracting the associated clause references from matched codewords and using them to access clauses directly. Under the chosen indexing scheme, encoded words are calculated to be one tenth of the size of the clauses on average. Therefore, comparing with unindexed databases, a possible ten fold improvement in clause retrieval rate is conceivable in our system. In addition to this, indexing will function as a clause retrieval filter (see later).

Another governing factor which seriously affects overall system performance in Prolog systems is the time required for unification. In database applications, unification is the fundamental operator for management functions, such as select, join and merge. In order for a Prolog database system to be efficient, the time required for unification has to be minimised. There are two means of achieving it, either by speeding up the unification process or by reducing the number of clauses to be unified. Both means will be exploited in our system. For fast unification, database clauses will be stored in compiled form. The advantages of compilation, as contrasted to interpretation, are well known for conventional languages, and compilation is no less of an advantage for Prolog. An interactive compiler-based Prolog system, Prolog-X [5,7] is used for our database. Under Prolog-X, clauses are compiled incrementally and the compiled codes are clustered according to clauses' names and arities. Clustering expedites unification by moving the processor to the neighbourhood of potentially unifiable clauses and confining the processor to performing unification on a selected set of clauses only. This effectively reduces the number of clauses that have to be searched among using unification. The search space for database clauses within a cluster will be further reduced by a 2-stage filtering mechanism. The 2-stage filter comprises indexing followed by preunification.

Index searching and preunification will be realised using custom hardware. The hardware is designed to operate in parallel and at real time, searching indices on-the-fly as they flow out from a disc.

3. The Clause Retrieval Processor

The special hardware unit (CLARE[†]), CLAuse Retrieval Engine will filter clause information from a large Prolog database, residing on discs, at maximum speed. In so doing only a small resolution set will be returned to the host for the final full unification. Filtering will be perform in two stages: indexing and pre-unification. Indexing based on superimposed code word plus mask bit (SCW) will be used to achieve partial matching. It will selectively locate all clauses within the database which are likely to unify with the query goal. On average, it can reduce the search space substantially (<0.1% of the original database). Nevertheless, this could still result in matches with thousands of clauses! The pre-unification stage will be incorporated to refine the search space further, possibly by reducing the search space by a factor of ten.

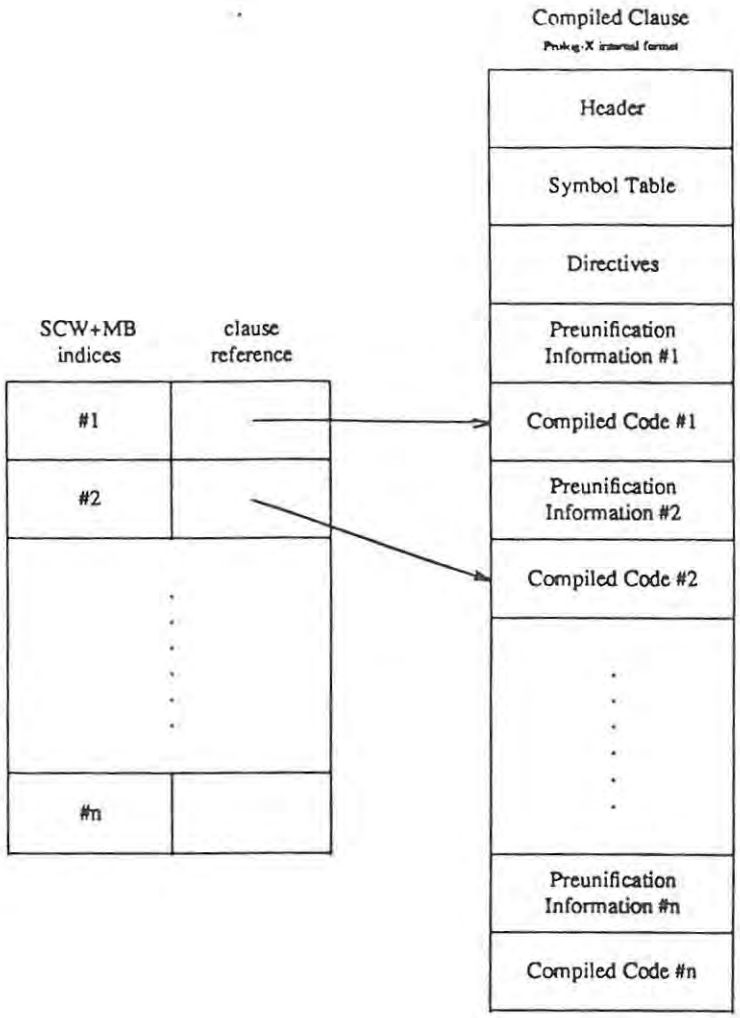


Figure 1: The CLARE Searchable File Organisation

The two filtering stages will operate multaneously in a streamline fashion. Indexing will ways be at least one clause ahead of pre-unification hich, in turn, will be at least one clause ahead of full ification in the host. This will be facilitated by two ffers sitting between the indexing/pre-ification/full-unification stages, collecting ermediate results.

The function of the CLARE will assume the llowing system organisation (figure 1): there will two separate files on two disc drives. Both discs ll have their own disc control units which will be ntrolled by the CLARE independently. One file will contain the indices and their associated clause references - the INDEX file; and the other will contain the compiled code of the database modules - the CLAUSE file. The INDEX file will be arranged, so that the clause references are in ascending order. For each entry in the INDEX file, the index will exist before the clause reference. The internal format of the CLAUSE file will be similar to PROLOG-X compiled code which is comprised of four sections : the header, the symbol table, the directives and the clause code. The clause code will be stored serially with respect to the INDEX file entries. Some pre-unification information will be included at the beginning of each clause. This could be the types of individual arguments, their hash values and their symbol table references.

bbreviations adopted in this document is listed in the appendix.

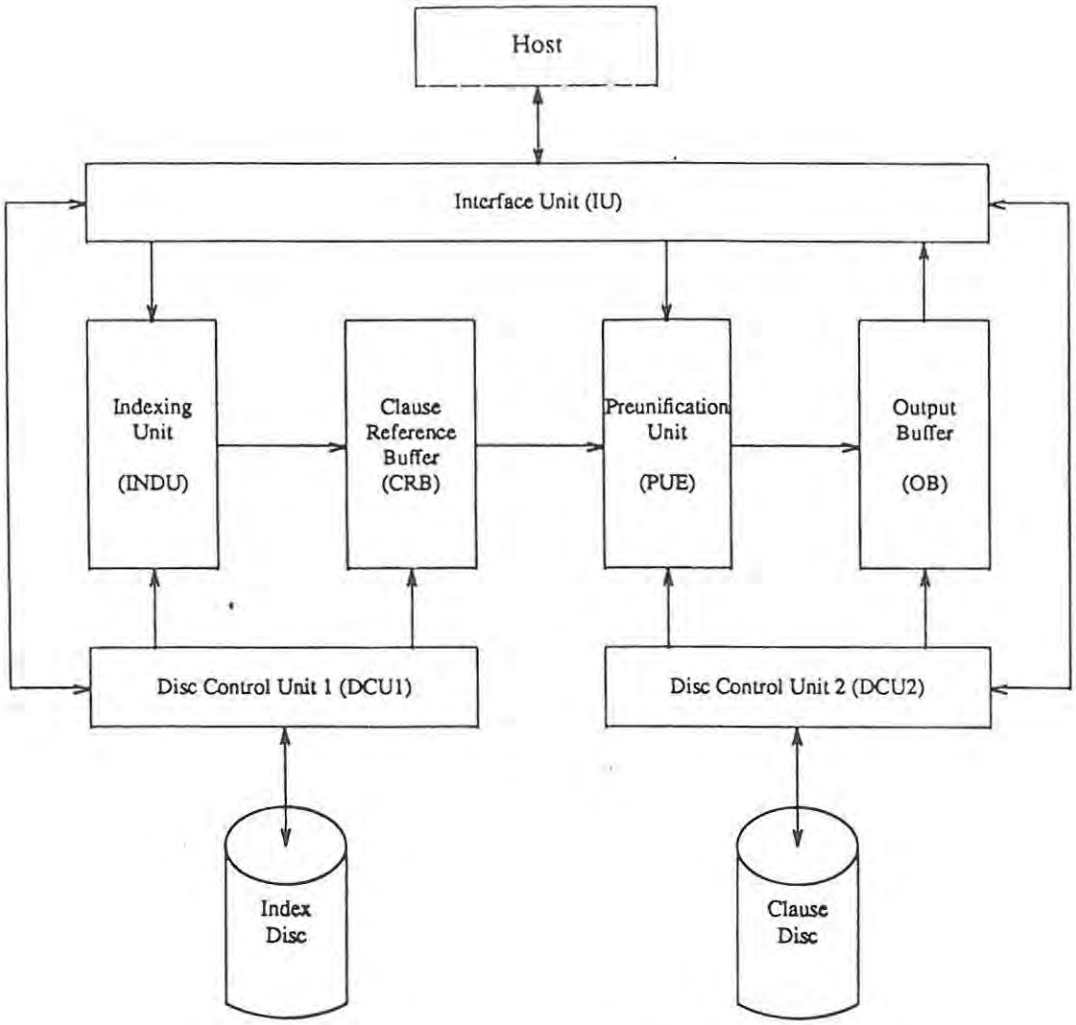


Figure2: The Overall Schematic Diagram of the CLARE Architecture.

3.1. The CLARE Architecture

The CLARE will sit between the host and the two disc drives. It will be under microprogram control. A software Clause Retrieval Server (CRS) in the host will be responsible for translating a query into the corresponding microprogram sequence to drive the CLARE hardware.

Internally, the CLARE will be divided into six functional units (Figure 2):

1) The Interface Unit (IU) will provide for communication between the host and the CLARE. It will contain some bidirectional registers with decoding logics directing traffic under a pre-defined protocol.

2) The INDexing Unit (INDU) will receive the superimposed code word plus mask bit of a query, use it as the base to match against the indices coming from the INDEX file disc and generate the appropriate control signals for the next stage.

3) The Clause Reference Buffer (CRB) will be a register file containing the references of the clauses which will have been reported as matches by the INDU.

4) The Pre-Unification Engine (PUE) will read in the clause references from the CRB, use them to extract clause code from the CLAUSE file disc and perform pre-unification.

The Output Buffer (OB) will store the entire block of a compiled clause which has satisfied the pre-unification criteria and make it ready to be sent to the host.

) The Disc Control Unit (DCU) will contain two microcoded disc controllers which will transfer information to the other CLARE units.

. The Operations of CLARE

1.1. Initialisation

In order to run a database, the file module containing the database will have to be explicitly loaded (using the Prolog-X's LOAD(database_module) command). Three types of database modules will be accounted for, depending on the number of clauses constituting each module: small (<5k), medium (5-100k) and large (>100k). The type of a module will be specified in the database header section. The CLARE will focus on large databases where the module is too huge to be completely transferred into the main memory. In this case, during a load module, only the symbol table will be read in from the disc†. A slight delay during system startup will have to be tolerated because of the time required for transferring the symbol table (estimated to be ~1M bytes for 100k clauses).

When a query is posed, it will be passed to the CRS. The CRS will interpret the query and generate two set of microinstructions. They will be loaded into the writable control stores of the INDU and the PUE separately. These microinstructions will govern the execution and control sequences of the units. In addition, the starting addresses of the two discs will be supplied. Using them the DCUs will move the disc heads into the appropriate locations.

† Sufficient physical memory will be required to store the symbol table.

4.2. Indexing

The Superimposed Code Word plus mask bit (SCW) indexing [6] approach will be used. Under this approach, each database clause head is encoded. Encoding is based on OR-ing (superimposing) individual code words of all attributes within a clause. An attribute code word is formed by setting k bits to 1 in a word of b bits width. The exact positions of 1's are calculated using a hashing function. Adjacent to the code word some masked bits are attached whose bit pattern shows the variable positions amongst the attributes. When a query is made, the arguments of the query are encoded using the same hashing function. A database code word is then compared with the first argument code of the query. The result is successful if either one of the following is true:

1) The query argument code is a subset of the database code word.

2) The mask bit of the database clause indicates that this is a variable position.

A database clause is considered to be a match if it compares successfully with all query arguments. Program 1 is a pseudo-C program describing the SCW matching operation.

The matching operation will be performed by the INDU hardware. The INDU hardware will be divided into two subunits: the control unit and the matching unit. The control unit will include a Writable Control Store (WCS), a microprogram sequencer and some testing circuitry for monitoring conditional signals. The control flow of a microprogram will be determined by these signals.

```
#define TRUE 1

for(;all database clauses;)
  for(;all query argument codes;)
    {
      if ( ( argument code is a subset of database code ) ||
           ( the corresponding mask bit is set ) ) return( TRUE );
    }
```

Program 1: Pseudo-C Program For SCW Partial Matching.

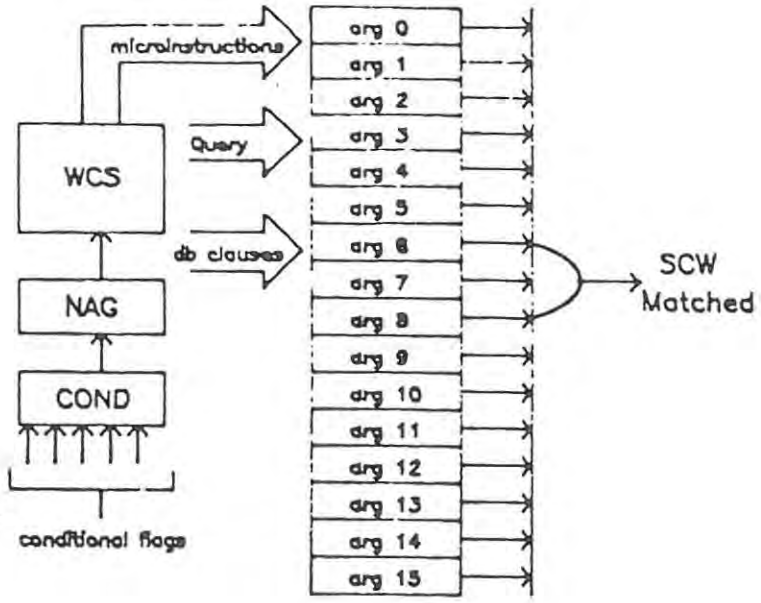


Figure 3a: The Indexing Unit (INDU) Hardware.

As shown in figure 3a, the matching unit of the INDU hardware consists of 16 identical modules. Their outputs are AND'ed together in order to produce the final result. Each module is responsible for making a comparison between the database code and a query argument. The position of the module is relevant because it corresponds with an argument position in the query. Within a module (figure 3b) there are six components : two 1 bit latches (IN and OUT), a Parallel In Serial Out (PISO) shift register, a 16-bit latch and two logic gates (an AND and a Subset‡). The first three items will function synchronously under a system clock. The IN latch will be rising edge triggered and the OUT latch will be triggered at the falling edge of the clock. The triggering sequence will ensure that all data is stable prior to comparison.

During initialisation the argument codes of a query will be loaded into their respective PISO shift register in parallel. At the same time, the 16 OUT latch will be preset (output 1) and the mask bit will be transferred into the 16-bit latch. Each bit of the 16-bit latch will then be used to control the preset pin of the OUT latch; so that the attribute positions which contain variables will be locked to '1's permanently. For those positions which are not variables, matching will be performed in a bit serial manner. The system clock will have to be faster (>2×) than the disc transfer speed. As a bit appears, it will be gated into the IN latch and at the same time the PISO will be shifted by one bit position. These two bits will be compared through the Subset logic gate which will produce a '1' if they are the input bit is a subset of the PISO output. This will be AND'ed with the output from the OUT latch which will contain the result of the previous cycle. Initially, the OUT latch will have been preset, therefore, the output of the AND gate will follow favourably to the output of the Subset gate. At a later stage, when a database code bit disagrees with a query code bit, a '0' will be permanently latched

‡ A Subset B = A OR B, which will be implemented using PLA.

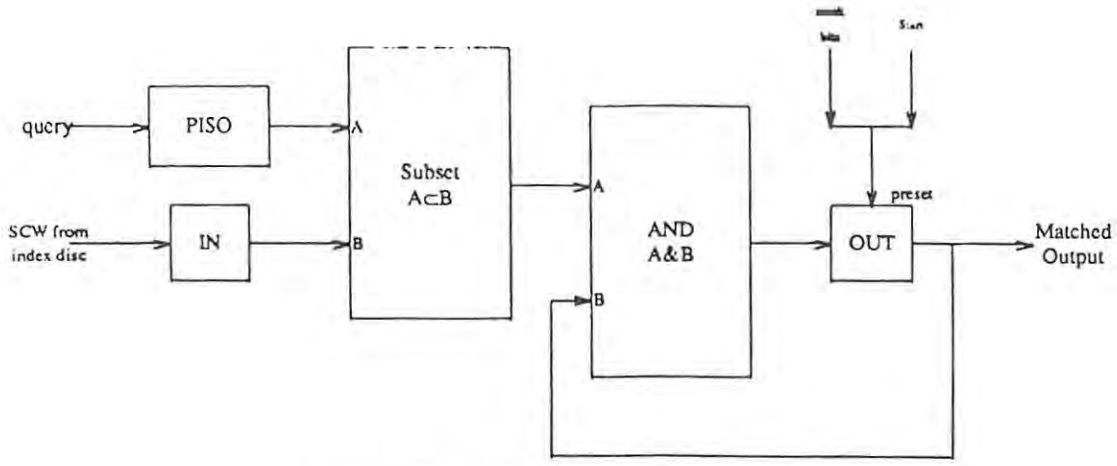


Figure 3b: An INDU argument Module.

OUT and from then on the AND gate will stantly output '0' until the OUT latch is reset again ore the start of the next database code. The results n all 16 OUT latches will be fed into a 16 input D gate. The database code word will be a match if ' is the result. If it is not a match, nothing will be e. The matching unit will then await the earance of the next index and the matching cycle l be restarted. If it is a match, however, then the B will be enabled which will permit the associated ase reference following to be stored.

. Clause Reference Buffering

The CRB will consist of several Serial In Parallel t (SIPO) registers and two special counters. The ct number of registers required will have to be ided. There will, however, be at least three isters; each of them will be big enough to contain lause reference and they will be clocked to the DEX disc controller. One counter, CNT1, will tain the number of the register in which the AUSE disc address currently operated on by the E is held. It is incremented after the clause is shed. The other counter, CNT2, will contain the nber of the next free register in which the next W matched clause disc address will be stored. The ue in the CNT1 will never exceed that of the T2. The third counter, CNT3, will contain the INDEX disc address of the clause which is next to be pre-unified. All three registers will initially be set to zero. Normally, the PUE will constantly be sampling a CLAUSE_AVAILABLE (see later) conditional flag. This flag will be disabled during startup and whenever the values of the two counters are the same and it will be set by the INDU as soon as the SIPO register, referenced by the CNT2, is filled. The PUE will then fetch the CLAUSE disc address which the register of the CNT1 contains and increment the CNT1. The values of the two counters will be compared at this time and the result will alter the state of the CLAUSE_AVAILABLE conditional flag.

4.4. Pre-Unification

The SCW indexing technique is only for partial matching. All matches produced from this method may not be definitely unifiable; nonetheless, it can effectively reduce a large database space to a manageable size. On average, the reduction should be as good as ~0.1% of the database. This, however, could give rise to thousands of clauses. In order to reduce the search space further (possibly by another factor of 10), a pre-unification stage has been introduced. The pre-unification algorithm/system is specially designed and has the following characteristics:

1) Pre-unification is a comparison of the types and the symbol table references of a query clause's arguments with a database clause head's arguments.

2) Lists and structures are only dealt with superficially. For lists, its type and the number of elements are tested; for structures, the functor name and its arity are checked. (This is mainly due to speed constraint).

3) Bound variables are accounted for. Some internal memory is dedicated for testing binding of variables (both database and query). Common variables are arranged to share the same memory location.

4) It is not full unification and it only provides a quick decision as to whether a database clause should be retrieved.

5) Operational speed is essential. The process is completely overlapped with data transfer from the CLAUSE disc. In the worst case, the process would be accomplished prior to the last byte of the database clause needing to be transferred. Thus the process is virtually transparent to the disc controller.

6) It favours a high false drop rate. The decision from preunification can terminate a disc transfer at any point. Therefore, the earlier the database and the query arguments disagree the quicker the preunification process.

7) A pre-unifiable clause does not imply a definite hit. If a false drop has been encountered, the pre-unification process can immediately restart and try the next SCW matched clause.

Pre-unification will be realised in hardware/firmware in the PUE. The PUE will, basically, be a customised bit slice microprocessor. It will consist of four functional parts (cf figure 4): the microprogram controller (or controller), the type comparator (TC), the dual ported memory (DRAM) and the processor (PROC).

4.4.1. The Microprogram Controller

The microprogram controller will have a next address generator (NAG), a CONDitional flag selector (COND) and a WCS. The function of the microprogram controller will be to provide the microinstruction addresses to the WCS. This is affected by the status of the conditional flags and the exception/interrupt signals generated by the other units. A conditional flag is a bit of information describing an operation. There will, at least, be the following conditional flags :

1,2) NOT_UNIFIABLE, UNIFIED - these signals will be asserted by the host if the current database clause head is (not) fully unified with the query clause.

3,4) NOT_PRE-UNIFIABLE, PRE-UNIFIED - these bits will be set if the current database clause fails (passes) the pre-unification test.

5,6) UNEQUAL_TYPE, EQUAL_TYPE - these bits will be set by the TC if the the types of the current arguments of the database clause and the query clause (dis)agree.

7,8) QUERY_VARIABLE, DATABASE_VARIABLE - these signal will be generated by the TC to indicate that the QUERY and the DATABASE arguments are variables.

9) OB_FULL - this will indicate that the Output Buffer (OB) is exhausted.

10) CLAUSE_AVAILABLE - this is generated by the CRB when the values of the two CRB counters are equal. Physically, it indicates the availability of a new database clause ready to be processed by the PUE.

A mechanism for subroutine calls will be available which is facilitated by the internal structure of the NAG. A 16-level LIFO memory inside the NAG allows stacking of 16 return addresses. During initialisation, the microprogram sequence will be

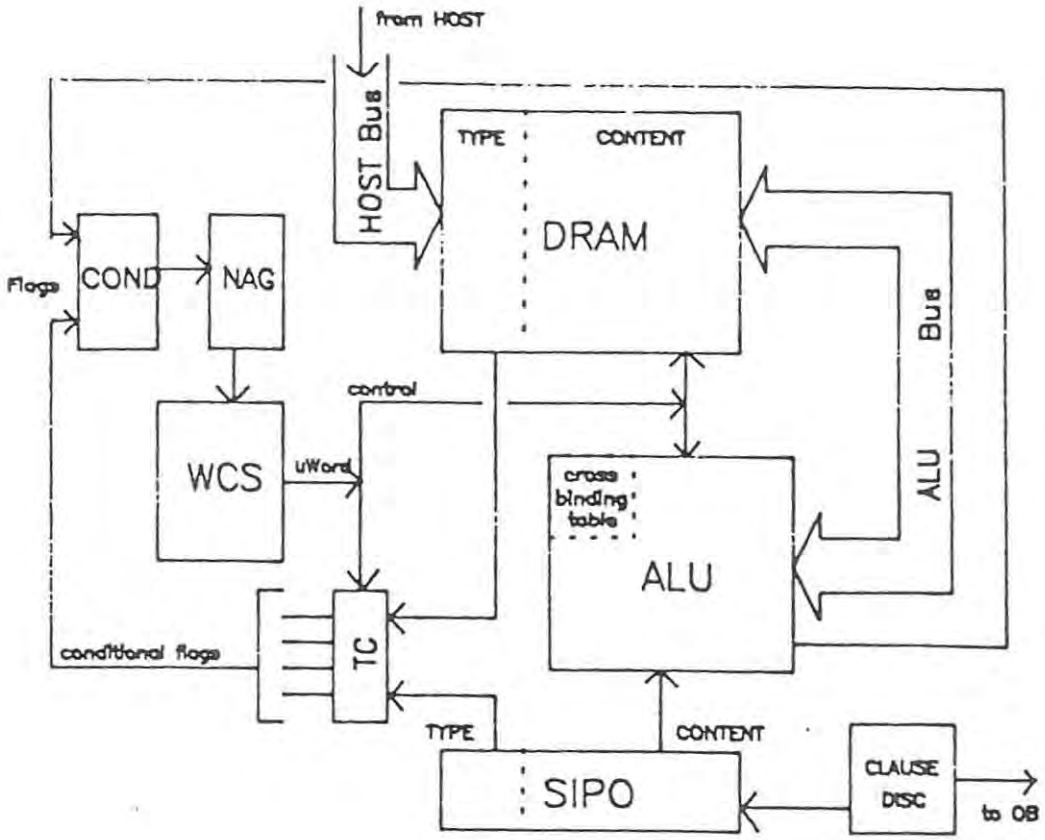


Figure 4: The Pre-Unification Engine (PUE) Hardware.

loaded into the WCS, all flags will be disabled and the NAG will be reset.

4.4.2. The Type Comparator (TC)

Pre-unification will be applied to all arguments sequentially. It is a two stage process. In the first stage, the database and query arguments' types will be checked. If they are compatible, the second stage will proceed and compare the contents of the arguments. This two stage execution sequence will be best realised in hardware using a pipeline technique. An overlapped type-checking architecture will be used. Effectively, the entire PUE hardware will be divided into two operational segments : the TYPE and the CONTENT. There will be a segmented SIPO register in the PROC which is responsible for collecting the database arguments from the CLAUSE disc. Segmentation will allow the type and the content of an argument to be read independently. The type will appear before the content during an argument transfer. The type part of the SIPO will be connected to one input of the TC. The other input of the TC will be taken from the type part of the DRAM. The DRAM will be set, at the initialisation stage, with the arguments of the query clause. Each argument in the DRAM will be referenced by the microprogram. As soon as the type of the SIPO is filled, a type comparison will be initiated. While the comparison is taking place the content of the database argument will be read into the rest of the SIPO register. The comparison at the TC will generate four possible results : UNEQUAL_TYPE, EQUAL_TYPE, QUERY_VARIABLE and DATABASE_VARIABLE. The last three results will be positive and the checking of the contents will proceed in the PROC. If the types disagree, the complete database clause will be condemned : the disc transfer operation will be terminated instantly and the various parts of the PUE will be reset. After this, the the PUE will attempt to fetch the next clause from the disc using the reference supplied by CNT1 in the CRB. If, however, CLAUSE_AVAILABLE is not

asserted, the PUE will cycle in a polling loop until the next clause is available.

4.4.3. The Dual-ported Random Access Memory (DRAM)

The DRAM will have two buses : one connected to the HOST and the other to the PROC. Each bus will consist of a bidirectional data port and an address port. Both the HOST and the PROC will be able to read/write to/from the DRAM independently. The microprogram will be arranged in such a way that contention is avoided.† In practice, the HOST bus will, only, be connected to the host at the initialisation stage. During which, the query arguments will be loaded. The DRAM will be allocated sequentially. This allocation will be done by the CRS. Each memory location will be associated with a query argument and the mapping will be recorded. This mapping information will later be used to generate the microprogram.

All query arguments will be looked up through the symbol table (this is assumed to have been read into the main memory during a LOAD(database_module)). If an argument is neither a variable nor a list and it exits in the symbol table , the associated disc reference will be transferred to the DRAM; otherwise, a new symbol will be created. This new symbol will be marked temporary so that it will not be stored upon a SAVE(database_module) command. If an argument is a variable, a DRAM location will be allocated and this location is recorded as usual. If a variable sharing the same name occurs, the same DRAM location will be used for this argument position. In so doing, all common query variables will be linked enabling testing of variable bindings in the query side. For example, the CRS will translate the query f(X,a,X) into f(DRAM 0 -> X, DRAM 1 -> a, DRAM 0). If a list is encountered as a query argument, the CRS will work out some structural value (possible, the number of elements, the number of levels, etc).

After initialisation, the host will relinquish the HOST bus. The address of the HOST bus will be part of the microprogram from the WCS and the type part of the data port will be treated as the input to the TC. During the pre-unification process, writing to the DRAM will be done by the PROC only. It will only happen when a variable argument is encountered - i.e. when QUERY_VARIABLE is set after a type comparison. In this case, both the type and the content of the database argument will be copied into the current DRAM location. After copying, this query will no longer have a variable type (providing the database argument itself is not a variable).

4.4.4. The Processor (PROC)

Physically there will be two major components in the PROC: a SIPO shift register and an Arithmetic and Logic Unit (ALU). The SIPO will provide the communication between the ALU and the DCU of the CLAUSE file. As explained previously, the type part and the content part of the SIPO can be loaded separately. The content part will be active while type comparison is working on the type part. The timing of loading process will be properly arranged so that a type comparison is finished before the content part of the SIPO is entirely filled. (The reason for this has been explained in Section 2.4.2). The roles of the ALU will be :

1) *To perform comparison on the contents of an argument.*

   A content comparison will be a very straight forward process. The query clause content will have been stored in the DRAM during initialisation. The content of a query argument will be a disc reference and if the content of the read-in database content has the same reference

† One port of a DRAM is usually designed to have a higher priority than the other; so that if both ports wishing to write to the same location, the one with the higher priority wins.

value, the comparison will be regarded as positive. While the content comparison is taking place, the rest of the database clause code will be transferred to the next stage (the OB). If comparison is negative, the CLAUSE DCU will be interrupted and the CLAUSE disc transfer will be terminated. Therefore no time will be wasted in collecting non-pre-unifiable clauses.

2) *To read and write from/to the DRAM.*
If the QUERY_VARIABLE flag is set after a type comparison, the database argument will be copied to the DRAM. The DRAM location will be pre-determined by the CRS software.

3) *To maintain a cross binding table for those database variables which binds with query variables.*
There will be some amount of internal memory (preferably associative) in the ALU to be used as the cross binding table. If the DATABASE_VARIABLE flag is set after a type comparison, the ALU will scan through the cross binding table to see if the database variable concerned has been bound to a query variable previously. If the database variable has not been previously bound, a new entry will be created in the cross binding table. An entry will consist of the database variable name (encoded) and the DRAM address of the query argument (cross link). On the other hand, if it is bound, the corresponding DRAM address (its cross link) will be extracted and the content of that DRAM location will be used for the content comparison.

4.5. Output Buffering

The mechanism of output buffering will be similar to clause retrieval buffering. Its role will be to ensure that the pre-unification will be performed on database clauses ahead of the full unification process at the host. There will be two counters, one will contain the OB location processing by the host and the other contain the OB location processing by the PUE. While the PUE is actively doing its duty, the rest of the database clause will be read into the OB location depicted in the second counter. As soon as it is filled, an OUTPUT_READY flag will be asserted. When the host is ready and OUTPUT_READY is asserted, it will read in the database clause according to the value indicated by the first counter and counter will be incremented at the same time. Shortly after this, the two counter values are compared. If they are equal, OUT_READY will be negated.

4.6. Backtracking

One of the advantages of the CLARE architecture is quick backtracking. When a database clause is a rule, a query that unifies with the rule head will reduce to the body of the rule. The body of the rule may consist of several clauses and each clause is queried in turn. Consider the following example,

QUERY	DATABASE
?- f(a).	#1 f(X):- g(X,Y), h(Y).
	#2 g(a,b).
	#3 g(a,c).
	#4 h(c).

Unification will reduce the query to #1 in the database. Clauses g and h will then be queried in turn. At first, g will unify with #2. At this point the values of CNT1 and CNT3 - they contain the CLAUSE and DISC addresses of the database clause next to be pre-unified, respectively - in the CRB will be saved by the host as the next choice point for g. (This corresponds to #3). Clause h will then be tried and it will fail because h(b) does not exist. The INDEX and CLAUSE discs will then be rotated to their previously left off locations and backtracking will be achieved in one head movement.

5. Discussions

For a very large database system (>1M clauses), SCW search would be very time consuming. One possible way to overcome this is to adopt secondary

indexing techniques. Indices in the database will be clustered. Passing the query SCW to a secondary index table would allow the system to locate only potential clusters. This would reduce the entire search space to a manageable size. This could be implemented into the CLARE by augmenting an extra secondary indexing unit (SINDU) before the INDU. The SINDU would mainly consist of some associative memory, large enough to store the secondary index table. The secondary index table will be stored on the INDEX disc and it will be loaded into the SINDU during initialisation. A query will then pass through the SINDU prior to any other operations. Its outputs will be successively used by the INDU to extract the corresponding clusters. Another possible modification would be to replace the PUE by a FUE (Full Unification Engine). This would however require a suitable algorithm. On the hardware side, some shared memory between CLARE and the host would be required for storing variable bindings.

The CLARE mechanism is not restricted to disc memory. The same philosophy could be applied to primary memory. To work with primary memory, the speed of operation would be greatly improved resulting in a much faster database system. ~200M bytes would be required for the complete INDEX and CLAUSE files. Alternatively, enduring a slightly slower operational speed, less memory would be required by leaving the CLAUSE file on disc. In that case ~20M bytes of primary memory would be required.

Moreover, it is conceivable that the CLARE could be regarded simply as a "black box". By "plugging" it into his computer, a user would convert it into a powerful Prolog database machine.

6. Conclusion

A novel hardware (CLARE) design has been proposed to achieve efficient handling of large Prolog database. The CLARE selectively retrieve disc resident clauses by a two stage filtering mechanism thus reduces the amount of full unification required by the host. The design of the two filtering stages is complete. Functional simulations of the filters have been performed separately which have verified their underlying algorithms. Shortage of project time has prevented detailed software simulation. The construction of the first stage filter is currently well under way. The host machine for development is a SUN3/160. On the software side, a database management system is being implemented. It is based on an existing Prolog system, namely Prolog-X which has a module facility, essential for large databases and it supports incremental compilation of clauses [7].

References

[1] R.A. Kowalski, "Logic as a Database Language", Internal Report, Imperial College, London (1981).

[2] J.C. Neves and M.H. Williams, "Towards a Co-operative Data Base Management System", Proceedings of Logic Programming Workshop ,pp341-370, Albufeira, Portugal, (1983).

[3] H. Taylor and M.H. Williams, "Implementing Parlog for Database Applications", Technical Report No. 87/10, Computer Science Department, Heriot-Watt University, Edinburgh, (1987).

[4] M.H. Williams, G. Chen, D. Ferbrache, P. Massey, S. Silvini, H. Taylor and K.F. Wong, "Prolog and Deductive Databases", Technical Report No. 87/4, Computer Science Department, Heriot-Watt University, Edinburgh, (1987). (due to appear in the journal of Knowledge-Based Systems in mid-1988)

[5] W.F. Clocksin, "Implementation Techniques for Prolog Databases", Internal Report, Imperial College, London, (1980).

[6] K. Ramamohanarao and J. Shepherd, "A Superimposed Codeword Indexing Scheme for Very Large Prolog Databases", Proceedings of the Third International Conference on Logic Programming, pp569-576, London, (July'1986).

7] W.F. Clocksin, "Design and Simulation of a Sequential Prolog Machine", New Generation Computing, vol. 3, pp101-120. (1985).

Appendix I: Abbreviations

ALU	Arithmetic and Logic Unit
CNT1	Counter #1 in the clause reference buffer
CNT2	Counter #2 in the clause reference buffer
CNT3	Counter #3 in the clause reference buffer
COND	CONDitional flag selector
CLARE	CLAuse Retrieval Engine
CRB	Clause Reference Buffer
CRS	Clause Retrieval Server
DCU	Disc Control Unit
DRAM	Dual-ported Random Access Memory
ENOR	Exclusive NOR logic gate
FUE	Full Unification Unit
IU	Interface Unit
INDU	Indexing Unit
NAG	Next Address Generator
OB	Output Buffer
PISO	Parallel In Serial Register
PROC	The processor within the Pre-Unification Unit
PUE	Pre-Unification Engine
SCW	Superimposed Code Word plus mask bits
SINDU	Secondary INDexing Unit
SIPO	Serial In Parallel Out register
TC	Type Comparator
WCS	Writable Control Store

Acknowledgements

The work discussed in this paper is part of a collaborative project between ICL and Heriot-Watt University, funded by an Alvey grant.

38. K.F.WONG and M.H.WILLIAMS, Design considerations for a Prolog engine, Proceedings of the Third International Conference on Data and Knowledge Bases, Jerusalem, Israel, 111-120 (1988).

Design Considerations for a Prolog Database Engine

K.F. Wong and M.H. Williams

Department of Computer Science
Heriot-Watt University
79, Grassmarket,
Edinburgh EH1 2HJ.

Abstract

One major hindrance to using Prolog for implementing large databases is the slow speed of unification. The size of physical memory required to store a large database renders main memory impractical and large sets of clauses need to be stored on secondary disk memory. Full unification performed directly on disk resident clauses would be unacceptably slow. For this reason, a special purpose machine is being designed which will provide fast disk filtering for a large Prolog database system. The system, known as CLARE (CLAuse Retrieval Engine), discriminantly retrieves all potentially unifiable clauses for queries and provides a small resolution set for full unification. The CLARE system is based on a two-stage filtering process and aspects of the design are discussed. Three system configurations suitable for the implementation of CLARE are studied. The studies enable the selection of the most cost-effective configuration applicable to a workstation-based Prolog database system.

1. Introduction

Prolog systems are generally fairly limited in their ability to handle large data sets. An evaluation of the capabilities of several existing systems to deal with large data sets has shown that they fail at around 60,000 clauses [1]. However, this is inadequate for many database applications, which may typically require a million clauses or more. With such large data sets it is necessary to rely on secondary storage.

One of the chief problems with Prolog is that when it searches for a clause, it uses the process of unification to match the goal against each clause in turn. This is a complex process of matching functor and arguments according to certain rules, and takes up a considerable proportion of query processing time. For a large database stored on disk, the time taken to perform this could well be intolerable.

One way of improving system performance is to adopt a hardware solution. Existing database machines retrieve data associatively using simple pattern matching techniques. However, these are not directly suitable for Prolog databases because the arguments of Prolog clauses may be variables or complex structures such as lists. On the other hand, from the Prolog point of view, several cases of specialised hardware for unification have been reported. Each of these is based on one of two design principles:

1 to expedite the process of unification e.g. the Hardware Unification Unit (HUU [2]) and the Syracuse Unification Machine (SUM [3]).

2 to minimise the scope of unification e.g. the Attached Index Machine (AIM [4]).

The first type of machine operates as a coprocessor closely coupled to a host. The host invokes the machine explicitly by supplying two unifying terms. The machine then performs unification and returns all necessary bindings as a result. Unification performed by this type of machine is incomplete. It cannot deal with lists and structures. The responsibility to unify such terms is conferred on the host.

The second type of machine is located between the memory system and the processor. Each clause in the database has associated with it some additional information. The role of the machine is to inspect the additional information for each clause and compare it against that of a query. When the two satisfy some matching criteria, it signals the host which then reads in the required clause. The comparison operation is repeated until a preset value or the end of the database is reached. Consequently, the host only needs to perform full unification on a subset of the database. The operation of this second type of machine is sometimes referred to as *partial test*

unification because it only notifies the host if two terms would be likely to bind without actually binding them.

A type two machine is being designed for a Prolog database system which caters for very large sets of disk resident clauses. The machine, CLARE (CLAuse Retrieval Engine), provides a filtering effect. It discriminantly retrieves all potentially unifiable clauses for queries and provides a small resolution set for full unification by the host. A description of the underlying searching mechanisms of CLARE is given in the next section. Section three explains three hardware configurations and assesses their suitabilities for realising CLARE. Some practical considerations for a CLARE prototype are outlined in section four.

2. Retrieval by Two Stage Filtering

A Prolog system is under development which will handle large sets of clauses efficiently [5]. The total system involves a combination of software and hardware. The software component is based on Prolog-X, a system developed by Clocksin [6]. Using Prolog-X, database clauses are compiled and stored in modules. These modules are divided into three categories: small (typically < 10k clauses) medium sized (10k-100k clauses) and large (> 100k clauses). Small modules are loaded into main memory when required, medium sized modules are partly loaded and large modules are disk resident.

The hardware component of the system consists of a special purpose Clause Retrieval Engine (CLARE) which is being designed to increase the performance of the system in handling large sets of clauses. CLARE is a type two machine which will selectively retrieve clauses on the fly from the disk. As database clauses stream from the disk, they are subjected to a two stage filtering process. The first stage is based on matching by *Superimposed Codeword plus Mask Bits* and the second stage on *Partial Test Unification.*

2.1. Superimposed Codeword plus Mask Bits Indexing

In the system under development, predicates with the same functor names and arities are stored in a compiled clause file. For fast searching in large files, a codeword is generated for each fact and rule head and these are maintained in a secondary file. The secondary file is effectively an index table associating codewords with clause addresses. When a query is given, the first stage filtering hardware scans through the secondary file and extracts the addresses of those clauses whose codewords successfully match with the codeword corresponding to that query. The size of a secondary file is generally much smaller than that of a compiled clause file, thereby enabling quicker retrieval to be achieved by scanning the former than by searching the latter exhaustively.

Originally, superimposed codeword indexing was designed for achieving fast retrieval in a relational database. To generate a superimposed codeword, each attribute of a tuple is put through a hashing function and the corresponding values are logically or'ed (superimposed) to obtain the resulting codeword. The same encoding technique is applied to a query. Variables in a query are allowed and they are simply not encoded. Effectively, variables are regarded as "don't cares". The query codeword is matched against every codeword in the database of a relation. A satisfier is discovered if the pattern of ones of a query codeword is a subset of the pattern of a database codeword. An example of superimposed codeword encoding and matching is given in appendix 1.

By itself, the superimposed codewords approach is incapable of representing embedded structures and variables in Prolog clauses†. To handle these, a modification to the codeword approach has been proposed [8], Superimposed referred to as Codeword plus Mask Bits (SCW+MB), which employs an additional field in the codeword to record positional information in a database clause. In its simplest form, one mask bit is used for each argument position. If a mask bit is set, the corresponding argument position is a variable and it will match with any query terms. The mechanism of the matching process is now slightly more complex. When a query is issued, hashing is performed upon each query argument to generate a set of binary codewords. Unlike ordinary Superimposed Codeword matching, these query codewords are not or'ed. Instead, they are compared with database codewords individually. A query argument is a hit if the pattern of 1s of its codeword is masked off by the database codeword or if its corresponding mask bit is set. A database clause is a potential match or satisfier if all query arguments are hits. The program given in figure 1, is an algorithmic representation of the matching process in C. To account for embedded structures, several mask bits are used. Similarly, encoding and matching are extended to functor names and nested arguments of an embedded structure. It is worth noting that mask bits are required by database clauses only and they do not exist in query codewords.

† *An example of embedded structures and variables in Prolog:*
Consider a restaurant predicate which takes four arguments, the restaurant's merchant number, its name, its address and the names of the dishes for which it is most popularly known, i.e. restaurant(Number,Name,Address,Popoular_dishes). Suppose there is a restaurant called Witchcraft whose merchant number is unknown. It is a subsidiary of the Thaumaturgy department store and the Witchcraft restaurant will therefore move to wherever the department store is. Three famous dishes are served in this restaurant. They are bat souffle, snake soup and roast lizard. A Prolog clause describing such a relation might be as follows:
restaurant(Unknown,witchcraft,wherever(thaumaturgy),[bat_souffle,snake_soup,roast_lizard]).
In this example, argument one is a variable and arguments three and four are embedded structures.

```
#define     BOOLEAN             unsigned char
#define     BITWISE_AND         &
#define     OR                  ||
#define     RIGHT_SHIFT_BY      >>

BOOLEAN partial_matching(query_argument_code,query_argument_count,database_code)
unsigned char  query_argument_code[];
unsigned char  query_argument_count;
unsigned char  database_code;
{
        BOOLEAN Satisfier;
        char count;

        for (count=1; count!=query_argument_count; count++)
        {
                if(((((query_argument_code[count] BITWISE_AND database_code))==database_code) OR
                  (((database_mask RIGHT_SHIFT_BY (count-1)) BITWISE_AND 1) == 1))
                        Satisfier=1; else { Satisfier=0; break; }
        }
        return(Satisfier);
}
```

Figure 1: Superimposed Codeword plus Mask Bit matching algorithm.

Argument matching in the first stage of CLARE is done in parallel. The design of the hardware matcher consists of 12 programmable logic devices; one for each argument term. These devices are programmed with the matching criteria. The outputs from individual matchers are logically and'ed to get the overall result. Details of the design are given in [10].

Information such as the hashing function adopted, the arity of a database predicate, the size of a codeword, its weight (the number of bits set) and the number of mask bits used per argument are maintained in a separate record which is loaded into the main memory at the start of a database access, They are consulted at beginning of a query in order to generate the required query codewords. The last three pieces of information are important design parameters. Equations relating the tolerable false drop rate, the average number of constant arguments in a query and the arity of the database clauses, have been formulated [8] for evaluating optimum values for these parameters.

2.2. Partial Test Unification

Indexing via Superimposed Codewords plus Mask Bits is a partial matching technique. Database satisfiers from the matching process are only potential unifiers to a query. Some of these satisfiers are false drops (or 'ghosts') and will fail at the subsequent full unification. False drops are derivable from two main sources: *non-unique encoding* and *restrictive codeword representation*. Non-unique encoding gives rise to the following situations:

(1) *Hash value clashing*. When the same hashed value is generated for two different attribute values. For example, consider a database which consists of a predicate name(Christian_name,Surname). Suppose that the hash values of annie and masson of name(annie,masson) are 01000001 and 00010100, respectively. The superimposed codeword for this clause would then be 01010101. A query name(nanie,samson) is posed. If the codewords for nanie and samson happen to be the same as those of annie and masson, then the database clause would be a false drop.

(2) *Position clashing*.
This is due to the loss of positional information during the process of superimposition. For example, clauses name(lindsay,william) and name(william,lindsay) might have the same codeword value. The query name(X,william) would match with both clauses. Nevertheless, a unification failure should be produced for the second clause.

(3) *Bit clashing*.
This is also caused by the process of superimposition and is best illustrated by an example. Suppose that a query name(tony,smith) was posed and that the binary codewords of tony and smith were 01010000 and 00000101, respectively. This query would create a false drop when it was matched against name(annie,masson), whose codeword was assumed in paragraph (1) above to be 01010101.

In view of the fact that the individual codewords corresponding to each argument of the query are retained separately and each combined with the codewords for the database clause, it is difficult to fully represent predicates with excessively large arities. For practical purposes it is necessary to limit the number of arguments of a predicate which are encoded. This limitation is generally determined at design time. In the CLARE system, a maximum of twelve arguments are accounted for. If a clause has more than twelve arguments, the additional arguments will not be used in determining the superimposed codeword. Inevitably, partial matching upon such clauses would produce false drops.

Another problem which cannot be handled by the superimposed codeword scheme is that of shared variables. Variables are ignored in the encoding process. This would create no problem if variables only occur once in a predicate. However, shared variables are common in Prolog. If no account is taken of shared variables, a large proportion of false drops could be generated. For example, the query married_couple(Same_surname,Same_surname) would result in the retrieval of the entire relation from the database, whilst in reality the resolution set should be very small.

Further refinement of the clauses which satisfy the first stage is provided by partial test unification using the second stage hardware. Using the clause references from the secondary index file, compiled clauses are extracted from the database. Facts and rule heads are compiled into pseudo in-line formats (PIF) ready for partial test unification. In the PIF format, an argument is represented by an 8 bit type tag followed by a 24 bit content field. At present, 102 types are supported: atom, integer, 32 types of structures, 32 types of lists, 32 types of lists with tail variables (or unlimited lists) and 4 types of variables. The second stage filter matches database arguments in PIF against those of the query according to a certain algorithm. Five levels of matching are identified in [9], depending on the depth of the two terms being compared:

- *Level 1* - type only.
- *Level 2* - type and content, ignoring complex structures.
- *Level 3* - type and content, catering for first level structures.
- *Level 4* - type and content, including full structures.
- *Level 5* - type, content with full structures and variable cross binding checks†.

It is expected that for very large databases, complex structures beyond the third level will be uncommon. Since the cost and complexity of the matching hardware to cater for levels four and five are high, a level three partial test unification algorithm is being adopted. In addition to the original algorithm, variable cross binding checks will be incorporated. At the hardware level, the second stage filtering hardware will be a microprogrammed bit slice processor. A simplified version of the matching algorithm is shown in figure 2.

After the second stage, the percentage of false drops will be reduced significantly, resulting in a manageable clause set for full unification.

3. System Configurations

CLARE is being designed to achieve hardware modularity. At the hardware interface level, it will appear as a special function circuit board. By connecting it to compatible computer systems, the latter will be able to execute Prolog efficiently with large data sets. At the machine level, three possible system configurations have been investigated for implementing CLARE.

3.1. A Single Disk Single Bus System (SDSB)

The functional mechanisms of the SDSB and the SDDB (see section 3.2) configurations assume a classical disk control protocol. In the SDSB configuration, figure 3, the host, the first stage filter (FS1), the second stage filter (FS2) and the disk control unit (DCU) exist as independent functional modules. They communicate with each other via a common channel - the global bus. To increase the portability of the system, a standard interface bus, e.g. VMEbus, could be used. The global bus works asynchronously. Arbitration facilities are implemented in order to ensure that at any one time only two modules can communicate. Each module includes a bank of memory which is assigned a unique set of system addresses. These banks of memory are shared and can be accessed both internally and externally. In practice, when a database query is made, the following execution cycles will be involved :

† A variable cross binding occurs when a database variable is bound to a query variable. Checking is essential to ensure that the cross binding relationship is consistent i.e. the variables are not instantiated to different terms. Referring to superimposed codeword index matching, cross binding is in fact an alternative form of shared variables and has the effect of creating undesirable false drops.

```
1       If both types are integers then
                compare their contents (numbers)

2       If both types are atoms then
                compare their contents (hashed values to the symbol table)

3       If both types are structures then
                compare their contents (functor names and arities &
                        contents of top level elements)

4       If both types are lists then
                compare their contents (length of list and types &
                        contents of top level elements)

5       If both types are variables or the clause type is a variable then
                check for the existence of the clause variable
                If it does not exist then
                        create a new entry in the hardware's variable store
                        associate it with the query term
                If it exists then
                        extract its association from the hardware's variable store
                        repeat the comparison operations again

6.      If the query is a VARIABLE then
                associate the variable to the clause term
```

Figure 2: Simplified partial test unification algorithm.

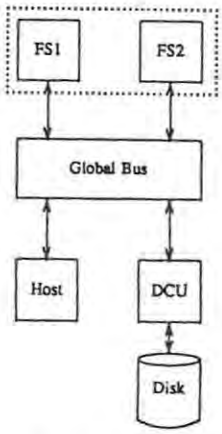


Figure 3: The Single Disk Single Bus System (the dotted box forms the CLARE boundary).

C1 The host will set the disk controller command block in its own memory. The DMA address of the command block will be set to the address in the FS1.

C2 Bus mastership will be acquired by the host. The host will communicate with the FS1 and program its argument space with the query SCW codewords.

C3 Bus mastership will persist with the host. The host will communicate with the FS2 and program its argument space with the required partial test information.

C4 The DCU will acquire bus mastership. It will perform DMA to the host and read in the disk controller command block. Using the disk address specified in the command block, the SCW information will be transferred from the disk to the FS1. The FS1 will capture disk references of those clauses whose SCW indexes match with the query codewords.

C5 At some point, either at the end of a track or when the hit quota is reached, the DCU will generate an interrupt. The FS2 will acknowledge the interrupt and acquire bus mastership. It will read in the disk references of the SCW match satisfiers individually and use them to create disk controller command blocks in its own memory space. At the end of its bus mastership, it will invoke the DCU.

C6 The DCU will acquire mastership and pass data (the compiled code) to the FS2. The FS2 will perform partial test unification as the data is placed into its memory. At the end of a clause, if the test is successful the FS2 will point to the next block of memory ready for the next test; otherwise, the internal address pointer will remain unchanged so that the next incoming compiled clause will overwrite the existing one. As soon as one satisfier (or a specified number of them) is discovered the FS2 will send an interrupt signal to the host.

C7 Upon receiving the signal, the host will acquire bus mastership and perform full unification on the potential unifiers.

The global bus traffic is very busy in the SDSB system. For optimum efficiency, the host should be running other tasks concurrently, when the bus is occupied by other modules during cycles C3-C6.

3.2. A Single Disk Dual Bus System (SDDB)

The SDDB and the SDSB configurations are similar. A schematic block diagram of the SDDB system is shown in figure 4. An extra bus is incorporated in the SDDB system to relieve the congested traffic on the global bus. Under this configuration, the global bus is primarily used for disk access. Communication between the host, the FS1 and the FS2 is provided by the local bus. Faster throughput can be achieved by closely coupling the bus to the functional modules; otherwise, a standard extension bus system can be used. Commercial dual bus systems are available, such as the VMEbus (global)/VMXbus (local) pair and the VMEbus (global)/proprietary P2 bus (local) pair installed in SUN workstations. The basic role of the local bus is for memory access. There are three non-overlapping communication paths using the local bus, namely a) host → FS1, b) host → FS2 and c) FS1 → FS2. During a query, the system will be engaged in the following execution cycles:

C1 The host will set the disk controller command block in its own memory. The command block will be programmed with the address of FS1 memory, so as to direct the destination of the subsequent DMA cycles.

C2 The host will communicate with the FS1 and program its argument space with the query SCW codewords via path a.

C3 The host will communicate with the FS2 and program its argument space with the required partial test information via path b.

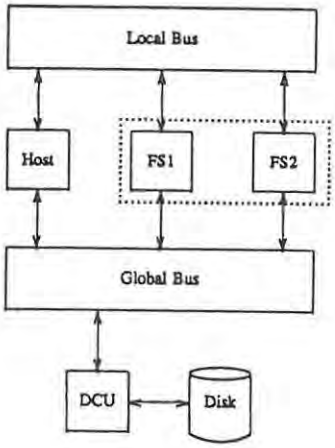


Figure 4: The Single Disk Dual Bus System (the dotted box forms the CLARE boundary).

C4 The DCU will acquire the global bus mastership. It will perform DMA to the host and read in the disk controller command block. Using the disk address specified in the command block, the SCW information will be transferred from the disk to the FS1. The FS1 will capture disk references of those clauses whose SCW indexes match with the query codewords.

C5 At some point, either at the end of a track or when the hit quota is reached, the DCU will generate an interrupt. The FS2 will acknowledge the interrupt. Through path c it will read in the disk references of the SCW match satisfiers individually and use them to create disk controller command blocks in its own memory space. At the end of this, it will invoke the DCU.

C6 The DCU will acquire global bus mastership and pass data (the compiled code) to the FS2. The FS2 will perform partial test unification as the data is placed into its memory. At the end of a clause, if the test is successful the FS2 will point to the next block of memory ready for the next test; otherwise, the internal address pointer will remain unchanged so that the next incoming compiled clause will overwrite the existing one. As soon as one satisfier (or a specified number of them) is discovered the FS2 will send an interrupt signal to the host.

C7 Upon receiving the signal, the host will access the FS2 memory via path b and subsequently, perform full unification on the potential unifiers.

While the DCU is accessing the global bus, there is nothing to hinder inter module communication using the local bus. Therefore cycles C4/C5 and C6/C7 could be overlapped which would expedite the overall filtering process. Also, if the local bus was closely coupled to the system, bus arbitration would be redundant resulting in faster execution.

3.3. A Dual Disk System

Two stage filtering is best realised by a dual disk configuration. A schematic block diagram of the dual disk configuration is shown in figure 5. One disk contains the index file and is connected to the first stage filter (FS1) only. The second stage filter (FS2) only operates on data coming from the second disk which consists of the compiled clause file. The two filters run concurrently and perform comparisons on their corresponding disks. Temporal parallelism is achieved by pipelining the two stages. As soon as a hit is found by the first stage, the second stage will be initiated. A detailed description of the system is given in [7].

This configuration will provide the fastest throughput at the expense of additional and complicated hardware. In order to achieve its full functionality, custom disk controllers would be desirable. Some additional control functions would be: bit serial comparison and disk access at sector boundaries. The dual disk approach would be acceptable to large mainframe systems, but in general would not warrant the additional cost.

3.4. The Clause Retrieval Server (CRS)

The communication between the host and CLARE is organised by a software module independent of the Prolog system - the Clause Retrieval Server (CRS). The idea of an independent CRS makes the underlying hardware transparent to the top level Prolog software. If CLARE was updated or replaced, modification to the CRS would be the only requirement and other software would remain unchanged. The CRS will be designed with a well defined, easily accessible interface so that other software can also utilise CLARE freely through it. The CRS maintains a query descriptor for each query. A query descriptor is a structure comprising the following entries:

1) Original disk position - this is equivalent to the starting address of a query.

2) Query matching references - there are two sets of references, one for each filtering stage.

3) Statistics - the number of uninspected hits remaining.

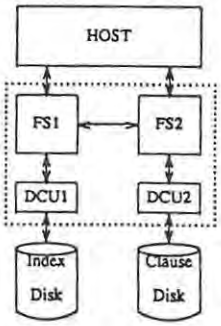


Figure 5: The Dual Disk System. The dotted box forms the CLARE boundary.

4) Operational mode - CLARE supports three operational modes: fetch_1: retrieve one resolvent; fetch_n: retrieve n resolvents and fetch_all: retrieve all resolvents.

5) Intermediate disk position - the point at which the last search terminates. This allows CLARE to resume operation upon encountering a false drop (ghost).

6) Intermediate results - pointers to the system memory which holds the partial unifiers associated with a query.

The inclusion of intermediate information assists the host in handling exceptions, system restart and programmable fetching.

3.5. Practical Considerations

The host system on which the Clause Retrieval Engine, CLARE, is currently being developed is the SUN3/160 workstation. For the architecture of CLARE, three general approaches to system configuration have been investigated. The improvement in performance at the expense of additional hardware offered by the third option (dual disk architecture) is not cost effective. The less complex SDSB and SDDB configurations are more applicable to small computers although their execution speeds are affected by the bus bottleneck. The existing bus system in a SUN workstation is the VMEbus paired with the SUN proprietary P2 bus. This makes the SDDB configuration the most attractive choice. However, the P2 bus information is not readily available at present. Therefore, in order to avoid modification to standard hardware, a pseudo SDDB system is being employed. The two filtering stages will be closely coupled but the communication between the host and the second stage (host→FS2) will be achieved through the global VMEbus.

For prototyping, an initial CLARE system, based on MSI/LSI technology, is being implemented Because of the limitation in packing, the FS1 and the FS2, although closely coupled, will have to be constructed on two separate boards. Both the FS1 and FS2 hardware are being designed as memory mapped modules. As far as the SUN CPU is concerned, they will be indistinguishable from additional external memory boards on the VMEbus. Viewed from the DCU, CLARE appears as a set of registers. All data from the disk will be checked by the DCU before it is written to the CLARE input registers thereby avoiding any need for CLARE to be aware of disk format information, e.g. correction codes. Only correct data will be placed at the input of CLARE. The first prototype of CLARE will be restricted to cater for sixteen argument terms because of the number of components involved. However, this restriction could be lifted in future developments.

The CLARE hardware is being designed to be driven by six machine commands: set_SCW_reference, set_timing, set_partial_test_unification_reference, set_fetch, read_result and execute. The set_timing command is used to program a timing circuit in CLARE whose role is to generate various control signals e.g end of mask bit (EOM). Other commands are self explanatory and affect the content of query descriptors in the CRS. The Prolog-X system will invoke these commands indirectly through the CRS which is the software front end to CLARE from the host. It is being written in C and designed to support a general internal data structure - the query descriptor - in order to allow the logic database management system to access CLARE transparently. Such a structure will account for intermediate information thus enabling easy system restart and restorable interrupts.

4. Conclusion

CLARE is a novel hardware engine designed to enhance the execution speed of Prolog when dealing with large clause sets. This is achieved by reducing the amount of full unification necessary on the large clause set by a two stage filtering process - superimposed codeword plus mask bits index matching followed by partial test unification. Two aspects of the design of CLARE have been discussed in this document: the filtering principles and the system configuration.

The superimposed codeword plus mask bits indexing technique adopted in the first stage filter provides a fast retrieval rate. It can be very discriminating and will work well in handling databases containing mostly ground facts. For higher level applications, such as large expert systems in which variables and complex structures will be widely used, partial matching by superimposed codeword plus mask bit indexing will be inadequate. A considerable proportion of false drops could result which would affect system performance seriously. The second stage filter based on partial test unification is employed to refine the resolution set, thus reducing the number of unnecessary unification operations to be performed by the host. Moreover, additional matching functions, anti-matching and integer range matching, are being incorporated in the second stage hardware design. These additional functions will speed up the performance and reduce the workload of database management software.

In a separate project a logic database management system is being developed using Prolog-X which will run on the SUN3 workstation, using the CLARE hardware through the CRS. The CLARE system itself is being designed as a memory mapped functional module inside the SUN3 workstation. The standard VMEbus specification is being adhered to. No modification of existing SUN hardware will be required.

Summarising, CLARE will offer the following advantages:

- Improved clause retrieval performance
- Versatility - conforming to the bus specification CLARE would be compatible to VMEbus based computers
- Ease of installation/replacement - CLARE will be readily installable on existing SUN3 workstations and can be easily replaced for future hardware maintenance
- Ease of software upgrade - through the use of a separate software interface, the top level software will not be confined.

Acknowledgements

The work discussed in this paper is part of a collaborative project between ICL and Heriot-Watt University, funded by the SERC and the DTI through an Alvey grant.

References

[1] Williams, M.H., Massey, P.A. and Crammond, J.A.: "Benchmarks for Prolog from a Database Viewpoint", Proceedings of Second ALVEY Workshop for SIGKME, 1987, ppC1-C8.

[2] Woo, N.S., "A Hardware Unification Unit: Design and Analysis", Proceedings of 12th International Symposium on Computer Architecture, June 1985, pp198-205.

[3] Robinson, P., "The SUM: An AI Coprocessor", BYTE, June 1985, pp169-180.

[4] Colomb, R.M., "A Clause Indexing System for Prolog Based on Superimposed Coding", DCS Report number 8506, September 1985, School of Electrical Engineering and Computer Science, University of New South Wales, Australia.

[5] Williams, M.H., "The Design of A Prolog Database Machine", Proceedings of the 1st Workshop for the Special Interest Group On Knowledge Manipulation Engines, Reading University, 5-6 January 1987, ppL1-L5.

[6] Clocksin, W.F., "Design and Simulation of a Sequential Prolog Machine", New Generation Computing, vol. 3, pp101-120 (1985).

[7] Wong, K.F. and Williams, M.H., "Limiting a Resolution Set in a Large Prolog Database Using Special Hardware", Proceedings of 1988 ACM SIGSMALL/PC Symposium on Personal and Small Computers, Cannes, France, May 4-6, 1988.

[8] Ramamohanarao, K. and Shepherd, J., "A Superimposed Codeword Indexing Scheme for Very Large Prolog Databases", Technical Report 85/17, 1985, Department of Computer Science, University of Melbourne, Australia.

[9] Ferbrache, D., "Partial Unification in Hardware and Software", Technical Memorandum TM-HW-CS-87-001, 11th June 1987, The Prolog Database Machine Group, Department of Computer Science, Heriot Watt University, Edinburgh, Scotland.

[10] Wong, K.F., "A Partial Matching Hardware for Selective Clause Retrieval in Large Prolog Databases" Technical Report 88/2, Department of Computer Science, Heriot Watt University, Edinburgh, Scotland.

Appendix I: An Example of Superimposed Codeword Encoding and Matching

Suppose, there is a "Restaurant" database relation which has the following four attributes : the restaurant's merchant number, name, address and the food for which it is famous. A tuple representing an Edinburgh based Witchcraft restaurant whose merchant number is 000001, which is renown for its delicate bat souffle is shown in the first entry of table A1.

Table A1: Relation = Restaurant			
Number	Name	Address	Famous Food
000001	Witchcraft	'Spellmarket, Edinburgh'	bat_souffle
000001	Witchcraft	'Spellmarket, Edinburgh'	snake_soup
000002	Halloween	'Magic Lane, London'	pumpkin_pie

Table A2: Superimposed Codewords for table A1	
SCW	table A1 entry
01110101	1
11010101	2
01110111	3

A superimposed codeword for each tuple is generated by passing each attribute values through a hashing function which produces four separate binary codewords. These binary codewords are logically or'ed (superimposed) to obtain the superimposed codeword. In this example, a codeword is 8 bits in length and two bits of the codeword are set by the hashing function. The diagram in figure A1 shows how the superimposed codeword for this example is generated.

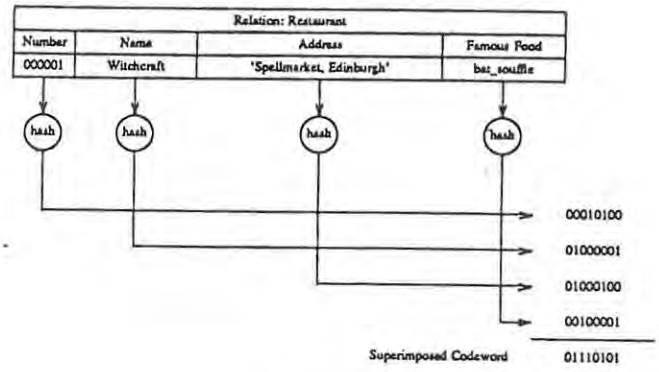


Figure A1: An Example of Superimposed Codeword Generation.

The same encoding method is used for queries. Suppose it is desired to discover the merchant number of a restaurant with the other three attributes known, e.g.

Restaurant.Number where
Restaurant.Name=Witchcraft &
Restaurant.Address='Spellmarket, Edinburgh' &
Restaurant.Famous_Food=bat_souffle

The binary codewords for Name, Address and Famous_Food are 01000001, 01000100 and 00100001, respectively. No binary codeword is generated for Address since it is unknown - it is a variable. Therefore, the superimposed codeword for this query is obtained from logically or'ing 01000001, 01000100 and 00100001, which gives 01100101. To achieve partial matching, the query codeword (01100101) is compared with the database codeword (01110101). In this example, the database tuple is a satisfier because the pattern of 1's in its codeword completely masks off that in the codeword for the query.

39. K.F.WONG and M.H.WILLIAMS, A type driven hardware engine for Prolog clause retrieval over a large knowledge base, Proceedings of the Sixteenth International Symposium on Computer Architecture, Jerusalem, 211-222 (1989).

A Type Driven Hardware Engine for Prolog Clause Retrieval over a Large Knowledge Base

Kam-Fai Wong[†] *and M. Howard Williams*[††]

† Software Engineering Department, Unisys Ltd., Livingston, EH54 7AZ, Scotland.

†† Dept. of Computer Science, Heriot-Watt University, Edinburgh EH1 2HJ, Scotland.

Abstract

Whereas existing Prolog systems are very effective at handling small knowledge bases, they are not very efficient at and often incapable of handling large sets of clauses. Large knowledge bases which may comprise millions of clauses and are shared by a number of users, may need to reside in secondary memory. In such cases exhaustive search is inordinately slow. Various approaches have been put forward for handling the problem, most of which involve *coupled* systems (loosely or tightly coupled). A Prolog data/knowledge based system which provides an *integrated* solution to the problem is being developed. An essential element in this system is the CLAuse Retrieval Engine, CLARE, which is a special purpose hardware engine designed to perform selective retrieval of data from disk in order to identify all potential clauses which will be required for full unification during a query. The engine consists of two separate hardware components, which together form a two-stage filtering configuration. This paper concentrates on the second stage filter which is concerned with partial test unification.

1. Introduction

One of the important application areas of Prolog is knowledge base implementation. Most existing systems for handling large knowledge bases [3,4] are constructed by *coupling*[1] a Prolog translator to a relational database manager. In a coupled knowledge based system, clauses are usually stored in one of two different databases : an Extensional Database (EDB) consisting of ground facts equivalent to tuples in a relational database and an Intensional Database (IDB) consisting of non-ground facts and rules. Such a system is usually based on the following assumptions :

(1) The EDB constitutes the bulk of the knowledge base.

(2) The IDB is small enough to fit into main memory.

(3) Mixed relations consisting of both ground facts and non-ground facts or rules do not occur.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1989 ACM 0884-7495/89/0000/0211$01.50

An exception to this is the EDUCE system [1] which stores facts and rules in a relational database which is tightly coupled to a Prolog system. One way to increase the performance of a coupled system is to employ a database machine to provide for rapid searching of disk files. The major role of the database machine is to expedite the data retrieval process by matching the argument terms of the disk resident facts against the argument terms of a query on-the-fly. Term matching in conventional database machines is based on equality testing which is a sufficient test only in the case of clauses which are ground facts whose arguments are purely atomic.

Equality testing is, however, inadequate for handling rules and non-ground facts which may contain variables and possibly also complex structures as their arguments. Evaluation of a Prolog query is based on *resolution* which invokes *unification* - a complex and time consuming term matching process. Compounding this with a low disk transfer rate, unification applied to a large set of disk resident rules would be impractically slow. For this reason, the decision is usually taken to separate rules and non-ground facts and store them separately in the IDB under the management of a Prolog system. On the other hand, retaining the IDB in main memory incurs a serious size restriction to the entire knowledge based system.

Another problem in a coupled knowledge base arises from the fact that in the Prolog system there is an ordering

associated with clauses which is used to effect the user in writing Prolog programs, whereas in the database part the ordering of clauses is determined by the system. For this reason mixed relations cannot be catered for in a uniform way and are usually disallowed. This difference in philosophy is quite acceptable for some applications but is a serious problem for a general purpose knowledge base.

In the future an increasing amount of semantic information will be included in knowledge based systems. For example, medium-size knowledge based systems are envisaged by D.H.D. Warren to be ".... of the order of 3000 predicates, 30000 rules, 3000000 facts, and 30 Mbytes total size" [5]. Clauses with rules and structures will not be uncommon. Rules and facts will coexist in the same file. Ordering of clauses, which contributes to the interpretation of the knowledge based system, will be important. For these reasons, the idea of coupling a Prolog system to a relational database will be inadequate, and purpose built systems will be required.

One way of providing a knowledge based system suitable for the future is to adopt the *integrated* implementation approach. In an integrated system, users have a uniform view of the knowledge base : rules and facts of a predicate are kept together in a user-specified order and they are managed by a single Prolog system. Compared with a coupled knowledge base, an integrated knowledge base is more flexible and can be engaged in a much wider scope of applications. However, existing systems are restricted to small applications due to their inefficient way of handling disk resident clauses†. One of the chief problems with Prolog is that when it searches for a clause, it uses the process of unification to match the goal against each clause in turn. This is a complicated process of matching functor and arguments according to certain rules, and takes up a considerable proportion of query processing time. For a large knowledge base stored on disk, the time taken to perform this could well be intolerable.

In order to provide for high-speed retrieval of clauses from large clause sets stored on disk, a special purpose Prolog data/knowledge base machine, CLARE - CLAuse Retrieval Engine, has been designed and is under development. A two-stage filtering process is used by CLARE which employs two pieces of dedicated hardware to search disk resident data on-the-fly. The first stage filter (FS1) is a hardware realisation of index searching based on *superimposed codeword plus mask bits* [2], while the second stage filter (FS2) is responsible for *partial test unification*. In the next section, an overview of the Prolog Data Base Machine (PDBM) project - a project which is aimed at developing the type of system to handle large knowledge bases - is outlined. The architecture of the second stage filter (FS2) of the CLARE hardware is described in detail in section three which also includes the timing calculations of the various hardware operations supported by the FS2.

† A set of benchmarks have been devised for evaluating the efficiency of various Prolog systems in handling data bases of different sizes [6]. The benchmark programs were run on a Sun3/160 workstation, equipped with 4MB of main memory, and revealed that the Prolog systems were unable to cope with more than about 60k clauses and even then the overhead of loading these clauses into main memory was very high [7].

2. The PDBM Project - A Brief Overview

The aim of the Prolog Data Base Machine (PDBM) project is to produce a Prolog system which will handle large sets of clauses efficiently [8]. This is achieved by a combination of software and hardware development. The software component is based on a C version of Prolog-X. Prolog-X is a Prolog compiler originally developed by Clocksin[9] and which now runs on ICL3900 and SUN systems. It uses Edinburgh Prolog syntax but with extension to handle modules. Using Prolog-X, clauses are compiled and stored in modules, each module containing one or more procedures. Modules are then classified into two types depending on their size, viz small modules which are loaded into main memory when required, and large modules which are disk resident.

The hardware component of the system consists of a special purpose Clause Retrieval Engine (CLARE) which is designed to increase the performance of the system in retrieval over large clause sets. CLARE selectively retrieves clauses on the fly from a disk. As clauses stream from the disk, they are subjected to a two stage filtering process, as follows :

2.1. Superimposed Code Word plus Mask Bits (SCW+MB)

Predicates with the same functor names and arities are stored in a compiled clause file. For fast searching in large files, codewords are generated for facts and rule heads and these are maintained in a secondary file. The secondary file is effectively an index table associating codewords with clause addresses. When a query is given, the first stage filtering hardware scans through the secondary file and extracts the addresses of those clauses whose codewords successfully match with the codeword corresponding to that query. The size of a secondary file is generally much smaller than that of a compiled clause file [10], thereby enabling quicker retrieval to be achieved by scanning the former than by searching the latter exhaustively. The superimposed codeword plus mask bits indexing scheme (SCW+MB) [11] is an extension of the basic superimposed codeword method which takes account of complex structures and variables which arise in Prolog clauses by the use additional field comprising a set of mask bits. Index matching is performed in parallel, using standard PLAs and MSI components [2], in the first stage of the CLARE engine.

Indexing via SCW+MB is a partial matching technique. Clauses satisfying the matching process are only potential unifiers for a query. Some of these satisfiers are false drops (or 'ghosts') and will fail at the subsequent full unification. False drops are derivable from two main sources [12]:

(1) *Non-unique encoding* in which the same codeword is shared by one or more clauses.

(2) *Restrictive codeword representation* which is caused by truncation of clauses with arities larger than the allowable hardware limit. (In the present system, only 12 arguments of a query is encoded).

Another problem which cannot be handled by the superimposed codeword scheme is that of shared variables. Variables are ignored in the encoding process. This would create no problem if each variable only occurs once in a predicate. However, if a variable occurs more than once in a predicate, a large proportion of false drops could be generated. For example, the query married_couple(Same_surname,Same_surname) would result in the retrieval of the entire predicate from the knowledge base, whilst in reality the resolution set should be very small.

2.2. Partial Test Unification

Further refinement of the clauses which satisfy the first stage is provided by partial test unification using the second stage hardware (FS2). Using the clause references from the secondary index file, compiled clauses are extracted from the knowledge base. Facts and rule heads are compiled into pseudo in-line formats (PIF) ready for partial test unification. In the PIF format, an argument is represented by an 8 bit type tag followed by a 24 or 32 bit content field with an optional 32 bit extension. At present, 107 data types are supported under the PIF format, see appendix 1. The FS2 filter matches clause arguments in PIF against those of the query following a set algorithm. Five levels of matching have been investigated which depends on the depth of the two terms being compared:

- *Level 1* - type only.
- *Level 2* - type and content, ignoring complex structures.
- *Level 3* - type and content, catering for first level structures.
- *Level 4* - type and content, including full structures.
- *Level 5* - type, content with full structures and variable cross binding checks†.

Since the cost and complexity of the matching hardware to cater for levels four and five are high, a level three partial test unification algorithm is being adopted. In addition to the original algorithm, variable cross binding checks have been incorporated. A simplified version of the matching algorithm is shown in figure 1. After the second stage, the percentage of false drops will be reduced significantly, resulting in a manageable clause set for full unification.

An independent software module, the Clause Retrieval Server (CRS), is being developed which links CLARE with the PDBM Prolog system. In practice, there will be four searching modes during a clause retrieval:

(a) By software only - the CRS performs all the search operations itself.

(b) Using FS1 only - the superimposed codeword hardware.

(c) Using FS2 only - the partial test unification hardware.

(d) Using both FS1 and FS2 - a two-stage hardware filter.

One of these modes will be selected depending on the nature of a query (e.g. whether it contains cross bound variables) and the knowledge base (e.g. whether it is rule or fact intensive). The CRS will also support simultaneous access by multiple clients which involves procedures for concurrency control and transaction handling.

At the hardware level, CLARE is incorporated in a SUN3/160 workstation. The communication between CLARE and the host takes place via the VMEbus interface. CLARE is memory mapped into the /dev/vme24d16, SUN's user space, using the mmap() system call [13]. Both filtering stages, FS1 and FS2, appear in the form of plug-in circuit boards. A common address space from ffff7e00(hex) to ffff7fff(hex) - 128k bytes in total - is shared by FS1 and FS2. The two filters are mutually exclusive. The selection between the two is governed by the third least significant bit, b_2, of an 8-bit control register - A 0 in b_2 selects FS1 and a 1 selects FS2.

3. The Second Stage Hardware (FS2)

Once a filtering stage is enabled, the setting of the first two significant bits of its control register determines the operation mode of the hardware. FS2 can be set to work in one of 4 operational modes :

Operational Mode	b0	b1
Read Result	0	0
Search	0	1
Microprogramming	1	0
Set Query	1	1

When a query is posed, it is translated into microprogram instructions. These instructions are loaded into the FS2 while it is set to Microprogramming mode. The FS2 is then switched to Set Query mode and all query arguments are written into the FS2 query memory. After performing the two operations, the FS2 is ready to perform a search. The DMA begin and end addresses of the disk transfer command block, where retrieved data will be placed, is specified to be the FS2 address space. A search is then initiated by setting the FS2 to Search mode. From then on the FS2 is, effectively, connected to the disk controller. At the end of the search, if a match has been found, the most significant bit (b_7) of the control register will be set. At this stage, in order to extract the potential answers to the query, the FS2 will be set to Read Result mode.

The overall architecture of the FS2 hardware is depicted in figure 2. It consists of four functional units. The Writable Control Store (WCS) provides the microprogram instructions to coordinate all the other units. Data streaming from the disk is retained in one of the two memory banks in the Double

† A variable cross binding occurs when a database variable is bound to a query variable. Checking is essential to ensure that the cross binding relationship is consistent i.e. the variables are not instantiated to different terms. Referring to superimposed codeword index matching, cross binding is in fact an alternative form of shared variable and has the effect of creating undesirable false drops.

Buffer (DB) while data stored previously in the other bank are subjected to partial test unification by the Test Unification Engine (TUE). The results of the test will decide which clause is to be captured in the Result Memory (RM).

3.1. The Writable Control Store (WCS)

The construction of the WCS, figure 3, is based extensively on AMD (Advanced Micro Devices, Inc) bit-slice components. The WCS consists of a bank of fast bipolar RAM which holds the microprogram instruction for coordinating the overall FS2 hardware during a query. A *one level pipeline based architecture* [14] is adopted by the WCS - the microprogram instruction appears one instruction ahead of the microprogram memory address. The RAM can hold a maximum of 2048 microprogram instructions, each 64 bits wide. Normally, the fast RAM is read only and is addressed by the Micro Program Controller (MPC). When the FS2 is operating in Microprogramming mode, the fast RAM is connected to the VMEbus. It will appear as normal memory to the SUN host and loading of microprograms is enabled. Two counters, one for the database input and one for the query, are used to keep track of the number of elements remaining during the matching of lists and structures. An 8 MHz clock is used to synchronise the various parts in the WCS.

At the beginning of a retrieval and after matching a clause, the MPC is engaged in a polling routine This routine repeatedly monitors the zeroth bit of the conditional code (CC) which is asserted when a new clause has been read into the Double Buffer and is ready to be examined. The output of the MPC which is the address in the fast RAM, can derive either from the MPC's internal counter or externally from the branch address field of the microprogram instructions when a branch or jump instruction occurs. Another external source comes from the output of the Map ROM. The Map ROM stores a list of jump vectors and its address port is connected to the db-data and Q-data bus from the Test Unification Engine. Only the type fields of the db-data and Q-data are effective. Depending on the combination of the type fields, different microprogram routines are invoked. The data types supported by the FS2 are classified into 3 categories (see appendix 1). Each category is handled differently :

Simple terms require simple matching.

Atoms, integers and floating point numbers are included in this category. Apart from the case of an integer constant (integer in-line) in which the content field is the value of the integer itself, the content fields of these data items are symbol table references. Equality is the testing condition when two arguments are compared against each other in this type.

Variable terms may require skipping, storing or fetch then match operation.

There are five variable types. An anonymous variable is effectively a don't care object and it causes a skip in the matching process. When an anonymous variable is encountered on either side during a match, the match will succeed immediately irrespective of the type of the other side. The other variable types account for the origin of a variable - whether it occurs in a query (QV) or in a data/knowledge base (DV). QV and DV are further divided into two groups in order to record their modes of occurrence. For example, 1st-QV will be the type of a variable when it is being referred to in the first time during the partial test unification process; if the same variable occurs again in the same query, it will be typed as Sub-QV (subsequent query variable). At compile time, the subsequent occurrences and the first occurrence of a variable have the same content field which is a pointer to the internal memory of the Test Unification Engine. First occurrences of QVs and DVs will invoke an instantiation routine which stores, respectively, the database and the query argument terms in the Q-Memory and DB-Memory of the Test Unification Engine (see later). QVs and DVs which occurred subsequently are handled specially. It involves two microprogram instruction cycles : firstly, the location of the internal memory of the Test Unification Engine, referenced to by the content field of the variable, is read; and secondly, that value is subject to partial test unification.

Complex terms require repetitive matching

A complex term can be a list or a structure. In-line complex terms can contain one to 32 constituent terms. During the matching process, the arities of the in-line complex terms of both the database and query (which need to be the same) will be loaded into respective counters. The matching process will be performed upon each constituent term pair. At the end of a comparison, both counters are decremented. This process will be repeated until the counters reach zero. Structures and lists with arities greater than 32 are represented, respectively, by structure and list pointers. Comparison of these types is identical to simple term matching. As a subset of lists, unlimited lists are defined. They are lists which contain a tail variable, e.g. [a,b|Tail]. The arities of the terms being compared may not be equal in this case. The arities are loaded into two counters and matching is repetitively carried out until the value of either counter is zero.

3.2. The Double Buffer and the Result Memory

The Double Buffer is separated physically into two identical halves. Each half consists of a bank of memory and an output register. During a search, one half is selected for input while the other is selected for output, and these roles alternated whenever the input is filled. The half which is selected for input accepts the data streaming in from disk. The data from the output register is the clause stored in the corresponding memory in the previous cycle. The output data is fed into the Test Unification Engine for matching with the pre-loaded query data. Selection control is basically generated by a toggle flip flop. Both inverting and non-inverting outputs of the flip flop are utilised in order to produce two non-overlapping clocks. The clock period is variable and is equal to the time taken for the Double Buffer to read in 2 clauses.

The Result Memory has a capacity of 32K bytes which is large enough to contain all clause satisfiers of one disk track - the worst case of a single FS2 search call. While disk data is transferring to the Double Buffer, a copy of the data is written into the Result Memory in parallel. The address of the Result Memory is derived from the Address Generator. The Address Generator consists of two counters. One is 6 bits wide and contributes to the upper 6 bits of the Result Memory address (A_9-A_{14}). This counter is incremented whenever a clause satisfier is found. The value of this counter at the end of a retrieval indicates the number of clause satisfiers. The other counter is 9 bits wide and forms the lower 9 bits of the Result Memory address (A_0-A_8). The second counter is always reset to zero after a clause has been examined. The schematic block diagram depicting the Double Buffer and the Result Memory is shown in figure 4.

3.3. The Test Unification Engine (TUE)

The Test Unification Engine (TUE), figure 5, consists of two banks of memory (DB Memory & Query Memory) a comparator (Comp), three registers (Reg1-3) and 6 selectors (Sel1-6). The DB Memory is a dual port memory which is used for storing bindings of database variables at run time. It is reset to pointing to itself at the beginning of each clause input. The Query Memory is pre-loaded at query time and it contains the query argument terms. The Comparator (Comp) is a standard ALS 8-bit comparator which generates a 1-bit HIT signal according to the values of its A and B inputs. The A and B inputs correspond to the data/knowledge base and query argument terms respectively. HIT is fed into the conditional code register (CC) of the WCS which will select of the next microroutine. When the FS2 is in Set Query mode, only the Query Memory is enabled. The right hand branches of Sel4 are selected to provide, respectively, the data and address to the Query Memory. The Query Memory is directly connected to the VMEbus interface and appears to the host as ordinary memory.

While the FS2 is in Search mode, operation of the TUE is controlled by microcode. Depending on the types of the terms being compared, different microprogram routines are invoked. The main objective of the microprogram routines is to ensure that the right data appears at the memory and the comparator. This is done by selecting the appropriate paths of the six selectors and enabling the appropriate registers. Seven basic hardware operations are provided in realising the partial test unification algorithm listed in figure 1.

3.3.1. MATCH

This is the simplest operation which is applied when the data/knowledge base and the query arguments are integers, atoms, floating point numbers, structures or lists (corresponding to cases 1 to 4 in the algorithm shown in figure 1). The routes in which query and knowledge base terms are passed to the Comparator are shown by the thick dotted lines in figure 6. Whenever a hardware operation is invoked, data/knowledge base and query arguments travel in two separate routes in parallel. The routes taken during the MATCH operation are :

(1) The database route originates from the Double Buffer, along the In-bus, through the left branch of Sel1 and finally appears at the A-port of the Comparator. From the timing specifications of individual devices, the time taken for data to appear at the A-port is calculated to be 40ns (see bottom of figure 6).

(2) The query route starts from the left branch of Sel6, from which micro bits 13-20 provide the address of the Query Memory. Data derived from the Query Memory enters the right branch of Sel3 and ends up at the B-port. The estimated propagation time is 75ns.

It takes 30ns for the comparator to generate a result. Comparison can not take place until both A and B inputs are ready. Therefore, although information travels on both routes in parallel, the longest routing time of the two should be taken in calculating the execution time of the MATCH operation. This gives an execution time of 105ns (75+30).

3.3.2. DB_STORE

This operation is applied when the data/knowledge base argument's type is first occurrence variable (1st-DV) - corresponding to case 5a of figure 1. The aim is to store the query argument in the location of the DB Memory which is addressed by the content field of the data/knowledge base argument. Figure 7 shows the routes taken by the database and query arguments. On the database side, data is extracted from the Double Buffer; it streams via the In-bus to the left branch of Sel1. Travelling via the left branch of Sel2, the data appears on the A address port of the dual ported DB Memory. There is only one input port in the DB Memory and it is connected to the output of Reg3. The content of Reg3 is taken from the output of the Query Memory whose location is specified by the left branch of Sel6. The time taken to execute this operation is 95ns.

3.3.3. QUERY_STORE

This operation is applied when the query type is a first occurrence variable (1st-QV) - case 6a of figure 1. The objective is to store the data/knowledge base argument in the location of the Query Memory which is addressed by the content field of the query argument. Referring to figure 8, the database route travels from the Double Buffer through the left branch of Sel1, then through the right branch of Sel5 and passes the left branch of Sel4 to appear at the input port of the Query Memory. The route which the query data takes is derived from the left branch of Sel6 and terminates at the address port of the Query Memory. The total execution time for this operation is 115ns.

3.3.4. DB_FETCH

When a subsequently occurring data/knowledge base variable (Sub-DV) is encountered during a search - case 5b of

figure 1, the DB Memory is accessed at the B-port in order to extract the binding of the variable for subsequent comparison. The routes taken by the data/knowledge base and the query arguments in this operation are depicted in figure 9. The database route starts from the Double Buffer which appears at the B address port of the DB Memory. The corresponding data is extracted from the DB Memory and it travels via the right branch of Sel1 to appear at the A-port of the Comparator. The query route is identical to the query route taken by the simple MATCH operation. The total execution time for this process is 105ns.

3.3.5. QUERY_FETCH

This operation is applied when the query type is a subsequently occurring variable (Sub-QV) - case 6b of figure 1. It is slightly more complicated to implement and requires two cycles for completion. In the first cycle, the query route (from the left branch of Sel6 → output of the Query Memory → right branch of Sel3 → right branch of Sel2 → A address port of DB Memory) provides the address to the A port of the DB Memory, figure 10. From the DB Memory the binding associated with the query variable is extracted and appears at the B-port of the Comparator via the left branch of Sel3 in cycle two. Running concurrently with the first and second cycles, the data/knowledge base argument is placed at the A-port of the Comparator. The time taken to execute this operation is 170ns.

3.3.6. DB_CROSS_BOUND_FETCH

DB_CROSS_BOUND_FETCH is required in situations when a data/knowledge base variable is initially bound to a query variable and the former is being used again in the same clause - cross binding (case 5c of figure 1). For example, it will be called for the second occurrence of the data/knowledge base variable A if the query f(X,a,b) is matched against the clause f(A,a,A). In the query side, data takes the simple route, i.e. ub13-20→left of Sel6→ output of Query Memory→right of Sel3→ A-port of Comp. Two cycles are required to establish the database route. In the first cycle, figure 11a, the data from the Double Buffer appears on the B address port of the DB Memory. From the DB Memory B data port, the reference to the cross bound variable is extracted and stored in Reg1. This reference is placed at the B address port in the second cycle, figure 11b. From the DB Memory the ultimate association is extracted and placed at the A-port of the Comparator via the right branch of Sel1. The total execution time is calculated to be 170ns.

3.3.7. QUERY_CROSS_BOUND_FETCH

The route taken by this operation is shown in figure 12. It is invoked when a query variable is initially bound to a data/knowledge base variable and the former is being used again in the same clause (case 6c of figure 1). This is the most complicated operation and requires 3 microprogram cycles. In the first cycle, the A-port of the Comparator is set up via the database route (Double Buffer→left of Sel1) and concurrently, the A port of the DB Memory is addressed by the data presented on the query route (i.e. ub13-20→left of Sel6→Query Memory→right of Sel3→right of Sel2). In the second cycle the query variable binding is extracted from the DB Memory A data port. This binding is recycled and is put back at the A address port (via left of Sel2→right of Sel2). The ultimate value for comparison is routed to the B-port of the comparator (via left of Sel3). The time required to perform the operation is calculated to be 235ns.

4. Conclusion

A special purpose hardware engine, CLARE, is being developed which will provide Prolog with a rapid clause retrieval rate in dealing with large data sets. CLARE consists of two hardware components and together they form a 2-stage filtering architecture. The hardware is designed to be linked to a M68020 host, via the VMEbus interface, in a SUN3/160 workstation. The prototype of the first stage filter (FS1) hardware has been fully developed. It can search data at a rate of up to 4.5Mbyte/sec. The design of the second stage filter (FS2) is complete and the partial test unification algorithm has been verified. The construction of the FS2 is well under way. Once the CLARE hardware is fully developed, it will be subjected to benchmark tests similar to the ones devised in [7].

Execution times for the various FS2 hardware operations are calculated and they are summarised in table 1. From the table it can be seen that, QUERY_CROSS_BOUND_FETCH is the most time consuming operation which takes 235ns. This gives the worst case execution rate of the FS2 to be approximately 4.25Mbytes/second. The SUN3/160 workstation, the target platform of CLARE, can be mounted with either a SCSI based disk system, e.g. Micropolis 1325, or a SMD+ based disk system, e.g. Futjitsu M2351A. Even if the second type of disk (which offers a faster transfer rate) is used and it is tuned to operate at its peak rate (circa 2Mbytes/second), the FS2 hardware can still filter disk data at a faster rate than it can be delivered from the disk.

Acknowledgement

The material described herein is based on work carried out in the Prolog Database Machine project. This project, is a collaborative project involving the Department of Computer Science, Heriot-Watt University and International Computers Ltd (ICL, UK), and is funded by the Science and Engineering Research Council (SERC) and the Department of Trade and Industry (DTI) of the United Kingdom, under the Alvey initiative.

Reference

[1] Bocca, J., "EDUCE A Marriage of Convenience: Prolog and a Relational DBMS", Internal report KB-9, European Computer-Industry Research Centre, Munich, Sept.'85.

[2] Wong, K.F. & Williams, M.H. "Partial Matching Hardware for Clause Retrieval in Large Prolog Databases", Technical Report 88/2, Dept. of Computer Science, Heriot-Watt University, Edinburgh, Scotland, 1988.

[3] Li, D., A Prolog Database System, Research Studies Press, London 1984.

[4] Berra, P.B., Chung, S.M. & Hachem N.I., "Computer Architecture for a Surrogate File to a Very Large Data/Knowledge Base", IEEE Computer, pp25-32, March'87.

[5] Warren, D.H.D., "Logic Programming and Knowledge Bases", Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems, pp69-72, Islamorada, Florida, February'85.

[6] Williams,M.H., Massey, P.A. & Crammond, J.A., "Benchmarks for Prolog from a Database Viewpoint", Proceedings of 2nd Alvey Workshop for SIGKME, ppC1-C8, Brunel University, England, 1987.

[7] Williams,M.H., Massey, P.A. & Crammond, J.A., "Benchmarking Prolog for Database Applications", Prolog and Databases: Implementations and New Directions, Ellis Horwood Series in AI, edited by P.M.D. Gray and R.J. Lucas, pp161-187, 1988.

[8] Williams, M.H., "The Design of a Prolog Database Machine", Proceedings of the 1st Workshop for SIGKME, ppL1-L5, Reading University, England, Jan'87.

[9] Clocksin, W.F., "Design and Simulation of a Sequential Prolog Machine", New Generation Computing, vol. 3, pp101-120, 1985.

[10] Wong, K.F. "Comment on *A Comparison of Concatenate and Superimposed Code Word Surrogate Files For Very Large Data/Knowledge Bases*", Technical Report 88/6, Dept. of CS, Heriot-Watt University, Edinburgh, Scotland, 1988.

[11] Ramamohanarao, K. & Shepherd, J. "A Superimposed Codeword Indexing Scheme for Very Large Prolog Database", Proceedings of 3rd International Logic Programming Conference, London, England, July'86.

[12] Wong, K.F. & M.H. Williams, "Design Considerations for a Prolog Database Engine", Proceedings of the 3rd International Conference on Data and Knowledge Bases, pp111-120, Jerusalem, Israel, 28-30 June'88.

[13] Sun Microsystems, Inc., "Writing Device Drivers for the Sun Workstations", Sun Workstation Manual, February'86.

[14] Advanced Micro Devices, Inc, "Architectures Using the 2910A", Bipolar Microprocessor and Logic Data Book, p(5)175 -177,1985.

Appendix I: Data Types Supported in the Pseudo In-line Format

Table A1: CLARE Data Type Scheme

Items	Type Tag (1b)	Content (b/3b)	Extension (4b)
Variables:			
Anonymous Var	0010 0000 (0x20)	-	-
First Query Var	0010 0111 (0x27)	Variable Offset (b)	-
Subsequent Query Var	0010 0101 (0x25)	Variable Offset (b)	-
First DB Var	0010 0110 (0x26)	Variable Offset (b)	-
Subsequent DB Var	0010 0100 (0x24)	Variable Offset (b)	-
Simple Terms:			
Atom Pointer	0000 1000 (0x08)	Symbol Table Offset	-
Float Pointer	0000 1001 (0x09)	Symbol Table Offset	-
Integer In-line	0001 nnnn (0x1N)	Least Significant Value -	
	(nnnn = most significant nibble)		
Complex Terms:			
Structure In-line	011a aaaa	Functor Symbol Table Offset	-
	(aaaaa=arity, ≤31)	Structure Elements Follow	
Structure Pointer	010a aaaa	Functor Symbol Table Offset	Pointer to Structure
	(aaaaa=arity, ≤31)		
Terminated List In-line	111a aaaa	-	-
	(aaaaa=arity, ≤31)	List Elements Follow	
Unterminated List In-line	101a aaaa	-	-
	(aaaaa=arity, ≤31)	List Elements Follow	
Terminated List Pointer	110a aaaa	Pointer to List (4b)	-
	(aaaaa=arity, ≤31)	Only Use in DB Argument	
Unterminated List Pointer	100a aaaa	Pointer to a List (4b)	-
	(aaaaa=arity, ≤31)	Only Use in DB Argument	

1 if both types are integers then
compare their contents (numbers)

2 if both types are atoms OR float then
compare their contents (hashed values to the symbol table)

3 if both types are structures then
compare their contents (functor names and arities &
contents of top level elements)

4 if both types are lists then
compare their contents (length of list and types &
contents of top level elements)

5 if the database type is a variable then
check for the existence of the clause variable
5a if it does not exist then
create a new entry in the DB variable store
associate it with the query term
5b if it exists then
extract its association from the DB variable store
5c if it is a variable itself then
fetch the ultimate association from the DB store
repeat the comparison operations again

6. if the query type is a variable then
check for the existence of the query variable
6a if it does not exist then
create a new entry in the Query variable store
associate it with the database term
6b if it exists then
extract its association from the Query variable store
6c if it is a variable itself then
fetch the ultimate association from the DB store
repeat the comparison operations again

Figure 1: The simplified partial test unification algorithm.

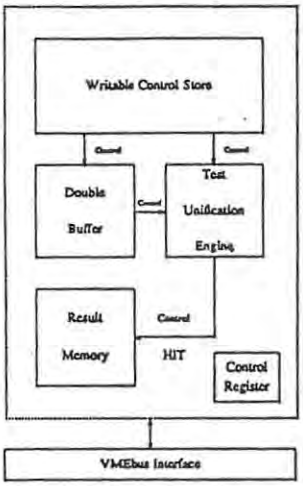


Figure 2: The overall architecture of the second stage filter hardware (FS2).

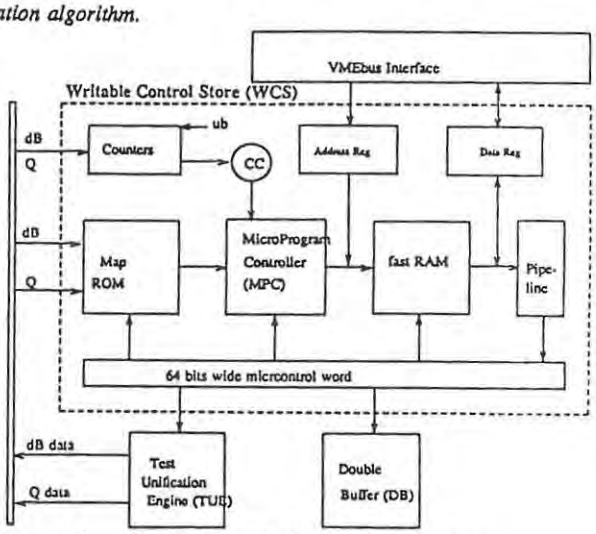


Figure 3: The Writable Control Store (WCS).

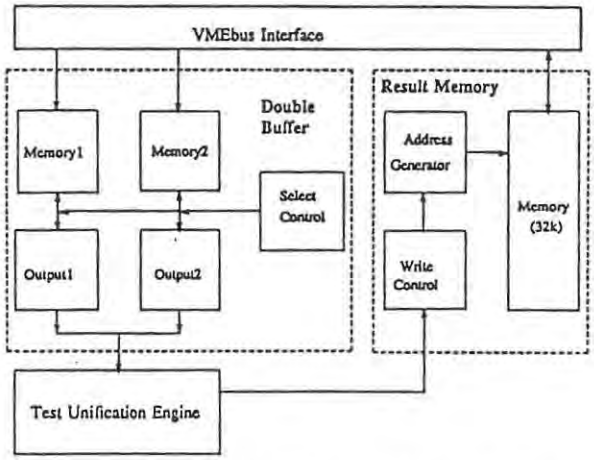


Figure 4: The Double Buffer and the Result Memory.

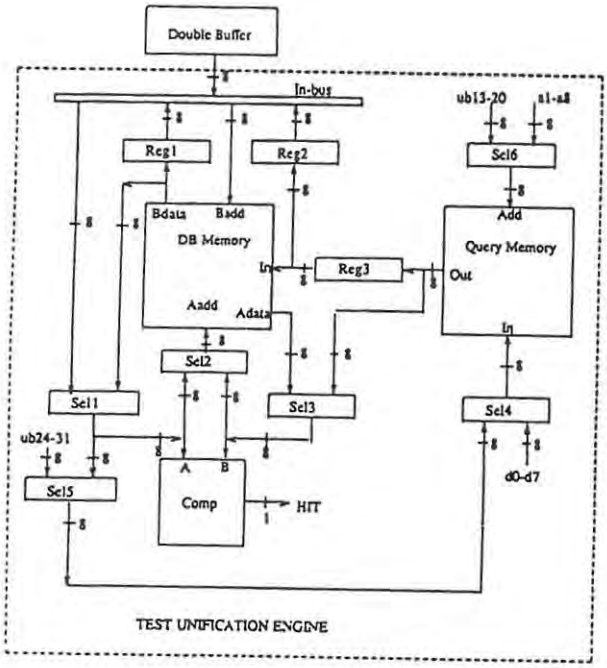
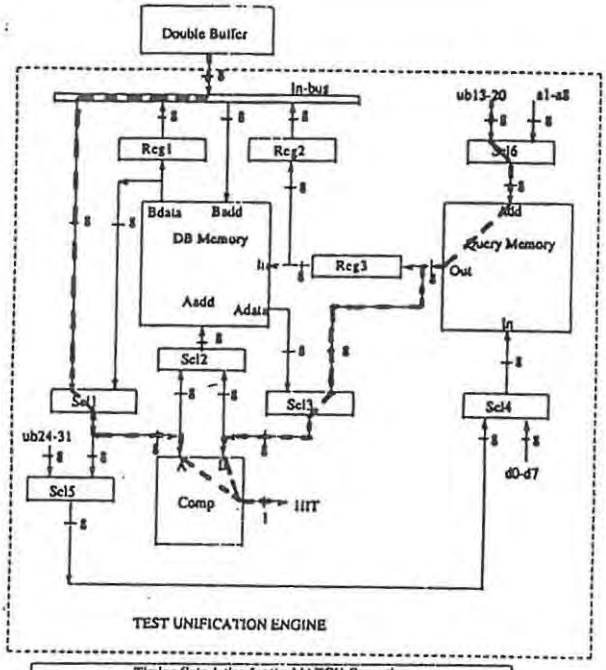


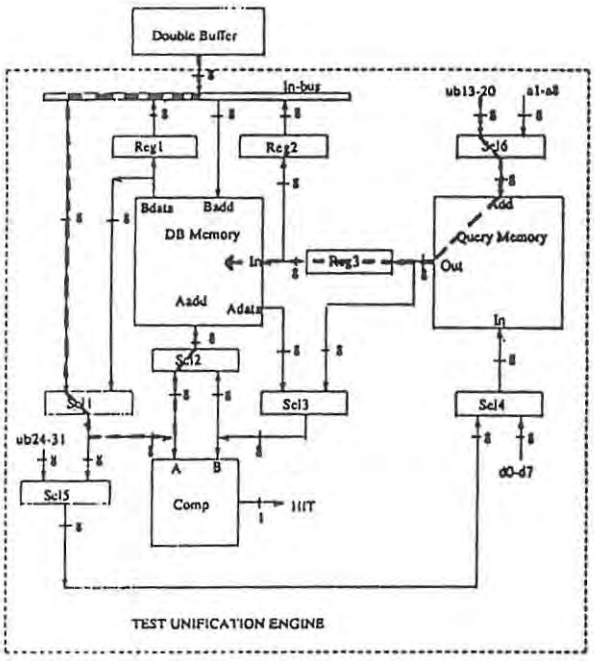
Figure 5: The Test Unification Engine.



Timing Calculation for the MATCH Operation							
database route :	Double Buffer	→	Sel1				
	20		20		(=40)		
query route :	Sel6	→	Query Memory	→	Sel3		
	20		35		20		(=75)
Comparison :	(=30)						

execution time = query route (75) + comparison (30) = 105ns.

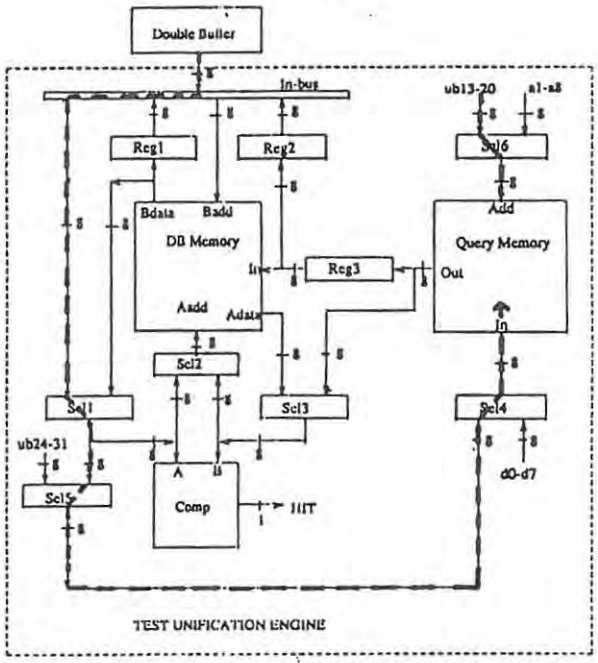
Figure 6: The MATCH Operation



Timing Calculation for the DB_STORE Operation							
database route :	Double Buffer	→	Sel1	→	Sel2		
	20		20		20		(=60)
query route :	Sel6	→	Query Memory	→	Reg3		
	20		35		20		(=75)
DB Memory write :	(=20)						

execution time = query route + DB Memory write = 95ns

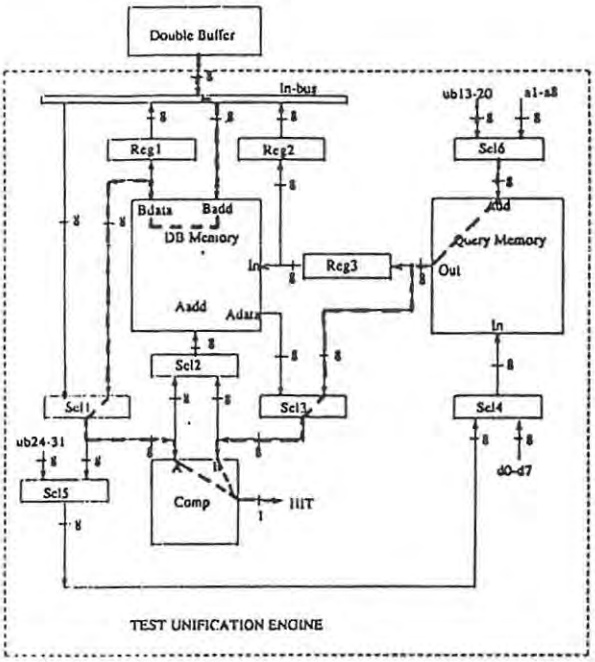
Figure 7: The DB_STORE Operation



Timing Calculation for the QUERY_STORE Operation								
database route :	Double Buffer	→	Sel1	→	Sel5	→	Sel4	
	20		20		20		20	(=80)
query route :	Sel6							
	20							
Query Memory write :	(=25)							

execution time = database route + Query Memory write = 105ns

Figure 8: The QUERY_STORE Operation

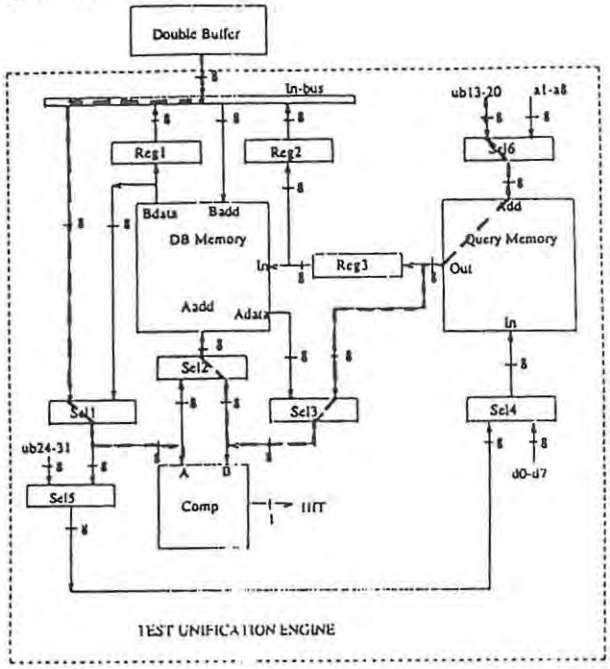


Timing Calculation for the DB_FETCH Operation							
database route :	Double Buffer	→	DB Memory	→	Sel1		
	20		25		20		(=65)
query route :	Sel6	→	Query Memory	→	Sel3		
	20		35		20		(=75)
comparison:	(=30)						

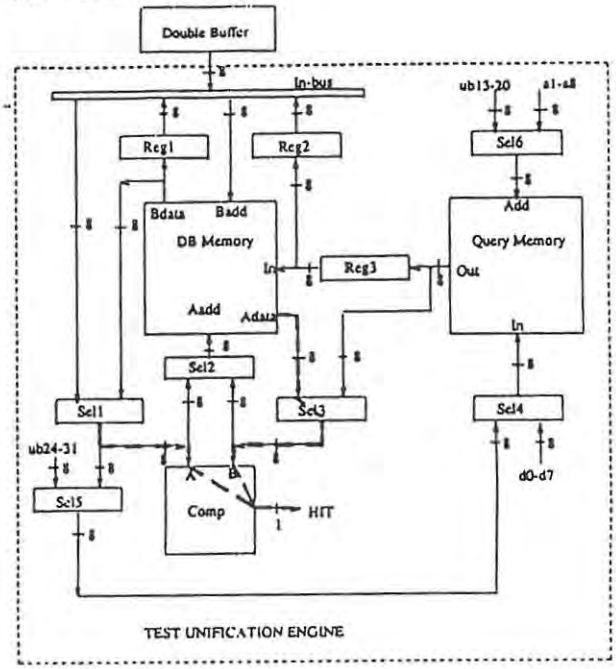
execution time = query route (75) + comparison (30) = 105ns.

Figure 9: The DB_FETCH Operation

(a) First Cycle



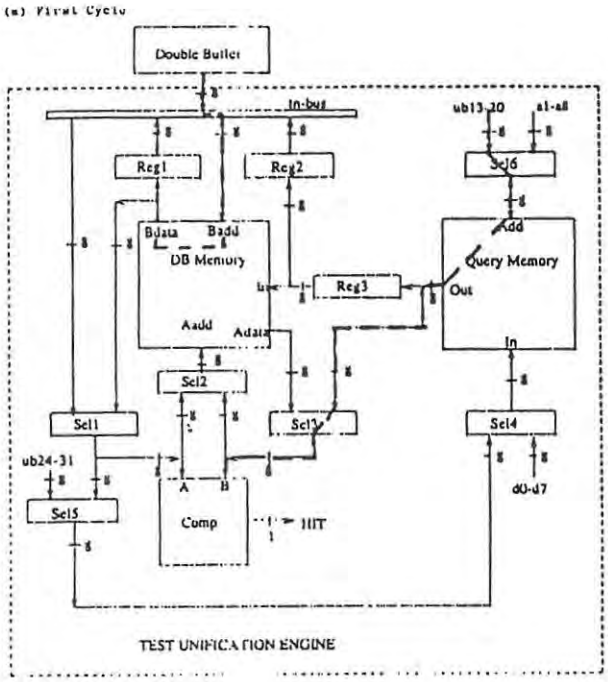
(b) Second Cycle

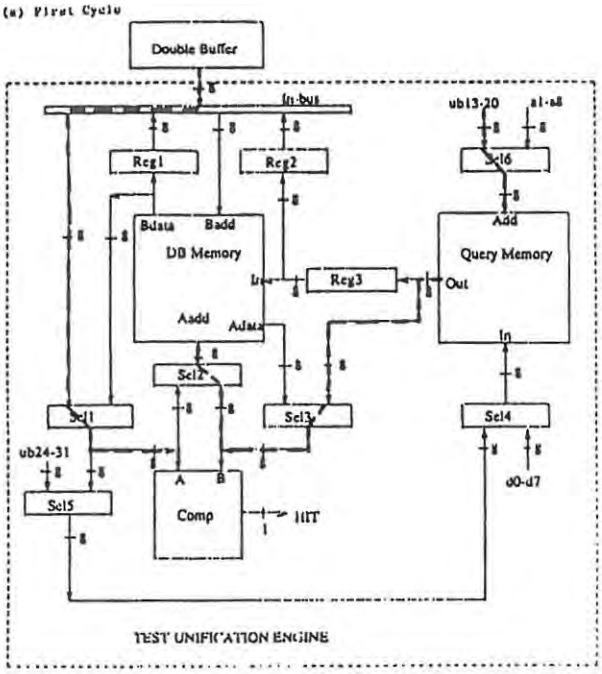


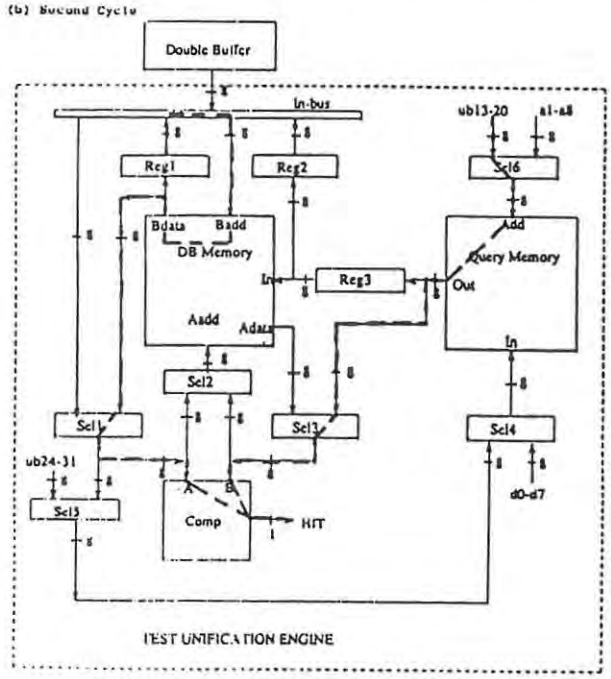
Timing Calculation for the QUERY_FETCH Operation										
First cycle										
database route :	Double Buffer	→	Sel1							
	20		20	(=40)						
query route :	Sel6	→	Query Memory	→	Sel3	→	Sel2	→	DB Memory	
	20		35		20		20		25	(=120)
Second cycle										
database route :	set at 1st cycle									
query route :	Sel3									
	20									
Comparison :	(=30)									

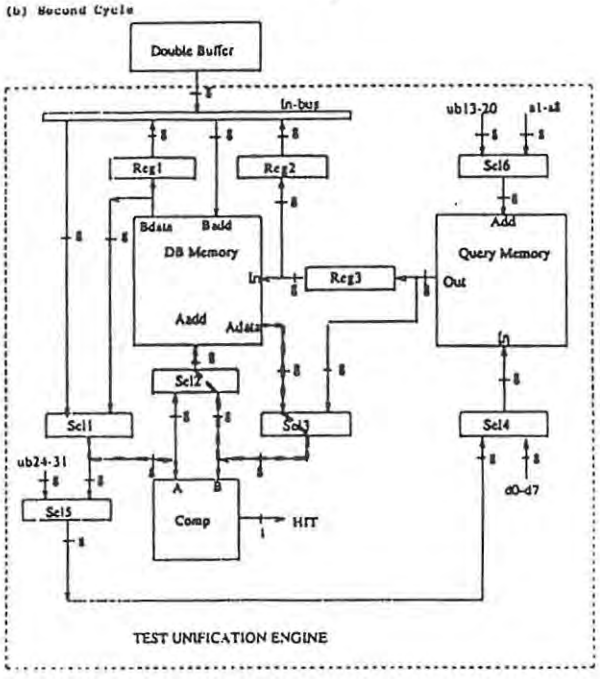
execution time = 1st cycle query route (120) + 2nd cycle query route (20) + comparison (30) = 170ns.

Figure 10: The QUERY_FETCH Operation







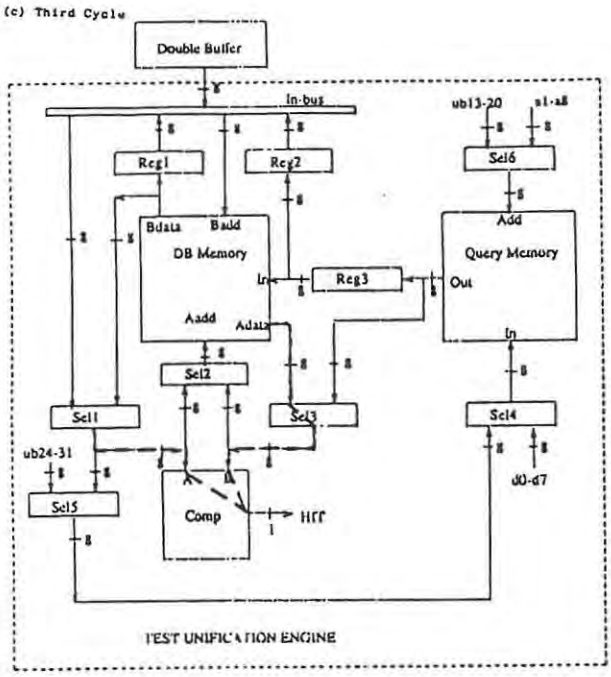


Timing Calculation for the DB_CROSS_BOUND_FETCH Operation						
First cycle						
database route :	Double Buffer	→	DB Memory	→	Reg1	
	20		25		20	(=65)
query route :	Sel6	→	Query Memory	→	Sel3	
	20		35		20	(=75)
Second Cycle						
database route :	Reg1	→	DB Memory	→	Sel1	
	20		25		20	(=65)
query route :	set at 1st cycle					
comparison :	(=30)					

Execution time = 1st cycle query route (75) + 2nd cycle database route (65) + comparison (30) = 170ns.

Figure 11: The DB_CROSS_BOUND_FETCH Operation

Figure 12: The QUERY_CROSS_BOUND_FETCH Operation



Timing Calculation for the QUERY_CROSS_BOUND_FETCH Operation								
First cycle								
database route :	Double Buffer	→	Sel1					
	20		20	(=40)				
query route :	Sel6	→	Query Memory	→	Sel3	→	Sel2	
	20		35		20		20	(=95)
Second cycle								
database route :	set in 1st cycle							
query route :	DB Memory	→	Sel3	→	Sel2			
	25		20		20	(=65)		
Third cycle								
database route :	set at 1st cycle							
query route :	DB Memory	→	Sel3					
	25		20	(=45)				
comparison :	(=30)							

execution time = query routes of (1st cycle (95) + 2nd cycle (65) + 3rd cycle (45)) + comparison (30) = 235ns

Figure 12: The QUERY_CROSS_BOUND_FETCH Operation

(Continuation)

Table 1: Execution Times of the FS2 Hardware Functions		
Corresponding Figure	Operations	Execution times (ns)
6	MATCH	105
7	DB_STORE	95
8	QUERY_STORE	115
9	DB_FETCH	105
10	QUERY_FETCH	170
11	DB_CROSS_BOUND_FETCH	170
12	QUERY_CROSS_BOUND_FETCH	235

40. K.F.WONG and M.H.WILLIAMS, CLARE - a clause retrieval engine for large Prolog databases, Proceedings of the IEE Colloquium on VLSI and Architecture for Symbolic Processing, IEE Digest No. 1989/37, 3/1-3/3 (1989).

CLARE - a CLAuse Retrieval Engine for Large Prolog Databases

Kam-Fai Wong and M.H. Williams†

1. Introduction

Although the logic programming language Prolog was originally designed to handle AI applications, its suitability for implementing database applications has been recognised for some while now. Its attractiveness for this purpose lies in the additional benefits which it can provide. It enables database systems to be developed which are considerably more powerful than current relational databases - systems which can store, retrieve and manipulate knowledge in the form of logical rules as well as conventional data in the form of facts. Such systems are generally referred to as deductive or logic database systems. However, existing Prolog systems are limited in their capacity to deal with large volumes of data.

In the future it is anticipated that there will be a need for large Prolog databases comprising millions of clauses (at the least). Warren [1] envisaged a medium sized Prolog database to contain some tens of thousands of rules and several million facts. For various reasons it may be desirable or necessary to keep such databases on secondary disc memory rather than in main memory. However, retrieval of clauses from disc may be very time consuming and this can seriously undermine the performance of such a database system. A suite of benchmark has been designed [2] to evaluate the performance of Prolog systems in handling large data set; when this was applied to several existing Prolog systems on a 4MB SUN3 workstation, none was able to handle more than 70,000 clauses.

The basic unit of information in Prolog is the clause. Each clause may be a rule or a fact (corresponding to a tuple in a relational database). The process of matching a Prolog query against a clause in the database, termed *unification*, is much more complex than the simple associative matching performed in relational database systems, and consumes a considerable proportion of the execution time of a Prolog process. For a large database stored on disc, the time taken to attempt to unify a query against each clause in the database in turn would be intolerable. This paper describes a special purpose engine which has been designed to overcome this problem.

2. System Overview

Two possible approaches which could be used to improve the efficiency of Prolog in handling large clause sets are :

(1) Improve the speed of unification

(2) Reduce the number of clauses

Both of these approaches have been adopted in the Prolog Database Machine (PDBM) system. The PDBM system provides a combined hardware/software solution to the problem. On the software side clauses are compiled incrementally and stored in clusters according to their name and arity (number of arguments). Storing them in compiled form facilitates direct unification at retrieval time and clustering localises the region for unification. Since clusters may be large, a dedicated hardware engine - CLAUse Retrieval Engine (CLARE) - is being developed which will selectively retrieve disc resident clauses. The communication between the Prolog system and CLARE is provided by a software clause retrieval server. The overall architecture is illustrated in figure 1.

CLARE is a coprocessor system which sits on the VMEbus of a SUN3/160 workstation. A 2-stage filtering process is used to achieve selective clause retrieval. Filtering is applied to disc-resident clauses as they pass from the disc on route to the host. The first stage filter (FS1) is based on *Superimposed Codeword plus Mask Bit (SCW+MB)* index matching technique and the second stage (FS2) is based on *Partial Test Unification*. At the beginning of a query, the clause retrieval server decides the degree of utilisation of CLARE. Depending on the nature of the query, there are four possible run time configurations: without CLARE - software only, FS1 only, FS2 only, and both filters.

† Department of Computer Science, Heriot Watt University, 79 Grassmarket, Edinburgh, EH1 2HJ.

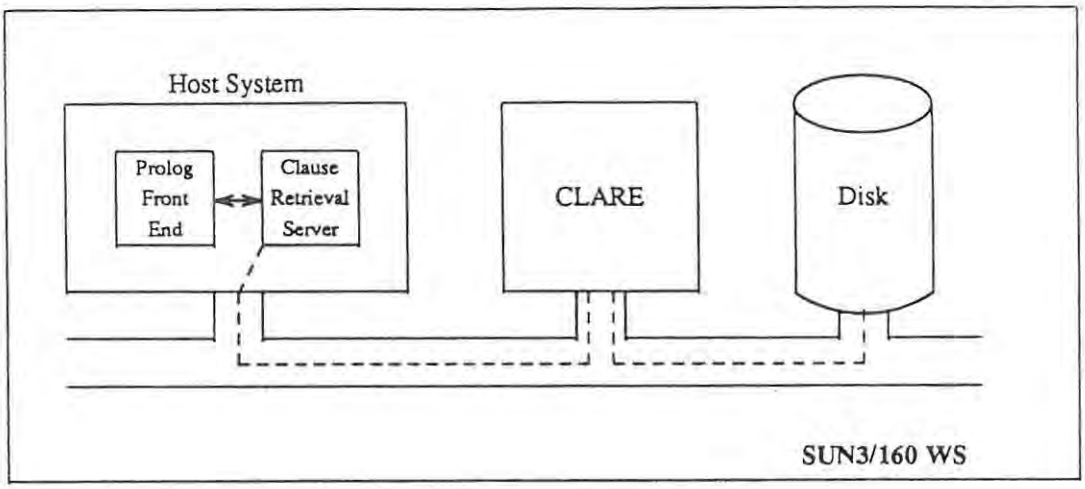


Figure 1: Overall architecture of the PDBM system.

3. First Stage Filter

To facilitate efficient information retrieval, indexing techniques are commonly used in conventional database systems (e.g. inverted list). However, most of the conventional indexing techniques are unsuitable for Prolog because they cannot handle variables and complex structures; even if some of them can, the size of the index file can be fairly large. For this reason, the SCW+MB indexing strategy has been adopted in the PDBM system. A SCW+MB codeword is formed by combining (or'ing) the binary codewords of a clause's arguments to form the SCW and by setting up a bit pattern which reflects positional information of the arguments to form the MB. When a query is posed, individual query arguments are hashed to form a set of binary codewords. If an argument is a variable, it is regarded as a don't care. The binary codewords of a query are not combined but are matched against database SCW+MB indexes in turn. A database index is a satisfier (or hit) if each binary codeword is either a subset of the SCW or the bit pattern on the MB indicates that the corresponding database argument is a variable.

SCW+MB index matching is performed in parallel by the FS1 hardware [4]. The FS1 is a 16 bit processor. It consists of 12 identical matching units and some result memory. Each unit is responsible for the matching of a query argument. It consists of a bank of query memory, an I/O register and a comparator. A query binary codeword is stored in the query memory which is mapped to the host's physical memory space. A maximum of 2048 bits can be stored in the query memory. The FS1 is logically connected to the disc controller via the I/O register. During a disc transfer, data is written to the register in 16 bit words. The outputs of the query memory and the I/O register form the two operands for the matching process executed by the comparator. The single bit outputs of the 12 matching units are connected together through a 13-input NAND gate. The output of the NAND gate is latched at the end of every database index. An index satisfier is signified by a '0' from the NAND gate. The result memory responds to this signal and captures the clause reference at the end of each index satisfier. These references are used later to retrieve the content of the clauses from the clause file. The FS1 performs index matching on a track-by-track basis. The result memory has been designed for the worst case and is large enough to accommodate the clause references of a track full of index satisfiers.

The FS1 hardware is fully developed using 180 LSI/MSI components. Low level hardware performance tests using a logic analyser shows that the maximum search rate of the FS1 hardware is 4.5M word/second [5]. PCB copies of the hardware are being constructed and they will be subjected to high level evaluation.

4. Second Stage Filter

SCW+MB indexing is an inexact matching technique. In practice, a small proportion of clauses resulting from SCW+MB indexing will fail to match the query during full unification. These clauses are commonly referred to as

'ghosts' or 'false drops'. In order to reduce the number of false drops, clauses extracted from the compiled file, based on the references of the index satisfiers collected from the first stage, are passed on to the second stage filter (FS2) where they are subjected to additional discrimination.

The FS2 hardware provides CLARE with enhanced selectivity based on a custom partial test unification algorithm. It operates on the clause file. A compiled clause in the clause file consists of 2 separate data segments : the *body segment* contains directly executable abstract machine code of the clause body and the *head segment* contains the list of the arguments appearing in the clause head. Head arguments are represented in a semi-compiled - Pseudo-In-line data Format (PIF). In PIF, an argument is represented as a tagged object which consists of a type tag followed by an identifier. At present, 107 data types are supported. An identifier may be a symbol table reference, a variable reference or a constant. Logically, the partial test unification algorithm is composed of a series of **if** *<condition>* **then** *<action1>* **else** *<action2>* control statements. The type tags of the query and the clause arguments are compared in the *<condition>*. If the result of the comparison is favourable, the clause currently under examination is retained; otherwise, the portion of the memory where it resides will be overwritten by the next clause. The algorithm is specially designed to handle multiple occurrences of variables and embedded complex structures which are the two main sources accounting for the FS1's false drops. At the implementation level, the FS2 hardware is being constructed using bit-slice devices. The partial test unification algorithm is microcoded into firmware and controls the hardware directly. The maximum search rate of the FS2 is estimated to be 4.3 Mbyte/second.

5. Summary

A dedicated Prolog database machine (CLARE) has been described. CLARE adopts a 2-stage filtering architecture and performs selective retrieval on disc resident clauses on-the-fly. The search rate of CLARE is fast enough to handle data transfers from existing single disc systems. Detailed descriptions of the FS1 and FS2 hardware units are given in references [4] and [6] respectively.

Acknowledgement

The Prolog Database Machine (PDBM) Project is a collaborative project with ICL and is sponsored by SERC and DTI under the ALVEY initiative (project number IKBS/037).

Reference

[1] Warren, D.H.D., "Logic Programming and Knowledge Bases", Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems, Florida, USA, pp69-72, (Feb'85).

[2] Williams, M.H., Massey, P.A. & Crammond, J.A., "Benchmarking Prolog for Database Applications", Prolog and Databases: Implementations and New Directions, Ellis Horwood Series in AI,pp161-187,(1988).

[3] Williams, M.H., "The Design of a Prolog Database Machine", Proceedings of the 1st Workshop of the SIGKME, Reading University, ppL1-L5,(Jan'87).

[4] Wong, K.F., Williams, M.H., "Index Matching Hardware for Selective Clause Retrieval in Large Prolog Knowledge Bases", submitted to IEEE Computer, (1989).

[5] Wong, K.F., Williams, M.H., "CLARE - A Prolog Database Machine" submitted to IEE Electronic Letters, (1989).

[6] Wong, K.F., Williams, M.H., "A Type Driven Hardware Engine for Prolog Clause Retrieval over a Large Knowledge Base", to be published in the 16th Annual International Symposium on Computer Architecture, IEEE/CS, (May28-June1'89).

41. K.F.WONG and M.H.WILLIAMS, CLARE - A Prolog database machine, in Proceedings of 1990 Sigsmall/PC Symposium on Small Systems, Arlington, Virginia, USA, 29-30 March 1990, 172-177 (1990).

CLARE - A Prolog Database Machine

Kam-Fai Wong

ECRC GmbH, Arabellastrasse 17, D8000 München 81, West Germany.

M. Howard Williams

Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ, Scotland.

ABSTRACT

The CLAuse Retrieval Engine (CLARE) is a coprocessor system which based on two-stage filtering, for handling large sets of disc resident clauses in Prolog database applications. The overall architecture and the timing measurements of the first stage CLARE hardware are reported. The timing measurements shows that the first stage CLARE hardware can perform search operations on data transferring at a rate up to 4.5 million double bytes per second.

Indexing Terms: Prolog database, knowledge base, Knowledge base machine, hardware index search

1. Introduction

The semantics of Prolog has been shown to be suitable for implementing various aspects of databases. However, at present the ideas are only being applied to small examples. This limitation is caused by the inefficiencies of existing Prolog systems in handling large sets of clauses [1]. A large Prolog database will consist of millions of clauses. The size of physical memory required to store the database has deemed main memory impractical in most computer systems. Therefore, clauses are usually placed on secondary disc memory. Clause retrieval, however, becomes extremely slow. Furthermore, the enormous amount of time required for unification upon a large set of clauses has drastically degraded system performance. Either of these factors leads to non-timely response which is irritating and in many cases intolerable. A special purpose CLAuse Retrieval Engine (CLARE) is being developed in order to increase the performance of a Prolog system in retrieval over large clause sets. A two-stage filtering approach is adopted by CLARE which employs two pieces of dedicated hardware to search disc resident clauses on-the-fly. The first stage hardware (FS1) is a hardware realisation of index searching based on *superimposed codeword plus mask bits* [2], while the second stage filter (FS2) is responsible for *partial test unification* [3]. The descriptions of the FS1 architecture and its timing are given in this paper.

Permission to copy without fee all or part of this matertial is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that the copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1990 ACM 089791-347-7/90/0003/0172 $1.50

2. Clause Indexing

In the Prolog database system under development, predicates with the same functor names and arities are stored in a compiled clause file. To achieve fast searching in large files, an index codeword is generated for each fact and rule head and these are maintained in a secondary file. The secondary file is effectively an index table associating codewords with clause addresses. When a query is given, the first stage filtering hardware (FS1) scans through the secondary file and extracts the addresses of those clauses whose codewords successfully match with the codeword corresponding to that query. The size of a secondary file is generally much smaller than that of a compiled clause file (<20%) [4], thereby enabling quicker retrieval to be achieved by scanning the former than by searching the latter exhaustively.

The Superimposed Code Word plus mask bit (SCW) indexing [5] approach is adopted. Under this approach, each database clause head is encoded. Encoding is based on OR-ing (superimposing) individual codewords of all atributes within a

clause. An attribute codeword is formed by setting k bits to 1 in a word of b bits width. The exact positions of 1's are calculated using a hashing function. Adjacent to the codeword some masked bits are attached whose bit pattern shows the variable positions amongst the attributes. When a query is made, the arguments of the query are encoded using the same hashing function. A database codeword is then compared with the first argument codeword of the query. The result is successful if either one of the following is true:

1) The query argument codeword is a subset of the database codeword.

2) The mask bit of the database clause indicates that this is a variable position.

A database clause is considered to be a match if it compares successfully with all query arguments The program given in figure 1, is an algorithmic representation of the matching process in C. Based on the program, matching is performed serially. since the database codeword (database_code) is matched against each query argument codeword (query_argument_code) one at a time, the matching process can be excessively slow. To speed up matching, comparisons between codewords is executed in parallel using the FS1 hardware.

3. The CLARE First Stage Filtering Hardware (FS1)

The FS1 hardware consists of two major functional units, figure 2: the Matching Unit and the Timing Control Unit. The Matching Unit is responsible to the parallel index matching operation between a database codeword and the query argument codewords. There are twelve RAMs within the Unit which forms the query space. Query codewords need be loaded into the RAMs before a search is initiated - one for each argument. The heart of the Matching Unit consists of twelve programmable array logic (PAL) devices. The contents of the PALs are effectively the digitised form of the matching algorithm, shown in figure 1. The outputs of the PALs are logically AND to form the final result of a match - the $\overline{HIT}$ signal. If the result is "positive", the corresponding clause reference is retained in the result memory; otherwise, the clause reference is discarded.

The Timing Control Unit provides the overall coordination of the FS1 hardware during index matching. When a database clause index is read from the disc, the signals generated from the unit signifies the field boundaries of the index. The format of an index file entry is fixed and takes the form of: <n> byte(s) of mask bits followed by <m> byte(s) of codeword followed by <o> byte(s) of clause reference. At the beginning of a query, the values of <n>, <m> and <o> are loaded into the Timing Memory in the Unit. Based on these values, the Unit will respectively generate the "Mask_bit", "Codeword" and "Clause_ref" marker signals during index matching. Different operations are performed under the marker signals (see the next section).

To initiate a search, the FS1 is programmed with the appropriate parameters. The FS1 address is then written to the disc controller as the data transfer destination address and a disk read operation is then initiated. From then on the disk controller regards the FS1 as if it was main memory and transfers disc resident data to the hardware using direct memory access (DMA).

A prototype of the FS1 is constructed using 188 off-the-shelf LSI/MSI ICs (mounted on a triple height speed wire board). It adopts the VMEbus interface standard and is connected to a SUN3/160 workstation with a 142M byte disc system. External devices, including the host and the disc controller, access the FS1 via the interface. The FS1 is designed to be a slave module on the bus so it cannot acquire bus mastership. There are several protocol for data transfer under the VMEbus specification. The A24/D16 protocol is adopted by the FS1. The hardware cofiguration of the CLARE prototype is shown in figure 3.

4. Testing

A demonstration program was written for testing the FS1 prototype. Using the program, a sample database containing 2048 clauses was generated. All database clauses are instances of the same predicate which consists of 12 arguments (arity=12). A secondary index file of the database was also generated. Each index entry encompasses 16 mask bits (only 12 bits are used), a 64 bit codeword and a 16 bit clause reference. These value were initially loaded into the timing memory. Alternative index satisfiers and failures were included deliberately in the database in order to generate a alternating $\overline{HIT}$ signal. A logic analyser (National Panasonic VP-3620A) was connected to various parts of the FS1 prototype for investigating the timing of the hardware. Figures 4a and 4b show two different instances of a snapshot taken from the logic analyser output. The logic analyser was set to operate under a 20MHz internal clock. (CLOCK = 50nsec). Only the 8 input lines on channel A of the logic analyser were utilised and they were programmed to trigger at the beginning of a new index entry (line#4, see below). The connections to the 8 input lines are summarised as follows :

Line#0	Data input ($\overline{DIN}$)
Line#1	Mask_bit ($\overline{MB}$, marker signal)
Line#2	Codeword ($\overline{CW}$, marker signal)
Line#3	Clause_ref ($\overline{CR}$, marker signal)
Line#4	Reset ($\overline{RESET}$, marker signal)
Line#5	Latched Hit signal ($*\overline{HIT}*$)
Line#6	Unlatched Hit signal ($\overline{HIT}$)
Line#7	Unused

Certain observations can be made from the figures. The FS1 hardware is a 16 bit machine. Each negative pulse in line#0 corresponds to a 16 bit data transfer from the disc to the FS1. It takes 225nsec - this is measured using a oscilloscope - to complete a 16 bit data transfer cycle. Data are input in pairs because the disc controller acquire VMEbus mastership for long enough to transfer a double word. Different portions of the snapshot were highlighted - areas enclosed between the vertical lines C and R - in figures 4a and 4b. In figure 4a, lines C and R enclose a transfer

of an entire index entry. It is worth noting that the index entry is transferred in six data transfer cycles ($\overline{DIN}$ of line#0 between C and R) which agrees with the format of the index of the sample database.

The sequence of event of the FS1 during index matching is clearly depicted in figure 4a. At the end of the previous index entry, the $\overline{RESET}$ pulse is asserted - line#4 on the left hand side of C. This has the effect of reseting the Timing Control Unit causing it to assert $\overline{MB}$ (line#1). Data input during $\overline{MB}$ is latched into the mask_bit register (MB reg). At the end of $\overline{MB}$, $\overline{CW}$ (line#2) is asserted. The 12 PALs are enabled and matching are performed. The outputs of the PALs are logically AND and produces $\overline{*HIT*}$ (line#6). At the end of the codeword, the rising edge of $\overline{CW}$ are used to latch $\overline{*HIT*}$ which produces $\overline{HIT}$ (line#5). At the same time, $\overline{CR}$ (line#3) is asserted. During $\overline{CR}$, data will be written to the result memory if $\overline{HIT}$ is low. The end of an index entry is signalled by the rising edge of $\overline{CR}$. The $\overline{RESET}$ pulse is generated and the matching process is repeated.

In practice, the index matching process for each word input from the disk occurs during $\overline{DIN}$ and requires 225ns. The matching of the index shown in figure 4a is a satisfier which is determined by the $\overline{HIT}$ (line#5) at the end (rising edge) of $\overline{CW}$ (line#2). Notice that the state of $\overline{HIT}$ stays the same until the end of $\overline{CW}$ of the next index entry; whereas $\overline{*HIT*}$ (line#6) is cleared the the beginning of every index entry. $\overline{CW}$ is pulled high as soon as a mismatch occurs. This is exemplified in figure 4b. The vertical line C in the figures marks the start of an index entry and a mismatch is detected at the first data input in the codeword phase, marked by line R.

5. Conclusion

The result of the test has shown that the FS1 hardware offers a maximum search rate of 4.5 Mword/sec which is much faster than the data transfer rate of any existing disk systems. The bottleneck of a FS1 based Prolog database, therefore, lies in disc throughput. One way to speed up disc transfers is to reduce the number of head movement. Various approaches for reducing disc head movement are under investigation.

Acknowledgements

The work described in this paper is part of a collaborative project between ICL(U.K.) Ltd. and Heriot-Watt University, funded by the Science and Engineering Research Council (SERC) and the Department of Trade and Industry (DTI) through the Avely Grant (numbered IKBS/037). Also, thanks are due to ECRC GmbH for sponsoring the first author to attend the conference.

Reference

[1] Williams, M.H, Massey P.A. and Cramond J.A., "Benchmarking Prolog for Database Applications",Prolog and Databases: Implementations and New Directions, Ellis Horwood Series in AI, (ed. P.Gray & R. Lucas), pp161-187, 1988.

[2] Wong, K.F. and Williams, M.H., "Index Matching Hardware for Selective Clause Retrieval in Large Prolog Knowledge Bases", December, 1988. (submitted to IEE Computer & Control journal U.K.)

[3] Wong, K.F. and Williams, M.H., "A Type Driven Hardware Engine for Prolog Clause Retrieval over a Large Knowledge Base", 16th Annual International Symposium on Computer Architecture, Jerusalem, Israel, pp211-222, 28May-1June'89.

[4] Wong, K.F., "Comments on 'A Comparison of Concatenated And Superimposed Code Surrogate Files For Very Large Data/Knowledge Bases'", Information Processing Letters, pp45-52, vol. 33, 27Oct'89.

[5] Ramamohanarao, K. and Shepherd, J., "A Superimposed Codeword Indexing Scheme for Very Large Prolog Databases". Technical Report 85/17. Dept. of C.S., University of Melbourne, Australia, 1985.

```
#define BOOLEAN        unsigned char
#define BITWISE_AND    &
#define OR             ||
#define RIGHT_SHIFT_BY >>

BOOLEAN index_matching(query_argument_code,query_argument_count,database_code)
unsigned char query_argument_code[];
unsigned char query_argument_count;
unsigned char database_code;
{
    BOOLEAN Satisfier;
    char count;

    for (count=1; count!=query_argument_count; count++)
    {
        if((((query_argument_code[count] BITWISE_AND database_code))==database_code) OR
          (((database_mask RIGHT_SHIFT_BY (count-1)) BITWISE_AND 1) == 1))
            Satisfier=1; else { Satisfier=0; break; }
    }
    return(Satisfier);
}
```

Figure 1: Superimposed Codeword plus Mask Bit matching algorithm.

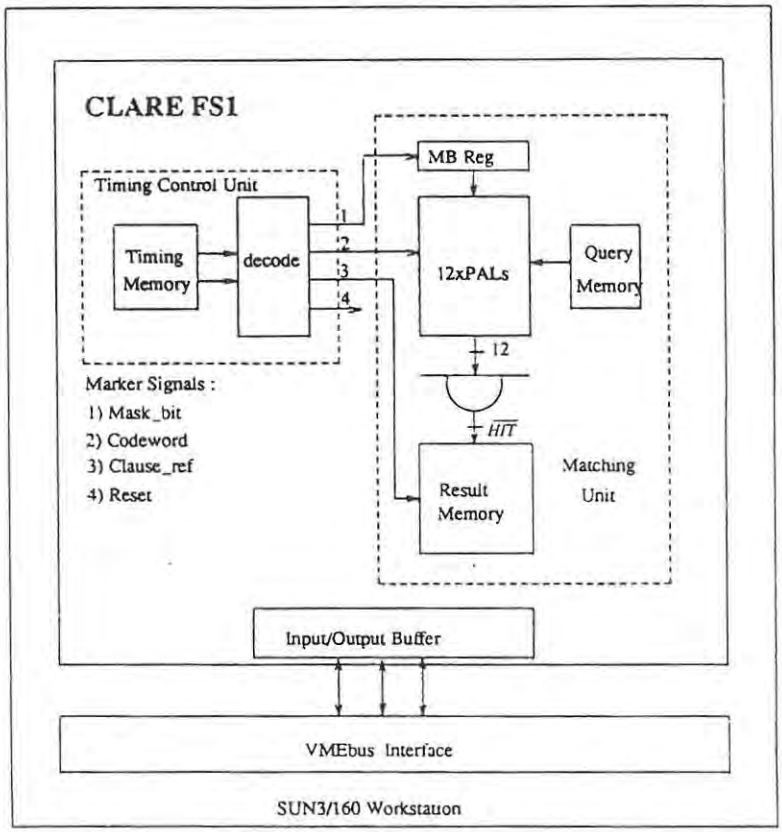


Figure 2: The schematic block diagram of the FS1 hardware.

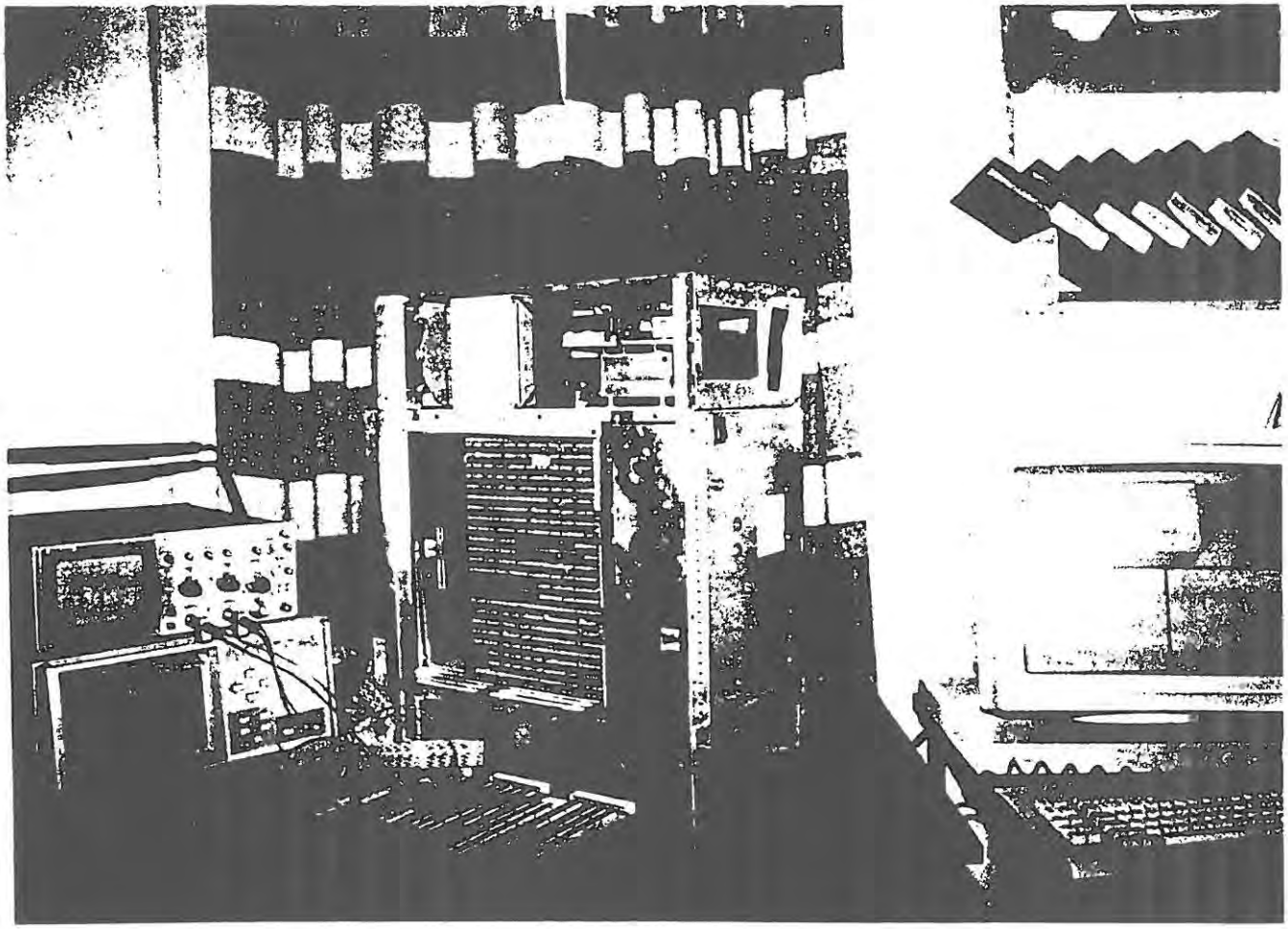
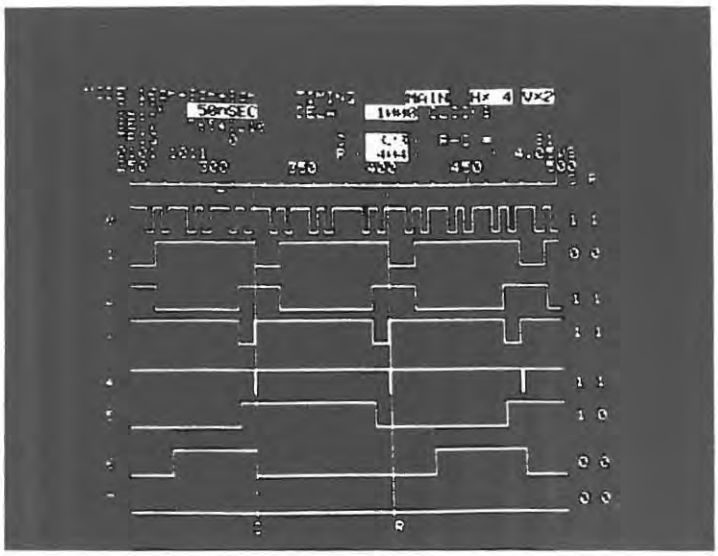
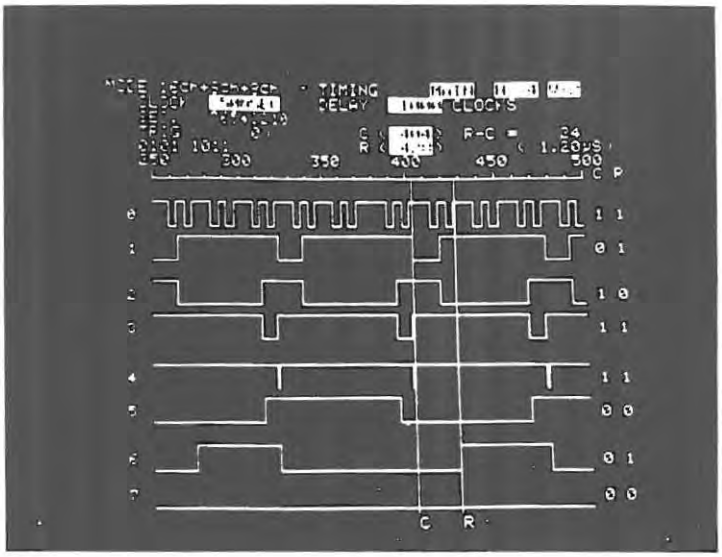


Figure 3: The hardware configuration of the CLARE prototype.



(a) Data transfer of one clause index to FS1 is between the vertical lines C and R.



(b) A failure match is shown between C and R

Figure 3: Snapshots of various internal signals in the CLARE FS1 hardware.

42. S.ZHOU and M.H.WILLIAMS, Assessment of two-level codeword indexing applied to Prolog databases, Information and Software Technology, vol. 33, 157-162 (1991).

Assessment of two-level codeword indexing applied to Prolog database

S Zhou and M H Williams

While interest in the use of Prolog for database applications is growing, the size of such applications is limited by the ability of current implementations of Prolog for handling disc-resident clauses. Various software and hardware approaches, such as codeword indexing, have been put forward to solve this problem. The paper reports on a study of the comparative performances of one-level and two-level codeword indexing. Some benchmarks have been used to evaluate the two strategies. The performance figures obtained show that the two-level codeword indexing strategy provides a more efficient retrieval method in most cases, though it has drawbacks in some cases. To improve the performance of the codeword indexing strategy, an improved approach is presented that combines both one-level and two-level indexing.

Prolog, databases, codeword indexing, database retrieval

A major difficulty that arises when searching large sets of clauses in Prolog is the time required to find a matching clause head for a given query, or to find that none exists[1,2]. While the number of clauses in a Prolog database is becoming larger and larger, the size of Prolog database applications is limited by the ability of current implementations of Prolog for handling disc-resident clauses. In an attempt to solve this problem, various software and hardware approaches have been put forward to improve the efficiency of retrieval, especially from disc[1,3,4].

In particular, there has been some interest in approaches based on the concept of a codeword, e.g., SCW(Superimposed codeword), FEW(field encoded codeword), etc.[1,3]. However, even these techniques can be slow for a large set of clauses as searching sequentially through a codeword index can be time consuming, especially in large databases. One way to reduce the time to search through the index file is to have a further index into the index file, called the 'block descriptor' or 'block index'. This is referred to as two-level codeword indexing[5].

This paper reports on a study of the comparative performances of one-level and two-level codeword indexing. An approach combining the two is presented that aims to improve the performance of the two-level codeword indexing scheme.

Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ, UK.
Paper submitted: 8 February 1990.
Revised version received: 22 June 1990.

IMPLEMENTATION OF TWO-LEVEL CODEWORD INDEXING

A Prolog system for developing database applications has been constructed based on three components[6,7]:

(1) A Prolog compiler, Prolog-X, which handles the execution of Prolog programs until disc-resident clauses are required, whereupon control is passed to (2).
(2) A Clause Retrieval Server, CRS, which is responsible for the storage and retrieval of clauses from disc. Where necessary, this invokes (3) to improve retrieval speed.
(3) A Clause Retrieval Engine, Clare, which is a hardware engine for fast search of a large set of clauses on disc.

The CRS provides various software strategies for searching clauses held on disc[8,9]. Two particular strategies of interest here are the superimposed codeword strategy and the field encoded codeword strategy.

Superimposed code word (SCW) strategy

The idea of SCWs was first proposed by Roberts[3] for the retrieval of partial-match queries from a relational database. Each record in the database is allocated a SCW, which is constructed by ORing together the codeword (hashed value) for each attribute in the record. This strategy was extended[5] to handle Prolog clauses by including a one-bit field, termed a mask bit, for each item in the clause head, which is set to 1 if the item is a variable and is zero otherwise. This is referred to as superimposed codeword plus mask bits (SCW + MB).

To deal with embedded structures, several mask bits are used (one bit per nested argument). Functor names and nested arguments of an embedded structure will also be encoded and matched.

To illustrate the process, consider a database of company/4 records, where each record has the following form

```
company(Name, Manager, Location, [Dept1, ..., Deptn])
```

Figure 1 shows the generation of SCW + MB for a database clause.

When a query is encountered, it is subjected to a simi-

 0950–5849/91/020157–07 © 1991 Butterworth–Heinemann Ltd

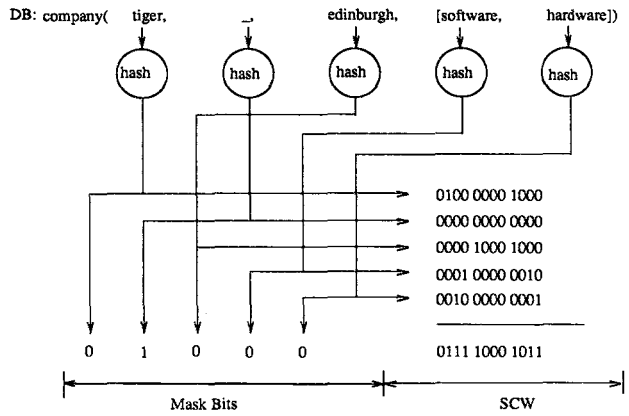


Figure 1. Generation of SCW + MB for database clause

lar treatment in that each query attribute is hashed to generate a corresponding binary codeword. In this case, however, mask bits are not required. Unlike simple SCW matching, these codewords are not ORed together, but instead are compared with database codewords individually. If each query codeword is a subset of the database codeword or the appropriate mask bit is set, the database clause corresponding to the database codeword is a satisfier.

SCW indexing does not guarantee that if the codeword in the index file of a clause matches a query codeword, then the clause matches the query. The only guarantee given is that if an index file codeword is not a satisfier then the clause cannot match the query. The retrieval of a record that fails the record comparison is called a false drop.

Field encoded codeword (FEW) strategy

In FEWs, the codeword allocated to each record in the database is partitioned into a number of fields, one for each attribute in the record. A smaller codeword (hashed value) is produced for each attribute and this is stored in the corresponding field of the FEW directly[1]. This approach can be used for Prolog clauses without introducing separate information on variables present. Any variables in the clause head are dealt with by setting each bit in the corresponding field to 1. Variables in the query are ignored.

The number of bits to be set in each partition of the codeword is

$$\frac{TotalWeight}{NumberOfArguments}$$

Structures (including lists) are dealt with by further subdividing both the weight and the bit string length.

Consider the previous example of the company/4 predicate, which describes the tiger company. An illustration of the generation of FEWs for the clause is given in Figure 2.

Comparison with other strategies

A number of methods have been developed to access records held on secondary storage. The inverted file[10] is

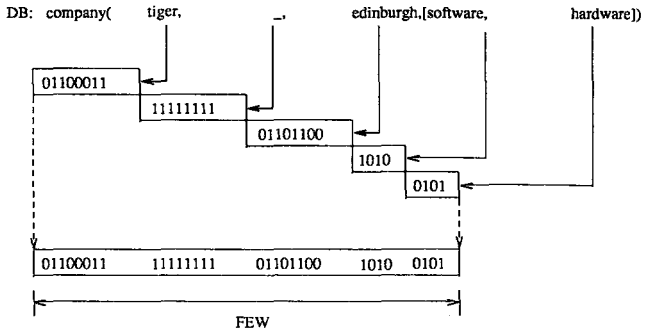


Figure 2. Generation of FEW for database clause

one. It sorts the values of a particular attribute and sets up an index of these values with pointers pointing to the corresponding records. Other methods divide the database into bins, each of which contains similar records[11]. Some hashing functions are applied to the attribute values to obtain the addresses of the bins. Each attribute in the record is assigned a probability, which indicates the likelihood of that attribute being specified in the query. Although these methods are suitable for relational databases, they are not suitable for Prolog as the order of the clauses is lost in the sorting process.

Codeword indexing, such as SCW and FEW, is a kind of multi-dimensional indexing strategy that constructs indices on all attributes. It does not depend on a particular attribute. The more attributes specified in the query, the more the search space is narrowed, and the faster is the retrieval. It can preserve the order in which the database clauses are stored. Thus it avoids the weaknesses of the previous strategies and provides a more suitable access method for Prolog databases.

Two-level codeword indexing

In this scheme, the database is partitioned into blocks. A block descriptor is then associated with each block, which is formed in a similar manner to the index codeword. The block descriptor is constructed using the attribute values from every record in the block.

For the study described here, a SCW index was used as the lower-level index for the clauses in a file. Then for each block of clauses, a high-level index was set up. For this higher-level index, both SCW and FEW strategies were tried, but the latter was more effective.

Thus when a clause is retrieved, its index codeword (SCW index) and block descriptor (FEW index) are generated. The block descriptor file is searched using the FEW index to find those blocks that contain potential matches. Then the individual blocks so identified are searched using the SCW index to find potential clause matches.

An illustration of two-level indexing can be found in Figure 3.

BENCHMARKS USED

Two different benchmarks have been used to assess the performance of the one-level and two-level codeword

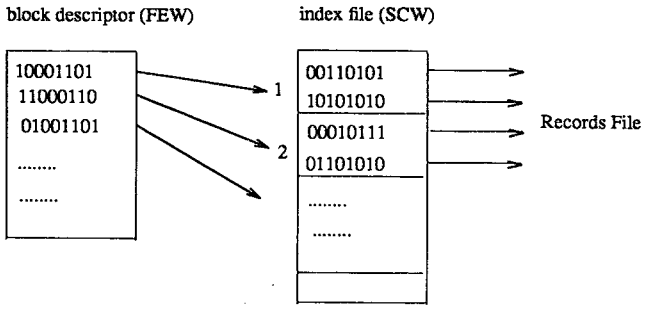


Figure 3. Two-level indexing

indexing strategies. These are taken from reported benchmarks[12,13]. Briefly, these consist of the following:

- The first is a database of vertices and edges comprising clauses of the form

  vertex(VERTEX_NAME, X, Y).

  which represent the vertices of an $n \times n$ grid of points, and

  edge(EDGE_NAME, VERTEX_NAME1, VERTEX_NAME2).

  which represent edges connecting the vertices. The database is set up using a Hilbert's space-filling curve algorithm to generate a set of edges corresponding to a Hilbert curve.
- The second is a database produced using a pseudo-random number generator. The generator is called as follows:

  dg(Name, Format List, Uniqueness List, Clause).

A complete description of the form of the arguments required by the data generator has been given[12,13]. The form chosen for benchmarking contains structures of the two most common forms: lists and terms.

For each benchmark, a number of databases of different size (ranging from 16 to 262144 clauses) were generated. These test databases were then exercised using the different retrieval strategies to ascertain the answers to questions such as:

- What is the retrieval time for the first solution/all solutions?
- What is the time to complete a full scan of the database (time to fail)?
- What is the retrieval time for a query with all ground terms?

RESULTS

One-level vs two-level strategies

Table 1 shows the times taken to find all solutions in searches of the databases based on:

(a) Benchmark 1 with retrieval on all arguments ground using the query

retrieve(DBASE_NAME, ALL, vertex('00',0,0),_)

(b) Benchmark 1 with retrieval on first argument using the query

retrieve(DBASE_NAME, ALL, vertex('00',_,_),_)

(c) Benchmark 1 with retrieval on third argument using the query

retrieve(DBASE_NAME, ALL, vertex(_,_,0),_)

(d) Benchmark 2 with retrieval on all arguments ground.

Considering the ratio of the time taken using the two-level strategy compared with the one-level approach, these show a variability in performance ranging from 0.07 to 1.70.

The time taken to fail was tested with the retrieval of a non-existent clause

vertex('11',1,1).

The results are shown in Table 2. Here the ratio of the time taken using the two-level strategy compared with the one-level approach ranged from 0.13 to 0.67.

Varying block size

The number of clauses stored in a block obviously influences the retrieval times. The larger the block size, the more clauses are stored in the block, the more bits will be set in the block descriptor, and hence the more

Table 1. Retrieval times (msec.) for four benchmarks for variety of database sizes using one-level and two-level strategies

No of clauses in databases	Benchmark 1						Benchmark 2	
	Retrieval on all arguments ground		Retrieval on first argument		Retrieval on third argument		Retrieval on all arguments ground	
	1-level	2-level	1-level	2-level	1-level	2-level	1-level	2-level
16	140	200	180	180	200	220	260	260
64	160	220	220	240	240	260	300	320
256	260	300	260	380	360	460	360	560
1024	500	640	520	680	740	560	540	920
4096	1400	800	1380	1760	1720	760	1660	1980
16384	4600	960	4420	5280	5420	1940	4920	5780
65536	20460	2860	21600	25260	20560	4580	20020	27860
262144	95740	7080	106800	122020	97800	19820		

Table 3. Retrieval times (msec.) for different block sizes for variety of database sizes using vertex databases

		Two-level							One-level
		Blocksize							
Benchmark	No of clauses in databases	10	30	40	50	60	70	90	
(a) Retrieval on all arguments ground	16	260	260	220	200	200	180	180	140
	64	280	280	260	220	260	260	220	160
	256	400	320	380	300	300	360	360	260
	1024	660	640	680	640	560	540	580	500
	4096	1120	940	840	800	760	680	700	1400
	16384	2240	1160	1040	960	940	920	1220	4600
	65536	10320	2900	2860	2860	2060	1940	2560	20460
(b) Retrieval on first argument	16	280	240	220	180	200	220	220	180
	64	260	300	260	240	300	300	260	220
	256	540	340	400	380	360	380	380	260
	1024	940	660	680	680	580	540	520	520
	4096	1660	1600	2100	1760	1680	1500	1420	1380
	16384	5660	5780	5240	5280	5180	4800	4940	4420
	65536	26220	25120	25340	25260	24300	23020	22520	21600
(c) Retrieval on third argument	16	300	260	280	220	280	280	280	200
	64	260	260	260	260	300	280	280	240
	256	320	400	400	460	360	340	340	360
	1024	420	420	440	560	520	500	780	840
	4096	1280	700	720	760	980	1620	1760	1720
	16384	1740	1140	1660	1940	2480	2600	2760	5420
	65536	6220	2280	4040	4580	4980	6400	7280	19980

Table 2. Retrieval times taken to fail (msec.) for variety of database sizes using vertex databases

	1-level	2-level
16	120	80
64	140	100
256	260	120
1024	500	140
4096	1060	320
16384	4460	780
65536	20420	2700

likely that this will result in a false drop. On the other hand, if a large block can be eliminated from the search, this should reduce search time significantly. At one extreme, the block size can be made so large that one block contains all clauses. In this case, the top-level search achieves nothing and the search reduces to a one-level search of a single block. At the other extreme, the block size can be reduced to a single clause. In this case, the top-level search reduces to a one-level search of clause codewords. Between these two extremes lie improved retrieval times.

Table 3 shows the times taken for the three types of search applied to benchmark 1. Once again times shown are those to find all solutions for block sizes in the range of 10 to 90 clauses.

The ratio of times taken using the two-level strategy with varying block sizes compared with the one-level approach ranges from 0.09 to 1.86 for (a), 1.00 to 2.08 for (b), and 0.11 to 1.50 for (c). From the Tables, optimum block sizes are different for different types of query. Thus if it is known that queries will be restricted to a particular type, the database can be optimized accordingly. However, to anticipate all types of queries, then a block size of 50 to 60 will provide adequate performance on all three tests.

Table 4. Retrieval times (msec.) for variety of database sizes (vertex databases) using caches

	Without caching		With caching	
	1-level	2-level	1-level	2-level
16	140	200	20	20
64	160	220	20	20
256	260	300	60	40
1024	500	640	220	40
4096	1400	800	840	80
16384	4600	960	3280	240
65536	20460	2860	19540	900

Effect of caching

A number of caching strategies are adopted within the CRS to minimize the overheads incurred due to system paging activity over large virtual memory spaces[6,7]. The effect of introducing a cache is quite obvious in Table 4, which shows the times taken for the retrieval of all solutions to a query with all arguments ground.

The results show that the advantage of the two-level strategy is even more obvious with caching.

IMPROVED APPROACH COMBINING ONE- AND TWO-LEVEL CODEWORD INDEXING STRATEGIES

The results of the previous section show that the two-level codeword indexing strategy is better than the one-level approach in most cases. However, when retrieving

on the first argument, as can be seen from Tables 1 and 3, the two-level strategy is slightly slower. On studying the domain of the first argument, it was observed that the range of values of the first argument within a block is quite large and thus the number of entries to the atom table which the first argument generated is quite large. This increases the likelihood that the block descriptor (FEW index) is almost filled with 1s, which will result in a large false drop rate. The same problem occurs when retrieving from the random number databases used in benchmark 2. This is a weakness of the two-level strategy. When using a FEW index as the higher-level codeword index, this problem will also occur when variables are present in the database clause head: if they are evenly distributed across the clause heads then a large number of bits will be set, resulting in a large number of false drops. However, it is likely that with facts there will be very few variables in the clause head and with rules the variables might occupy the same field.

Several cases in which the one-level strategy shows advantages over the two-level approach include

- If the values of a specific argument within a block range over a large domain, the block descriptor for this block is likely to be filled with a large number of 1s, resulting in a large false drop rate when searching through the higher-level index file.
- When retrieving from small databases, there might be little advantage to be gained by a two-level approach and in the extreme the additional overhead will lead to larger search times.
- If the query contains no ground term, say, an all-variable query, such as vertex(_,_,_), all the clauses will be retrieved. In this case, the one-level indexing strategy will be more efficient.

In the improved approach, the query is preprocessed to switch between one-level and two-level indexing strategies under different circumstances. For example, if the query is an all-variable query, the one-level indexing strategy will be selected.

A counter has been set up for each argument of the clauses in the database. The counter stores the percentage of bits that are set to 1 in the field of the block descriptor representing that argument in the clauses. If the percentage exceeds a certain limit, this means that most bits in that field are filled with 1s. A large false drop rate will be produced if retrieving on that argument.

The query is preprocessed to find out the argument(s) on which retrieval is to be based. If only one argument is found to be significant in the retrieval, as in the query of vertex('00',_,_), the counter of that argument should be checked to see if it exceeds a certain limit. If it does, the one-level strategy will be selected; otherwise, the two-level strategy will be selected. If several arguments are involved, as in the query of vertex('00',_,0), the counters of these arguments should be checked sequentially. If one of them is below a certain limit, the two-level strategy will be selected; otherwise, the one-level strategy will be selected.

Table 5. Retrieval times (msec.) for variety of database sizes (vertex databases) adopting improved approach

	Retrieval on all arguments ground	Retrieval on first argument	Retrieval on third argument
16	200	200	220
64	200	260	300
256	320	340	420
1024	660	680	580
4096	840	1640	1040
16384	1080	5000	2080
65536	2880	23660	4740
262144	8620	106880	20400

RESULTS OF IMPROVED APPROACH

The same set of benchmarks is applied to the improved approach with a block size of 50. A strategy is selected according to the size of database, type of query, and contents of the corresponding counters.

Table 5 shows the times taken for the three types of search applied to benchmark 1. The times shown are those to find all solutions.

The ratio of times taken using the improved approach compared with the one-level strategy ranges from 0.09 to 1.42. It is quite obvious that the improved approach has combined the advantages of both the one-level and two-level indexing strategies, although there is a small overhead when preprocessing the queries. The results show that in retrievals in which all arguments are ground or the third argument is ground, when the database contains fewer than 1024 clauses, the improved approach adopts the one-level indexing strategy, which is better under this circumstance; when the size is 1024 or more, it adopts the two-level indexing strategy, which is better under that circumstance. In retrieval in which the first argument is ground, the one-level strategy is adopted to avoid the large false drop rate when searching through the block descriptor file resulting from the two-level indexing strategy.

USING CLARE HARDWARE

Clare is a special-purpose hardware engine that improves the slow clause retrieval rate of Prolog in dealing with large data sets. The hardware is designed to be linked to a M68020 host, via the VMEbus interface, in a Sun 3/160 workstation[4,14–16].

The four types of benchmark tests used in the previous section have been applied to the Clare hardware search, and Table 6 shows the resulting times.

From these results, the two-level strategy performs even better than the hardware search in some cases, while Clare shows advantages in retrievals on the benchmarks using random number databases and other cases like retrieval on the first argument. Thus Clare can be used to overcome the weaknesses of the two-level indexing strategy. An improved approach combining the two-level strategy and the one-level strategy using Clare has also been developed. Table 7 shows the times taken for the three types of search adopting this improved approach.

Table 6. Retrieval times (msec.) for four benchmarks for variety of database sizes using Clare hardware

	Benchmark 1			Benchmark 2
	Retrieval on all arguments ground	Retrieval on first argument	Retrieval on third argument	Retrieval on all arguments ground
16	180	180	200	200
64	220	200	220	260
256	220	220	240	280
1024	420	440	620	360
4096	1580	1240	1900	1040
16384	1640	1900	2500	2820
65536	7940	15700	9360	11740
262144	39060	51680	35220	

Table 7. Retrieval times (msec.) for variety of database sizes (vertex databases) adopting improved approach combining two-level approach and one-level approach using Clare

	Retrieval on all arguments ground	Retrieval on first argument	Retrieval on third argument
16	200	200	240
64	240	260	320
256	320	300	440
1024	680	520	600
4096	880	1200	960
16384	1080	2000	2080
65536	2920	15740	4820
262144	8640	52320	20300

CONCLUSION

Adopting the two-level indexing strategy, it is possible to retrieve Prolog clauses efficiently from large databases and improve the performance of disc-resident clause handling. However, the two-level strategy shows weakness in some cases, such as when the database is small, the query consists of all variables, or the domain of an argument is large. By judicious choice between one-level and two-level indexing strategies, a more efficient retrieval system can be developed.

Some work has been done to compare the codeword indexing technique with the BANG file approach[17]. It has been noticed that the BANG file structure performs better than the one-level and two-level SCW indexing in some cases when retrieving on a large domain argument or the query contains a small number of variables. However, the BANG file structure suffers two major drawbacks. First, it currently does not support the handling of variables and complex structures in the clause heads. Second, it does not preserve the order in which the database clauses are stored.

Hardware search enhances the execution speed of Prolog when dealing with large clause sets by reducing the amount of full unification necessary on the large clause sets. The weakness of the two-level indexing strategy can be remedied if it is combined with hardware search to improve the performance when retrieving on large domain arguments. A possible approach is to use the software SCW or FEW indexing strategy to retrieve the matching block descriptors, and then use the hardware search to retrieve the matching clauses from these blocks.

ACKNOWLEDGEMENTS

The authors thank the Prolog Database Machine Group, Department of Computer Science, Heriot-Watt University for their support and advice.

REFERENCES

1 **Wise, M J and Powers, D M W** 'Indexing PROLOG clauses via superimposed code words and field encoded words' in *Proc. IEEE Conf. Logic Programming* Atlantic City, NJ, USA (January 1984) pp 203–210

2 **Williams, M H** 'The design of a Prolog database machine' in *Proc. 1st Workshop for SIGKME* Reading, UK (January 1987) pp L1–L5

3 **Roberts, C S** 'Partial-match retrieval via the method of superimposed codes' in *Proc. IEEE* Vol 67 No 12 (December 1979)

4 **Wong, K F and Williams, M H** 'Design considerations for a Prolog database engine' in *Proc. 3rd Int. Conf. Data and Knowledge Bases* Jerusalem, Israel (28–30 June 1988) pp 111–120

5 **Ramamoharao, K and Shepherd, J** 'A superimposed codeword indexing scheme for very large Prolog databases' in *Proc. 3rd Int. Logic Programming Conf.* London, UK (July 1986)

6 **Massey, P A, Ferbrache, D J and Williams, M H** 'A transaction system for a multi-user Prolog database system' *Internal report* Heriot-Watt University, Edinburgh, UK (May 1989)

7 **Massey, P A, Wong, K F, Ferbrache, D J and Williams, M H** 'Adapting Prolog for large knowledge based applications' *Internal report* Heriot-Watt University, Edinburgh, UK (1989)

8 **Ferbrache, D J, Massey, P A, Taylor, H, Wong, K F and Williams, M H** 'CRS clause format' *Internal report* Heriot-Watt University, Edinburgh, UK (1988)

9 **Ferbrache, D J** 'Clause retrieval server technical specification: user call protocol specification' *Internal report* Heriot-Watt University, Edinburgh, UK (December 1987)

10 **Knuth, D E** *The art of computer programming – sorting and searching* Addison-Wesley (1969) pp 559–562

11 **Ramamoharao, K and Lloyd, J W** 'Partial-match retrieval using hashing and descriptors' *ACM Trans. Database Syst.* Vol 8 (1983) pp 552–576

12 **Williams, M H, Massey, P A and Crammond, J A** 'Benchmarking Prolog for database applications' in *Prolog and databases: implementations and directions* Ellis-Horwood (1988) pp 161–187

13 **Williams, M H, Massey, P A and Crammond, J A** 'Benchmarks for Prolog from a data viewpoint' in *Proc. 2nd Workshop for SIGKME* Brunel University, UK (28–29 May 1987)

14 **Wong, K F and Williams, M H** 'A type driven hardware engine for Prolog clause retrieval over a large knowledge base' in *Proc. 16th Annual Int. Sympos. Computer Architecture* Jerusalem, Israel (28 May–1 June 1989) pp 211–222

15 **Wong, K F and Williams, M H** 'Partial matching hardware for selective clause retrieval in large Prolog databases' *Internal report* Heriot-Watt University, Edinburgh, UK (1988)

16 **Wong, K F and Williams, M H** 'Limiting a resolution set in a large Prolog database using special hardware' in *Proc. 1988 ACM SIGSMALL/PC Sympos. Small and Personal Computers* Cannes, France (4–6 May 1988) pp 133–145

17 **Freeston, M** 'The BANG file: a new kind of grif file' in *Proc. 1987 ACM-SIGMOD Int. Conf. Management of Data* San Francisco, CA, USA (May 1987)

43. J.CAMACHO GONZALEZ,M.H.WILLIAMS and I.E.AITCHISON, Evaluation of the effectiveness of Prolog for a CAD application, IEEE Computer Graphics and Applications, vol. 4, no. 3, 67-75 (1984).

Fifth-generation computers will probably be based on logic programming languages like Prolog rather than on Fortran-type languages. Are such languages suitable to 3-D CAD applications?

Evaluation of the Effectiveness of Prolog for a CAD Application

J. Camacho Gonzalez, M. H. Williams, and I. E. Aitchison
Heriot-Watt University

Prolog is a programming language based on predicate logic.[1-4] Since its development in 1970, it has been used in a wide range of applications, including relational databases, natural language processing, expert systems, and architectural design.[5-8]

Recently, the importance of Prolog has risen dramatically as a result of the Fifth-Generation Computer Project, a Japanese effort to produce the next generation of computer systems.[9] The set of core languages for these machines, known as the Fifth-Generation Kernel Languages, is a set of logic programming languages modeled on Prolog.

In view of this new-found importance, it is natural to consider the suitability of Prolog for handling various applications.[10,11] One such application is computer-aided design. CAD tools, such as two-dimensional drafting packages, three-dimensional geometric modeling packages, etc., are common in the engineering industry today.[12,13] These packages are normally used interactively by a person sitting at a graphics workstation, which typically consists of a graphics screen, a VDU, and a graphics tablet. Traditionally, such packages have been implemented in Fortran or similar languages.

Since Prolog is already implemented on sequential computers, we felt it would be useful to study whether Prolog could be used as a language for programming CAD applications and, if so, how it could be used. To obtain a fair perspective, we compared it with an appropriate conventional (imperative) programming language.

We chose Pascal, because it is well structured, embodying all the control structures required for structured programming and providing a wide range of data structures. Like Prolog, Pascal has been chosen as a model for programming languages; the most notable example is Ada. The ability of Pascal to allow definition of complex data structures is a particular advantage in CAD work, since in many applications the object being designed is complex and must be represented internally by complex data structures.

Prolog: an overview

Prolog differs significantly from conventional programming languages. It has none of the constructs normally associated with such languages: goto, if-then-else, while loop—not even an assignment statement. A Prolog program comprises a set of Horn clauses,[14-16] each of which is in one of two forms. A Horn clause of the form

p:– q1, q2, . . . , qn.

is referred to as a *rule* and is interpreted as "*p* is true if all of *q*1, *q*2, . . . , *qn* are true." The form

p.

is a *fact* and is unconditionally true.

Once a program has been set up, it can be queried by typing a question of the form

?– p.

This question can involve more than one goal. When asked a question, the Prolog program uses the known clauses to satisfy the goals. A goal is satisfied when a fact that matches it is found or when a rule is found whose left side matches the goal and whose right side, treated as a goal, is satisfied.

0272-1716/84/0300-0067501.00 © 1984 IEEE

When an attempt to match a goal fails, the evaluation backtracks to the most recent evaluation step and continues from there. If no solution can be found that satisfies the current query, the query fails.

The clauses are always tested for matching in the order in which they appear in the database. Prolog programmers often exploit this order by placing rules that must be evaluated first ahead of any default rule that applies when the other rules have failed. The symbol !, referred to as the "cut operator," is used as a backtracking control device to prevent use of a later rule in cases where an earlier rule has been successfully applied. (This is a highly controversial feature of Prolog.)

As illustration, consider the facts, the rule, and the comments in Table 1. The facts represent information relevant to the triangle in Figure 1; the rule defines a property of its components.

Given this information, the user can ask such questions as

- ? – vertex(v1, 2, 1, 0). [Does a vertex $v1$ with coordinates 2, 1, 0 exist?]
- ? – vertex(V, X, 1, 0). [Looks for any vertex with y coordinate 1 and z coordinate 0 and returns its x coordinate.]
- ? – connected(e1, e2). [Are edges $e1$ and $e2$ connected?]

An application: Pascal vs. Prolog

Among current systems designed to aid the production of engineering drawings, many are simply tools that enable the draftsman to draw or modify orthographic projections of objects. Since no attempt is made to construct a complete three-dimensional model of an object, it is not possible for such systems to generate arbitrary views or sections of an object or compute such properties as the object's weight or the position of its center of gravity.

Several recent packages, however, allow the user to design three-dimensional objects. With these systems, one can create a stored three-dimensional model that can be used to produce any required view or parameter. These geometric modeling systems are usually large and complex.

We have been studying the implementation of a geometric modeler on a small computer with particular emphasis on the development of facilities to update three-dimensional models by means of operations on two-dimensional views. From this study, we developed a package for interactive design of two-dimensional pictures consisting of lines, arcs, and circles. Package facilities enable the user to create pictures, edit them, store them in and retrieve them from a file, etc.

We believe this two-dimensional drawing aid is a realistic and useful CAD application that serves as a good example to test the suitability of potential CAD programming languages.

The system is currently implemented on a PDP-11/34 computer system running under Unix. The workstation is a minimal graphics configuration, consisting of a Tektronix 4051 storage tube terminal with a joystick for input of graphical information.

The present system is oriented toward two-dimensional technical drafting in mechanical engineering. The draftsman works with two-dimensional representations of three-dimensional objects by means of three orthogonal projections. He uses the system to create a set of two-dimensional views of the object he is trying to represent; from these views, the system constructs a complete 3-D representation.

Facilities for manipulating 2-D views. The facilities in the general 2-D drafting system are aimed at allowing the user to input graphical information into the computer and file it. The system also permits the user to fetch previously filed views, manipulate them, and incorporate them into another view or return them to a disk file.

Specifying lines. When the system is entered, the work space is empty. To build a view, the user must add lines to this work space by specifying the end points of the lines. The system keeps constant note of the most recently defined point; to specify a line, the user gives the other end point. Points can be specified in four ways:

(1) by entering the absolute coordinates at the keyboard,

Table 1.
Information relevant to the triangle in Figure 1.

FACT/RULE	COMMENTS
vertex (v1,1,1,0).	Vertex v1 has coordinates 1,1,0.
vertex (v2,2,2,0).	Vertex v2 has coordinates 2,2,0.
vertex (v3,3,1,0).	Vertex v3 has coordinates 3,1,0.
edge (e1,v1,v2).	Edge e1 links vertices v1 and v2.
edge (e2,v2,v3).	Edge e2 links vertices v2 and v3.
edge (e3,v3,v1).	Edge e3 links vertices v3 and v1.
connected (E1, E2):-edge (E1,__, V), edge (E2,V,__).	Edge E1 is connected to edge E2 if the final vertex of E1 is the initial vertex of E2.

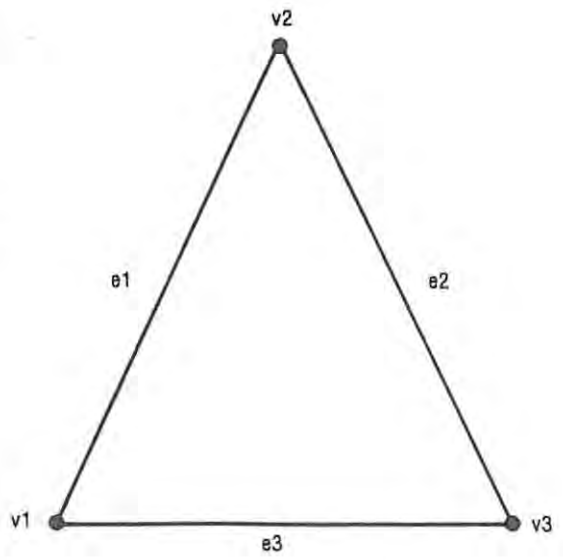


Figure 1. Triangle represented as a set of Prolog clauses.

(2) by indicating the position of the point on the screen with the joystick,

(3) by using an existing vertex in the figure (the joystick can be used to position the cursor at the vertex), or

(4) by finding a point on an existing line (again using the joystick to position the cursor on the line).

Having set up a point, the user can *move* or *draw*.

- *Move:* Move to that point, which becomes the current point.
- *Draw:* Draw a line from the current point. Various options exist in the way this line can be drawn, including, among others: draw to a point, draw a line of given length and direction, draw a horizontal or vertical line (to a point or to a given length), draw a line parallel to some specified line (to a point or to a given length), and draw a sequence of lines.

Modifying and filing figures. In addition to facilities for creating views, there are commands for modifying and filing figures. The most obvious is the equivalent of the backspace facility in text-processing systems; it allows a user to cancel the last operation performed.

Other operations include

- *Point edit:* A point can be moved independently, thus stretching or contracting any lines connected to it.
- *Line edit:* A line can be deleted or edited by replacing all or part of it with several lines (Figure 2).
- *Ext_op:* Extend operations allow the user to extend any line intersected by a horizontal or vertical line set up by the user (Figure 3).

The designer can retrieve previously defined figures from disk and use them to draw new figures. In particular, it is possible to construct a compound figure from several components stored in a disk file.

Interface to a 3-D system. The 2-D drafting system can be used to produce the normal orthogonal front elevation, end elevation, and plan views of a three-dimensional object. These can be used as figures in their own right. However, it is more advantageous to use the information inherent in these views to produce a true 3-D model, which can in turn be used to generate any required view or evaluate other parameters.

It should be possible, at any design stage, to edit the model by going back to the defining 2-D views and editing them. If the 2-D system is to be used as input to the modeler, it must carry out a certain amount of preprocessing to extract further connectivity information needed in generating a 3-D view. Thus, if the system is to be used as a front end to the modeler, it is operated in a special mode, one in which this additional information is computed as the user builds his views.

This additional information consists of

- paths,
- connected vertices,
- neighbor paths, and
- equivalent vertices.

Paths. A path is a cyclic sequence of directed edges that marks the boundary of an area in a view. For example, some paths in Figure 4, View A, are

- [6, 7, 9, 8, 6]
- [4, 5, 2, 3, 7, 6, 4], and
- [1, 2, 5, 4, 1].

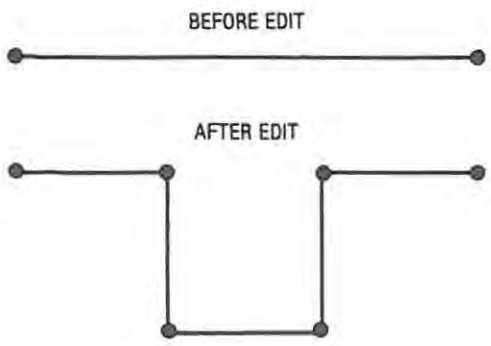


Figure 2. Illustration of line-edit facility.

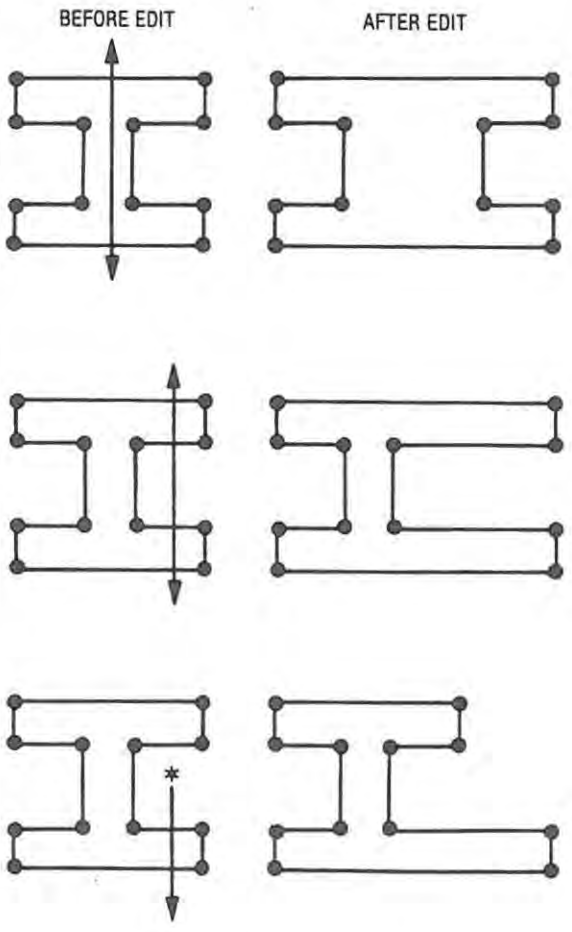


Figure 3. Some examples of the extend facility.

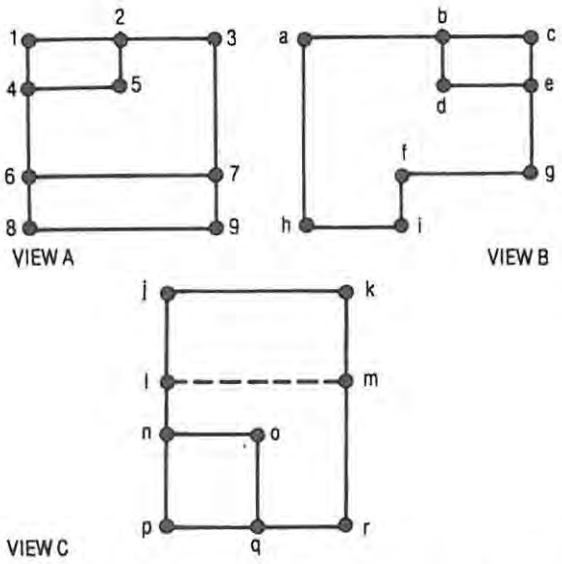


Figure 4. Three two-dimensional views of an object.

Table 2.
An example of the relation Vertex.

PASCAL	PROLOG
VERTEX = array [size] of record x,y:integer; view:tname end;	vertex (Name,X,Y,View).

NAME	X COORDINATE	Y COORDINATE	VIEW
vertex1	200	300	viewa
vertex2	400	400	viewa
vertex3	300	200	viewa

Table 3.
An example of the relation Edge.

PASCAL	PROLOG
EDGE = record name, v1,v2, view:tname; flag: ^ list; m,b:real end;	edge(Name,Flag,V1,V2,M,B,View).

NAME	FLAG	V1	V2	M	B	VIEW
edge1	edge3	vertex1	vertex2	m1	b1	viewa
edge2	edge1	vertex2	vertex3	m2	b2	viewa
edge3	edge2	vertex3	vertex1	m3	b3	viewa

Table 4.
An example of the relation View.

PASCAL	PROLOG
VIEW = record name : tname; le, lv: ^ list end;	view(Name, LE, LV).

NAME	LE	LV
view1	(edge1, . . . , edge4)	(v1, . . . , v4)
view2	(edge5, . . . , edge10)	(v5, . . . , v10)

Connected vertices. Two vertices are connected if a common edge or a sequence of colinear edges lies between them. In Figure 4, View A,

- vertex 6 is connected to [1, 4, 8, 7],
- vertex 4 is connected to [1, 6, 8, 5], and
- vertex 7 is connected to [3. 9, 6].

Neighbor paths. If a view of an object contains more than one path, each path will have a set of neighboring paths. Two paths are said to be neighbors when they have one or more edges in common. In Figure 4, View A,

- path [1, 2, 5, 4, 1] is a neighbor of path [4, 5, 2, 3, 7, 6, 4], and
- path [4, 5, 2, 3, 7, 6, 4] is a neighbor of paths [1, 2, 5, 4, 1] and [6, 7, 9, 8, 6].

Equivalent vertices. Each vertex in an orthogonal view of an object must correspond to one or more vertices in another orthogonal view, which could project onto the original vertex. This set of corresponding vertices is called the equivalent vertices of the original vertex. For example, Figure 4 shows a set of three views of a simple body.

The equivalent vertices of vertex *p* in Figure 4, View C, are

- [*c, e, g*] in View B and
- [1, 4, 6, 8] in View A.

Equivalent vertices of the vertex *q* in View C are

- [*c, e, g*] in View B and
- [2, 5] in View A.

Internal representation of views. Any two-dimensional view can be described in terms of information about its vertices and edges. In our implementations, three relations, Vertex, Edge, and View, describe 2-D drawings. Each relation is defined as a record in Pascal and as a fact in Prolog.

Vertex. In a 2-D drawing, *x* and *y* coordinates determine a vertex. The relation Vertex contains four attributes: Name, *x* coordinate, *y* coordinate, and the view in which it occurs. Name is the key of this relation. Table 2 shows an example of the relation Vertex.

Edge. Any edge can be described by its two end points. In the Edge relation, the attribute Name is the key; attributes *V*1 and *V*2 are the origin vertex and the destination vertex, respectively. Additional attributes are *M* and *B*, coefficients of the equation of the line, and View and Flag. Table 3 shows an example of the relation Edge; ^ list is a pointer to the Edge structure.

View. A sequence of edges and vertices describes a view. In the View relation, the attribute Name is the key, and the attributes *LE* and *LV* contain the lists of the edges and vertices forming the view. Table 4 shows an example of this relation.

Representation in Pascal. At the beginning of a session, the work space is empty and ready to receive data from either the user or an existing figure stored in a disk file. As

the designer enters data (vertices and edges), the system loads them, in order, into the work space. The figure will be represented by an ordered and closed sequence of edges. This feature helps the system to modify the figure easily and quickly. Figure 5a shows a rectangle; Figure 5b represents the structure used in the Pascal implementation.

If the user wants to modify the figure via line- or point-edit commands, a second structure is created. At the end of the edit, the system adds the new part into the work space of the first structure or replaces part of the old structure with the new material.

Two-dimensional figures can be used as figures in their own right or as views of some 3-D object, which can be reconstructed from the 2-D views. Since the latter requires much more information to be stored, the user must indicate to the system whether the 2-D figure he wishes to construct is a stand-alone figure or a view of some 3-D object.

In the internal representation of a view, any edge common to two faces is represented as a pair of edges. For example, suppose the user wants to draw a straight line across the object in Figure 6a. Using the line-edit facility, he can draw a new edge between vertices 9 and 10, as shown in Figure 6b. At this stage, the user sees the view as in Figure 6b; the system represents it internally as in Figure 6c.

Any holes in the object are considered as sets of edges and vertices. For purposes of the 3-D representation, each view is regarded as a set of paths. Figure 7a shows a view; Figure 7b shows the paths that constitute it.

Representation in Prolog. The Prolog database replaces the work space concept; all information for each view is stored in the database. Two predicates, *asserta* and *assertz,* are used to add new clauses to the database. Asserta inserts the clauses at the beginning of the database; assertz adds them to the end. Otherwise, the two are the same. Another predicate, *retract,* removes clauses from the database.

Figure 8a illustrates a rectangle; Figure 8b shows how it might be represented as a structure in the database.

The principal elements of two-dimensional views or figures are vertices and edges. A vertex has four attributes: the internal name assigned to it, the x and y coordinates, and the view in which it occurs, i.e.,

vertex (Name, X, Y, View).

An edge has associated information that determines its position in the view and its boundaries, namely,

edge (Name, Flag, Vertex1, Vertex2, M, B, View).

As before, Name is the internal name given to the edge. For each edge, the attribute Flag contains the name of the next edge in the path. This attribute is used to determine all the paths of any view.

The two-dimensional system is responsible both for providing facilities to draw the orthogonal views of a body and for deducing all the secondary information about paths and faces that the modeler requires in order to construct a 3-D representation of the body.

Typical algorithms and their implementation. The effectiveness of Prolog and Pascal for implementing the CAD system we have described can be compared according to the traditional measures of time and space. The values quoted here were obtained by comparing a particular Prolog system (an interpreter) and a particular Pascal system (a compiler), both running on the same machine, a PDP-11/34. Although this comparison is clearly inadequate as an absolute measure, since the per-

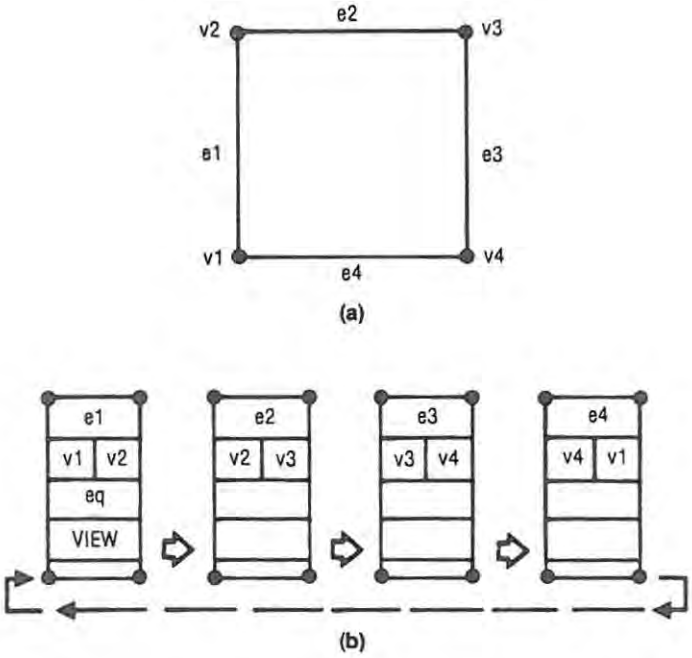


Figure 5. A simple rectangle (a); internal representation in Pascal (b).

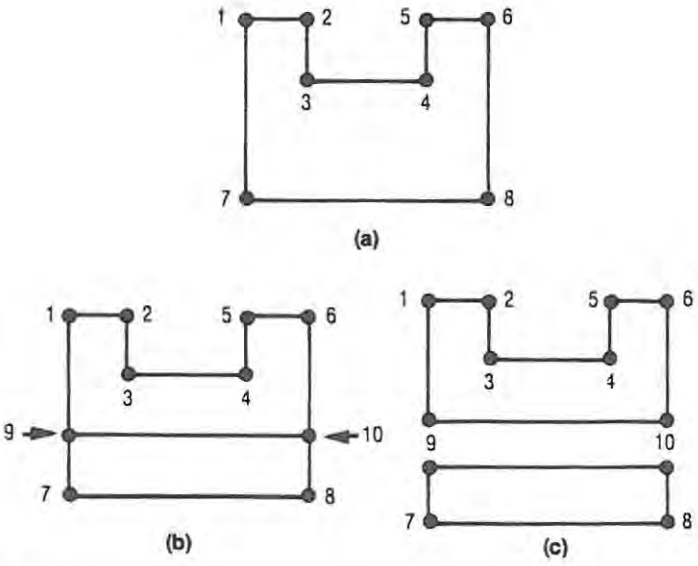


Figure 6. Initial view (a); modified view as seen by the user (b); modified view as represented within the system (c).

formance of individual compilers can vary considerably, it does indicate relative performance of the two systems in this type of application.

Finding a vertex. Suppose one wishes to find a particular vertex whose approximate coordinates are indicated via the joystick. The two systems carry out this task as shown below.

PASCAL

```
procedure findpoint (var ver:integer; var found: boolean);

begin
  ver: = 1; found: = false;
  while (ver < = indvertex) and (not found) do
    with vertices[ver] do
    begin
      if (((x + 10) > xpoint) and ((x - 10) < xpoint) and
        ((y + 10) > ypoint) and ((y - 10) < ypoint)) then
            found: = true {Flag to know if found}
        else ver: = ver + 1
    end
end;
```

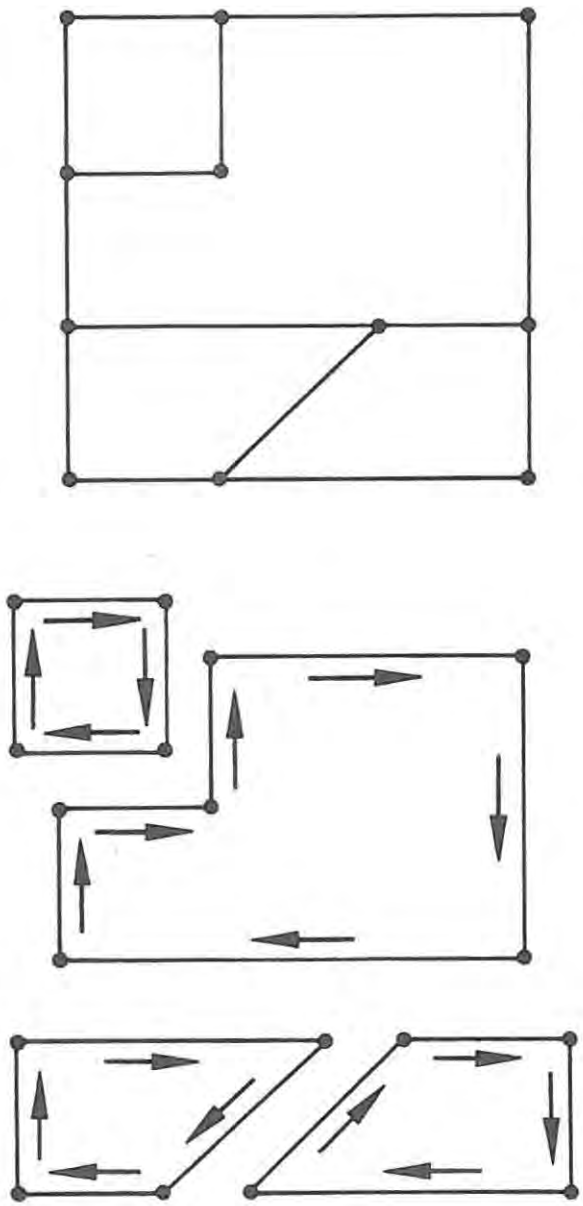


Figure 7. Different paths forming a view.

PROLOG

```
find_a_point(V, X, Y): -
      vertices(V, A, B, C), A + 10 > X, A - 10 < X,
      B + 10 > Y, B - 10 < Y, !.
```

The Pascal procedure *findpoint* looks for any vertex sufficiently close to the current position of the cursor, represented by *xpoint* and *ypoint* (global variables). The set of vertices is contained in the array *vertices*, and x and y are the coordinates of each vertex.

The algorithm operates by scanning from the beginning of the array, comparing, one at a time, the coordinates *xpoint ypoint* against the x and y coordinates of each vertex. A relatively small interval around each vertex is included, to take account of the approximate positioning obtained with the joystick. If the procedure succeeds, the Boolean variable *found* will be set to true and the integer variable *ver* will contain the number of the vertex that was the object of the search.

For each iteration of the loop, six comparisons are performed, two in the head of the loop and four in the body. With an array of 50 vertices, this algorithm takes an average of 20 CPU units to find the right vertex. It uses 234 bytes of compiled code.

The corresponding clause in Prolog, *find_a_point*, achieves the same effect through different logic. It states that

V is a point close to coordinates X and Y if
  V is the name of the vertex with coordinates
    A and B and view C,
  and $A + 10 > X$,
  and $A - 10 < X$,
  and $B + 10 > Y$,
  and $B - 10 < Y$.

The first goal to be satisfied, *vertices* (V, A, B, C), matches the first vertex in the database. The variables V, A, B, and C are instantiated with the name and coordinates of the vertex. Each succeeding goal ($A + 10 > X$, $A - 10 < X$, . . .) is tested in turn until the end of the clause is reached or until a goal fails. In the former case, the rule has succeeded; in the latter case, an alternative match for the first goal—*vertices* (V, A, B, C)—is tried and the whole process repeats.

In Prolog the search is performed automatically and requires no special algorithm. Also, the code is simpler and clearer than in Pascal. To execute this predicate on a data structure of 50 vertices, the algorithm takes an average of 13 CPU units and uses 164 bytes of interpreted code.

Erasing the last element. Another algorithm erases the last element drawn by the user. This "back-one-step" facility is similar in the two languages.

PASCAL

```
procedure lastelement;

begin
  if last = nil then writeln('** NO ELEMENTS **')
  else begin
    if last = view[group] then begin
      indvertex: = last ^.v1;
      view[group]: = nil;
      last: = nil;
      end
    else begin
      last ^.next: = view[group] ^.next;
      indvertex: = last ^.v2;
      view[group]: = last;
      while last ^.next < >view[group] do last: = last ^.next
    end;
    xlast: = vertices[indvertex].x;
    ylast: = vertices[indvertex].y;
    findvertex: = indvertex;
    writeln('** LAST ELEMENT HAS BEEN REMOVED **')
  end
end;
```

PROLOG

```
last_element: - retract(last_point(_,_,_)),
      retract(vertices(_,_,_,_)),
      vertices(Ver, X, Y, Z), asserta(last_point(Ver,X,Y)),
      write('**LAST ELEMENT HAS BEEN
        REMOVED**'), nl.
last_element: - write('** NO ELEMENTS **'), nl.
```

In Pascal each path of a view is stored as a list of edges. In finding the last element, the system handles three cases: a list with no elements, a list with one element, and a list containing two or more elements. The first is trivial and involves no action. In the second, the element is removed and the pointers are reset to nil. In the third, the element is erased and the pointers are reset to their previous values. The algorithm takes 20 CPU units and occupies 312 bytes of code.

In Prolog the most recent element is always at the beginning of the database. Therefore, the problem is reduced to simple retract predicates, which are fast. This algorithm takes 10 CPU units and has 124 bytes of code.

Finding an edge. During editing, the user often must find an edge, given only the name of one of its end points.

PASCAL

```
function auxlookfor(p:tyedge):boolean;

begin
  if p = nil then auxlookfor: = false
  else begin
    search: = p ^.next;
    while(search < >p)and(search ^.v2 < >ver) do
      search: = search ^.next;
    auxlookfor: = search ^.v2 = ver
  end
end;
```

PROLOG

```
edges(Name, Flag, V1, V2, Eq).
```

In Pascal a Boolean function returns the direction of the element, if it is found, as a parameter. The global variable *ver* contains the final vertex of the edge the algorithm is seeking. This algorithm takes an average of 16 CPU units to search through 15 edges on a list. It occupies a total of 184 bytes.

In Prolog the process is considerably simpler. For the same data as the Pascal example, it requires four CPU units and 24 bytes of code.

Displaying a figure. Displaying a figure is another frequently performed action.

PASCAL

```
procedure drawfigure;

procedure drawto(p:tyedge);

var q:tyedge;

begin
  q: = p;
  repeat
    with q ^ do
    begin
      xlast: = vertices[v1].x;
      ylast: = vertices[v1].y;
      xpoint: = vertices[v2].x;
      ypoint: = vertices[v2].y;
```

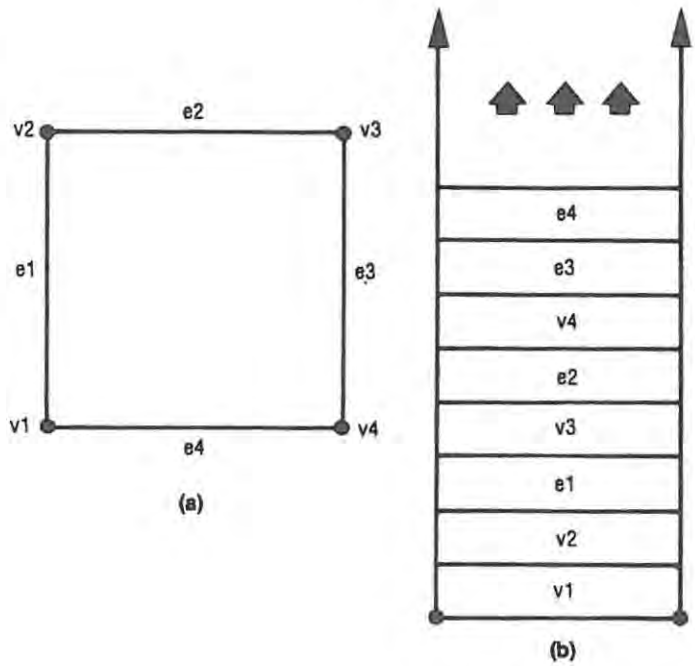


Figure 8. A simple rectangle (a); internal representation in Prolog (b).

```
      write(chr(entergraphmode));
      transmicoor(xlast,ylast);
      transmicoor(xpoint,ypoint);
      write(chr(leavegraphmode))
    end;
    q:=q^.next;
  until q=p;
end;

begin
  clearscreen;
  for group:=first to second do
    if view[group]<>nil then
      drawto(view[group]);
  for i:=1 to indhole do
      drawto(holes[i])
end;
```

PROLOG

```
draw_to(V1,V2):-vertices(V1,X1,Y1,_), vertices(V2,X2,Y2,_),
      entergraphmode,
      transmicoor(X1,Y1),
      transmicoor(X2,Y2),
      leavegraphmode.

draw_figure:- clearscreen,
      edges(_,_,V1,V2,_,_,_), draw_to(V1,V2), fail.
draw_figure.
```

The two algorithms use the routine *transmicoor* to send the coordinates to the Tektronix terminal and the routine *clearscreen* to clear the screen.

In Pascal the structure is stored as a set of lists representing the paths; Prolog stores the vertices and edges in the database in order of entry.

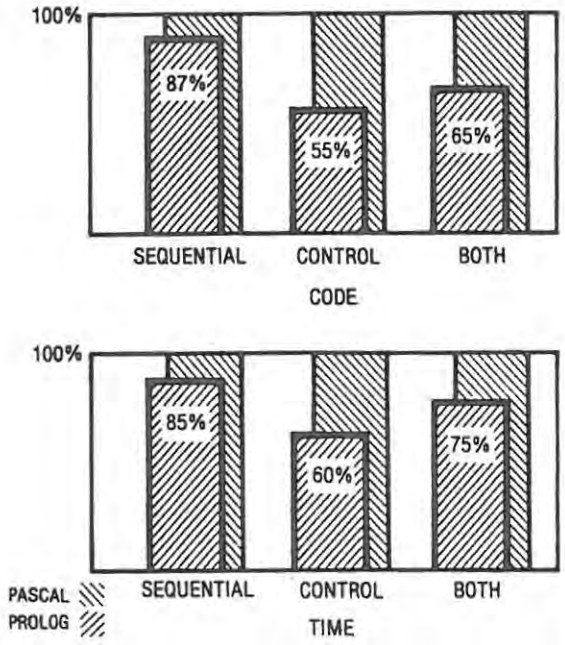


Figure 9. Code size and execution time of Prolog segments as a proportion of the code size and execution time of corresponding Pascal segments.

For a figure with 30 edges, the Pascal algorithm takes an average of 610 CPU units (approximately 20 CPU units for each edge) and requires 296 bytes. The Prolog clauses take an average of 285 CPU units (approximately nine CPU units for each edge) and occupy 138 bytes.

Development time. Another aspect of comparison is more subjective: development time. The time taken to analyze the problem, code it, and implement it with the descriptive approach of Prolog was about 65 percent of that required by the prescriptive approach of Pascal.

Whole-system comparisons. Finally, let us compare the complete systems. For this purpose, we examined those parts that are essentially sequential separately from those involving more complex control structures.

The two bar charts in Figure 9 illustrate the relative amounts of object code and execution time required by Pascal and Prolog to execute the types of algorithms discussed above. For the particular translators used, Prolog took less memory space and spent less time than did Pascal. Although the smaller memory requirements were not surprising, the reduced execution time was unexpected, particularly in view of the fact that the Prolog system used was an interpreter. It would be wrong to place undue emphasis on this result without taking account of the efficiency of individual translators; nevertheless it does show that Prolog can handle CAD applications as effectively as Pascal, if not more so. In both cases, it was easy to access the data structure and perform operations. User interaction is identical in the two implementations.

We also compared the two systems' source size. The unit of measure used for Pascal was an instruction—e.g., X:=a*b-3;, drawto(faces[i]). For Prolog the unit was a predicate—e.g., vertices(V, X, Y, View), find_a_point (V, X, Y). The Pascal system has 1560 units, compared with Prolog's 979.

Conclusion

This comparison has shown that Prolog is not only more concise, more readable, and clearer than Pascal, but also more efficient, at least on the compilers we used. Furthermore, Prolog programs were developed more quickly and with minimum error.

We implemented the two systems on a PDP-11/34. The NU7 Prolog interpreter developed at Edinburgh University handled the Prolog system, and the VU Pascal compiler implemented by Vrije University compiled the Pascal system. In view of our results, the three-dimensional modeler will be implemented by means of the UNSW Prolog system from Saint Joseph's University on a Zilog Z8000 microcomputer. Our experience with Prolog suggests that its use for the CAD portion of the Japanese Fifth-Generation Computer Project is likely to be a success. ■

References

1. D. H. D. Warren, "Implementing Prolog—Compiling Predicate Logic Programs," technical reports 39 and 40, Dept. of Artificial Intelligence, Univ. of Edinburgh, Edinburgh, U.K., 1977.
2. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, 1981.
3. D. H. Warren, L. M. Pereira, and F. Pereira, "Prolog—the Language and Its Implementation Compared to Lisp," *Sigplan Notices*, Vol. 12, No. 8, Aug. 1977, pp. 109-115.
4. D. McDermott, "The Prolog Phenomenon," *Sigart Newsletter*, No. 73, July 1980, pp. 16-20.
5. H. Gallaire and J. Minker, eds., *Logic and Data Bases*, Plenum Press, New York, 1978.
6. F. C. N. Pereira and D. H. D. Warren, "Definite Clause Grammars for Language Analysis—A Survey of the Formalism and a Comparison with Augmented Transition Networks," *Artificial Intelligence*, Vol. 13, No. 3, May 1980, pp. 231-278.
7. K. L. Clark and F. G. McCabe, *Prolog: A Language for Implementing Expert Systems*, Dept. of Computing, Imperial College, London, Nov. 1980.
8. Z. Markusz, "Application of Prolog in Designing Many-Storied Dwelling Houses." *Proc. Logic Programming Workshop*, Debrecen, Hungary, July 14-16, 1980, pp. 249-260.
9. T. Moto-Oka et al., "Challenge for Knowledge Information Processing Systems," in *Fifth Generation Computer Systems*, North-Holland, Amsterdam, 1982, pp. 3-89.
10. E. Santane-Toth and P. Szeredi, "Prolog Applications in Hungary," in *Logic Programming*, K. L. Clark and S. A. Tarnlund, eds., Academic Press, New York, 1982, pp. 19-32.
11. W. T. Zaumen, "Computer Assisted Circuit Evaluation for VLSI," SRI International internal report, Sept. 1982.
12. W. Fitzgerald, F. Gracer, and R. Wolfe, "GRIN: Interactive Graphics for Modeling Solids," *J. IBM Research and Development*, Vol. 25, No. 4, July 1981, pp. 281-294.
13. J. W. Boyse and J. E. Gilchrist, "GMSolid: Interactive Modeling for Design and Analysis of Solids," *IEEE Computer Graphics and Applications*, Vol. 2, No. 2, Mar. 1982, pp. 27-40.
14. J. A. Robinson, *Logic: Form and Function*, North-Holland, Amsterdam, Jan. 1979.
15. R. A. Kowalski, *Logic for Problem Solving*, Elsevier, North-Holland, Amsterdam, 1979.
16. W. F. Clocksin and J. D. Young, "Introduction to Prolog, a 'Fifth-Generation' Language," *Computerworld*, Aug. 1983, pp. 1-16.



Juan Camacho Gonzalez is with the software research group at the CRIL (Conception et Réalisation Industrielle de Logiciel) in Paris. He received the BSc degree in electrical engineering from the ITR No. 7, Madero, Mexico, in 1976; the MSc degree in electrical engineering from the CIEA-IPN, Mexico City, in 1979; the docteur-ingénieur in software engineering from the ENST, Paris, in 1981; and the PhD in computer science from Heriot-Watt University, Edinburgh, in 1983.



M. Howard Williams is a professor and head of the Department of Computer Science at Heriot-Watt University, Edinburgh, Scotland. Prior to 1980 he was a professor of computer science at Rhodes University, Grahamstown, South Africa, where he was active in the Computer Society of South Africa and acted as editor of the society's journal, *Quaestiones Informaticae*. His main research interests are programming languages and compilers, logic programming, expert systems, and computer-aided design.

Williams received an honors degree in physics and a PhD degree in ionospheric physics from Rhodes University. He is a member of the ACM and CSSA and a fellow of the BCS.



Ian E. Aitchison is a lecturer in the Department of Computer Science, Heriot-Watt University, Edinburgh, Scotland. After a period in the engineering industry, he joined the Mathematics Department at the Heriot-Watt University in 1964. His research interests include computer graphics and computer-aided design.

Aitchison received the BSc in mathematics from Heriot-Watt University in 1964. He is a member of the British Computer Society and an associate fellow of the Institute of Mathematics and Its Applications.



Symposium topics included AI in medicine, automated medical records, a video teaching tool for diabetics, mapping the healing of burned patients using Apple II graphics, a computerized nursing care plan, and a general psychiatric database for research use. 989 pp.

Order #503

PROCEEDINGS—7th Annual Symposium on Computer Applications in Medical Care
October 23-26, 1983

Nonmembers—$75.00
Members—$45.00

Order from IEEE Computer Society Order Dept.
PO Box 80452, Worldway Postal Center
Los Angeles, CA 90080 USA
(714) 821-8380

44. M.H.WILLIAMS, Towards deductive database systems, Proceedings of the Third Portuguese Conference on Artificial Intelligence (EPIA), University of Minho, Braga, Portugal, 365-374 (1987).

Towards Deductive Database Systems

M.H.Williams

Computer Science Dept., Heriot-Watt University, Edinburgh EH1 2HJ, UK.

Abstract - The storage, retrieval and manipulation of large quantities of knowledge as opposed to more conventional data types presents a challenging problem. A first approximation to solving this problem can be obtained by extending existing systems. However, there are fundamental theoretical problems which need to be solved to produce a clean model which is efficient to implement, to replace the relational model. Some of the issues are considered here.

Paper presented at the 3rd Portuguese Conference on Artificial Intelligence (EPIA-87).

1. DATABASES AND AI

The rapid expansion of research into knowledge based systems coupled with the tremendous growth in expert systems which is anticipated, has led to increasing interest in knowledge bases themselves. From the point of view of databases knowledge represents a new and interesting form of data to store and manipulate. With this in mind researchers are seeking to extend the ideas of databases to cater for this more general data type.

Of the different ways of representing knowledge, first order logic is the easiest one to relate to databases. One reason for this is that relational database theory is firmly based on logic, relational calculus being a restricted form of first order logic [1]. For this reason much of the current research on extending databases is concerned with extending them in the direction of predicate logic.

This research can be divided into two categories, viz.

(a) Theory-based research, aimed at establishing the theoretical foundations for extended database systems and developing systems along these lines.

(b) Pragmatic research, aimed at producing extended database systems from current systems by ad hoc approaches.

This paper explores some of the issues and problems associated with this research.

2. GENERALITY OF CLAUSES

A deductive database is a database which is used to store, retrieve and manipulate logical clauses. One decision which is crucial to such a system is the form of the clauses to be catered for.

The most general form is full clausal form (equivalent to first order predicate calculus) [2,3], in which each clause is a disjunction of literals (negated or unnegated atomic formulae), e.g.

$$\neg P_1 \vee \ldots \vee \neg P_m \vee Q_1 \vee \ldots \vee Q_n$$

or, equivalently, an implication with joint conditions and alternate conclusions, i.e.

$$P_1 \wedge \ldots \wedge P_m \rightarrow Q_1 \vee \ldots \vee Q_n$$

Full clausal form can be broken down into a number of categories, depending on the values of m and n, as follows

n \ m	0	1	>1
0	$\rightarrow$	$P_1 \rightarrow$ Simple negation	$P_1 \wedge \ldots \wedge P_m \rightarrow$ Compound negation
1	$\rightarrow Q_1$ Fact	$P_1 \rightarrow Q_1$ Simple rule	$P_1 \wedge \ldots P_m \rightarrow Q_1$ Compound rule
>1	$\rightarrow Q_1 \vee \ldots \vee Q_n$ Indefinite fact	$P_1 \rightarrow Q_1 \vee \ldots \vee Q_n$ Simple indefinite rule	$P_1 \wedge \ldots \wedge P_m \rightarrow Q_1 \vee \ldots \vee Q_n$ Compound indefinite

2.1 The Closed World Assumption

One of the fundamental assumptions made in database theory is the Closed World Assumption (CWA), which states that if a relationship between individuals is not known to hold then it is assumed to be false [4]. For example, if a database contains the set of facts $\{on(a,b), on(b,c)\}$ then the CWA allows one to conclude $\{\neg on(a,c), \neg on(b,a), \neg on(c,a), \neg on(c,b), \ldots\}$. Clark [5] showed that the CWA is equivalent to the assumption of negation as failure, an assumption by which a logical formula is assumed to be false if it cannot be proved to be true. The opposite of the CWA is the Open World Assumption (OWA) in which, unless the negation of a predicate is explicitly stated, one cannot assume its existence

For most applications, if one were not to adopt the CWA, the amount of data which would need to be stored explicitly would rise astronomically. On the other hand, if one adopts the CWA, it does place considerable restrictions on the formats of clauses which one can store in the database. In particular

(a) If one permits clauses for which $n=0$ and $m \geq 1$, i.e.

$$P_1 \wedge \ldots \wedge P_m \rightarrow \qquad m \geq 1$$

the system will have to cope with two different types of negation - explicit negation by a clause of this form and implicit negation by the CWA. The main reason for wanting to do this is to distinguish between something which is definitely known to be false and something which is unknown. Thus one now has three categories to deal with:

- definitely false
- unknown but assumed to be false
- definitely unknown

For this one requires a hybrid assumption somewhere between the CWA and the OWA. To retain the CWA, it is necessary to restrict consideration to clauses for which $m \neq 0$.

(b) If one permits clauses for which $n > o$, the CWA can lead to inconsistencies. For example, given the clause

$$p(a) \vee p(b)$$

one cannot deduce that $p(a)$ is definitely true, and hence by the CWA one can conclude $\neg p(a)$ is true. Similarly one can deduce that $\neg p(b)$ is true. Hence one can conclude

$$\neg p(a) \wedge \neg p(b)$$

which contradicts the initial clause. Thus it is necessary to restrict clauses to $n \not> 1$.

For these reasons, in order to apply the CWA, clauses must have the form

$$P_1 \wedge \ldots \wedge P_m \rightarrow Q_1 \qquad m \geq 0$$

Clauses with this form are referred to as definite clauses. A Horn clause is either a definite clause or a formula of the form

$\neg(A_1 \wedge \ldots \wedge A_n)$

which states that at least one of the A_i has value false.

Hence a deductive database which satisfies the CWA consists of a set of definite clauses while queries have the form

$\neg(A_1 \wedge \ldots \wedge A_n)$

Such databases are referred to as definite deductive databases or Horn clause databases.

Thus although the CWA enables query evaluation to be carried out more efficiently, it does limit the kind of information which can be represented. In particular, it excludes two important categories of information. In order to overcome this a number of different approaches have been proposed recently in order to attempt to represent and reason with negative, indefinite and default information. All of these can be viewed as relaxations of the CWA.

2.2 Minimal models

Another way of explaining paragraph (b) of the previous section is in terms of minimal models. For example, consider the set of clauses

$p(X) \rightarrow q(X)$
$p(a)$
$r(b)$

This set of clauses tells one that p(a), q(a) and r(b) are all true but says nothing about p(b), q(b) or r(a). Any model of this set of clauses must therefore include the fact that p(a), q(a) and r(b) are true, and may make any assumptions about p(b), q(b) or r(a). The largest possible model, M_{max}, assumes that all six literals are true. A model m is said to be a minimal model if no proper subset of m is a model of the set of clauses. In the case of this example, {p(a), q(a), r(b)} is a minimal model.

Now consider the database

$p(x) \rightarrow q(x)$
$p(a) \vee p(b)$
$r(b)$

for which r(b) is definitely true and it is known that p(a) and p(b) cannot both be false. In this case possible models include {r(b),p(a),q(a)}, {r(b),p(b),q(b)}, {r(b),p(a),q(a),p(b),q(b)}, etc. For this example there are two minimal models, viz.

{r(b), p(a), q(a)} and {r(b),p(b),q(b)}

It can be shown that a definite deductive database has a unique minimal model, and that querying a database under the CWA is equivalent to evaluating the query with respect to the minimal model. However, if a database contains an indefinite clause, there may not be a unique minimal model. The lack of a unique minimal model is what gives rise to inconsistencies when the CWA is applied to the set of clauses.

In order to deal with the problem of a set containing more than one minimal model, Minker [6] proposed an extension to the CWA known as the Generalised Closed World Assumption (GCWA). This defines the semantics of a database containing both definite and indefinite clauses in terms of a set of minimal models as follows. A formula is true in the database if it is true in all minimal models, it is false in the database if it is false in all minimal models, and it is indefinite or unknown if it satisfies neither condition.

Grant and Minker [7] have investigated query evaluation under the GCWA for a restricted set (relational databases with indefinite assertions) and give an algorithm for finding all possible minimal answers to a query, but even for this special case their algorithm is exponential. To handle the full set of definite and indefinite clauses they propose an extension in which the deductive database is used to generate a new database containing only definite and indefinite assertions, to which their algorithm can then be applied. An alternative approach put forward by Loveland [8] involves splitting a database DB containing definite and indefinite clauses into a family of databases DB_i each containing only definite clauses so that some query Q succeeds from DB if and only if it succeeds from each of the DB_i. However, both techniques are impractical for a database of any size.

One consequence of minimal model semantics is that all disjunctions are treated as exclusive disjunctions. For example, consider the set of clauses

$p(a) \vee p(b)$
$p(a)$

This set has only one minimal model, namely $\{p(a)\}$, and hence p(b) will be assumed to be false.

The GCWA applies only to databases containing definite and indefinite clauses, i.e. non-negative clauses. Gelford and Pzymusinska [9] have extended this and propose a careful closure procedure which will handle simple negative clauses provided that they are ground. They modify Loveland's evaluation algorithm for their purposes but once again the result is extremely inefficient.

A further extension to the GCWA has been proposed by Yahya and Henschen [10] and Naqvi [11] to include any negative clauses (simple or compound) provided that they are ground clauses. The resulting formalism is referred to as the Extended General Closed World Assumption (EGCWA). The extension which they propose is as follows:

> A negative ground clause $\neg P_1 \vee \ldots \vee \neg P_n$ may be assumed if and only if it is true in every minimal model of the database, i.e. no minimal model of the database contains all atoms $P_1 \ldots, P_n$

To illustrate this consider the following set of clauses

$p(a) \vee p(b)$
$p(c)$

This has two minimal models $\{p(a), p(c)\}$ and $\{p(b), p(c)\}$. By the EGCWA one can assume negative clauses such as: $\neg p(a) \vee \neg p(b)$, $\neg p(c) \vee \neg p(d)$, etc.

For a given database DB, let $\overline{DB}$ be the set of all minimal ground negative clauses that can be assumed from DB under the EGCWA. If every element in $\overline{DB}$ is an atom, the EGCWA reduces to the GCWA, and if DB is definite, it reduces to the CWA. However, no method has yet been proposed for implementing the EGCWA with any acceptable degree of efficiency. Since it is based on minimal model semantics the EGCWA also effectively treats all disjunctions as exclusive disjunctions.

A slightly different extension of minimal model semantics has been proposed in which negative literals are given a different priority depending on where they occur in the formula. This includes the positivistic model semantics of Bidoit and Hull [12] and the perfect model semantics of Przymusinski [13].

Other general approaches to handling more general sets of clauses than the class of definite clauses include Bossu and Siegel's system of non-monotonic reasoning called subimplication [14,15] and McCarthy's circumscription [16,17].

3. EXTENDING EXISTING SYSTEMS

The simplest way of producing a database system capable of storing, retrieving and manipulating rules as well as facts is to extend existing systems. Three basic strategies suggest themselves:

(a) Starting with a relational database system, extend the system to handle simple (recursive) rules and to perform inference where necessary.

(b) Starting with a Prolog system, extend it to include a database facility.

(c) Starting with a Prolog system and a relational database system, couple the two together in some way.

Consider each of these in turn.

3.1 Extending a relational database

The aim of this approach is to adapt an existing relational database system to enable it to provide for the storage and retrieval of simple recursive rules as well as a simple inference mechanism.

Because of the importance of security and integrity in database systems, one of the prime concerns in including recursion in databases has been that of detecting termination. In the case of programming languages such as Prolog, it is the programmer's responsibility to ensure that infinite loops do not occur, and to deal with them if they do. In a database system tighter controls are necessary as the system cannot permit one user to get into an infinite loop and thereby lock out all other users from accessing the data.

A number of approaches have been put forward recently for solving this problem [18,19]. These methods generally make the assumption that the database is function-free and comprises only finite relations [20]. However, all carry a considerable overhead and are costly in terms of both storage and processing time.

One such approach is Vieille's Query/Subquery approach (QSQ) [21]. If a clause

a(X,Y)←b(X,Z) & a(Z,Y)

is invoked by some query then b(X,Z) is said to be a subquery and a(Z,Y) a recursive subquery of the query a(X,Y). Two subqueries are equal if they have the same name and identical constant arguments (in the equivalent positions).

The state of processing is defined by a pair (Q,R) where Q is a set of subqueries to be answered and R the set of answers to be found. In order to detect termination it is necessary to store temporarily the set of subqueries Q and the set of answers R accumulated throughout the processing of the whole query. Each time a new tuple or subquery is generated the sets Q and R are searched to ensure that the tuple or subquery has not been encountered previously.

It will be evident from this that the overhead in employing this approach to evaluate a query could be very high. This overhead can be reduced by combining the approach with other intelligent strategies. For example, most of these methods use only the rules and facts in the database and do not consider any additional knowledge on the structure of the data. An exception to this is the approach described by Wicovsky et al [22] in which special integrity constraints are used to maintain the acyclic structure of the data and these same constraints are used to reduce the burden of detecting termination.

Until a more efficient approach can be found for dealing with recursion within the database system itself, this way of extending existing systems is unlikely to match the flexibility and generality of the other two approaches.

3.2 Extending a Prolog system

Another possible strategy is to take an existing Prolog system and build into it the appropriate routines to provide access to and control of files along the lines of a database system. However, to build up such a set of routines from scratch is a large task and makes this approach less attractive than the following one.

3.3 Coupling together a Prolog system and a database system

Each Prolog system has associated with it an internal database or knowledge base in which the clauses are stored. However, Prolog systems are not designed to cope with large numbers of clauses. A set of simple benchmarks designed to evaluate the performance of Prolog for database applications [23] shows up the limitations of most Prolog systems. Using these on several Prolog systems running on a 4Mb SUN system revealed that they are unable to cope with more than about 60,000 clauses and even then the overhead of loading these clauses into memory is very high [24].

Thus the idea behind linking a Prolog system to an existing database system is to take advantage of the storage and retrieval facilities of the database to store the clauses and to use the inference mechanism of Prolog to execute them. The linkage between the two components may be effected by either a tight coupling or a loose coupling. However, the definitions of tight and loose coupling seem to vary from one system to the next.

A classic example of a coupled system is the EDUCE system developed at the ECRC [25]. This involves the integration of Prolog with the relational database management system Ingres using both forms of coupling. Loose coupling, between Prolog and Ingres, set up as two separate tasks running under UNIX, is effected by means of pipes; one pipe is reserved for queries passing from Prolog to Ingres and another for replies travelling in the reverse direction. Tight coupling has been achieved by extracting the access routines from Ingres and embedding them into the Prolog system. Thus Educe offers a hybrid strategy for query evaluation, providing for both tuple-oriented retrieval (returning a single tuple at a time) or set-oriented retrieval (in which the data base management system is used to fetch the set of all tuples which satisfy the query).

The chief advantage of this type of approach is that the two main components of such a system, the Prolog compiler and the database system are usually existing systems which are adapted to effect such a linkage. This facilitates rapid development of the composite system. However, since the components have been designed for different purposes, the way in which they are linked together may lead to peculiarities of behaviour, unnecessary constraints and inefficiencies.

DEFICIENCIES OF A SIMPLE EXTENSION

The main disadvantage of extending existing systems is that they do not cater for the naive user. The interface which the user sees is that of Prolog, a language which is unsuitable for the casual user. Moreover, Prolog does not offer a great deal of protection to the user but relies on the user to ensure that he does not get himself into a closed loop.

There are a number of variations on this basic technique, aimed at improving the user interface. For example, an existing database query language may be embedded into the Prolog system to provide a more user friendly interface for simple queries. This approach has been adopted in the EDUCE system.

An alternative strategy is to construct a deductive or logic database management system in Prolog which will run on a Prolog-database system and provide other database facilities such as integrity constraints (geared to clauses rather than simple facts), security or authorization (applied to clauses), etc. However, this raises a number of research issues.

CONCLUSION

The extension of an existing database system to provide a rule handling capability or the coupling of existing Prolog and database systems together provides a pragmatic approach to producing a more powerful database system capable of handling knowledge. However, such systems are limited in two respects.

(a) They cater only for definite clauses and therefore exclude two important categories of knowledge.

(b) The user interface is at the level of Prolog which is not suitable as a query language for a database system.

In order to generalize the internal representation used, it is necessary to determine the theoretical foundations of the model used, and, in particular, the assumptions made concerning negation. It is also essential to find efficient algorithms for realizing these assumptions.

The development of more general query languages raises fundamental issues such as how integrity constraints or authorization constraints are applied to rules. Equally problems arise when values may be stored explicitly as well as derived implicitly. Updates may change explicit facts making them inconsistent with tuples derived implicitly from rules.

At this stage a number of problems exist and no clear theory has yet emerged which will produce a clean and efficient implementation and which will be the natural successor to the relational model.

1. P.M.D.Gray, Logic, Algebra and Databases, Ellis Harwood, Chichester, 1984.
2. R.A.Kowalski, Logic for Problem Solving, Elsevier, 1979.
3. W.F.Clocksin and C.S.Mellish, Programming in Prolog, Springer-Verlag, New York, 1981.
4. R.Reiter, On Closed World Data Bases, in Logic and Databases (ed.H.Gallaire and J.Minker), Plenum Press, New York, 1978, 55-76.
5. K.L.Clark, Negation as failure, in Logic and Databases (ed.H.Gallaire and J.Minker), Plenum Press, New York, 1978, 293-322.
6. J.Minker, On indefinite databases and the closed world assumption, Proc.of Sixth Conference on Automated Deduction, LNCS 138, Springer-Verlag, Berlin, 1982, 292-308.
7. J.Grant and J.Minker, Answering queries in indefinite databases and the null value problem, Advances in Computing Research, vol.3, 1986, 247-267.
8. D.Loveland, Automated Theorem Proving: A Logical Basis, North-Holland, New York, 1978.
9. M.Gelford and H.Przymusinska, Negation as Failure: Careful Closure Procedure, Artificial Intelligence, vol.30, 1986, 273-287.
10. A.Yahya and L.J.Henschen, Deduction in non-Horn Databases, Journal of Automated Reasoning, vol.1, 1985, 141-160.
11. S.A.Naqvi, Some extensions to the Closed World Assumption in Databases, Proc.International Conference on Database Theory, Rome, 1986.
12. N.Bidoit and R.Hull, Positivism vs Minimalism in Deductive Databases, Proc.Fifth Conference on Principles of Database Systems, Cambridge, Mass., 1986, 123-132.
13. T.C.Przyminski, On the Semantics of Stratified Deductive Databases, in Foundations of Deductive Databases and Logic Programming, ed.J.Minker, 1986.
14. G.Bossu and P.Siegel, Nonmonotonic Reasoning and Databases, in Advances in Database Theory, vol.2, ed.H.Gallaire, J.Minker and J.Nicolas, Plenum Press, New York, 1984, 239-284.
15. G.Bossu and P.Siegel, Saturation, Nonmonotonic Reasoning and the Closed World Assumption, Artificial Intelligence, vol.25, 1985, 13-63.
16. J.McCarthy, Circumscription in a form of non-monotonic reasoning, Artificial Intelligence, vol.13, 1980, 27-39.
17. J.McCarthy, Circumscription and other non-monotonic formalisms, Artificial Intelligence, vol.13, 1980, 171-172.
18. F.Bancillion and R.Ramakrishnan, An Amateur's Introduction to Recursive Query Processing Strategies, ACM SIGMOD, 1986.
19. L.Vieille, Recursive Axioms in Deductive Databases: Various Solutions, Technical Report KB-6, ECRC, Munich, May 1985.
20. D.E.Smith, M.R.Genesereth and M.L.Ginsburg, Controlling Recursive Inference, Artificial Intelligence, 30, 1986.
21. L.Vieille, Recursive Axioms in Deductive Databases: The Query/Subquery Approach, Proc.of First International Conference on Expert Systems, 1986, 4.
22. K.Wicovsky, P.F.Thanisch and M.H.Williams, Improved Recursion Handling through Integrity Constraints, Internal Report, Computer Science Dept., Heriot-Watt University, 1987.
23. M.H.Williams, P.A.Massey and J.A.Crammond, Benchmarks for Prolog from a Database Viewpoint, Proc.of Second Alvey Workshop for SIGKME, Alvey Directorate, May 1987, C1-C8.
24. M.H.Williams, P.A.Massey and J.A.Crammond, Benchmarking Prolog for Database Applications, Technical Report, Computer Science Department, Heriot-Watt University, Edinburgh, 1987.
25. J.Bocca, EDUCE A Marriage of Convenience: Prolog and a Relational DBMS, Proc.of Third Symposium of Logic Programming, Salt Lake City, Utah, USA, 1986.

45. P.CHISHOLM, G.CHEN, D.FERBRACHE, P.THANISCH and M.H.WILLIAMS, Coping with indefinite and negative data in deductive databases: a survey, Data and Knowledge Engineering, vol. 2, 259-284 (1987).

Data & Knowledge Engineering 2 (1987) 259–284
North-Holland

Coping with indefinite and negative data in deductive databases: A survey

P. CHISHOLM, G. CHEN, D. FERBRACHE, P. THANISCH and M.H. WILLIAMS

Department of Computer Science, Heriot-Watt University, Edinburgh EH1 2HJ, United Kingdom

Abstract. A problem with current database systems is the limitation placed on the type of data which may be represented and manipulated within such systems. In an attempt to broaden this to a wider class of data (i.e. rules as well as facts) and a more powerful set of manipulations, the concept of a deductive database was introduced. However, for the sake of efficiency the type of rule which is allowed in a deductive database is restricted in form. This paper surveys a number of attempts to move towards less restrictive forms of rules in deductive databases which allow indefinite and negative data to be handled.

Keywords. Deductive database, negative data, indefinite data, query evaluation.

1. Introduction

In 1985 and 1986, a number of papers were published describing approaches to the problem of handling indefinite and negative data in deductive databases. Newcomers to this field will find it difficult to compare the approaches. Firstly, there are few cross-references between these papers. (This is not surprising, in view of the time delay between submission and publication). Secondly, these papers tend to concentrate on presenting the ideas, rather than dealing with such 'niceties' as implementability. A related point (and this is a criticism of computer science research in general) is that original research publications tend to describe what *can* be done using a new technique. Consequently, the reader is left unclear as to what cannot be done. Finally, the authors of the various ideas make different assumptions about the database. As far as is possible, we have attempted to make a consistent set of assumptions for purposes of comparability.

Whereas relational databases are grounded in the relational theory developed by Codd [6], deductive databases are based on predicate logic. For this purpose formulae in first-order predicate calculus are converted to clausal form, i.e. they are represented as a set of clauses where each clause is a disjunction of literals (negated or unnegated atomic formulae), e.g.

$$\neg P_1 \vee \cdots \vee \neg P_m \vee Q_1 \vee \cdots \vee Q_n ,$$

or equivalently, a clause is an implication with joint conditions and alternate conclusions, i.e.

$$P_1 \wedge \cdots \wedge P_m \rightarrow Q_1 \vee \cdots \vee Q_n .$$

A database can be represented by a collection of clauses of this form. One could also restrict attention to some subclass of clauses; say clauses where n is restricted to 1, i.e. clauses of the

0169-23X/87/$3.50 © 1987, Elsevier Science Publishers B.V. (North-Holland)

form

$$P_1 \wedge \cdots \wedge P_m \rightarrow Q ,$$

Such restrictions limit the kind of information which can be represented but allow query evaluation to be carried out more efficiently.

Whatever representation is used, there are certain fundamental assumptions which are generally made about real world data in order to construct a practical database system [9]. These are

- *Unique name assumption*: each object appearing in the database is identified by a unique name. Unfortunately, there are exceptions to this rule, e.g. a criminal database in which a single subject may be referenced by several aliases.
- *Domain closure assumption*: the universe is restricted to contain only those objects named in the database. Thus if an object does not appear in the database, it does not exist.
- *Closed world assumption* (CWA): if a relationship between individuals is not known to hold, then it is assumed to be false. If the database contains the set of facts $\{on(a,b), on(b,c)\}$ then the CWA allows us to conclude $\{\neg on(a,c), \neg on(b,a), \neg on(c,a), \neg on(c,b), \ldots\}$.

However, when these assumptions are applied to deductive data bases, problems can arise. In particular, the CWA can lead to inconsistencies when applied to indefinite data. For example, given the (indefinite) clause

$$learns(jean,latin) \vee learns(jean,greek) ,$$

one cannot deduce that $learns(jean,latin)$ is true, and hence by the CWA one can assume $\neg learns(jean,latin)$ to be true. The formula $\neg learns(jean,greek)$ can be similarly deduced. Hence one can conclude

$$\neg(learns(jean,latin) \vee learns(jean,greek))$$

which contradicts the initial clause. Another problem arises with negative data in that if negative data are stored explicitly, it would imply that missing data should be treated as unknown rather than false.

The obvious alternative to the closed world assumption is the open world assumption in which, unless the negation of a predicate is explicitly stated, one cannot assume its existence. However, for most applications this assumption is impractical since the requirement to state explicitly each relationship which does not hold as being false, leads to an inordinately large amount of negative information requiring storage. In view of this problem, the rules stored in a deductive database system are generally restricted to clauses of the form:

$$P_1 \wedge \cdots \wedge P_m \rightarrow Q .$$

Such a database system is referred to as a definite deductive database [11], and for this restricted form the CWA can be adopted. Unfortunately the restrictions of a deductive database exclude two important categories of information, namely negative and incomplete information. This paper surveys a range of different approaches which attempt to represent and reason with negative, indefinite and default information. All of these approaches can be viewed as relaxations of the CWA, while retaining the domain closure and unique name assumptions.

2. Background and terminology

The following standard example will be used to illustrate the terminology and when discussing the various proposed models for deductive databases. It is taken from [4] and concerns students taking language courses. Informally, the information in the database is

- every student of the 3rd course learns latin or greek,
- nobody learns greek and german at the same time,
- modern languages and dead languages are languages,
- toto, jean and paul are students,
- jean and toto are students of the 3rd course,
- english and german are modern languages,
- latin and greek are dead languages,
- paul learns english and german,
- toto does not learn english.

The required relations are

$le(x, y)$	x learns y,
$s(x)$	x is a student,
$3rd(x)$	x is a student of the 3rd course,
$l(x)$	x is a language,
$lm(x)$	x is a modern language,
$ld(x)$	x is a dead language.

The constants of the database are

$$\{eng, ger, lat, gre, paul, toto, jean\}$$

This set of constants is called the *domain* of the database. In general, the domain of a database may also contain functions, but most work in the field of relational and deductive databases assumes the absence of functions.

An *atomic formula* is an expression of the form

$$P(t_1, \ldots, t_n)$$

where P is an n-ary predicate name and each t_i is a term, i.e. a constant or a variable (we consider function-free databases only). The t_i are the arguments of P. The predicate corresponds to a database relation and the arguments to objects from the domain of the database. For example, the atomic formula

$$le(paul, eng)$$

consists of the predicate name *le* and arguments *paul* and *eng*. An atomic formula is said to be *ground* if its arguments are all constants.

An *assertion* is a formula of the form

$$\rightarrow A$$

where A is an atomic formula. This asserts the formula A to be true. For example,

$$\rightarrow le(paul,eng)$$

is interpreted as meaning $le(paul,eng)$ has value true. In the context of databases, this is equivalent to stating that the tuple $(paul,eng)$ is an instance of the relation le.

An assertion is *ground* if its atomic formula is ground. The atomic formula in an assertion may contain variables; for example,

$$\rightarrow le(paul,x)$$

contains the variable x. Here x is interpreted as standing for any object of the database. The logical meaning of this formula is thus: for any object a, the formula $le(paul,a)$ has value true. With regards to databases, this is interpreted as: for any object a in the domain of the database, the tuple $(paul,a)$ is an instance of the relation le. Formulae containing variables make more general statements than ground formulae, e.g. $le(paul,x)$ is more general than $le(paul,eng)$.

The term *fact of the database* will be used to refer to any ground atomic formula which is either stored explicitly in the database as a ground assertion, or can be derived from a non-ground assertion or more general rule. For example if,

$$\rightarrow le(paul,x)$$

is an assertion in the database, then $le(paul, eng)$ is a fact of the database.

A *literal* is either an atomic formula or a negated atomic formula (i.e. an atomic formula preceded by the symbol $\neg$). For example,

$$\neg le(paul,eng)$$

is a literal whose logical meaning is: the formula $le(paul,eng)$ has value false. Alternatively, $le(paul,eng)$ is not a fact of the database.

A *definite clause* is a formula of the form

$$A_1 \wedge \cdots \wedge A_n \rightarrow B$$

where $\wedge$ and $\rightarrow$ denote conjunction and implication respectively, and the A_i and B are atomic formulae. Logically this is interpreted as: if each A_i has value true, then B has value true. With respect to databases: if each A_i is a fact of the database, then B is a fact of the database. When n equals 0, a definite clause reduces to an assertion.

A *Horn clause* is either a definite clause or a formula of the form

$$\neg(A_1 \wedge \cdots \wedge A_n)$$

where each A_i is an atomic formula. Logically this means: at least one of the A_i has value false. Alternatively, at least one of the A_i is not a fact of the database. Each A_i consists of a relation name and a tuple. One of the tuples does not satisfy the relation it is paired with. The programming language Prolog is based on Horn clause logic. Programs are expressed as definite clauses and formulae of the form $\neg(A_1 \wedge \cdots \wedge A_n)$ are interpreted as objects to be evaluated.

A *clause* is a formula of the form

$$\neg A_1 \vee \cdots \vee \neg A_m \vee B_1 \vee \cdots \vee B_n$$

where $\vee$ denotes disjunction and the A_i and B_j are atomic formulae. This is usually written as

$$A_1 \wedge \cdots \wedge A_m \rightarrow B_1 \vee \cdots \vee B_n .$$

It is interpreted as stating that if each A_i has value true, then at least one of the B_j has value true. Alternatively, if each A_i is a fact of the database, then one of the B_j is a fact of the database.

Every formula of first-order logic can be translated to a collection of clauses. The collection of clauses is called the *clausal form* of the formula. Formulae in clausal form can be used in theorem provers whose sole rule of inference is the resolution principle of Robinson [34]. Interpreters for Prolog use a form of resolution called SLDNF-*resolution* [19] SLDNF-resolution is restricted to the Horn clause subset of clausal form and employs the *negation as failure* principle. Under *negation as failure*, a formula is assumed to be false if it cannot be proved to be true.

The two basic approaches to evaluating logical formulae are model theory (in which each formula is assigned a truth value), and proof theory (in which a collection of formulae (axioms) and proof rules are given which may be used to infer other formulae). The model and proof theories are connected by the soundness and completeness results for the proof theory. The proof theory is *sound* if every provable formula is assigned the value true under the model theory. The proof theory is *complete* if every formula assigned the value true under the model theory is provable. Query evaluation procedures for deductive databases are essentially proof theories. The facts and rules of the database are the axioms of the proof theory. The evaluation procedure corresponds to the interference rules. The query is a formula to be proven from the axioms and rules. The statements of the soundness and completeness results for deductive databases are dependent on the formalism used but they have the same purpose as for proof theory. Soundness involves showing that anything which can be deduced is consistent with what is already known. Completeness involves showing everything that should be deducible can actually be deduced.

A database is defined to be *indefinite* if there are answers to a query that have the form

$$P(a_1, \ldots, a_n) \vee P(b_1, \ldots, b_n)$$

where $a_i \neq b_i$ for some i and neither $P(a_1, \ldots, a_n)$ nor $P(b_1, \ldots, b_n)$ individually are answers to the same query. Reiter [32] has shown that databases consisting purely of Horn clauses cannot have indefinite answers. He also shows Horn clause databases are consistent with the CWA but non-Horn clause databases can be inconsistent with the CWA (see the introduction).

A feature of traditional logics is that they have the property of *monotonicity*. Monotonicity is defined as follows: let S and T be arbitrary sets of formulae of some logic and let A be any formula provable from S. The logic is *monotonic* if A can be deduced from the set of formulae $S \cup T$.

Horn clause logic together with the CWA gives a non-monotonic logic. Let the sets of clauses S and T be $\{p \vee \neg q\}$ and $\{q\}$ respectively. Under the CWA, $\neg p$ is a consequence of S but p is a consequence of $S \cup T$, i.e. $\neg p$ is not a consequence of $S \cup T$.

The *Herbrand base* of a database is the set of all possible combinations of relation names applied to domain objects respecting the arities of the relations. For the example database, the Herbrand base is the set of atoms

$$\{s(eng),s(ger),s(lat),s(gre),s(paul),s(toto),s(jean),$$
$$3rd(eng),\ldots,3rd(jean),$$
$$l(eng),\ldots,l(jean),$$
$$lm(eng),\ldots,lm(jean),$$
$$ld(eng),\ldots,ld(jean),$$
$$le(eng,eng),\ldots,le(eng,jean),$$
$$\ldots$$
$$le(jean,eng),\ldots,le(jean,jean)\}$$

A *Herbrand interpretation* of a database is a subset of the Herbrand base. For example, the set of atoms

$$\{s(toto),s(jean),s(paul),3rd(jean),3rd(toto),lm(eng),lm(ger),ld(lat),ld(gre),$$
$$l(eng),l(ger),l(lat),l(gre),le(paul,ent),le(paul,ger),le(toto,lat),le(jean,gre)\}$$

is a Herbrand interpretation of the database. Intuitively, the meaning is that all atoms appearing in the Herbrand interpretation are true, or are facts of the database, and all other atoms are false.

Let p be a formula concerned with the relations of some database, DB, and I be an interpretation of DB. I is said to *satisfy* p if the value of p is true whenever the value of each atom in I is true. If I satisfies p, then I is a *model* of p. A database, DB, is a set of formulae. Thus, an interpretation of DB is a model of DB if it satisfies each formula in DB.

The information in the example is formalised by the following first-order formulae:

$$(\forall x)(3rd(x)\rightarrow(le(x,lat)\vee le(x,gre)))$$
$$\neg(\exists x)(le(x,gre)\wedge le(x,ger))$$
$$(\forall x)(lm(x)\vee ld(x)\rightarrow l(x))$$

$s(toto)$	$s(jean)$	$s(paul)$
$3rd(jean)$	$3rd(toto)$	
$lm(eng)$	$lm(ger)$	
$ld(lat)$	$ld(gre)$	
$le(paul,eng)$	$le(paul,ger)$	$\neg le(toto,eng)$

The above Herbrand interpretation is also a model of the example database. On the other hand, any interpretation not containing the atom $le(paul,eng)$ is not a model; its absence means the value of the atom $le(paul,eng)$ in the interpretation is false, so the database formula $le(paul,eng)$ would not be satisfied.

A database is *consistent* if it has a model. The *extension* of an (n-ary) relation in an interpretation is the set of n-tuples which satisfy the relation. The extension of le in the interpretation of the example database is the set of pairs

$$\{(paul,eng),(paul,ger),(toto,lat),(jean,gre)\}$$

The following standard queries, expressed as first-order formulae, will be considered with respect to this database. It is required that any free variables in queries be instantiated by the evaluation process. Viewing the database as a first-order theory, the answers to the queries with respect to classical logic are given:

(1) list those things that do not learn greek

$$\neg le(x,gre)$$

In classical logic, this formula is true in any interpretation which assigns the constant *paul* to the variable x.

(2) list those students who learn latin or greek but not english

$$s(x) \wedge (le(x,lat) \vee le(x,gre)) \wedge \neg le(x,eng)$$

This formula is true in any interpretation which assigns the constant *toto* to the variable x.

(3) for which students and languages is it the case that the student learns either the particular language or greek

$$s(x) \wedge l(y) \wedge (le(x,y) \vee le(x,gre))$$

This formula is true in any interpretation which assigns one of the pairs of contants *paul/eng*, *paul/ger*, *toto/lat* or *jean/lat* to the variables x and y respectively.

3. Extended program clauses

Lloyd and Topor [20–22] propose that deductive databases be based on a form of logic which is more expressive than Prolog but which can be directly translated to Prolog for query evaluation. The advantages of such databases are that clauses and queries may be expressed more naturally but do not need any more advanced theorem proving than SLDNF-resolution [19] which is simple and can be efficiently implemented. However, it does not relax the restrictions imposed by the CWA in any way.

An *extended program clause* has the form

$$A \leftarrow B$$

where A is an atomic formula and B is a (possibly absent) arbitrary first-order formula. An *extended goal* has the form

$$\leftarrow A$$

where A is an arbitrary first-order formula. A set of rules is given for transforming extended program clauses and goals to *general program clauses* and *goals*. General program clauses and general goals have the same form as extended program clauses and extended goals but the formula A may only be a conjunction of literals. General program clauses and goals are essentially those of Prolog (with the predicate *not*). This allows extended goals to be evaluated from extended programs (finite sets of extended program clauses) by first transforming to general program clauses and goals.

Example 3.1 (Lloyd and Topor [20]). The subset operation can be defined by the extended program

$$x \subseteq y \leftarrow (\forall u)(u \in y \leftarrow u \in x)\,.$$

The corresponding general program is

$$x \subseteq y \leftarrow \neg p(x,y)\,,$$
$$p(x,y) \leftarrow \neg(u \in y) \wedge u \in x\,.$$

Evaluating goals by transforming to Prolog means the convention for negative data is negation as failure and theorem proving is SLDNF-resolution. Negation as failure and SLDNF-resolution are both sound for extended programs. The technique requires a *safe* computation rule. A computation rule decides, given a collection of subgoals, which one to attempt next. A safe computation rule is one which only selects negative literal goals if they are *ground* (contain no variables) and, having selected such a goal, completely evaluates it before continuing with any other goals. The standard Prolog left-to-right computation rule is not safe. The disadvantage of a safe computation rule is evaluation may be delayed permanently. If the goals to be satisfied are non-ground negative literals, nothing can be done. The evaluation of the general form of extended goals with a computation rule which is unsafe can give answers which are wrong.

Deductive databases are considered to be extended programs in which all extended program clauses are typed [21] Types provide a natural way of expressing the domain concept of relational databases. A *database clause* has the form

$$A \leftarrow B$$

where A is an atomic formula and B is a (possibly absent) function-free typed first-order formula. A *query* has the form

$$\leftarrow A$$

where A is a function-free typed first-order formula. Standard techniques of logic are used to transform typed formulae to equivalent untyped formulae (by introducing type predicates). Thus, query evaluation consists of first transforming the database clauses and the query to their untyped forms. These untyped forms are extended program clauses and an extended goal whose evaluation is described above. Query evaluation in databases is sound with respect to goal evaluation in extended programs.

The translation from typed to untyped formulae means all variables in the resulting formulae appear in some type predicate. This ensures:

(a) a safe computation rule cannot delay evaluation permanently. The type predicates associated with the variables in a non-ground literal can be used to obtain ground instances of the literal.

(b) query evaluation always produces ground answers. The type predicates associated with the variables in a query can be used to obtain ground answers.

(c) the standard Prolog left-to-right computation rule can be made safe. Simply reorder the goals so that the type predicate associated with each variable in a literal comes before the literal.

With regard to negative and indefinite data in deductive databases, this formalism is found to be lacking in expressive power. Negative data cannot be stored. The form of rule simply does not allow it. Neither is it possible to store indefinite data. The clause $A \vee B$ is classically equivalent to $\neg A \rightarrow B$, suggesting that $A \vee B$ can be represented in the database by the clause $B \leftarrow \neg A$ (where B would have to be an atomic formula). Using SLDNF-resolution, evaluation of the goal B would succeed and evaluation of the goal A would fail with respect to this clause. Thus, $A \vee B$ has not been correctly represented, what has been represented is the fact B.

Negative and indefinite queries are allowed. A query of the form $\neg P(x)$ can be evaluated. The presence of type predicates and a safe computation rule ensures the variable x will be instantiated. Using SLDNF-resolution, evaluation of non-ground negative literals can only answer yes or no: variables cannot be instantiated. Indefinite queries of the form $A \vee B$ can

also be expressed. Such a goal would only succeed if either A succeeds or B succeeds. It may be that neither A nor B succeeds but $A \vee B$ is a logical consequence of the database. This situation cannot be handled.

Using this formalism, the standard database cannot be represented. It is not sufficiently expressive to handle negative and indefinite data. The basic problem is that the convention for negative data is negation as failure. With this convention, being able to specify negative data explicitly is of limited use and truly indefinite data cannot be handled consistently.

4. Generalised closed world assumption

Minker [27] proposes extending Reiter's closed world assumption [32] to handle indefinite databases. He calls this formalism the *generalised closed world assumption* (GCWA).

A feature of definite deductive databases is they have what is known as the *model intersection property* [7]. That is, the intersection of all models of a definite deductive database is itself a model of the database; namely, the *minimal model*. Querying a database under the CWA is equivalent to evaluating the query with respect to the minimal model.

Indefinite databases do not have the model intersection property. For example, the database consisting of the single clause

$$P(a) \vee P(b)$$

has three possible models

$$\{P(a)\} \qquad \{P(b)\} \qquad \{P(a),P(b)\}$$

The intersection of these three models is the empty set which is not a model of the database. The inconsistency which arises when the CWA is applied to indefinite databases is a result of this lack of a minimal model. Query evaluation under the CWA (or the related notion of negation as failure) implicitly assumes the existence of a unique minimal model; an assumption which is invalid for indefinite databases.

For indefinite databases, Minker modifies the notion of minimal model as follows.

Definition 4.1. Let DB be a database and db be a model of DB. db is a *minimal model* of DB if no proper subset of db is a model of DB.

Under this definition there may be more than one minimal model. From the example above, $\{P(a)\}$ and $\{P(b)\}$ are minimal models but $\{P(a),P(b)\}$ is not (both of the minimal models are proper subsets of it). In the first minimal model, $P(a)$ is true and $P(b)$ is false. In the second minimal model, $P(a)$ is false and $P(b)$ is true. Different answers may be obtained from the same query when it is evaluated with respect to different minimal models. It is not possible to choose an arbitrary minimal model for query evaluation. One must, in some sense, evaluate a query with respect to every minimal model. Thus, query evaluation on indefinite databases is inherently less efficient than a query evaluation on definite databases (where there is only one minimal model).

The motivation behind this definition of minimal model is that those atomic formulae appearing in all minimal models are true, those atomic formulae appearing in no minimal models are false, and those atomic formulae appearing in some minimal models but not in others have unknown values. Let DB be an indefinite database and Q a query over DB. Q is

true with respect to DB if Q is true with respect to all minimal models of DB, Q is false with respect to DB if Q is false with respect to all minimal models of DB, and Q is indefinite if Q is true with respect to some minimal models of DB and false with respect to others.

Consider the set of all minimal models of a database. The intersection of all members in this set contains those atomic formulae which are true in all minimal models (call it Γ). Similarly, consider the set whose members are the complements of the minimal models. The intersection of all members of this set contains those atomic formulae which are false in all minimal models (call it $\bar{\Gamma}$). $\bar{\Gamma}$ is said to constitute the generalised closed world assumption for the database, i.e. those atomic formulae which may be assumed to be false. The atomic formulae appearing neither in Γ nor in $\bar{\Gamma}$ are said to be indefinite, i.e. their truth values are unknown.

In addition to the above semantic definition of the GCWA, Minker also gives a corresponding syntactic definition. Consider a database DB and the collection of all clauses of the form

$$A_1 \vee \cdots \vee A_n$$

where each A_i $(1 \leq i \leq n)$ is an atomic formula such that

(1) $A_1 \vee \cdots \vee A_n$ is derivable from DB,

(2) the disjunction of any proper subset of $\{A_i, \ldots, A_n\}$ (for example $A_2 \vee \cdots \vee A_n$) is not derivable from DB.

Denote by E the collection of all atomic formulae that appear in a clause of the above form. The collection of all atomic formulae which do not appear in E constitute the GCWA for DB. Intuitively, E contains all those atomic formulae which *may be* true. Therefore, the complement of E contains atomic formulae which must be false. Minker proves that the semantic and syntactic definitions of the GCWA are equivalent.

Grant and Minker [13] investigate query evaluation under the GCWA. They give an algorithm to find all possible minimal answers to a query. It is essentially a generate and test algorithm, i.e. possible minimal answers are generated and then tested to determine if they are truly answers. Grant and Minker restrict their attention to relational databases (which may contain indefinite assertions), but even in this subcase their algorithm is exponential. They briefly describe a possible extension to allow deductive databases to be handled. This extension involves using the deductive rules to generate a new database which only contains assertions (possibly indefinite), then use the aforementioned algorithm. However, this technique can considerably enlarge the database placing a further burden on the query evaluation process.

Loveland [23] shows that a non-Horn database DB can be split into a family of Horn databases DB_i such that some query Q succeeds from DB iff it succeeds from each of the DB_i. Minker notes that the technique could be used for query evaluation under the GCWA. Gelfond and Przymusinska [12] investigate this further and generalise Minker's GCWA. They formulate a *careful closure procedure* which adds to a database the negations of those ground atomic formulae which can be assumed to be false. Their modification allows application of the negation as failure rule with respect to a set P of (not necessarily all) predicates of a database. Thus, Minker's GCWA is a special case of Gelfond and Przymusinska's careful closure procedure. The soundness and completeness of the careful closure procedure are established. They modify Loveland's evaluation algorithm for their purposes and show it is sound with respect to the careful closure procedure for a wide class of databases. However, the technique of splitting an indefinite database into a family of definite databases is inefficient. For example, consider a database containing some collection C of definite clauses plus the indefinite clause $A_1 \vee A_2$. The splitting process produces the two databases

$$C \cup \{A_1\} \qquad C \cup \{A_2\}$$

If there are a number of indefinite clauses, the size of the family of definite databases is the product of the number of positive literals in the indefinite clauses, i.e. let there be p indefinite clauses in the database and denote the number of positive literals in each indefinite clause by n_i $(1 \leq i \leq p)$, then the family of definite databases contains $\prod_{i=1}^{p} n_i$ members. Thus, the complexity of the splitting process is exponential in the number of indefinite clauses in the database. In addition to the burden of splitting the database, query processing involves evaluating some queries with respect to *all* members of the family of definite databases.

The minimal model semantics of the GCWA interprets $\vee$ as *exclusive* whenever possible. For example, let DB be $\{a, a \vee b\}$. The only disjunction DB can derive is $a \vee b$ and a is derivable in DB. Under the GCWA, one can assume $\neg b$. This is simply because the derivability of a implies $a \vee b$, so that $\neg b$ can be assumed without inconsistency. Thus, the interpretation of $\vee$ as exclusive or is a direct consequence of the minimal model semantic framework of the GCWA.

In summary, the GWCA is an extension of the CWA which can represent arbitrary clauses. This is clearly more general than the CWA which is only applicable to Horn clauses.

5. Extending GCWA: Non-atomic negative clauses as answers

Yahya and Henschen [36] and Naqvi [30] have extended the GWCA to accept non-atomic ground negative clauses, i.e. to decide whether one can assume a non-atomic ground negative clause $\neg L_1 \vee \cdots \vee \neg L_n$ when n may be greater than one. The formalism is called the *extended GCWA* (EGCWA). As for the GCWA, the problem is discussed from both the semantic (model theoretic) and syntactic (proof theoretic) viewpoints. Both Yahya and Henschen, and Naqvi give similar semantic and syntactic definitions, and prove the equivalence of the two definitions.

Definition 5.1 (Semantic definition). A non-atomic negative ground clause $\neg L_1 \vee \cdots \vee \neg L_n$ may be assumed if and only if it is true in every minimal model of DB, i.e. no minimal model of DB contains all atoms $L_1, \ldots, L_n$.

Definition 5.2 (Syntactic definition). A non-atomic negative ground clause $\neg L_1 \vee \cdots \vee \neg L_n$ may *not* be assumed if and only if there exists a positive or empty clause K such that for all i $(1 \leq i \leq n)$

(1) $DB \vdash L_i \vee K$, and
(2) $DB \nvdash K$.

The following example illustrates the above definition. Let $DB = \{p, q, r \vee s\}$. DB has two minimal models $\{p,q,r\}$ and $\{p,q,s\}$. One can assume the following negative clauses:

$$\neg p \vee \neg t,$$
$$\neg p \vee \neg u,$$
$$\neg r \vee \neg s,$$
$$\neg r \vee \neg q \vee \neg s,$$
$$\neg p \vee \neg r \vee \neg q \vee \neg s,$$
$$\ldots$$

In the last three clauses, every atom in the clause belongs to some minimal model, but no single minimal model contains all the atoms.

Let DB be a database and $\overline{DB}$ be the set of all minimal ground negative clauses that can be assumed from DB under the EGCWA. A ground negative clause is said to be minimal if no part of it can be assumed. The EGCWA shares the following properties with the GCWA [27, 36]:

(1) If DB is consistent, $DB + \overline{DB}$ is consistent.

(2) An interpretation M is a minimal model of DB if and only if it is a model of $DB + \overline{DB}$.

(3) If DB is consistent, then $DB + \overline{DB}$ is maximally consistent. That is, let C be a negative clause not derivable from $DB + \overline{DB}$. If C is added to $DB + \overline{DB}$, then there exists some empty or positive clause K (not derivable from $DB + \overline{DB}$) which is derivable from $DB + \overline{DB} + C$.

$$\forall C \exists K (DB + \overline{DB} \nvdash C \rightarrow (DB + \overline{DB} + C \vdash K) \wedge (DB \nvdash K)) .$$

This formula states that adding any new negative clause not already derivable leads to inconsistency, establishing a completeness result for the EGCWA.

The EGCWA reduces to the GCWA if every element in $\overline{DB}$ is an atom, and reduces to the CWA if DB is Horn. The EGCWA is thus a consistent extension of GCWA and CWA.

However, the implementability of the EGCWA is doubtful. The algorithm presented by Yahya and Henschen can only decide when one can assume a ground negative atomic clause $\neg c$. Furthermore, it is unlikely that the algorithm can be implemented efficiently.

Let DB be a database containing no *pure* negative literals. A negative literal $\neg p$ is pure in DB if there are no positive occurrences of p in DB. Let $C_1, \ldots, C_n$ be those clauses in DB containing a positive occurrence of the atom c, and $C_i' = C_i - c$ (i.e. C_i with c removed). Let DB' be the database resulting from removing each occurrence of $\neg c$ from each clause in $DB - \{C_1, \ldots, C_n\}$.

Theorem 5.3 (Yahya and Henschen [36]). *A negative atom $\neg c$ can be assumed if and only if every model of $DB' + \overline{DB'}$ is also a model of the clause set $\{C_1', \ldots, C_n'\}$.*

The essential part of the algorithm obtained from the above theorem can be described as: if, for each C_i', one can either derive the empty clause from $DB' + \neg C_i'$, or in case C_i' contains some negative atoms, one can derive the empty clause from $DB' + \overline{DB'} + \neg C_i'$, then $\neg c$ can be assumed. The algorithm works iteratively on each i $(1 \leq i \leq n)$ and tries to derive the empty clause from DB' and $\overline{DB'}$.

Example 5.4 Let DB be the database $\{a \vee \neg m, a \vee b \vee m, b \vee c, \neg c \vee e, \neg c \vee d \vee m, \neg a \vee n, \neg m \vee k, \neg m \vee \neg k\}$. If one requires to determine whether or not $\neg a$ can be assumed, one has

$$\begin{aligned}
&C_1 = a \vee \neg m , \\
&C_2 = a \vee b \vee m , \\
&C_1' = \neg m , \\
&C_2' = b \vee m , \\
&D_1 = \neg a \vee n , \\
&D_1' = n , \\
&DB' = \{n, b \vee c, \neg c \vee e, \neg c \vee d \vee m, \neg e \vee b, \neg m \vee k, \neg m \vee \neg k\} .
\end{aligned}$$

One can derive the empty clause from $DB' + \neg C_1'$ and from $DB' + \neg C_2'$, and hence conclude that $\neg a$ can be assumed.

The algorithm involves deriving an empty clause from a set of first-order formulae. For function free databases, the problem is decidable but exponential in time. For this reason, it is unlikely that the algorithm can be put to practical use.

As with the CWA and the GCWA, the EGCWA is nonmonotonic. Like the GCWA, one problem associated with the EGCWA is that the operator $\vee$ is interpreted as exclusive or. The extension of the GCWA to allow negative clauses containing more than one atom means that $\vee$ is treated as exclusive or in more circumstances without causing inconsistency. For example, if $p \vee q$ is derivable but neither p nor q is individually derivable, one can assume $\neg(p \wedge q)$ provided that it does not lead to inconsistency (i.e. $p \wedge q$ does not violate the database). Let DB be the database $\{a \vee b, b \vee c, c \vee a\}$. Under the EGCWA, one cannot assume $\neg(a \wedge b), \neg(b \wedge c)$, or $\neg(c \wedge a)$ because DB can derive the clause $(a \wedge b) \vee (b \wedge c) \vee (c \wedge a)$. Intuitively, to make DB valid, at least two of the atoms must be true at the same time. However, although all three of them might be simultaneously true, this cannot be proved and the assumption $\neg(a \wedge b \wedge c)$ can be made without inconsistency.

Another problem is whether the EGCWA can be used to handle general laws. Yahya and Henschen tackle the problem of general laws involving query evaluation of indefinite databases. Consider a database that contains some non-Horn clauses as general laws, e.g.

$$L_1 \wedge L_2 \wedge \cdots \wedge L_n \rightarrow M_1 \vee M_2 ,$$

which is equivalent to

$$\neg L_1 \vee \neg L_2 \vee \cdots \vee \neg L_n \vee M_1 \vee M_2 .$$

In order to answer the query M_2, one can rewrite the above rule to produce

$$L_1 \wedge L_2 \wedge \cdots \wedge L_n \wedge \neg M_1 \rightarrow M_2$$

in which the evaluation of a negative atom M_1 is required.

However, in many cases, the EGCWA alone is inadequate to solve the problems of the evaluation of general laws. For example [36], let DB be the set

$$\neg a(x) \vee p(x) \vee q(x) ,$$
$$\neg b(y) \vee p(y) \vee q(y)$$

and suppose that one uses the first clause to define p and the second to define q. A deadlock will be reached when one tries to assume $\neg q(x)$ in the first clause and $\neg p(y)$ in the second. Therefore, database users must decide what kinds of general laws must not be treated as derivation laws for defining new information, but rather as integrity constraints. Thus, the EGCWA can only solve a specific problem in the query evaluation process in the whole non-Horn database system. Yahya and Henschen give an algorithm to decide which class of general laws can be treated as data derivation rules and which must be treated as the integrity constraints. That algorithm may help users to avoid the deadlock situation described above. However, the nature of the non-Horn database has inevitably limited the application of the EGCWA to general non-Horn database systems.

Naqvi [30] has proposed extending the GCWA to allow non-ground (as well as non-atomic) negative clauses. However, in the absence of functions, any non-ground database can be represented by an equivalent ground database. Thus, Naqvi's proposal is no more general than Yahya and Henschen's, though it does allow the database to be represented more concisely and naturally.

6. A further extension of minimal model semantics

Another extension of the GCWA, or more precisely an extension of the minimal model semantics, has been proposed by several researchers [1, 12, 31]. This concerns the interpretation to be placed on a deductive rule clause in which negative literals appear in the premise part. Consider a rule of the form

$$A_1 \wedge A_2 \wedge \cdots \wedge A_m \wedge \neg B_1 \wedge \neg B_2 \wedge \cdots \wedge \neg B_n \rightarrow C_1 \vee C_2 \vee \cdots \vee C_p$$

where $m,n \geq 0$, $p \geq 1$, and A_i, B_j and C_k denote atoms. Normally, this can be transformed into an equivalent formula of the form

$$A_1 \wedge A_2 \wedge \cdots \wedge A_m \rightarrow B_1 \vee B_2 \vee \cdots \vee B_n \vee C_1 \vee C_2 \vee \cdots \vee C_p .$$

The equivalence of the above two clauses is based on the interference system of first-order logic. However, as a result of the non-logical treatment of negation, the above two clauses have different meanings. The following example illustrates the difference.

Example 6.1 Assume that every third year student must learn Latin unless they are explicitly exempt from doing so, Jean is a third year student, and Paul learns German. The database can be expressed by the following clauses:

$3rd(x) \wedge \neg exempt(x) \rightarrow le(x,lat)$
$3rd(jean)$
$le(paul,ger)$

Under the GCWA, the database is equivalent to:

$\neg 3rd(x) \vee exempt(x) \vee le(x,lat)$ (*equivalent to* $3rd(x) \rightarrow exempt(x) \vee le(x,lat)$)
$3rd(jean)$
$le(paul,ger)$

The database has two minimal models:

$\{3rd(jean), exempt(jean), le(paul,ger)\}$ and

$\{3rd(jean), le(jean,lat), le(paul,ger)\}$

Since Jean is not explicitly exempt from learning Latin, she must do so. Hence, the first minimal model is an incorrect representation of the database because it states that Jean is exempt from learning Latin. Only the second minimal model reflects the meaning of the database correctly, it expresses that Jean learns Latin. Przymusinski [31] points out that for a database containing clauses with negated atoms in the premise part, only some minimal models (perhaps none of them) reflect the correct meaning of the database. Such minimal models are called *perfect models* and the meaning of a perfect model, the *perfect model semantics*, correctly describes the intended meaning of the database.

The perfect model semantics employs a special *negation as failure* rule to handle the negated atoms in the premise part when the CWA rule is used. A priority relation is defined over the predicates such that the negated predicates in the premise part have a higher

priority than the consequent predicates. Thus, generally speaking, the negated predicates in the premise part should be minimised before the consequent predicates. The perfect model semantics is therefore equivalent to the minimal model semantics if no negated atoms appear in the premise part of any clause. The perfect model semantics is defined according to the priority relation over the predicates.

7. Subimplication

Bossu and Siegel's approach [3, 4] to query answering in the presence of negative and indefinite information is based on the contents of the intersection of the set of minimal models of the database. The answer to a query can be 'yes', 'no', or 'indefinite' according to whether the query is true in, respectively, all, none, or some of the minimal models of the database. Similarly, when a list of answers is to be given in response to a query, the list is divided into two sections; the *definite* section and the *indefinite* section.

Bossu and Siegel have developed a system of non-monotonic reasoning called *subimplication*. This form of implication has a decidable algorithm provided that certain syntactic restrictions are put on both database clauses and queries. In order to describe these restrictions, some additional terminology is required.

A *positive* formula is a formula semantically equivalent to a formula which contains neither the symbol $\neg$ nor the symbol $\rightarrow$. A *negative* formula is the negation of a positive formula. Note that, by this definition, there are some formulae which are neither positive nor negative, for example $\forall x(p(x) \rightarrow q(x))$. A *positive* clause has no negative literals. A *negative* clause has no positive literals.

Bossu and Siegel restrict the contents of the database to *groundable* clauses. A clause is *groundable* if every variable which appears in a positive literal also appears in a negative literal. Thus a groundable clause may be

(i) a ground clause,
(ii) a negative clause or
(iii) a clause semantically equivalent to a sentence of the form:

$$\forall x_1 \cdots \forall x_p(l_1 \wedge \cdots \wedge l_n \rightarrow l'_1 \vee \cdots \vee l'_m)$$

where the l_i and the l'_j are positive literals and every variable x_k appears in at least one of the l_i.

(The reader should note that Bossu and Siegel's definition of a groundable clause [4] is wrong.)

Although the database is restricted to groundable clauses, queries can be expressed in a more general form called *groundable formulae*. Let $\forall \boldsymbol{X}$ denote a (possibly empty) sequence of universally quantified variables. A clause $c = l_1 \ldots l_n$ can be identified with the formula $\forall \boldsymbol{X}(l_1 \vee \cdots \vee l_n)$ (the variables of c are quantified in $\forall \boldsymbol{X}$). The empty clause and the empty disjunction of formulae will be identified with the formula $\mathbf{0}$ (= false). The empty conjunction of formulae will be identified with the formula $\mathbf{1}$(= true).

The set, $\boldsymbol{E}$, of groundable formulae is the smallest set of sentences such that

(1) the formula $\mathbf{1}$ is in $\boldsymbol{E}$.
(2) $\forall \boldsymbol{X}(c \vee \neg g)$ is in $\boldsymbol{E}$ if
  (i) c is a disjunction of negative literals,
  (ii) g is a conjunction of elements of $\boldsymbol{E}$,
  (iii) every variable quantified in $\forall \boldsymbol{X}$ appears in at least one literal of c.

In the most general case, a groundable formula q has the form

$$q = \forall x_1 \ldots \forall x_p(\neg l_1 \vee \cdots \vee \neg l_n \vee \neg(g_1 \wedge \cdots \wedge g_m)) .$$

The $\neg l_i$ are negative literals and the g_j are elements of $\boldsymbol{E}$.

Example 7.1 *Does every student learn at least one language?*

$$\forall x(student(x) \rightarrow \exists y(language(y) \wedge learns(x, y))) .$$

In order to write this as a groundable formula, we let

$$g = \forall y(\neg language(y) \vee \neg learn(x, y)) .$$

The groundable formula can then be written as

$$\forall x(\neg student(x) \vee \neg g) .$$

Bossu and Siegel introduce the notion of *subimplication* denoted $]=$, for use with minimal Herbrand models. Let DB be a consistent set of groundable clauses and q be a groundable formula. $DB \,]= q$ iff every minimal Herbrand model of DB is also a model of q. They define $DB \,]\neq q$ to mean that there exists a minimal Herbrand model of DB which does not satisfy q. Subimplication has the following properties:

(1) If $DB \models q$, then $DB \,]= q$.
(2) if q is a positive formula, then $DB \,]= q$ iff $DB \models q$.
(3) $DB \,]= q \rightarrow DB \,]\neq \neg q$.

Note that it is possible for both $DB \,]\neq q$ and $DB \,]\neq \neg q$ to hold simultaneously. This will be the case if there exists a minimal model of DB that satisfies q and a minimal model of DB that does not satisfy q.

With regard to the queries in our standard example, we note the following points. All clauses in the database are groundable and all three queries can be converted into groundable formulae (examples of similar conversions can be found in Bossu and Siegel's papers). We note that the first query is *not* a groundable clause. On the other hand, the clause $s(x) \wedge \neg le(x, gre)$ is groundable. The first query yields the same answers as one would expect from classical logic. This is because the literal $\neg le(paul, gre)$ is in the intersection of the four minimal Herbrand models of the database. Evaluation of the second query yields the answers *toto* and *jean*. Although $\neg le(jean, eng)$ is not explicitly stated in the database, it can be deduced because $le(jean, eng)$ does not appear in any minimal model. In answering the third query, subimplication will produce two lists of answers. The first list comprises the pairs *paul/eng*, *paul/ger*, *toto/lat*, and *jean/lat*. These assignments cause the query formula to be true in all of the minimal Herbrand models of the database. The second list comprises the pairs *toto/gre* and *jean/gre*. These assignments cause the query formula to be true in some minimal Herbrand models and false in others.

In conclusion, Bossu and Siegel's notion of subimplication is coherent. The syntactic restrictions on the database and query clauses ensure that the evaluation is always computable. With regard to the complexity, in the worst case, the running time of the implication algorithm can be an exponential function of the size of the database. On the other hand, the performance of the algorithm depends on the extent to which the database is non-Horn. The performance deteriorates as the number of non-Horn clauses increases.

8. Circumscription

Circumscription is a non-monotonic formalism due to McCarthy [24, 25]. It was designed to allow one to 'jump to certain conclusions' from given facts even though the conclusions are not strict logical consequences of the known facts. For example, it allows one to conclude that all entities which satisfy some predicate P from a given collection of facts and rules are the only entities which satisfy P. Alternatively, suppose one wants an entity to satisfy P unless it is prevented from doing so by condition Q. Circumscription may be used to conjecture that the only thing that can prevent an entity satisfying P is condition Q holding.

Circumscription can be used with the standard rules of inference of first-order logic. The addition of circumscription to first-order logic produces a non-monotonic system. Enriching the collection of facts and rules means more entities may satisfy P, thus entities which were previously conjectured *not* to satisfy P are no longer applicable. The situation is similar to adding negation as failure to Horn clause logic with the resolution rule of inference.

Let DB be the conjunction of all clauses in a database, P a predicate symbol, and A a predicate expression of the same arity as P. Denote by $DB(A)$ the result of replacing all occurrences of P in DB by A.

Definition 8.1 The *circumscription* of P in DB is the sentence schema

$$(DB(A) \wedge (\forall x)(A(x) \rightarrow P(x))) \rightarrow (\forall x)(P(x) \rightarrow A(x)) .$$

This says: if the database produced by substituting A for P is consistent, and the entities satisfying A are a subset of the entities satisfying P, then the entities satisfying P are a subset of the entitites satisfying A. Essentially, the claim is being made that P is in some sense minimal, i.e. the only entities satisfying P are those that have to from DB. The extension of P is as small as possible while retaining consistency.

Example 8.2 Let DB be

$$s(toto) \wedge s(jean) \wedge s(paul) ,$$

DB asserts *toto*, *jean*, and *paul* are students. Suppose one wants *toto*, *jean*, and *paul* to be the only students. Circumscribing the predicate s in DB gives the schema

$$(A(toto) \wedge A(jean) \wedge A(paul) \wedge (\forall x)(A(x) \rightarrow s(x))) \rightarrow (\forall x)(s(x) \rightarrow A(x)) .$$

Make the definition

$$A(x) \equiv (x = toto \vee x = jean \vee x = paul) .$$

From this definition, simplification by standard logical techniques produces the formula

$$(\forall x)(s(x) \rightarrow (x = toto \vee x = jean \vee x = paul))$$

which asserts that *toto*, *jean*, and *paul* are the only students in existence. We could have circumscribed s in the entire example database and obtained the same result because s does not appear anywhere else.

Circumscription may be used with arbitrary first-order formulae. The following example illustrates circumscription with respect to indefinite data.

Example 8.3 Let DB be

$$le(toto,lat) \vee le(toto,gre)$$

asserting that *toto* learns either *latin* or *greek*. Suppose one wants to conjecture that if something learns *latin*, then it does not learn anything else. Circumscribing *le* in DB gives

$$((A(toto,lat) \vee A(toto,gre)) \wedge (\forall x,y)(A(x,y) \rightarrow le(x,y))) \rightarrow$$
$$(\forall x)(le(x,y) \rightarrow A(x,y)) .$$

Make the definition

$$A(x,y) \equiv le(x,lat)$$

and substitution followed by simplification gives

$$(\forall x)(le(x,lat) \rightarrow (\forall y)(le(x,y) \rightarrow y = lat)) .$$

McCarthy [24] proves a soundness result for circumscription. For this purpose he introduces the notion of a model minimal with respect to a predicate. Let I and J be models of DB and P a predicate symbol occurring in DB. I is a *submodel of* J *in* P if

(i) I and J have the same domain,

(ii) for any predicate symbol Q not equal to P, the extension of Q in I is equal to the extension of Q in J,

(iii) the extension of P in I is a subset of the extension of P in J.

A model I of DB is *minimal in* P if any submodel of I in P is equal to I. McCarthy shows: any instance of the circumscription of P in DB is true in all models of DB minimal in P.

However, minimal models do not always exist and such a situation can lead to inconsistency. Etherington et al. [8] investigate the class of formulae for which circumscription cannot introduce inconsistencies. They show that consistency is preserved when *universal theories* [4] are circumscribed. A first-order theory is universal if each of its formulae is equivalent to a formula of the form $(\forall \bar{x})A$ where $\bar{x}$ is a tuple of variables and A is a quantifier free formula. Lifschitz [17] extends this class to *almost universal* theories. A first-order theory is almost universal with respect to a predicate P if each of its formulae is equivalent to a formula of the form $(\forall \bar{x})A$ where A contains no positive occurrences of the predicate P within the scope of a quantifier.

The sentence schema produced by circumscribing P in DB, namely

$$(DB(A) \wedge (\forall x)(A(x) \rightarrow P(x))) \rightarrow (\forall x)(P(x) \rightarrow A(x))$$

is second order. A denotes an arbitrary formula so the schema is implicitly universally quantified over a second-order variable. Lifschitz [16] investigates the problem of computing circumscription and shows that in some cases it is possible to replace circumscription by an equivalent first-order formula. However, the problem of evaluating the first-order formula remains.

There is a clear connection between circumscription and applying the CWA to a predicate.

Both have the effect of minimising the set of objects which satisfy the predicate. Lifschitz [15] proves that if the CWA is consistent with some database *DB*, then the result of circumscribing all predicates in *DB* and applying the CWA to *DB* are equivalent theories. Thus, the CWA is a special case of circumscription.

Reiter [33] discusses the connections between circumscription and Clark's predicate completion [5]. Let *DB* be a database Horn with respect to the predicate *P*. *DB* is said to be Horn with respect to *P* if every clause of *DB* contains at most one positive occurrence of *P* (assuming *DB* is in clausal form). The circumscription of *P* in *DB* logically implies the completion axiom for *P*. Thus, circumscription subsumes predicate completion for a wide class of first-order theories. Reiter gives the most accessible introduction to circumscription we have found in the literature and discusses one of its main problems. In order to reason with circumscription, it is necessary to determine a suitable formula to substitute for the predicate parameter *A*. This is, to quote Reiter [33], "the central computational problem with circumscription", and he likens it to the problem of determining suitable predicates on which to perform induction in, say, number theory.

So far, this section has restricted attention to a particular form of circumscription called predicate circumscription [24]. We concluse with a brief description of the other forms of circumscription.

- *Domain circumscription*: this is nothing more than the domain closure assumption and is subsumed under predicate circumscription.
- *Protected circumscription*: suppose a database contains the two clauses

  $3rd(jean) \qquad 3rd(toto)$

  and the only other object in the domain is *paul*. Under the CWA, one can deduce $\neg 3rd(paul)$. If, for some reason, it is not desirable to automatically deduce $\neg 3rd(paul)$ from a failure to prove $3rd(paul)$, the constant *paul* can be *protected* in the relation $3rd$. Thus, although any attempt to satisfy $3rd(paul)$ will fail, it will be noted that *paul* is protected in $3rd$ preventing $\neg 3rd(paul)$ from being deduced. If the domain was extended to include the constant *peter* (not protected in $3rd$), it would still not be possible to deduce $\neg 3rd(paul)$ but $\neg 3rd(peter)$ would be deducible (because $3rd(peter)$ fails and *peter* is not protected in $3rd$).

  Minker and Perlis [28] extend predicate circumscription to handle protected data, i.e. the protected data for a predicate is exempt from the circumscription of that predicate. They provide a model theory and prove the consistency of protected circumscription. Minker and Perlis investigate the evaluation of protected circumscription, [29] but only in the restricted case of Horn databases.
- *Formula circumscription*: in predicate circumscription, a particular predicate (or sequence of predicates) is circumscribed to minimise its extension. Formula circumscription [26] extends this notion to allow arbitrary formulae to be circumscribed. Thus, the set of objects which satisfy some general formula is minimised in formula circumscription. Protected circumscription is subsumed under formula circumscription.
- *Prioritised circumscription*: let *DB* be a database and *P* and *Q* be two predicate symbols. The circumscription of *P* in *DB* causes *P*'s extension to be minimised. *Q*'s extension is similarly minimised. However, the simultaneous circumscription of *P* and *Q* in *DB* may cause conflict, i.e. combining the minimisations of *P* and *Q* may lead to inconsistency. Prioritised circumscription [26] overcomes this problem by ordering the predicates. Thus, if the circumscription of two distinct predicates causes conflict, the one with the highest priority is preferentially minimised. Prioritised circumscription generalises formula circumscription.

9. Negation as inconsistency

Gabbay and Sergot [10] have proposed a basis for logic programming which they term *negation as inconsistency*. It is a more expressive formalism than Prolog. It uses a convention for negative data which is closer to classical logic than negation as failure. This formalism is also applicable to deductive databases.

For simplicity, only the propositional (variable free) case will be described. A *goal* has the form

A	where A is an atomic formula,
$\neg A$	where A is a goal,
$A_1 \wedge A_2$	where A_1 and A_2 are goals.

A *clause* has the form

A	where A is an atomic formula,
$A \leftarrow B$	where A is atomic and B is a goal.

A *database* is a pair of the form (Π, Σ) where Π is a set of clauses and Σ is a set of goals. The clauses in Π are just the facts and rules of the database, as in Prolog. Σ contains goals which are *not* to succeed. This is logically equivalent to adding the negation of each goal in Σ to Π. Intuitively, Π is the set of positive information and Σ is the set of negative information.

The convention for negative data, negation as inconsistency, from which the formalism takes its name is a stronger fragment of classical logic than negation as failure. A database (Π, Σ) is defined to be *inconsistent* if some goal A in Σ succeeds from the database Π. Therefore, a piece of data can only be negated if adding it to the positive part of the database causes the database to become inconsistent.

Query evaluation under negation as inconsistency is carried out as follows (assuming evaluation with respect to the database (Π, Σ)):

(a) a goal of the form A (where A is an atomic formula) succeeds if either A is in Π or $A \leftarrow B$ is in Π and B succeed;

(b) a goal of the form $A_1 \wedge A_2$ (where A_1 and A_2 are goals) succeeds if A_1 and A_2 succeeds;

(c) a goal of the form $\neg A$ (where A is a goal) succeeds if the database produced by adding A to Π, denoted $(\Pi + A, \Sigma)$, is inconsistent.

Note that any goal of the form $\neg A$ can be written as $\neg(\wedge B_i \wedge \wedge \neg C_j)$, where the B_i are atomic formulae and the C_j are goals. In practice, the B_i would be added to Π and the C_j to Σ. Evaluation rules (a) and (b) are as for SLD-resolution. The reasoning behind rule (c) is as follows:

> The database (Π, Σ) is assumed to be consistent. Add the goal A to Π. Show that some goal B in Σ succeeds. Since all goals in Σ are required *not* to succeed, by definition, the database $(\Pi + A, \Sigma)$ is inconsistent. This inconsistency must have arisen from adding the goal A to Π. Therefore, we deduce $\neg A$.

Query evaluation under negation as inconsistency is sound with respect to classical logic and complete in the following sense:

Let (Π, Σ) be a database and A a goal. Let $C(\Pi, \Sigma)$ denote the following formula of classical logic:

$$C(\Pi, \Sigma) \equiv \bigwedge_{x \in \Pi} x \wedge \bigwedge_{y \in \Sigma} \neg y$$

Then the following holds:

if $C(\Pi, \Sigma) \vdash A$, then $\neg\neg A$ succeeds from (Π, Σ).

When extending negation as inconsistency to handle variables, the query evaluation process is largely as described above. The major difference is the extra burden of keeping track of the various variable substitutions and ensuring that cumulative substitutions on variables are consistent. Query evaluation under negation as inconsistency remains sound when variables are admitted to clauses and goals. No completeness result is given, though any such result could be no stronger than that for the propositional case.

It is clearly not possible to implement negation as inconsistency as efficiently as SLD-resolution. Theorem proving and substitution management for evaluation rules (a) and (b) are as for SLD-resolution. In the absence of negative goals, SLD-resolution is all that is required. The inefficiency is introduced by rule (c). The evaluation of a negative goal involves adding new facts to the database, some to the positive part and some to the negative part. Negative goals may be conjoined and nested so the database will be continually changing during query evaluation. The database will effectively operate as a stack, information being pushed when a negative goal is to be evaluated and popped when the goal has been satisfied (or evaluation has failed).

The burden of managing substitutions is also increased. Consider the goal $\neg(P(x) \wedge \neg Q(x))$. Evaluation rule (c) says add $P(x)$ to the positive part of the database and $Q(x)$ to the negative part, then show the database is inconsistent. $P(x)$ and $Q(x)$ are in different parts of the database but share a common variable. Cumulative substitutions made on this variable, via $P(x)$ or $Q(x)$, must be consistent in the same way as the occurrence of the same variable in more than one subgoal.

A database is defined to be inconsistent when one of the goals in the negative part of the database can be evaluated successfully. When a database is to be shown inconsistent, which goal in the negative part should be chosen for evaluation? An implementation of negation as inconsistency would require some strategy for selecting such a goal—just as SLD-resolution requires a computation rule. The ability of this selection rule to choose an appropriate goal from the negative part of the database could significantly affect efficiency. The success of the Prolog computation rule, for example, is largely dependent on how the programmer orders subgoals in a rule. It would also be necessary to know which goals in the negative part had been attempted for backtracking purposes.

Negation as inconsistency is an expressive formalism. Both negative and indefinite data can be stored. To store the information $\neg A$ merely add A to the negative part of the database. To store the indefinite data $A \vee B$, add $\neg A \wedge \neg B$ to the negative part of the database (because $A \vee B$ is equivalent to $\neg(\neg A \wedge \neg B)$). Negative and indefinite queries can be made. Negative queries are evaluated directly, by evaluation rule (c) and indefinite queries are transformed as above to give a negative query and then evaluated.

That negation as inconsistency is weaker than classical negation is seen by a simple example [10]. Consider the database with the information

$$A \vee B \qquad A \rightarrow C \qquad B \rightarrow C$$

Under classical negation, C is clearly a logical consequence of this database. Querying the database with C under negation as inconsistency would fail (note that failure means *unable to prove*, not *is false* as is the case with negation as failure). The database is represented

Π	Σ
$C \leftarrow A$ $C \leftarrow B$	$\neg A \wedge \neg B$

The query C matches with the head of the clause $C \leftarrow A$ giving the subgoal A. No further evaluation can be done; none of the evaluation rules apply. The same situation arises if C is matched with the clause $C \leftarrow B$. However, the goal $\neg\neg C$ would succeed under negation as inconsistency. It is a negative goal so $\neg C$ is added to Π, which in practice means add C to Σ. The goal $\neg A \wedge \neg B$ in Σ will then succeed from this extended database giving the required inconsistency and justifying $\neg\neg C$.

Consider the example database. It can be represented in the current formalism by

Π	Σ
$l(x) \leftarrow lm(x)$ $l(x) \leftarrow ld(x)$ $s(toto)$ $s(jean)$ $s(paul)$ $3rd(toto)$ $3rd(jean)$ $lm(eng)$ $lm(ger)$ $ld(lat)$ $ld(gre)$ $le(paul,eng)$ $le(paul,ger)$	$3rd(x) \wedge \neg le(x,lat) \wedge \neg le(x,gre)$ $le(x,gre) \wedge le(x,ger)$ $le(toto,eng)$

The standard queries are now evaluated with respect to this database. The way information is added to and removed from the database during query evaluation is described via an auxiliary dynamic database. π and σ denote the information added to the positive and negative parts of the database respectively.

(1) Query 1 is expressed as

$$\neg le(u,gre)$$

This is a negative query. Rule (c) is used giving the extended database.

π	σ
$le(u,gre)$	

Choose $le(x,gre) \wedge le(x,ger)$ from Σ to show the inconsistency. By rule (b), the goals $le(x,gre)$ and $le(x,ger)$ are considered individually.

(i) $le(x,gre)$ is satisfied immediately from π by rule (a)–u and x share.

(ii) $le(x,ger)$ is satisfied immediately from Π by rule (a)–x and u are instantiated to *paul*.

The goal $\neg le(u,gre)$ succeeds with u instantiated to *paul*.

(2) Query 2 is expressed as

$$s(u) \wedge \neg(\neg le(u,lat) \wedge \neg le(u,gre)) \wedge \neg le(u,eng)$$

By rule (b), each of the three conjuncts is evaluated individually.

(i) $s(u)$ is satisfied immediately from Π by rule (a)–u is instantiated to *toto*.

(ii) The instantiation from (i) means the goal becomes

$$\neg(\neg le(toto,lat) \wedge \neg le(toto,gre))$$

By rule (c) the database is extended to

π	σ
	$le(toto,lat)$ $le(toto,gre)$

Choose $3rd(x) \wedge \neg le(x,lat) \wedge \neg le(x,gre)$ from Σ to show the database is inconsistent. Three subgoals are produced by rule (b).

(ii.i) $3rd(x)$ is immediately satisfied from Π by rule (a)–x is instantiated to *toto*.

(ii.ii) The instantiation from (ii.i) means the goal becomes ¬le(toto,lat). By rule (c) the database is extended to

π	σ
$le(toto,lat)$	$le(toto,lat)$ $le(toto,gre)$

Choose $le(toto,lat)$ from σ to show database is inconsistent. It is immediately satisfied from π by rule (a).

(ii.iii) $\neg le(toto,gre)$ satisfied as for (ii.ii).

(iii) The instantiation from (i) means the goal becomes

$$\neg le(toto,eng)$$

By rule (c), extend the database to

π	σ
$le(toto,eng)$	

The goal $le(toto,eng)$ from Σ is immediately satisfied from π by rule (a) establishing the inconsistency.

The goal $s(u) \wedge \neg(\neg le(u,lat) \wedge \neg le(u,gre)) \wedge \neg le(u,eng)$ succeeds with u instantiated to *toto*.

(3) Query 3 is expressed as

$$s(u) \wedge l(v) \wedge \neg(\neg le(u,v) \wedge \neg le(u,gre))$$

Consider the first two conjuncts $s(u) \wedge l(v)$. s is satisfied by three constants directly from Π by rule (a)–*toto*, *jean* and *paul*. l is satisfied by four constants via the rules $l(x) \leftarrow lm(x)$ and $l(x) \leftarrow ld(x)$ and the facts concerning *lm* and *ld*–*eng*, *ger*, *lat* and *gre*. Therefore, using SLD-resolution and standard Prolog backtracking, the variables u and v would be successively instantiated to the constants

u	*toto*	*toto*	*toto*	*toto*	*jean*	*jean*	*jean*	*jean*	*paul*	*paul*	*paul*	*paul*
v	*eng*	*ger*	*lat*	*gre*	*eng*	*ger*	*lat*	*gre*	*eng*	*ger*	*lat*	*gre*

Query evaluation consists of generating these pairs and testing the third conjunct with the

various instantiations. The pairs for which the query will succeed under negation as inconsistency are the same as for classical negation—*paul/eng*, *paul/ger*, *toto/lat* and *jean/lat*.

(i) For the pair *paul/eng*, the goal is

$$\neg(\neg le(paul,eng) \wedge \neg le(paul,gre))$$

By rule (c) the database is extended to

π	σ
	$le(paul,eng)$ $le(paul,gre)$

Choose $le(paul,eng)$ from σ to show the inconsistency. This is immediately satisfied from Π by rule (a).

Satisfying the goal for the pair *paul/ger* follows the same pattern.

(ii) For the pair *toto/lat*, the goal is

$$\neg(\neg le(toto,lat) \wedge \neg le(toto,gre))$$

This is the same as subgoal (ii) of query (2). Evaluation follows exactly the same path and the goal is satisfied. Evaluation with the pair *jean/lat* follows a similar pattern.

10. Discussion

In order to produce a database system that is genuinely more powerful than a relational database system, one must either relax the CWA or represent indefinite information. If neither of these is done, Vardi [35] has shown it is possible to simulate a logic database by a relational database.

Minker's GCWA [27] strictly generalises the CWA. It allows the construction of databases in which one can explicitly store and make queries about negative and indefinite information. The minimal model semantics on which it is based is a natural extension of the minimal model semantics for Horn databases and is relatively easy to grasp. It remains to be shown that efficient implementations of the GCWA exist. The algorithms of Grant and Minker [13] and Gelfond and Przymusinska [12] are both exponential. Strictly speaking, the GCWA allows one to infer certain negative atomic formulae, unit negative clauses, by default. Yahya and Henschen's contribution [36], the EGCWA, is to extend the GCWA to allow one to directly infer non-unit negative clauses by default. Their algorithm, as with the others for the GCWA, is exponential. With respect to implementation, the most promising work seems to be that of Bossu and Siegel. Their notion of subimplication is nothing more than an implementation of the GCWA under certain conditions. They place syntactic restrictions on the kinds of formulae which can be used to represent database clauses and queries. These restrictions allow query evaluation to be carried out more efficiently. On the negative side, their query evaluation algorithm becomes inefficient as the amount of indefiniteness in the database increases. Furthermore, the syntactic relations placed on queries are likely to be confusing to end users.

Circumscription, first suggested by McCarthy [24] is a powerful and flexible formalism. Its various forms subsume many of the other systems described here. A major problem with

circumscription is that it is difficult to obtain an intuitive grasp of how it can be used in the practical setting of databases. The descriptions of circumscription in the literature do not help in this respect; they concentrate on the formal aspects and keep intuitive motivation to a minimum. Little work has been carried out on the problems of evaluation under circumscription. Recently, the assumptions under which the prefect models approach, circumscription and default logic are equivalent have been investigated. For example, prefect models have a semantics that is equivalent to a form of prioritised circumscription [1, 2].

Gabbay and Sergot's negation as inconsistency [10] is set apart from the other formalisms in that it is monotonic. It is appealing because of its purity, but the lack of any facility for inferring by default (which would make it non-monotonic) is a major hurdle in its practical application to databases. If the formalism could be combined with some kind of default reasoning, it would be a promising candidate. The evaluation algorithm and the ramifications of the completeness result would need further investigation.

In conclusion, a variety of formalisms have been discussed which, to various degrees, are applicable to representing negative and indefinite information in databases. The major problem seems to be the complexity of query evaluation in the presence of indefinite data. It is notable that the only formalism discussed here for which an efficient implementation does exist is that of Lloyd and Topor [20]; a system which neither relaxes the CWA nor allows the representation of indefinite data.

One of the assumptions made in this survey is that the data in the database, though indefinite, is correct. Consequently, we have not included any discussion on the interesting results (obtained by, for example, Levesque [14]) on treating the data in the database as beliefs about the real world. Another omission from this survey is the important work done on incomplete information databases by Lipski [18]. Lipski's work is mainly of relevance to databases conforming to the relational model [6].

Acknowledgment

We would like to thank the referees for their constructive comments which have helped to remove some inaccuracies and greatly improve the presentation of this paper.

References

[1] N. Bidoit and R. Hull, Positivism vs. minimalism in deductive databases, *Proc. 5th Conference on Principles of Database Systems*, Cambridge, MA (1986) 123–132.

[2] N. Bidoit and C. Froidevaux, Minimalism subsumes default logic and circumscription in stratified logic programming, 350, L.R.I., Université de Paris-Sud, 1987.

[3] G. Bossu and P. Siegel, Nonmonotonic reasoning and databases, in: H. Gallaire, J. Minker and J. Nicolas (Eds.), *Advances in Database Theory, Vol. 2* (Plenum Press, New York, 1984) 239–284.

[4] G. Bossu and P. Siegel, Saturation, nonmonotonic reasoning and the closed-world assumption, *Artificial Intelligence 25* (1985) 13–63.

[5] K.L. Clark, Negations as failure, in: H. Gallaire and J. Minker (Eds.), *Logic and Data Bases* (Plenum Press, New York, 1978) 293–322.

[6] E.F. Codd, A relational model of data for large shared data banks, *Comm. ACM 13* (1970) 377–387

[7] M.H. van Emden and R.A. Kowalski, The semantics of predicate logic as a programming language, *J. ACM 23* (4) (1976) 733–742.

[8] D.W. Etherington, R.E. Mercer and R. Reiter, On the adequacy of predicate circumscription for closed-world reasoning, *Comput. Intelligence 1* (1985) 11–15.

[9] R.A. Frost, *Introduction to Knowledge Base Systems* (Collins, 1986).

[10] D.M. Gabbay and M.J. Sergot, Negation as inconsistency, I, *J. Logic Programming 3* (1) (1986) 1–35.

[11] H. Gallaire, J. Minker and J.-M. Nicolas, Logic and databases: a deductive approach, *ACM Comput. Surveys 16* (2) (1984) 153–185.
[12] M. Gelfond and H. Przymusinska, Negation as failure: Careful Closure Procedure, *Artificial Intelligence 30* (1986) 273–287.
[13] J. Grant and J. Minker, Answering queries in indefinite databases and the null value problem, *Adv. Comput. Res. 3* (1986) 247–267.
[14] H.J. Levesque, Making believers out of computers, *Artificial Intelligence 30* (1986) 81–108.
[15] V. Lifschitz, Closed world databases and circumscription, *Artificial Intelligence 27* (1985) 229–235.
[16] V. Lifschitz, Computing circumscription, *Proc. 9th International Joint Conference on Artificial Intelligence, Vol. 1* (1985).
[17] V. Lifschitz, On the satisfiability of circumscription, *Artificial Intelligence 28* (1986) 17–27.
[18] W. Lipski, Incomplete information in databases, *Théorie Des Bases De Donnés*, Benodet, France, (1984) 77–90.
[19] J.W. Lloyd, *Foundation of Logic Programming* (Springer, Berlin, 1984).
[20] J.W. Lloyd and R.W. Topor, Making Prolog more expensive, *J. Logic Programming 1* (3) (1984) 225–240.
[21] J.W. Lloyd and R.W. Topor, A basis for deductive database systems, *J. Logic Programming 2* (2) (1985) 93–109.
[22] J.W. Lloyd and R.W. Topor, A basis for deductive database systems II, *J. Logic Programming 3* (1) (1986) 55–67.
[23] D. Loveland, *Automated Theorem Proving: A Logical Basis* (North-Holland, New York, 1978).
[24] J. McCarthy, Circumscription- a form of non-monotonic reasoning, *Artificial Intelligence 13* (1980) 27–39.
[25] J. McCarthy, Circumscription and other non-monotononic formalisms, *Artificial Intelligence 13* (1980) 171–172.
[26] J. McCarthy, Applications of circumscription to formalizing common-sense knowledge, *Artificial Intelligence 28* (1986) 89–116.
[27] J. Minker, On indefinite databases and the closed world assumption, *Proc. 6th Conference on Automated Deduction*, Lecture Notes in Computer Science *138* (Springer, Berlin, 1982) 292–308.
[28] J. Minker and D. Perlis, Applications of protected circumscription, *Proc. 7th International Conference on Automated Deduction* (Springer, Berlin, 1984).
[29] J. Minker and D. Perlis, Computing protected circumscription, *J. Logic Programming 2* (1985) 235–249.
[30] S.A. Naqvi, Some extensions to the closed-world assumption in databases, *Proc. International Conference on Database Theory*, Rome, Italy (1986).
[31] T.C. Przymusinski, On the semantics of stratified deductive databases, in: J. Minker (Ed.), *Foundations of Deductive Databases and Logic Programming* (1986).
[32] R. Reiter, On closed world data bases, in: H. Gallaire and J. Minker (Eds.), *Logic and Data Bases* (Plenum Press, New York, 1978) 55–76.
[33] R. Reiter, Circumscription implies predicate completion (Sometimes), *Proc. International Joint Conference on Artificial Intelligence* (1982) 418–420.
[34] J.A. Robinson, A machine-oriented logic based on the resolution principle, *J. ACM 12* (1) (1965) 23–41.
[35] M.Y. Vardi, Querying logical databases, *Proc. 4th ACM Symposium on Principles of Database Systems* (1985) 57–65.
[36] A. Yahya and L.J. Henschen, Deduction in non-Horn databases, *J. Automated Reasoning 1* (1985) 141–160.

46. M.H.WILLIAMS and K.A.NICHOLSON, An approach to handling incomplete information in databases, The Computer Journal, vol. 31, no. 2, 133-140 (1988).

An Approach to Handling Incomplete Information in Databases

M. H. WILLIAMS AND K. A. NICHOLSON
Computer Science Department, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ

In a database missing values may be denoted by the use of a special null symbol. However, between the extremes of complete information and missing information one may have various states of incomplete or partial information. One of the advantages which logic databases will have over relational databases is the ability to handle incomplete information in a more helpful way than simply through the use of null values. This paper considers three forms of incomplete information and describes how the query language Query-by-Example can be extended to cater for these forms. The implementation of these ideas in Prolog is described to illustrate how these concepts can be realised in a logic programming language.

Received May 1986, revised November 1986

1. INTRODUCTION

If one constructs a database model of a perfect world, the value of each item of information will be known exactly; however, if one wishes to model a real-world situation, exact knowledge of each attribute is not always possible. In a real world the values of some attributes will be known exactly with absolute certainty while other values may be completely unknown. This latter situation is usually taken care of by the introduction of a null value.[1]

Between these two extremes of complete information and total absence of information, one may have various states of partial or incomplete information. This is a situation in which, although the value is not known exactly, sufficient is known about it to be useful (i.e. not to classify it as completely unknown).

For example, consider a database containing details of patients in a hospital. When a new patient is admitted, the data on that patient are entered into the database system. If the patient has been involved in an accident and is unconscious on arrival, it may not be possible to capture some of this data. In particular, the age of the patient may be unknown and, if this is required, a null value may have to be given to indicate that the value is unknown. Yet it is not true that the patient's age is completely unknown – it should certainly be possible to distinguish a baby from an old man and hence place some limits on this attribute. This could be invaluable if, say, the pharmaceutical division of the hospital needs to issue a prescription for the patient and wants to check that the dosage is correct.

Equally there may be uncertainty about a patient's symptoms. The patient may not have taken note of the frequency of some symptom and can only place some limits on its value. Alternatively the extent of some symptom (e.g. severity of pain) may be in doubt. There may even be some doubt about the diagnosis. In each case a null value would be far from helpful.

For this reason various workers (e.g. Bowers,[2] Babad and Hoffer[3]) have studied the problem of representing and manipulating incomplete information. The present paper is concerned with how incomplete information can be handled in a database system implemented in the logic programming language Prolog.[4] In particular three specific types of incomplete information are considered. The query language Query-by-Example (QBE)[5] has been used as the basis of this study, and extensions are described which enable incomplete information to be dealt with in a natural and simple manner. These extensions can be mapped into Prolog in a simple but effective way.

2. LIPSKI'S MODEL OF INCOMPLETENESS

In the mathematical model of a database with incomplete information proposed by Lipski,[6] an *information system* or *system* is defined as a triple

$$\langle X, (D_i)_{i\in I}, U\rangle$$

where X is the set of objects under consideration,
I is a set of attributes relating to these objects,
D_i is the set of attribute values (domain) for attribute i, and
U is a function which associates with any $i\in I$ and $a\in D_i$ a set of objects $U(i,a)\subseteq X$ for which attribute i *possibly takes value* a.

Another function $u(i,a)$, determined by $U(i,a)$, is the set of objects for which the value of attribute i *is known to be* a:

$$u(i,a) = X\setminus\bigcup\{U(i,b): b\in D_i \wedge b \neq a\}$$

For example
$U(colour, red)$ = set of objects for which the colour might possibly be *red*,
$u(colour, red)$ = set of objects for which the colour is known to be *red*.

This approach is clearly attribute-oriented; an alternative view of the same information, but from the point of view of objects, is obtained from the sets

$$\beta_i(x) = \{a\in D_i: x\in U(i,a)\}$$

which contains all possible values which attribute i can assume for the object x. These sets $(\beta_i)_{i\in I}$ are referred to as the classification associated with the system.

A system is said to be complete if each set $\beta_i(x)$ consists of a single element of D_i, or equivalently if the two functions u, U are identical (i.e. $u = U$). In other words, it is complete if there is no uncertainty about the attributes of each object in the system. A system which is not complete is said to be incomplete.

A query in such a system is either a term (a description of a subset), $\langle term\rangle$, or a formula (some fact concerning

the system as a whole – a 'yes–no' query), ⟨*form*⟩. It may be defined as follows

$$\langle query\rangle ::= \langle form\rangle \mid \langle term\rangle$$
$$\langle form\rangle ::= \neg\langle form\rangle \mid \langle form\rangle \vee \langle form\rangle \mid \langle form\rangle \wedge \langle form\rangle \mid \langle form\rangle \Rightarrow \langle form\rangle \mid (\langle form\rangle) \mid \mathbf{true} \mid \mathbf{false} \mid \langle term\rangle = \langle term\rangle$$
$$\langle term\rangle ::= -\langle term\rangle \mid \langle term\rangle + \langle term\rangle \mid \langle term\rangle . \langle term\rangle \mid \langle term\rangle \rightarrow \langle term\rangle \mid (\langle term\rangle) \mid \varnothing \mid X \mid \langle descriptor\rangle$$

Where $\neg$, $\vee$, $\Rightarrow$ are the usual logical connectives and $-$, $+$, $.$, $\rightarrow$ the set operators (complementation, union, intersection and '$(X \backslash A) \cup B$'). A descriptor is defined to have the form $\langle i, A\rangle$ where $i \in I$ and A is a subset of D_i. In a complete system the descriptor $\langle i, A\rangle$ stands for the set of objects for which the value of i is in A. Examples of descriptors include

$$\langle colour, \{green\}\rangle \quad \text{or} \quad \langle COLOUR = green\rangle$$

$$\langle length, \{x \in R^+ : x \leqslant 30\}\rangle \quad \text{or} \quad \langle LENGTH \leqslant 30\rangle, \quad \text{etc.}$$

In a complete system the *value* of any query, denoted by $\|Q\|$, is the response of the system to the query, and this has its usual interpretation. However, in the case of an incomplete system, different interpretations are possible. Two limiting internal interpretations of a query Q are

(*a*) the *lower value* of Q, $\|Q\|_*$, which is the set of objects which from the information available cannot fail to satisfy the query Q, and

(*b*) the *upper value* of Q, $\|Q\|^*$, which is the set of objects which from the information available could possibly satisfy the query Q.

Lipski studies these two limiting interpretations and defines a number of useful relationships which assist in evaluating such a query.

3. CLASSES OF INCOMPLETE INFORMATION

As defined in the previous section, each $\beta_i(x)$ represents an attribute state, i.e. a set of possible values which attribute i may have for object x. This may be single-valued, in which case the attribute state consists of a set with only one value (e.g. $\beta_{SEX}(x) = \{male\}$) or, in an incomplete system, may have more than one value (e.g. $\beta_{COLOUR}(x) = \{black,red,blue\}$).

In the approach studied in this paper, three specific categories of incomplete information are considered. These are as follows.

(*a*) *Sets.* The most obvious form of incomplete information is a set of possible values. This is relevant if the domain is an unordered set of values and the expected value of the attribute lies within some proper subset of this set. It will be assumed that all possible values of this subset are equally probable. For example, if the colour of car y is known to be either white or cream but it is not certain which, the value of $\beta_{COLOUR}(y)$ will be {*white*, *cream*}.

(*b*) *Simple ranges.* Another obvious form of incomplete information is a simple numeric range. This type of construct is completely determined by the values of its lower and upper bounds. If an attribute is assigned a simple range it is assumed that the value of the attribute lies somewhere between the lower bound and the upper bound (inclusive) and that all values in this range are equally probable. An example of a simple range might be the age of a patient. Normally one would expect the age of a patient to be given as a single numeric value; however, if the age is not known exactly, it could be specified as a range.

(*c*) *Multiple ranges.* The natural extension of a simple numeric range is a set of non-contiguous ranges. Although this is not nearly as common as the simple range, there are circumstances in which it is useful. In particular, an event may be known to have occurred in one of several possible discrete intervals of time but it is not known for certain which.

4. SELECTORS FOR INCOMPLETE INFORMATION

In Section 2 the notion of a descriptor was introduced as part of the query format. A descriptor is used to define a selector which, when applied to the set of objects X, returns those objects which satisfy that selector. Following Lipski, it is assumed that each descriptor defines a selector S involving one particular attribute i, and is denoted by $\langle i, S\rangle$. Thus given the descriptor $\langle j, S\rangle$, when S is applied to X, it selects all objects for which $\beta_j(x)$ satisfies S. Selectors for the three classes of incomplete information described in the previous section are as follows.

(*a*) *Sets.* In establishing selectors for sets, typical operations which one might expect are:

(1) The ability to select objects x by describing their $\beta_i(x)$ sets directly. An example of a simple selector for sets might be

$$SEX = \{male\}$$

which will select all objects for which $\beta_{SEX}(x) = \{male\}$. In the case of an incomplete system, one might have a selector such as

$$COLOUR = \{blue,green\}$$

which will select all objects x for which $\beta_{COLOUR}(x) = \{blue,green\}$, i.e. objects whose colour is definitely either blue or green but is is not certain which. In general, this form of selector must specify a list of values v_j and is used to select those objects x for which $\beta_i(x)$ consists of exactly the same list of values.

(2) The ability to cater for incomplete set descriptions. For example, the notation $\{X\}^+$ might be used to denote any set which contains X. Thus

$$COLOUR = \{blue,green\}^+$$

will select all objects for which $\beta_{COLOUR}(x)$ contains both blue and green and possibly other values too (e.g. $\beta_{COLOUR}(x) = \{blue,green,red\}$).

(3) The ability to select objects by specifying the value with which they do not match. Within a set description a value v might be preceded by the operator '**not**' to indicate that it will match any value other than v. For example

$$COLOUR = \{\mathbf{not}\ red\}$$

is interpreted as selecting any object x for which $\beta_{COLOUR}(x)$ contains a single value other than red. This interpretation follows from the fact that '**not** *red*' represents any single colour other than red.

(*b*) *Simple ranges.* A simple selector for a range

might involve one of the relational operators $<$, $\leqslant$, $>$, $\geqslant$, $=$ or $<>$. For example

$$AGE < 30$$

might select all objects for which the attribute *AGE* has value less than 30. Once again it is necessary to distinguish between objects for which the attribute *AGE* is certainly less than 30 and those for which it could possibly be less than 30.

(*c*) *Multiple ranges*. The properties of selectors for this type of construct are similar to those for simple ranges.

5. QBE EXTENSIONS FOR UPDATES AND SIMPLE QUERIES

This section considers the actual query-language extensions proposed for QBE to enable it to handle the constructs outlined in Section 3. Normally, in a QBE query each attribute contains either a constant value or a variable. In order to introduce the three types of incomplete information, a format is required for representing data of each of the three types. This is necessary for inserting or updating tuples which contain attributes whose values are only partially known.

To query a relation containing incomplete information, further extensions are required. The simplest approach is to define a set of special functions for each type of construct. Alternatively one may create two general functions (pos and def) which take more general arguments and process them according to the type of incomplete information concerned. Both approaches are described here.

Once again consider each construct in turn.

(*a*) *Sets*. In order to enter or change information of this type the operations of insertion, deletion and update may be extended to permit data items of the form

$$"\{"\langle item\rangle\{","\langle item\rangle\}"\}"$$

Thus suppose that one has a table of descriptions of suspicious vehicles provided by witnesses in the vicinity of a crime, and suppose that one wishes to add the reported sighting by the witness Jones of a white, cream or yellow vehicle which was either an estate car or a hatchback. One may have an entry such as

suspvehicle	witness	colour	type
i.	*jones*	*{white,cream, yellow}*	*{estate, hatchback}*

The parts in italics denote user-specified entries, the remainder being displayed by the system. Deletion and update operations are handled in a similar fashion.

If a particular attribute which is to be printed is found to contain a set of values, this is displayed by the system in the same form. For example, if one wished to display the details contained in the suspvehicle relation one might enter

suspvehicle	witness	colour	type
p.			

and the system might respond by displaying

witness	*colour*	*type*
smith	blue	saloon
doe	{blue,green}	panel-van
jones	{white,cream,yellow}	{estate,hatchback}
evans	black	taxi
:	:	:

The problem arises when one wishes to query such a system. Suppose one wishes to determine who reported sighting a blue vehicle, and poses the query

suspvehicle	witness	colour	type
	p.X	*blue*	

The meaning of this query is obvious in the case of Smith (who satisfies the query) and Evans (who does not). However, its meaning is not clear in the case of Doe – it could be interpreted as giving the name of each witness who reported seeing a vehicle whose colour was definitely blue (in which case Doe does not qualify) or whose colour could possibly be blue (in which case he does). For this reason a query of this type will be flagged as an error if it encounters an incomplete value (in this case, a set). In order to express queries involving incomplete information in the form of sets, four special functions are introduced, namely:

makeset(*Var*) – if *Var* is a term (as opposed to a set) then return the set comprising this single term (i.e. {*Var*}) otherwise return *Var*

int(*Var*1,*Var*2) – returns the intersection of *makeset*(*Var*1) and *makeset*(*Var*2)

sub(*Var*1,*Var*2) – if *makeset*(*Var*1) is a subset of *makeset*(*Var*2) then return **true** otherwise return **false**.

iexact(*Var*) – if *Var* is a set then return **false** else return **true**.

Thus one may pose queries such as

suspvehicle	witness	colour	type
	p.X	*Y*	

CONDITIONS
sub(*white*,*Y*)

which will display the names of all people who claim that the vehicle which they saw could possibly have been white (i.e. $\beta_{COLOUR}(x) = \{white\}^+$), and

suspvehicle	witness	colour	type
	p.X	*Y*	

CONDITIONS		
sub(*white*,*Y*)	*and*	*sub*(*Y*,*white*)

which will display the names of those who maintain that the vehicle was definitely white (i.e. $\beta_{COLOUR}(x) = \{white\}$).

Although these functions are adequate for handling such queries, they are not always the simplest and most easily readable form for users and could lead to mistakes. To avoid this problem, two functions *pos* and *def* are introduced (analogous to Lipski's $\| \ \|_*$ and $\| \ \|^*$), each of which takes as argument a condition of the form

$$\langle variable\rangle\langle relational\ operator\rangle\langle expression\rangle$$

Thus

pos(⟨*condition*⟩) is true if the condition could possibly be true otherwise it is false,

def(⟨*condition*⟩)is true if the condition is definitely true otherwise it is false.

In the case of sets, the relational operator is one of =, < >, *in* or *notin* and the expression consists of a variable, a constant or a set. Thus one may formulate queries such as

suspvehicle	witness	colour	type
	p.X	*Y*	

CONDITIONS
def(*Y* = *black*)

which will display the names of all witnesses who claim that the vehicle which they saw was definitely black (i.e. $\beta_{COLOUR}(x) = \{black\}$), or

suspvehicle	witness	colour	type
	p.X	*Y*	

CONDITIONS
pos(*Y* = {*white*,*yellow*})

which will display the names of those who claim that the vehicle could possibly be either white or yellow (i.e. any entry for which $\beta_{COLOUR}(x) = \{white,yellow\}^+$). As an example of a more complex query, one may wish to display the name of each witness who reported a vehicle which definitely had one of the colours which Jones reported seeing, i.e.

suspvehicle	witness	colour	type
	jones	Y	
	p.X	*Y*1	

CONDITIONS
def(Y1 in Y)

A complete definition of these operators is given in Table 1.

Table 1. Definitions of functions *def* and *pos* for sets

def(*Var*1 = *Var*2)	*iexact*(*Var*1) **and** *iexact*(*Var*2) **and** *Var*1 = *Var*2
def(*Var*1 < > *Var*2)	*int*(*Var*1,*Var*2) = [ ]
pos(*Var*1 = *Var*2)	*int*(*Var*1,*Var*2) < > [ ]
pos(*Var*1 < > *Var*2)	**not** *iexact*(*Var*1) **or not** *iexact*(*Var*2) **or** *Var*1 < > *Var*2
def(*Var*1 **in** *Var*2)	**sub**(*Var*1,*Var*2)
pos(*Var*1 **in** *Var*2)	**int**(*Var*1,*Var*2) < > [ ]
def(*Var*1 **notin** *Var*2)	**int**(*Var*1,*Var*2) = [ ]
pos(*Var*1 **notin** *Var*2)	**not sub**(*Var*1,*Var*2)

(*b*) *Simple ranges.* In this instance the operations for insertion, deletion and update are extended by the introduction of data items of the form

⟨*number*⟩"."."⟨*number*⟩

For example, in order to add to the database a new patient whose age is uncertain but is known to lie in the range 25 to 35 (inclusive), one might enter

casualties	patientno	name	age	weight
i.	*125*	*smith*	*25..35*	*190*

Deletions and updates are extended in the same way.

When an attribute is printed, if it is found to contain a simple range this will be displayed in the form

⟨*number*⟩"."."⟨*number*⟩

For example, if one entered the command

casualties	patientno	name	age	weight
p.				

the system might return with

patientno	name	age	weight
118	doe	20..30	140..150
120	brown	72	127
124	taylor	15	120..130
125	smith	25..35	190

As with sets, additional functions are introduced to facilitate the formulation of queries involving simple ranges. These are

imin(*Y*) – if *Y* is a range then return the lower bound of *Y* else return *Y*.
imax(*Y*) – if *Y* is a range then return the upper bound of *Y* else return *Y*.
iave(*Y*) – if *Y* is a range then return midpoint value of *Y* else return *Y*.
iexact(*Y*) – if *Y* is a range then return **false** else return **true**.

Using these one may formulate queries such as

casualties	patientno	name	age	weight
		p.X		*Y*

CONDITIONS
imin(*Y*) < = *127* *and* *imax*(*Y*) > = *127*

which will print the names of all patients whose weight could possibly be 127 lb, and

casualties	patientno	name	age	weight
		p.X		*Y*

CONDITIONS
imin(*Y*) = *127* *and* *imax*(*Y*) = *127*

which will print the names of those whose weight is exactly 127 lb.

Once again , seeking a more readable notation, the two functions *def* and *pos* may be used with conditions of the form

⟨*variable*⟩⟨*relational operator*⟩⟨*expression*⟩

where ⟨*relational operator*⟩ may be any of <, ⩽, >, ⩾, = or < >. For example

casualties	patientno	name	age	weight
		p.X		*Y*

CONDITIONS
def(*Y* = *125*)

would display the names of all patients whose weight is definitely 125 lb (i.e. exactly 125 lb), whereas

casualties	patientno	name	age	weight
		p.X		*Y*

CONDITIONS
def(*Y* > *125*)

would display the names of all patients whose weight is definitely over 125 lb (i.e. those whose weight is known exactly and is greater than 125 lb as well as those whose

weight is not known exactly but whose lower limit exceeds 125 lb), and

casualties	patientno	name	age	weight
		p.X		*Y*

CONDITIONS
pos(*Y* > 125)

would display the names of all patients whose weight may possibly exceed 125 lb (i.e. those whose weight is known exactly and is greater than 125 lb, as well as those whose weight is not known exactly but whose upper limit exceeds 125 lb). The effects of these two functions are defined in Table 2.

Table 2. Definition of functions *def* and *pos* for simple range

def(*Var* = *Exp*)		*iexact*(*Var*) **and** *iexact*(*Exp*) **and** *Var* = *Exp*
def(*Var* > *Exp*)	**or** ⩾	*imin*(*Var*) > *imax*(*Exp*)
def(*Var* < *Exp*)	**or** ⩽	*imax*(*Var*) < *imin*(*Exp*)
def(*Var* < > *Exp*)		*imin*(*Var*) > *imax*(*Exp*) **or** *imax*(*Var*) < *imin*(*Exp*)
pos(*Var* = *Exp*)		*imax*(*Var* ⩾ *imin*(*Exp*) **and** *imin*(*Var*) ⩽ *imax*(*Exp*)
pos(*Var* > *Exp*)	**or** ⩾	*imax*(*Var* > *imin*(*Exp*)
pos(*Var* < *Exp*)	**or** ⩽	*imin*(*Var*) < *imax*(*Exp*)
pos(*Var* < > *Exp*)		**not** *iexact*(*Var*) **or not** *iexact*(*Exp*) **or** *Var* < > *Exp*

The expression on the right-hand side of the condition may be any general expression, provided that the values of any variables contained therein are not ranges, or alternatively it may have a simple form in which the value of one of the variables may be a range. For example, the query

casualties	patientno	name	age	weight
		doe	*S*	
		p.X	*T*	

CONDITIONS
pos(*S* < *T*)

will display the names of all patients who could possibly be older than Doe.

(*c*) *Multiple ranges.* A multiple range is treated as a set of simple ranges. For example, suppose that a record is kept of the last time a patient was visited by a doctor. Due to unforeseen circumstances, the only information available on a particular admission was that the patient was visited either during the afternoon round (2–4 p.m.) or during the early evening (6–8 p.m.). This might be entered into the database as follows:

visit	patient	time	doctor
i.	*macbeth*	[*14..16, 18..20*]	*crippen*

or, a little less messily, as

visit	patient	time	doctor
i.	*macbeth*	*X*	*crippen*

CONDITIONS
X ← [*14..16, 18..20*]

Multiple-range queries are similar to those for single ranges, with the following basic functions:

imin(*X*) – if *X* is a set of ranges then return overall minimum of *X* else as before.
imax(*X*) – if *X* is a set of ranges then return overall maximum of *X* else as before.
iexact(*X*) – if *X* is a set of ranges then return **false** else as before.

Only *iave*(*X*) is not defined for a set of ranges and will produce an error.

The functions *def* and *pos* are defined in much the same way as for ranges (see Table 3). Thus if one wishes to determine who could possibly have been visited before 4 p.m. one may enter

visit	patient	time	doctor
	p.X	*Y*	

CONDITIONS
pos(*Y* < 16)

or if one wishes to know the latest time that each patient could have been visited one may enter

visit	patient	time	doctor
	p.X	*p.imax*(*Y*)	

Table 3. Definitions of functions *def* and *pos* for multiple range

def(*Var* = *Exp*)		*iexact*(*Var*) **and** *iexact*(*Exp*) **and** *Var* = *Exp*
def(*Var* > *Exp*)	**or** ⩾	*imin*(*Var*) > *imax*(*Exp*)
def(*Var* < *Exp*)	**or** ⩽	*imax*(*Var*) < *imin*(*Exp*)
def(*Var* < > *Exp*)		value of *Exp* lies outside any subrange of *Var*
pos(*Var* = *Exp*)		value of *Exp* overlaps at least one subrange of *Var*
pos(*Var* > *Exp*)	**or** ⩾	*imax*(*Var*) > *imin*(*Exp*)
pos(*Var* < *Exp*)	**or** ⩽	*imin*(*Var*) < *imax*(*Exp*)
pos(*Var* < > *Exp*)		**not** *iexact*(*Var*) **or not** *iexact*(*Exp*) **or** *Var* < > *Exp*

6. COMPLEX QUERIES

The facilities for constructing queries outlined in the previous section adequately cater for all simple queries. However, a problem arises in the case of complex queries.

Suppose that one wishes to retrieve from some database all people who are definitely male or who definitely have blood group AB. The condition part for such a query may be written as

def(*SEX* = *male*) **or** *def*(*BLOODGROUP* = *AB*)

This query will select any object x for which $\beta_{SEX}(x) = \{male\}$ or $\beta_{BLOODGROUP}(x) = \{AB\}$, i.e.

$$\{x \in X: \beta_{SEX}(x) = \{male\}\} \cup \{x \in X: \beta_{BLOODGROUP}(x) = \{AB\}\}$$

On the other hand, if one wishes to retrieve all people who definitely have blood group O or AB, there are two possible interpretations as to what might be intended, as follows.

(*a*) Retrieve all people whose blood group is known definitely and who have either O or AB as their blood group. For this case the query

$$def(BLOODGROUP = O) \quad \textbf{or} \quad def(BLOODGROUP = AB)$$

will select all objects for which $\beta_{BLOODGROUP}(x)$ is either $\{O\}$ or $\{AB\}$, i.e.

$$\{x \in X: \beta_{BLOODGROUP}(x) = \{O\}\} \cup \{x \in X: \beta_{BLOODGROUP}(x) = \{AB\}\}$$

(*b*) Retrieve all people whose blood group is definitely either O or AB. This would include those whose blood group is known definitely (first interpretation) as well as those whose blood group is uncertain but is one of O or AB, i.e.

$$def(BLOODGROUP = O) \textbf{ or } def(BLOODGROUP = AB) \textbf{ or } def(BLOODGROUP = [O,AB])$$

which will select all objects x such that

$$\{x \in X: \beta_{BLOODGROUP}(x) = \{O\}\} \cup \{x \in X: \beta_{BLOODGROUP}(x) = \{AB\}\} \cup \{x \in X: \beta_{BLOODGROUP}(x) = \{O,AB\}\}$$

Following Lipski's approach, the query

$$def(BLOODGROUP = O \textbf{ or } BLOODGROUP = AB)$$

will be given interpretation (*b*) above.

The problem is exacerbated in cases of more complex queries. For example, suppose one has a database of possible donors who are willing to donate their organs for transplant in the fullness of time. The information on any individual donor may be incomplete. Further, suppose that one wishes to search the database for a possible donor for an eye transplant. To this end one may wish to retrieve all persons who definitely have blood group O or A and have either brown or hazel-coloured eyes. The basic form of this query is

$$(BLOODGROUP = O \textbf{ or } BLOODGROUP = A) \text{ and } (COLOUR = brown \textbf{ or } COLOUR = hazel)$$

The simplest interpretation of this query

$$(def(BLOODGROUP = O) \textbf{ or } def(BLOODGROUP = A)) \textbf{ and } (def(COLOUR = brown) \textbf{ or } def(COLOUR = hazel))$$

would extract those items x given by

$$(\{x \in X: \beta_{BLOODGROUP}(x) = \{O\}\} \cup \{x \in X: \beta_{BLOODGROUP}(x) = \{A\}\}) \cap (\{x \in X: \beta_{COLOUR}(x) = \{brown\}\} \cup \{x \in X: \beta_{COLOUR}(x) = \{hazel\}\})$$

However, there may be individuals with either blood group unknown but known to be one of O or A, or colour unknown but known to be one of brown or hazel, or both; in other words, the complete set of individuals who might satisfy the query is

$$(\{x \in X: \beta_{BLOODGROUP}(x) = \{O\}\} \cup \{x \in X: \beta_{BLOODGROUP}(x) = \{A\}\} \cup \{x \in X: \beta_{BLOODGROUP}(x) = \{O,A\}\}) \cap (\{x \in X: \beta_{COLOUR}(x) = \{brown\}\} \cup \{x \in X: \beta_{COLOUR}(x) = \{hazel\}\} \cup \{x \in X: \beta_{COLOUR}(x) = \{brown,hazel\}\}$$

Consequently, this is the interpretation given to the query

$$def((BLOODGROUP = O \textbf{ or } BLOODGROUP = A) \textbf{ and } (COLOUR = blue \textbf{ or } COLOUR = hazel))$$

This can be summarized as follows:

$def(i = A \textbf{ or } i = b)$ selects $\{x \in X: \beta_i(x) = \{A\}\} \cup \{x \in X: \beta_i(x) = \{B\}\} \cup \{x \in X: \beta_i(x) = \{A,B\}\}$

$def(i = A \textbf{ or } j = B)$ selects $\{x \in X: \beta_i(x) = \{A\}\} \cup \{x \in X: \beta_j(x) = \{B\}\}$

$def(i = A \textbf{ and } j = B)$ selects $\{x \in X: \beta_i(x) = \{A\}\} \cap \{x \in X: \beta_j(x) = \{B\}\}$

$def(i = A \textbf{ and } i = B)$ is meaningless

$pos(i = A \textbf{ or } i = B)$ selects $\{x \in X: \beta_i(x) = \{A\}^+\} \cup \{x \in X: \beta_i(x) = \{B\}^+\}$

$pos(i = A \textbf{ or } j = B)$ selects $\{x \in X: \beta_i(x) = \{A\}^+\} \cup \{x \in X: \beta_j(x) = \{B\}^+\}$

$pos(i = A \textbf{ and } j = B)$ selects $\{x \in X: \beta_i(x) = \{A\}^+\} \cap \{x \in X: \beta_j(x) = \{B\}^+\}$

$pos(i = A \textbf{ and } i = B)$ selects $\{x \in X: \beta_i(x) = \{A\}^+\} \cap \{x \in X: \beta_i(x) = \{B\}^+\}$
i.e. $\{x \in X: \beta_i(x) = \{A,B\}^+\}$

In addition the operator '**not**' may be handled as follows:

$$def(\textbf{not } A) = \textbf{not}(pos(A))$$

$$pos(\textbf{not } A) = \textbf{not}(def(A))$$

Thus a complex query may have the form

$$def(\langle condition \rangle) \text{ or } pos(\langle condition \rangle)$$

where $\langle condition \rangle$ is any expression consisting of one or more terms of the form described in the previous section combined by means of the operators **and**, **or** and **not**, i.e.

$\langle condition \rangle ::= \textbf{not}\langle condition \rangle \mid (\langle condition \rangle) \mid \langle condition \rangle \textbf{and} \langle condition \rangle \mid \langle condition \rangle \textbf{or} \langle condition \rangle \mid \langle term \rangle$
$\langle term \rangle ::= \langle var \rangle \langle relop1 \rangle \langle exp1 \rangle \mid \langle var \rangle \langle relop2 \rangle \langle exp2 \rangle$
$\langle relop1 \rangle ::= < > \mid \textbf{in} \mid \textbf{notin} \mid =$
$\langle exp1 \rangle ::= \langle var \rangle \mid \langle const \rangle \mid \{\langle constlist \rangle\} \mid union(\langle exp1 \rangle, \langle exp1 \rangle)$
$\langle relop2 \rangle ::= < \mid <= \mid > \mid >= \mid = \mid <>$
$\langle exp2 \rangle ::= \langle exp2 \rangle + \langle exp2 \rangle \mid \langle exp2 - \rangle \langle exp2 \rangle \mid \langle exp2 \rangle * \langle exp2 \rangle \mid \langle exp2 \rangle / \langle exp2 \rangle \mid (\langle exp \rangle) \mid \langle term2 \rangle$
$\langle constlist \rangle ::= \langle constlist \rangle, \langle const \rangle \mid \langle const \rangle$
$\langle term2 \rangle ::= \langle var \rangle \mid \langle const \rangle \mid imin(\langle var \rangle) \mid imax(\langle var \rangle) \mid iave(\langle var \rangle)$

In order to implement this, each query with a complex condition is analysed as follows.

(1) Move negation inwards using the transformations

$$\textbf{not } (A \textbf{ and } B) \rightarrow \textbf{not } A \textbf{ or not } B$$

$$\textbf{not } (A \textbf{ or } B) \rightarrow \textbf{not } A \textbf{ and not } B$$

removing any double negations.

(2) If any term (in the sense of the above definition) is preceded by **not**, remove the operator **not** and reverse the relational operator in the term.

(3) If the query is a *def* query convert the condition to

conjunctive normal form; if it is a *pos* query convert the condition to disjunctive normal form.

(4) Within each set of terms joined by '**or**' in conjunctive normal form or by '**and**' in disjunctive normal form, group together any terms which involve the same attribute. Now move the *def/pos* function to precede each term or group of terms referring to an attribute.

7. IMPLEMENTATION IN PROLOG

An implementation of the query language Query-by-Example has been developed in the language Prolog (Neves *et al.*)[7] and has been used as the basis of various studies.[8,9,10] In this system each tuple of a relational database is mapped into a Prolog ground clause. For example, the tuple for witness Smith in relation suspvehicle of Section 5 would be represented as

suspvehicle(*smith*,*blue*,*saloon*).

In order to extend this system to handle incomplete information, a tuple containing one or more attributes which are only partially known is represented as a Prolog rule. Thus, if the value of an attribute is specified as a set of possible values

$$\{x, \ldots, y\}$$

the attribute will be represented by some variable, say X, and the right-hand side will include the predicate

$$set(X,[x, \ldots, y]).$$

To illustrate this, the tuples of the relation *suspvehicle* given in Section 5 would be stored as

suspvehicle(*smith*,*blue*,*saloon*).
suspvehicle(*doe*,*X*,*panel-van*):−*set*(*X*,[*blue*,*green*]).
suspvehicle(*jones*,*X*,*Y*):−*set*(*X*,[*white*,*cream*,*yellow*]),
set(*Y*,[*estate*,*hatchback*]).
⋮

In the case that the value of an attribute is described as a simple numeric range

$$n..m$$

the attribute will again be represented by a variable, say X, and the right-hand side will include the predicate

$$range(X,n,m)$$

For example, the entry for patient number 125 in Section 5 would be stored as

casualties(125,*smith*,*X*,190):−*range*(*X*,25,35).

A multiple range is handled in much the same way as a simple range, except that the second and third parameters of the predicate range represent the lowest and highest values in the set of ranges while a fourth parameter holds the list of gaps in the range. Thus the example given in Section 5 may be represented as

visit(*macbeth*,*X*,*crippen*):−*range*(*X*,14,20,[16,18]).

A query to the database is translated into what is effectively a Prolog query. Thus a query such as that given in Section 5 to display the names of those witnesses who reported sighting a blue vehicle might be translated as

answer(*X*):−*suspvehicle*(*X*,*blue*,–).
?-*answer*(*X*).

For each ground clause the right-hand side of the rule defining *answer*(*X*) either matches or fails to match the clause, thereby providing the values of *answer*(*X*). In the case of a rule, if *suspvehicle*(*X*,*blue*,–) matches the left-hand side of the rule, the right-hand side of the rule will be executed, resulting in calls to the predicates *range* and/or *set*.

Queries involving the special functions *makeset*,*int*,etc., are mapped into Prolog queries in a similar fashion. For example, the query

suspvehicle	witness	colour	type
	p.X	*Y*	

CONDITIONS
sub(*white*,*Y*)

is translated as:

answer(*X*):−*suspvehicle*(*X*,*Y*,–),*sub*(*white*,*Y*).
?-*answer*(*X*).

Queries involving the functions *pos* and *def* are translated using the definitions in Tables 2 and 3 to produce Prolog queries involving the functions *imax*, *imin*,*sub*,etc. Thus the query

casualties	patientno	name	age	weight
		p.X	*Y*	

CONDITIONS
def(*Y* = 50)

might be translated to:

answer(*X*):−*casualties*(–,*X*,*Y*,–),*iexact*(*Y*),*Y* = 50.
?-*answer*(*X*).

And so on.

The predicates *set* and *range* are defined within the system. Each predicate checks that the first argument of the call is a variable, and if it is not, a suitable error message is printed and the clause fails. If it is a variable, the set or range described in the remaining arguments is assigned to that variable.

The data for the casualties relation given in Section 5 would be represented internally as

casualties (118,*doe*,*X*,*Y*):−*range*(*X*,20,30),*range*(*Y*,140, 150).
casualties(120,*brown*,72,127).
casualties(124,*taylor*,15,*X*):−*range*(*X*,120,130).
casualties(125,*smith*,*X*,190):−*range*(*X*,25,35).

If the user wishes to display the names of all patients whose weight is definitely 127 lb he may write

casualties	patientno	name	age	weight
		p.A		*B*

CONDITIONS
def(*B* = 127)

which in turn may be translated to Prolog as

answer(*A*):−*casualties*(–,*A*–,*B*),*iexact*(*B*),*B* = 127
?-*answer*(*A*).

When this is applied to the database, the goal casualties(–,*A*,–,*B*) will match with the left-hand side of the first clause, instantiating *A* with *doe*, *X* with the anonymous

variable –, and B with Y, so that the right-hand side of the clause becomes

$$range(-,20,30),range(B,140,160)$$

The first call assigns the range 20–30 to the anonymous variable, the second assigns the range 140–160 to B. Thus the goal casualties $(-,A,-,B)$ will succeed. The system will then call $iexact(B)$ with B instantiated to the range 140–160, which must fail.

The second clause in the definition of casualties matches with casualties $(-,A,-,B)$, instantiating A with *brown*, and B with 127, and returns the value **true**. This time the call to $iexact(B)$ will succeed, as will the final condition, causing the value *brown* to be recorded as a valid answer. And so on.

8. CONCLUSION

In the real world there are many instances in which complete information on a particular attribute for some object is not available; nevertheless, some information does exist which could be helpful and is worthwhile recording. Such information, referred to as incomplete or partial information, can take various forms. In order to handle this type of information in a database system, it is necessary to provide appropriate data structures and to extend the database query language accordingly.

This paper focuses on one approach to handling incomplete information based on Lipski's model and describes how the language QBE can be extended to cater for this. This approach caters for three basic types, namely sets, simple ranges and multiple ranges. Mechanisms are discussed for handling both simple and complex queries.

One assumption underlying these forms of incomplete information is that all values in some range or set are equally probable. Work is continuing on the case where this is not true and probabilities need to be associated with individual values from the range or set.

An implementation of the query language QBE, which has been developed in the logic programming language Prolog and used as the basis of various studies, has been extended to incorporate the ideas outlined in this paper. This system is currently running on a High Level Hardware Orion and a Vax 11/750.

REFERENCES

1. E. F. Codd, A relational model of data for large shared data banks. *Communications of the ACM* **13**, 377–387 (1970).
2. D. S. Bowers, A database architecture for aggregate incomplete data. *The Computer Journal* **27**, 294–300 (1984).
3. Y. M. Babad and J. A. Hoffer, Even no data has a value. *Communications of the ACM* **27**, 748–756 (1984).
4. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*. Springer-Verlag, Heidelberg, (1981).
5. M. M. Zloof. Query-by-Example: a data base language. *IBM Systems Journal* **16**, 324–343 (1977).
6. W. Lipski, On semantic issues connected with incomplete information in databases. *ACM Transactions on Database Systems* **4**, 262–296 (1979).
7. J. C. Neves, S. O. Anderson and M. H. Williams, A Prolog implementation of Query-by-Example. *Proceedings of the 7th International Computing Symposium*, pp. 318–332. Teubner, Stuttgart (1983).
8. J. C. Neves and M. H. Williams, Towards a co-operative data base management system. *Proceedings of the Logic Programming Workshop 83*, pp. 341–370. Universidade de Lisboa, Portugal (1983).
9. M. H. Williams, J. C. Neves and S. O. Anderson, Security and integrity in logic data bases using Query-by-Example. *Proceedings of the Logic Programming Workshop 83*, pp. 304–340. Universidade de Lisboa, Portugal (1983).
10. M. H. Williams and I. M. Pattison, Reorganisation in a Simple Data Base System, **16**(8), 719–729 (1986). *Software Practice and Experience*).

Announcement

2–3 JUNE 1988

The First International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, The University of Tennessee Space Institute, Tullahoma, TN, USA

Aim and scope

The objective of this conference is to provide a forum for the international community (industrial, engineering, academic and government representatives) to (i) present and discuss their contributions; (ii) overview the recent successes of AI & ES in the industrial and engineering environment, (iii) form mutual cooperative research groups in the areas of interest, and (iv) discuss the future directions of research and development. Original papers will be presented in areas relating AI & ES technology to engineering and industrial applications. The papers, concerning basic research, tools and techniques, will focus on one or more aspects of knowledge acquisition, representation and learning; inference, reasoning and searching techniques; heuristic programming; AI languages; hardware and software implementation; validation and evaluation; planning and problem solving; scheduling; automatic programming; cognitive modelling; natural language and speech; perception and robotics; image processing and understanding; and intelligent distributed systems. The application areas include, but are not limited to, the following:

- Fault monitoring and diagnosis
- Computer-aided design
- Automatic controls
- Computer-aided manufacturing
- Computer engineering
- Software engineering
- Systems engineering
- Medical and biomedical engineering
- Aerospace engineering
- Oceanic and atmospheric engineering
- Chemical, petroleum and mining engineering
- Mechanical and electrical engineering
- Intelligent industrial robots
- Robotic vision
- Mobile robots and autonomous navigation
- Aircraft and flight management
- Space station automation
- Space support and satellite autonomy
- Spacecraft controls and operations
- Operations in nuclear or hazardous environments
- Command control and communications
- Man–machine interface

Sponsorship

IEA–AI/ES–88 is sponsored by the University of Tennessee Space Institute, in cooperation with the American Association for Artificial Intelligence, ACM-SIGART, the National Society of Professional Engineers, and the Biomedical Engineering Society.

Further information

Further information may be obtained from and additional correspondence should be directed to:
Professor Moonis Ali, Conference Program Chairman, The University of Tennessee Space Institute, Tullahoma, TN 37388, USA

47. M.H.WILLIAMS and Q.KONG, Incomplete information in a deductive database, Data and Knowledge Engineering, vol. 3,197-220 (1988).

Data & Knowledge Engineering 3 (1988) 197–220
North-Holland

Incomplete information in a deductive database

M. Howard WILLIAMS and Qinzheng KONG
Computer Science Department, Heriot-Watt University, Edinburgh EH1 2HJ, Scotland, U.K.

Abstract. The problem of handling incomplete information efficiently has been studied in the context of relational databases by various workers. However, dealing with incomplete information in a deductive database is a more difficult problem. This paper discusses the problem of interpretation of incomplete information in a deductive database and puts forward a model to assist in understanding the different interpretations and their effect on the evaluation of queries.

Keywords: Incomplete information, Deductive database.

1. Introduction

When the idea of relational databases was first put forward by Codd [2], the assumption was made that the value of each attribute recorded would be known exactly. This is unrealistic in many cases and for this reason Codd later proposed an extended version of the relational model which takes account of simple null values [3]. Between these two extremes of exact information about an attribute value and no information about a value, there are various instances of incomplete information.

Two obvious forms of incomplete information are *sets* and *ranges*. To illustrate these consider a situation in which the hair colour and height (in inches) of a number of suspects is stored in a relation *suspect* which may be represented in tabular form as

suspect	suspect_no	hair_colour	height
	1	fair	69
	2	brown	72

or as a set of facts as follows

suspect(1, *fair*, 69) .

suspect(2, *brown*, 72) .

Suppose that one wishes to record that suspect number 3 is reported to have either brown or black hair (it is not certain which) and to be between 69 and 72 inches tall (the exact value being unknown). This might be expressed as

suspect(3, {*brown*, *black*}, 69–72) .

where {*brown*, *black*} stands for the fact that the hair colour is either brown or black, and 69–72 for the fact that the height is between 69 and 72 inches tall.

Lipski [4] developed a model based on the relational model which catered for attribute values of either of these two types. In this model each tuple of a relation consists of a set of attribute values, each of which is a constant, a set or a range. Queries applied to a database containing incomplete information must specify whether the answer required is the set of values which might possibly satisfy the query or the set which definitely will satisfy it. For example, if one wished to find all possible suspects who might possibly be 70 inches tall, suspect 3 would qualify; however, if one wanted all suspects who are definitely known to be 70 inches tall, suspect 3 would not be included.

The extension of this model to deductive databases is non-trivial. The problem arises when variables and incomplete data are both present in a clause. This is discussed in detail in the following sections. Sections 2, 3 and 4 discuss the interpretations of incomplete information arising in facts, rules and queries, and Section 5 gives the associated model interpretations. Section 6 focusses on two different queries which require information with different certainty, while Section 7 looks at the evaluation of such queries in a system containing incomplete information.

2. Interpretation of incomplete information

The introduction of the notions of incomplete information into a deductive database leads to problems of interpretation which do not arise in the case of the relational model. The basic problem occurs with the introduction of variables into clauses and may be seen from the following two examples.

(a) Suppose that one has a relation

$$manager(Name, Age, Grade, Employeename)\,.$$

which specifies that an individual identified by *Name* whose age is given by *Age* and whose grade is *Grade* is the manager of some other individual identified by *Employeename*. In this case, if one has an individual John whose grade is 10, whose age is not known exactly but lies somewhere in the range 60 to 65 and who is the overall manager (i.e. he is everybody's manager), one might express this as:

$$manager(john,\ 60\text{–}65,\ 10,\ X)\,.$$

From the meaning intended this clause might be translated into full first order logic as follows:

$$\exists_{A\in\{60,61,62,63,64,65\}}\forall_X\ manager(john,\ A,\ 10,\ X)\,.$$

In other words the individual John has a unique value for age (although this value is not known exactly) and is manager to all other individuals in the system.

Incomplete information which is interpreted in this manner (i.e. with the existential quantifier preceding any universal quantifiers) can be regarded as a special kind of value which lies part way between an exact value and a null value and which ranges over a specified subset of the attribute domain.

(b) Suppose that one wishes to use an *is_a* relation to express the fact that everybody in the database is either a member of staff or a student. This can be written as:

$$is_a(X, \{staff, student\}) .$$

If this were expressed in full first order logic, it might be written as:

$$\forall_X \exists_{Y \in \{staff,\ student\}}\ is_a(X, Y) .$$

In this case it is assumed that there are a whole set of individuals, each of whom has its own particular value which is unknown. This is equivalent to

$$\forall_X (is_a(X, staff) \vee is_a(X, student)) .$$

and hence removing the universal quantifier:

$$is_a(X, staff) \vee is_a(X, student) . \qquad (*)$$

Thus the difference between these two examples is that in the first case the clause stands for a group of alternative clauses

$$\forall_X\ manager(\ john, 60, 10, X) \text{ or } \ldots\ldots \text{ or } \forall_X\ manager(\ john, 65, 10, X)$$

but not a clause

$$\forall_X (manager(\ john, 60, 10, X) \vee manager(\ john, 61, 10, X) \vee \ldots\ldots$$
$$\vee\ manager(\ john, 65, 10, X))$$

whereas in the second case the clause stands for a single clause

$$\forall_X (is_a(X, staff) \vee is_a(X, student)) .$$

It is possible (though less likely) that both examples of incomplete information could occur together in a clause. It is even possible to construct a more complex example in which one has three items of incomplete information and two variables and the resulting clause has the form

$$\exists_{A \in \{\cdots\}} \forall_X \exists_{B \in \{\cdots\}} \forall_Y \exists_{C \in \{\cdots\}} p(\ldots) .$$

To take account of these two extreme cases in the first instance, and the more general case in the second instance, the following is proposed:

1. If one or more variables are present in a clause then the default assumption will be that existential quantifiers will follow universal quantifiers. For this case no special annotation will be introduced. For example, the clause:

   $$is_a(X, \{staff, student\}) .$$

   will be interpreted as:

   $$\forall_X \exists_{Y \in \{staff,\ student\}} is_a(X, Y) .$$

2. If one or more variables are present in a clause and an instance of an existential

quantifier should precede all universal quantifiers, the argument concerned is preceded by an asterisk "*". For example

$$manager(john, *60\text{–}65, 10, X).$$

will be interpreted as:

$$\exists_{A\in\{60,61,62,63,64,65\}} \forall_x\, manager(john, A, 10, X).$$

3. If one or more variables are present in a clause and an instance of an existential quantifier should occur somewhere in between the leftmost universal quantifier and the rightmost one, then the argument concerned will be preceded by $*n$ where n indicates the position in the list of quantifiers (1 = first). For example

$$p(X, Y, *2\{a, b\}).$$

will be interpreted as:

$$\forall_X \exists_{A\in\{a,b\}} \forall_Y p(X, Y, A).$$

4. If no variable is present there is no difference in interpretation and once again the default format will be used. For example

$$manager(john, 60\text{–}65).$$
$$suspect(5, \{brown, black\}, 57\text{–}60).$$

3. Positive and negative literals

The occurrence of an indefinite constant such as $\{a, b\}$ in the head of a clause (i.e. a positive literal) e.g.

$$is_a(X, \{staff, student\}).$$

has already been discussed and shown to be interpreted as:

$$\forall_X \exists_{Y\in\{staff,\, student\}} is_a(X, Y).$$

However, consider the interpretation to be given to an indefinite constant occurring in the body of a clause (i.e. in a negative literal). To illustrate this consider the rule

$$p(X) \leftarrow q(X, \{a, b\}).$$

i.e.

$$p(X) \vee \neg q(X, \{a, b\}).$$

Assuming that all quantifiers have been moved to the beginning of a clause, there are two possible interpretations of this rule, viz.

(a) $\exists_{U\in\{a,b\}}\forall_X(p(X)\leftarrow q(X, U))$.

i.e.

$$[\forall_X(p(X)\leftarrow q(X, a))]\vee[\forall_Y(p(Y)\leftarrow q(Y, b))].$$

In other words two separate rules are possible and although one of these rules is known to exist, it is not known which it is. Thus in order to prove that the clause p is definitely true one has to prove that p is true for each rule separately.

(b) $\forall_X\exists_{U\in\{a,b\}}(p(X)\leftarrow q(X, U))$

i.e.

$$\forall_X[(p(X)\leftarrow q(X, a))\vee(p(X)\leftarrow q(X, b))].$$

Using truth tables this can be shown to be equivalent to

$$\forall_X(p(X)\leftarrow q(X, a)\wedge q(X, b)).$$

In other words a single rule exists from which $p(X)$ can be concluded only if $q(X, a)$ and $q(X, b)$ both succeed.

The difference between these two interpretations can be illustrated in the context of the following database:

$$p(X)\leftarrow q(X, \{a, b\})$$
$$q(c, a)$$
$$q(d, b).$$

Using the first interpretation the set of clauses consists either of

$$p(X)\leftarrow q(X, a)$$
$$q(c, a)$$
$$q(d, b).$$

which will succeed with $X = c$, or of

$$p(X)\leftarrow q(X, b)$$
$$q(c, a)$$
$$q(d, b).$$

which will succeed with $X = d$. As a result the query as a whole succeeds with $X = \{c, d\}$. However, if one applies the second interpretation, the same X must satisfy

$$p(X)\leftarrow q(X, a) \quad \text{and}$$
$$p(X)\leftarrow q(X, b)$$

with the result that in this case the query will fail.

As an example, consider the rule

$$eligible(X) \leftarrow qualifications(X, \{a_level, highers\}) .$$

The interpretation of this rule following the above line of reasoning is that someone is eligible provided that that person has a suitable qualification. However, it is not certain at this stage whether a-level (English school qualification) constitutes an acceptable qualification, or whether highers (Scottish school qualification) is that necessary qualification. In this case, the second interpretation is intended, i.e. in order to prove that any particular individual is definitely eligible, one must show that the person concerned has both a-level and highers qualifications.

Another possible condition which one might wish to express is that if any one of a set of options is satisfied then the conclusion can be drawn. For example, if either highers or a-level qualifications are sufficient to ensure eligibility, one might express this as:

$$eligible(X) \leftarrow qualifications(X, a_level) \vee qualifications(X, highers)$$

or as a pair of implications without disjunction:

$$eligible(X) \leftarrow qualifications(X, a_level)$$
$$eligible(X) \leftarrow qualifications(X, highers) .$$

However, as this case is already in Horn clause form it is not necessary to consider it further.

If more than one variable is present, more interpretations are possible. Once again assume that all quantifiers have been moved to the beginning of the clause and that all universal quantifiers occur in the order that the corresponding variables appear for the first time within the clause. Consider the following example involving two variables:

$$p(X, Y) \leftarrow q(X, Y, \{a, b\}) .$$

In this case there are three possible interpretations, viz.

(a) $\exists_{U \in \{a,b\}} \forall_X \forall_Y (p(X, Y) \leftarrow q(X, Y, U))$
$\equiv [\forall_X \forall_Y (p(X, Y) \leftarrow q(X, Y, a))] \vee [\forall_A \forall_B (p(A, B) \leftarrow q(A, B, b))] .$

In other words two separate rules are possible.

(b) $\forall_X \exists_{U \in \{a,b\}} \forall_Y (p(X, Y) \leftarrow q(X, Y, U))$
$\equiv \forall_X (\forall_Y (p(X, Y) \leftarrow q(X, Y, a)) \vee \forall_Z (p(X, Z) \leftarrow q(X, Z, b))) .$

(c) $\forall_X \forall_Y \exists_{U \in \{a,b\}} (p(X, Y) \leftarrow q(X, Y, U))$
$\equiv \forall_X \forall_Y ((p(X, Y) \leftarrow q(X, Y, a)) \vee (p(X, Y) \leftarrow q(X, Y, b))) .$

As with facts we will assume that if no special annotation precedes an argument, then the default assumption is that the corresponding existential quantifier will follow all universal quantifiers. For example,

$$eligible(X) \leftarrow qualifications(X, \{a_level, highers\}) .$$

is equivalent to

$$\forall_X \exists_{Y \in \{a_level, highers\}} (eligible(X) \leftarrow qualifications(X, Y)) .$$

If an existential quantifier should precede all universal quantifiers, the argument concerned is preceded by an asterisk. For example

$$p(X, Y) \leftarrow q(X, Y, {}^*\{a, b\}) .$$

is equivalent to

$$\exists_{U \in \{a,b\}} \forall_X \forall_Y (p(X, Y) \leftarrow q(X, Y, U)) .$$

If an instance of an existential quantifier should occur in the nth position in the list of quantifiers (between the leftmost and rightmost universal quantifiers) then the corresponding argument should be preceded by $^*n$. For example,

$$p(X, Y) \leftarrow q(X, Y, {}^*2\{a, b\}) .$$

will be interpreted as:

$$\forall_X \exists_{U \in \{a,b\}} \forall_Y (p(X, Y) \leftarrow q(X, Y, U)) .$$

4. Queries and resolution

Suppose that one has a set of clauses $S = \{C_1, C_2\}$ where

$$C_1 = r \vee \neg p(X_1, \ldots, X_n)$$
$$C_2 = p(Y_1, \ldots, Y_n)$$

and the query

$$? - r$$

i.e. one wants to prove r from S. Using the resolution principle of Robinson [6], one can achieve this by proving that $S' = S \cup \{\neg r\}$ is unsatisfiable, i.e. that the empty clause $\square$ can be deduced from S'. Such a proof may proceed as follows:

(1)	$\neg r$	from S'
(2)	$r \vee \neg p(X_1, \ldots, X_n)$	from S
(3)	$p(Y_1, \ldots, Y_n)$	from S
(4)	$\neg p(X_1, \ldots, X_n)$	from (1), (2)

Now if a most general unifier [1] θ can be found such that when it is applied to the literals $\neg p(X_1, \ldots, X_n)$ and $p(Y_1, \ldots, Y_n)$ one obtains the complementary pair $\neg p(A_1, \ldots, A_n)$ and $p(A_1, \ldots, A_n)$ where $(p(A_1, \ldots, A_n) = p(X_1, \ldots, X_n)\theta = p(Y_1, \ldots, Y_n)\theta$ then from (3) and (4) one can deduce the empty clause.

This is the situation for a database in which each argument or attribute is either a variable or a constant representing a complete value. However, if one allows constants representing indefinite data, the task becomes more complicated. This leads to the questions:

1. How is the process of proof by refutation affected if a query contains incomplete data?
2. What will be the answer produced by a query in an incomplete system?

In a later section we will distinguish between two different types of query: definite queries and possible queries. However, before introducing this distinction we will concentrate on the first of these two types which corresponds to the normal process of logical deduction.

A query of the form

$$? - p(X)$$

can be translated as

$$\neg \exists_X p(X)$$

or equivalently

$$\forall_X \neg p(X)$$

and added to the database for resolution against the clauses in the database.

Following this interpretation, the query

$$? - p(\{a, b\})$$

will be translated as

$$\neg(\exists_{U \in \{a,b\}} p(U))$$

or

$$\neg(p(a) \vee p(b))$$

which is equivalent to

$$\neg p(a) \wedge \neg p(b)$$

and adding this to the database will give rise to the two clauses

$$\neg p(a)$$
$$\neg p(b)$$

Inconsistency can be established if any of the following three clauses is present in the database:

$$p(a)$$
$$p(b)$$
$$p(\{a, b\})$$

Thus the query $? - p(\{a, b\})$ will succeed for set S if $p(a)$ is in S ($p(a)$ is known to be true)

or $p(b)$ is in S ($p(b)$ is known to be true) or $p(\{a, b\})$ is in S ($p(X)$ is true where X is either a or b but it is not known which).

If variables are present in a query containing incomplete information, the same set of interpretations are possible as for facts and rules. Thus the following three cases arise:

1. If one or more variables are present in a query and no special annotation precedes an incomplete constant, the existential quantifier for the constant will follow the existential quantifiers of the variables. Thus

$$? - p(X, \{a, b\}, Y)$$

will be translated as

$$\neg\exists_X\exists_Y\exists_{U\in\{a,b\}} p(X, U, Y)$$

or

$$\forall_X\forall_Y\neg(\exists_{U\in\{a,b\}} p(X, U, Y))$$

or

$$\forall_X\forall_Y\neg(\, p(X, a, Y) \vee p(X, b, Y))\,.$$

2. If one or more variables are present in a query and an incomplete constant is preceded by an asterisk "*", the existential quantifier for the constant will precede the quantifiers of the variables. Hence

$$? - p(X, {}^*\{a, b\}, Y)$$

may be interpreted as

$$\neg\exists_{U\in\{a,b\}}\exists_X\exists_Y p(X, U, Y)$$

which is equivalent to

$$\neg(\exists_X\exists_Y p(X, a, Y) \vee \exists_A\exists_B p(A, b, B))$$

or

$$\forall_X\forall_Y\neg p(X, a, Y) \wedge \forall_A\forall_B\neg p(A, b, B\ )\,.$$

3. If variables are present in a query and an existential quantifier corresponding to an incomplete constant should occur somewhere in between the leftmost and the rightmost quantifiers corresponding to variables, then the argument concerned will be preceded by $*n$ where n indicates the position in the list of quantifiers. For example

$$? - p(X, {}^*2\{a, b\}, Y)$$

will be interpreted as

$$\neg\exists_X\exists_{U\in\{a,b\}}\exists_Y p(X, U, Y)$$

or

$$\forall_X\neg(\exists_Y p(X, a, Y) \vee \exists_Z p(X, b, Z))$$

which is equivalent to

$$\forall_X(\forall_Y \neg p(X, a, Y) \wedge \forall_Z \neg p(X, b, Z)) .$$

In this case, however, the order of existential quantifiers can be changed without changing the meaning of the clause. Thus

$$\neg \exists_{U \in \{a,b\}} \exists_X \exists_Y p(X, U, Y)$$

can be transformed to

$$\neg \exists_X \exists_{U \in \{a,b\}} \exists_Y p(X, U, Y)$$

or to

$$\neg \exists_X \exists_Y \exists_{U \in \{a,b\}} p(X, U, Y) .$$

Consequently, the query

$$? - p(X, \{a, b\}, Y)$$

is equivalent to

$$? - p(X, {}^*\{a, b\}, Y)$$

and to

$$? - p(X, {}^*2\{a, b\}, Y)$$

and hence one can ignore the latter two forms and consider only incomplete constants without special annotation.

5. Model interpretations

One way of visualizing the ideas of incomplete information is through models. Following this approach a ground fact such as $p(x_1)$ is represented as $p(x_1)$ and generates a single model (*Fig. 1a*). A fact involving a variable, e.g. $p(X)$, will be represented as a collection of facts, one for each value of the variable, once again constituting a single model (*Fig. 1b*). An incomplete constant $\{x_1, x_2\}$ will give rise to separate models, one for each value in the constant; an example for the case $p(\{x_1, x_2\})$ is illustrated in *Fig. 1c*.

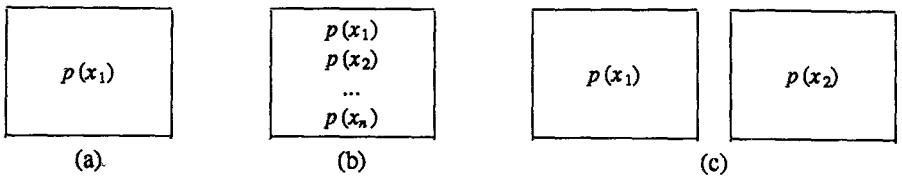


Fig. 1. Model representations of (a) $p(x_1)$, (b) $p(X)$, (c) $p(\{x_1, x_2\})$.

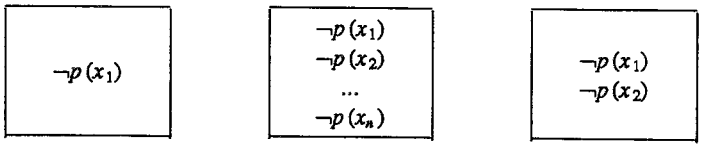


Fig. 2. Model representations of (a) $? - p(x_1)$, (b) $? - p(X)$, (c) $? - p(\{x_1, x_2\})$.

A query can be represented in a similar fashion. For example, a ground query such as $? - p(x_1)$ is represented by a single model in which the term $p(x_1)$ is negated (*Fig. 2a*). The query $? - p(X)$ is expanded into a whole set of queries, one for each value of X, each of which is negated (*Fig. 2b*). The query $? - p(\{x_1, x_2\})$ is equivalent to

$$? - p(x_1) \vee p(x_2)$$

and when negated becomes

$$\neg p(x_1) \wedge \neg p(x_2) .$$

This is represented as a single model with the two subqueries as separate terms (*Fig. 2c*).

The application of the query $? - p(x_1)$ to the database

$$p(x_1)$$

corresponds to merging the model in *Fig. 2a* with that in *Fig. 1a* as shown in *Fig. 3a*. In this case the terms $\neg p(x_1)$ and $p(x_1)$ can be resolved to yield the empty clause. Likewise $? - p(x_1)$ can be applied to the database

$$p(X)$$

by merging *Fig. 2a* with *Fig. 1b* (see *Fig. 3b*), once again enabling resolution to produce the empty clause. If $? - p(x_1)$ is applied to the database consisting of

$$p(\{x, x_2\})$$

the model in *Fig. 2a* is merged with each of the models in *Fig. 1c* as shown in *Fig. 3c*. In this case one of the models will yield the empty clause but the other will not. This corresponds to the case where one can neither prove nor disprove the query – the result is unknown.

Likewise if the query $? - p(X)$ is applied to each of the above three cases, the model in *Fig. 2b* is merged with each of the models in *Fig. 1*. In this case all three will succeed. The same is true of the query $? - p(\{x_1, x_2\})$.

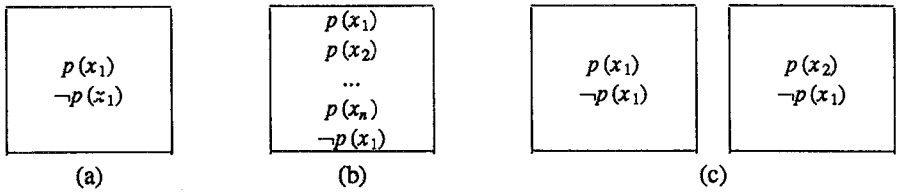


Fig. 3. Applying the query $? - p(x_1)$ to (a) $p(x_1)$, (b) $p(X)$, (c) $p(\{x_1, x_2\})$.

Now consider the different classes of incomplete information arising in facts. First, by way of contrast, the model corresponding to the fact $q(X, a, Y)$ is given in *Fig. 4a*. This consists of a single model. The second example is that of $q(X, \{a, b\}, Y)$ which has been shown to be equivalent to

$$\forall_X \forall_Y (q(X, a, Y) \vee q(X, b, Y)) .$$

As with $q(X, a, Y)$, the clause is expanded to give a set of clauses with each possible combination for the values of X and Y. However, in this case a separate model is created for each possible assignment of a or b to the second argument position of the set of clauses. The result is shown in *Fig. 4b*. Assuming that both X and Y can take n possible values, the number of models generated by this clause is 2^{n^2}.

The clause $q(X, {}^*\{a, b\}, Y)$ is equivalent to

$$\forall_X \forall_Y q(X, a, Y) \vee \forall_A \forall_B q(A, b, B) .$$

In this case there are only two models – the one in which the second argument is a for all possible combinations of the values of X and Y, and the other in which the second argument is b. This case is shown in *Fig. 4c*. The final case, $q(X, {}^*2\{a, b\}, Y)$ is equivalent to

$$\forall_X (\forall_Y q(X, a, Y) \vee \forall_Z q(X, b, Z))$$

and is intermediate between the previous two cases. Here all possible combinations of a or b for different values of the first variable are provided, while values of the second variable are taken together as a set (*Fig. 4d*). And again if X can assume n different values, this case will comprise 2^n models.

The same approach can be applied to rules. An example of the simplest type of rule is

$$p(X, Y) \leftarrow q(X, a, Y) .$$

This is represented in a single model by expanding it for all possible combinations of values for X and Y as shown in *Fig. 5a*. The rule

$$p(X, Y) \leftarrow q(X, \{a, b\}, Y)$$

which has been shown to be equivalent to

$$\forall_X \forall_Y (p(X, Y) \leftarrow q(X, a, Y) \vee (p(X, Y) \leftarrow q(X, b, Y)))$$

is treated in the same way as the corresponding fact example, i.e. by expansion in terms of all possible combinations of values for X and Y and creation of a separate model for each possible assignment of a or b to the second argument position of the set of clauses (see *Fig. 5b*). This gives a model which is consistent with the corresponding case for facts (*Fig. 4b*). However, it was noted that the rule

$$p(X, Y) \leftarrow q(X, \{a, b\}, Y)$$

is also equivalent to

$$\forall_X \forall_Y (p(X, Y) \leftarrow q(X, a, Y) \wedge q(X, b, Y)) .$$

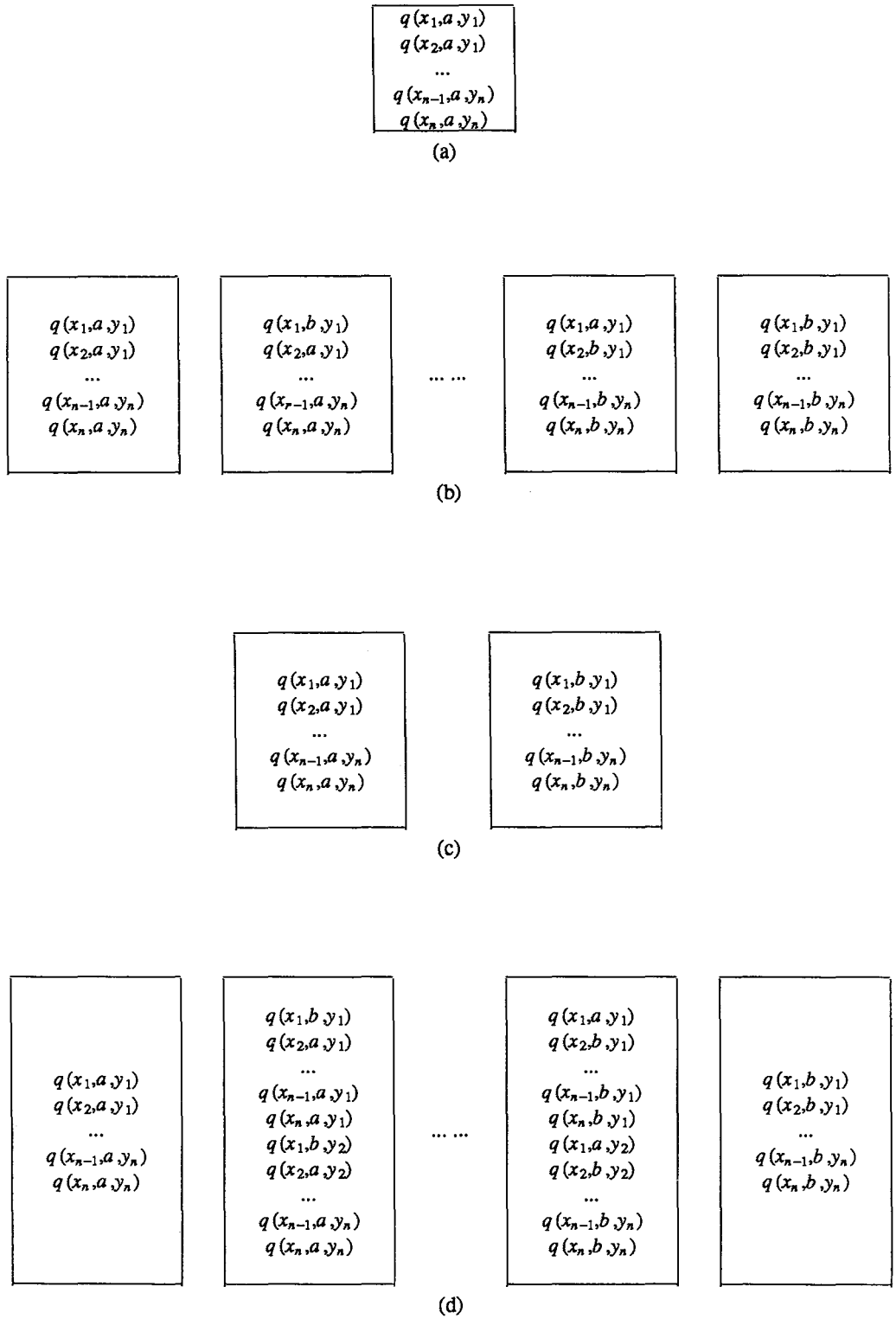


Fig. 4. Model representations of (a) $q(X, a, Y)$ (b) $q(X, \{a, b\}, Y)$ (c) $q(X, {}^*\{a, b\}, Y)$ (d) $q(X, {}^*2\{a, b\}, Y)$.

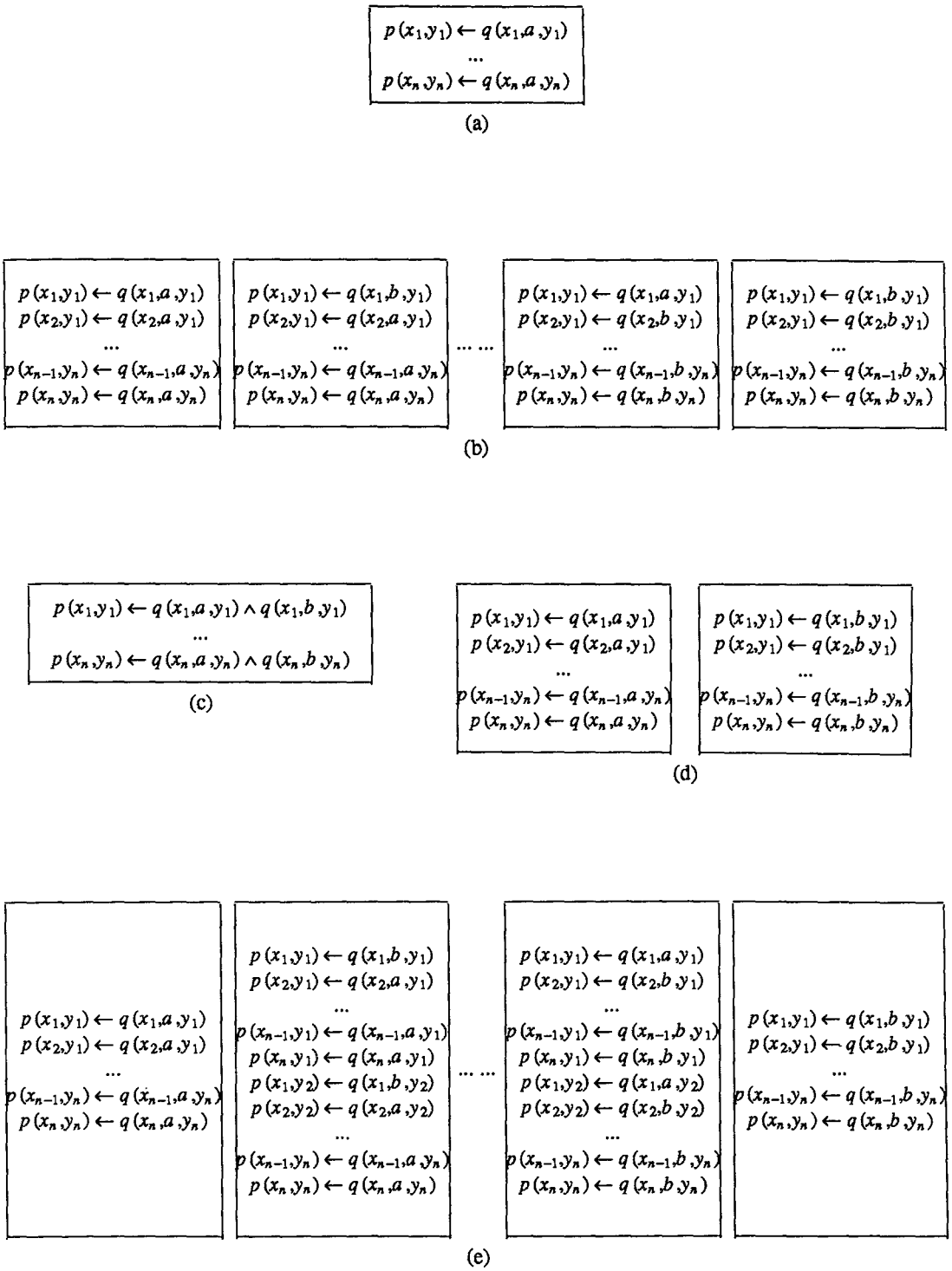


Fig. 5. Model representations of (a) $p(X, Y) \leftarrow q(X, a, Y)$, (b) $p(X, Y) \leftarrow q(X, \{a, b\}, Y)$, (c) $p(X, Y) \leftarrow q(X, \{a, b\}, Y)$ (alternative representation), (d) $p(X, Y) \leftarrow q(X, *\{a, b\}, Y)$, (e) $p(X, Y) \leftarrow q(X, *2\{a, b\}, Y)$.

This results in a single model containing all possible expansions of the rule for different combinations of values for X and Y, as illustrated in *Fig. 5c*. Although the latter appears to be rather different from *Fig. 5b*, the two are equivalent from the viewpoint of resolution. For example, if one wants to prove $p(x_i, y_i)$ from *Fig. 5b*, one needs to prove it in each model separately. Half of the models will enable $p(x_i, y_i)$ to be concluded if $q(x_i, a, y_i)$ is true, the other half if $q(x_i, b, y_i)$ is true. Thus in order to conclude $p(x_i, y_i)$, both $q(x_i, a, y_i)$ and $q(x_i, b, y_i)$ must be true. This is what is stated in *Fig. 5c*.

The rule

$$p(X, Y) \leftarrow q(X, {}^*\{a, b\}, Y)$$

is equivalent to

$$\forall_X \forall_Y (p(X, Y) \leftarrow q(X, a, Y)) \vee \forall_A \forall_B (p(A, B) \leftarrow q(A, b, B))$$

and as for the equivalent case for facts is represented by a pair of models as shown in *Fig. 5d*. Finally, the rule

$$p(X, Y) \leftarrow q(X, {}^*2\{a, b\}, Y)$$

which is equivalent to

$$\forall_X (\forall_Y (p(X, Y) \leftarrow q(X, a, Y)) \vee \forall_Z (p(X, Z) \leftarrow q(X, b, Z)))$$

is expanded to create separate models for all possible combinations of a or b for different values of the first variable, X, while values of the second variable are taken together as a set, as demonstrated in *Fig. 5e*.

Queries containing incomplete information are handled in a natural extension to the representations given in *Fig. 2*. Thus the query

$$? - q(X, a, Y)$$

can be regarded as a set of queries of the form

$$\forall_X \forall_Y \neg q(X, a, Y)$$

and is represented as a single model as shown in *Fig. 6a*. The query

$$? - q(X, \{a, b\}, Y)$$

is equivalent to a set of queries of the form

$$\forall_X \forall_Y \neg (q(X, a, Y) \vee q(X, b, Y))$$

and this reduces to a single model as illustrated in *Fig. 6b*. On the other hand, the query

$$? - q(X, {}^*\{a, b\}, Y)$$

which is equivalent to

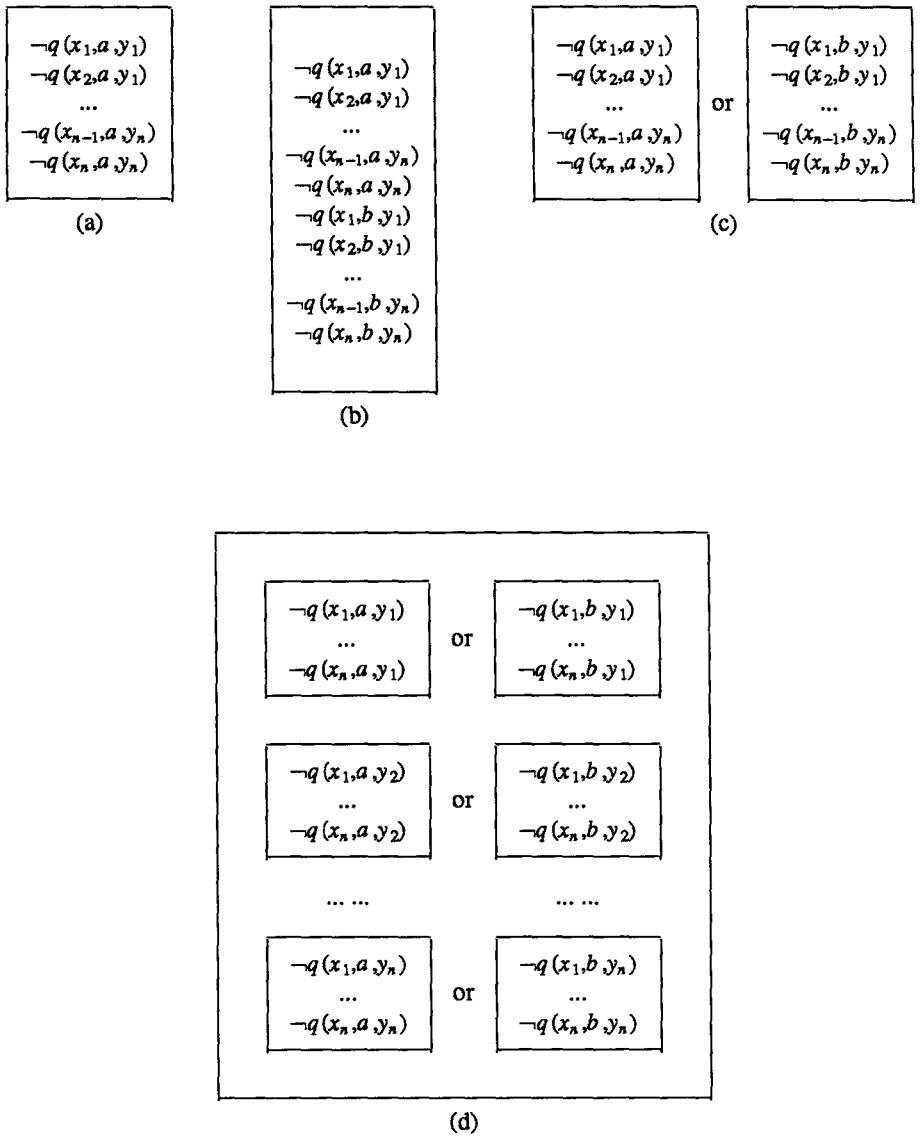


Fig. 6. Model representations of (a) $? - q(X, a, Y)$, (b) $? - q(X, \{a, b\}, Y)$, (c) $? - q(X, *\{a, b\}, Y)$, (d) $? - q(X, *2\{a, b\}, Y)$.

$$\forall_X \forall_Y \neg q(X, a, Y) \wedge \forall_U \forall_V \neg q(U, b, V)$$

corresponds to the pair of models shown in *Fig. 6c*. The query

$$? - q(X, *2\{a, b\}, Y)$$

which can be interpreted as

$$\forall_X (\forall_Y \neg q(X, a, Y) \wedge \forall_Z \neg q(X, b, Z))$$

is once again intermediate between the previous two cases and reduces to a single model in which clauses are grouped together into subsets. Each subset of clauses has an alternative subset associated with it.

As before, if a query is applied to a database, the corresponding models are merged. Thus if the query $? - q(X, a, Y)$ (*Fig. 6a*) is applied to each of the examples given in *Fig. 4*, it will succeed for the first case but will be unknown for each of the remaining cases since the empty clause is not derivable from the model in which the second parameter of each clause is b but is derivable from all of the other models.

If the query

$$? - q(X, \{a, b\}, Y)$$

which is represented by the model in *Fig. 6b*, is applied to each of the cases in *Fig. 4*, it will succeed in every case. The query

$$? - q(X, {}^*\{a, b\}, Y)$$

is represented by two alternative models. When this is applied to a database each database model is merged separately with the two alternative query models, if either of the resulting models succeeds, the query as a whole succeeds for that database model. Thus this query will succeed for all four cases given in *Fig. 4*. Since the query

$$? - q(X, {}^*2\{a, b\}, Y)$$

is represented in a way which is intermediate between the previous two cases, it can easily be verified that it too will succeed for all four cases shown in *Fig. 4*.

This confirms the result that $\{a, b\}$, $^*\{a, b\}$ and $^*2\{a, b\}$ all produce the same effect in a query containing variables.

6. Possible and definite queries

One further complication in interpreting queries lies in the certainty which one associates with the answer. Thus, in posing a query

$$? - p(a)$$

does one want to know whether $p(a)$ is definitely true or whether $p(a)$ could possibly be true (i.e. is definitely not false)? Lipski [5] identified these two limiting interpretations as:

(a) the lower value of Q, $\|Q\|_*$, which is the set of objects which from the information available, cannot fail to satisfy the query Q, and

(b) the upper value of Q, $\|Q\|^*$, which is the set of objects which from the information available, could possibly satisfy the query Q.

If a query is to be applied to a database containing incomplete information and it is not made clear whether the lower or upper value of the query is required, the query is ambiguous. This problem can be overcome by either declaring such a query to be invalid or by assuming that it is to be interpreted in a particular way, e.g. as lower value.

Lipski's queries were formulated in terms of comparisons at the attribute level. For example, given a database with complete information containing the relations *bloodgroup*(*Patient*, *Bloodgp*) and *description*(*patient*, *Colour_of_eyes*, *Colour_of_hair*), if one wished to find all patients who are definitely known to have blood group AB and blue eyes, one might express the condition part of such a query as

$$Bloodgp = AB \wedge Colour_of_eyes = blue\ . \tag{Q1}$$

However, this paper will adhere to a notation based on clauses and hence such a query will be rewritten as

$$?- bloodgroup(X, B) \wedge description(X, C, Y) \wedge B = ab \wedge C = blue$$

or more simply as:

$$?- bloodgroup(X, ab) \wedge description(X, blue, Y)\ .$$

For a database with incomplete information one must specify whether one wants the upper or lower value. For this purpose two functions will be used: *pos* (standing for possible) and *def* (standing for definite). These two functions can be applied either to a whole query or to the arguments of a query (individually or in an expression). When the function *def* (or *pos*) is applied to a query it states that the query requires an answer which is definitely (or possibly) true in the database. For example, if a query which seeks any patient who has blood group AB, is preceded by the *def* function

$$?- def(bloodgroup(X, ab))$$

the database returns those values which definitely satisfy the query, whereas if it is preceded by the *pos* function

$$?- pos(bloodgroup(X, ab))$$

the database returns all possible answers. If the functions are applied to the arguments of a query, such a query is interpreted as requiring an answer which satisfies the conditions specified by the functions. Thus if one wants to find any patient who definitely has blood group AB and blue eyes, one may formulate a query as

$$?- bloodgroup(X, def(ab)) \wedge description(X, def(blue), W)$$

or

$$?- bloodgroup(X, Y) \wedge description(X, Z, W) \wedge def(Y = ab \wedge Z = blue)\ .$$

This is equivalent to Lipski's

$$\|Bloodgp = AB \wedge Colour_of_eyes = blue\|_*\ .$$

Similarly if one wants any patient who possibly has blood group AB and blue eyes, one can formulate this as

$$?- bloodgroup(X, pos(ab)) \wedge description(X, pos(blue), W)\ .$$

Using this approach one might even combine *pos* and *def* functions within a query. For example, if one wants to determine all patients who are definitely known to have blood group AB and may possibly have blue eyes, one might write

$$? - bloodgroup(X, def(ab)) \wedge description(X, pos(blue), W)$$

The semantics of *pos* and *def* functions applied to the arguments can be related to upper and lower values as follows:

$p(a_1, \ldots, a_{i-1}, def(b), a_{i+1}, \ldots, a_n)$ is equivalent to
$p(a_1, \ldots, a_{i-1}, X, a_{i+1}, \ldots, a_n) \wedge def(X = b)$ or in Lipski's notation $\|X = b\|_*$

$p(a_1, \ldots, a_{i-1}, pos(b), a_{i+1}, \ldots, a_n)$ is equivalent to
$p(a_1, \ldots, a_{i-1}, X, a_{i+1}, \ldots, a_n) \wedge pos(X = b)$ or in Lipski's notation $\|X = b\|^*$

where X does not occur in $a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n$.

As discussed in Section 4, one may wish to use incomplete information within the arguments of a query. For example, one may wish to find any patient whose blood group is definitely either A or O but it does not matter which it is. Such a query may be written as:

$$? - bloodgroup(X, Y) \wedge def(Y = a \vee Y = o)$$

or

$$? - bloodgroup(X, def(\{a, o\})) .$$

7. Query evaluation

The previous sections concentrated on the interpretations to be given to incomplete information arising in facts, rules and queries. This leads naturally to the question of *how to perform query evaluation in a system containing incomplete information* or more specifically *what the answer would be to a query in such a system.*

Before discussing this, the term *complete* database will be used to refer to a database comprising a set of clauses which contain no incomplete information or disjunctions. Likewise an *incomplete* database is one which does contain incomplete information or disjunctions.

As noted earlier a clause containing an incomplete constant for some attribute A may be regarded as being equivalent to a set of different possible clauses, each with a complete constant for A. In turn this may be viewed as giving rise to a set of complete databases, each containing one instance of the clause with a complete constant for A. If there is more than one clause containing incomplete constants, then the set of possible databases must include every permutation of clauses with complete constants. For example, a database DB containing the single clause $p(\{a, b\}, c)$, i.e.

$$DB: p(\{a, b\}, c)$$

is equivalent to a database with a disjunctive clause

$$DB: p(a, c) \vee p(b, c)$$

which can, in turn, be regarded as a set of alternative complete databases

$$DB_1 : p(a, c)$$
$$DB_2 : p(b, c)$$

This means that either one has a database DB_1 which contains clause $p(a, c)$ or one has database DB_2 which contains clause $p(b, c)$.

If one views an incomplete database as a set of complete databases, then in each alternative database a query can be evaluated using a resolution based query evaluation mechanism. Thus in order to prove that a query is definitely true in the database (DB) one must prove that it is true in all alternative databases (DB_1 and DB_2). If such a query contains variables then each evaluation in the alternative databases will give rise to a ground substitution of the variables, and the combination of all such ground substitutions constitutes an answer to the query. In order to prove that a query is possibly true in the database one needs to prove that it is true in at least one of the alternative databases (DB_1 or DB_2). If this type of query contains variables then the ground substitution of the variables in any alternative database constitutes an answer to the query.

If the query itself contains incomplete data, it may be regarded as a set of alternative queries. If any one of these queries can be proved from the database then the whole query is proved.

These ideas can be formalised to give a set of answer expressions. When an exact query Q is applied to a complete database DB, an answer is a substitution

$$\theta = \{t_1/v_1, t_2/v_2, \ldots, t_m/v_m\}$$

in which each t_i is a term, each v_i a variable and $v_i \neq v_j$ if $i \neq j$, such that

$$DB \vdash Q\theta$$

if either the query or the database contains incomplete information then the answer may be a set of alternative substitutions

$$\theta_1 + \theta_2 + \cdots + \theta_n$$

such that

$$DB \vdash Q\theta_1 \vee Q\theta_2 \vee \cdots \vee Q\theta_n$$

Now consider the answer to a query, Q, applied to a database, DB. Let θ be a ground substitution of the variables, and $Ans(Q)$ be an answer set of Q. The answer to a query can be given as follows.

(a) In the case of a complete database, it does not make any difference whether or not the query is prefixed by *pos* or *def*, since the two are equivalent and will yield the same answer, i.e.

$$Ans(Q) = Ans(def(Q)) = Ans(pos(Q)).$$

The answer is defined as:

1. For a single query Q, the answer set is:

$$Ans(Q) = \{\theta \mid DB \vdash Q\theta\} .$$

2. If the query is a disjunction $Q_1 \vee Q_2$, then the answer is:

$$Ans(Q_1 \vee Q_2) = \{\theta \mid DB \vdash Q_1\theta \text{ or } DB \vdash Q_2\theta\} .$$

(b) If the database is incomplete, the two forms of query ($pos(Q)$ and $def(Q)$) may lead to different answer sets. Let $A \vee B$ represent a disjunctive clause (obtained by rewriting a clause containing incomplete data) in the database DB, and let Δ represent the remaining clauses in the database, i.e. $DB = \Delta + \{A \vee B\}$. Let DB_1 denote $\Delta + \{A\}$ and DB_2 denote $\Delta + \{B\}$. Then

1. The answer to a single query preceded by a *def* function $def(Q)$ is:

$$Ans(def(Q)) = \{\theta \mid DB \vdash Q\theta\} \cup \{\theta_1 + \theta_2 \mid DB_1 \vdash Q\theta_1 \text{ and } DB_2 \vdash Q\theta_2\} .$$

2. The answer to a disjunctive query preceded by a *def* function $def(Q_1 \vee Q_2)$ is:

$$Ans(def(Q_1 \vee Q_2)) = \{\theta \mid DB \vdash Q_1\theta \text{ or } DB \vdash Q_2\theta\} \cup \{\theta_1 + \theta_2 \mid DB_1 \vdash Q_1\theta_1 \text{ and } DB_2 \vdash Q_2\theta_2\} .$$

3. The answer to a single query preceded by a *pos* function $pos(Q)$ is:

$$Ans(pos(Q)) = \{\theta \mid DB_1 \vdash Q\theta \text{ or } DB_2 \vdash Q\theta\}$$

4. The answer to a disjunctive query preceded by a *pos* function $pos(Q_1 \vee Q_2)$ is:

$$Ans(pos(Q_1 \vee Q_2)) = \{\theta \mid DB_1 \vdash Q_1\theta \text{ or } DB_1 \vdash Q_2\theta \text{ or } DB_2 \vdash Q_1\theta \text{ or } DB_2 \vdash Q_2\theta\}$$

This approach may be extended to handle the case where the functions *def* and *pos* are applied to individual arguments of a query. For example, queries can have the form

$$?- q(X, def(\{a, o\}), def(b)) ,$$
$$?- q(X, pos(\{a, o\}), def(b)), \text{ or}$$
$$?- q(X, def(a), pos(b)) .$$

Using the same database as above, the latter query will be interpreted as:

$$Ans(q(X, def(a), pos(b))) = \{t/X \mid DB \vdash q(t, a, b)\} \cup \{t_1/X + t_2/X \mid DB_1 \vdash q(t_1, a, C_1) \text{ and } DB_2 \vdash q(t_2, a, C_2) \text{ and } (C_1 = b \vee C_2 = b)\} .$$

To illustrate these ideas consider the following example database. Suppose that the database contains four relations and one rule. The relation *patient*(*Name*, *Age*, *Sex*, *Blood_group*) specifies the name, age, sex and blood group of each patient and has the following data:

patient(*tom*, 51, *m*, *a*) .

patient(*john*, {51, 52}, *m*, {*a*, *ab*}) .

patient(*tim*, 51, *m*, {*a*, *o*}) .

patient(*peter*, 56, *m*, *o*) .

The relation *symptom*(*Patient*, *Symptom*) specifies the symptoms pertaining to each patient, and consists of

symptom(*tom*, *headache*) .

symptom(*john*, *chest_pain*) .

symptom(*tim*, *chest_pain*) .

symptom(*peter*, *chest_pain*) .

The relation *sign*(*Patient*, *Sign*) specifies the evidence of the patient's symptom and has the following information:

sign(*john*, *haemoptysis*) .

sign(*tim*, {*haemoptysis*, *cough*}) .

sign(*peter*, *haemoptysis*) .

The relation *pathogeny*(*Patient*, *Causative_agent*) lists the possible causative agents to which each patient has been exposed and has the following records:

pathogeny(*john*, *radiation*) .

pathogeny(*tim*, *smoking*) .

pathogeny(*tim*, *radiation*) .

pathogeny(*peter*, *smoking*) .

pathogeny(*peter*, *radiation*) .

When possible causes of lung cancer were being investigated in the fifties a number of possible causative agents were identified including smoking and exposure to radiation. It was known[†] that one of these agents caused lung cancer but it was not known which. Thus if it was known that a person had been exposed to all such agents then it was certain that he had been exposed to the cause of lung cancer. For purpose of illustration only two agents will be considered, namely smoking and exposure to radiation. Thus the rule *diagnosis*(*Name*, *lung_cancer*) states that the diagnosis for a given patient is likely to be carcinoma of the lung if he is more than 50 years of age, complains of severe chest pain, has coughed up blood (haemoptysis) and has a history of exposure to some possible causative agent which might be cigarette smoking or radioactive fallout from atomic tests, etc. It can be expressed as:

[†] As the number of agents under investigation was very large, it seems reasonable to assume that this was true.

diagnosis(*Name*, *lung_cancer*):-
patient(*Name*, *Age*,_,_), *Age* > 50 ,
symptom(*Name*, *chest_pain*) ,
sign(*Name*, *haemoptysis*) ,
pathogeny(*Name*, {*smoking*, *radiation*}) .

A set of queries which might be applied to this database is as follows:

Q1: A definite query

$$? - def(\, patient(X, 51, _, a))$$

seeks any patient who is definitely known to be 51 years old and has blood group A. The answer to this query is Tom.

Q2: A possible query

$$? - pos(\, patient(X, 51, _, a))$$

seeks any person who is possibly 51 years old and has blood group A. The answer to this query contains the three individuals Tom, John and Tim.

Q3: A query having the form

$$? - patient(X, def(51), _, pos(a))$$

seeks any person who is definitely 51 years old and may possibly have blood group A. According to the records in the database the answer might be Tom and Tim.

Q4: A definite query having the form

$$? - def(diagnosis(X, lung_cancer))$$

seeks any patient who definitely has a diagnosis of lung_cancer. The answer to this is Peter.

Q5: A possible query

$$? - pos(diagnosis(X, lung_cancer))$$

seeks any patient who is diagnosed as possibly having lung_cancer. The answer to this is Peter, Tim and John.

8. Conclusion

The way in which incomplete information can be handled in a relational database has been dealt with by Lipski. This paper explores how this idea can be taken a step further and incorporated into a deductive database.

The first observation to be made concerns the different interpretations of incomplete information in a deductive database. By introducing a notation for describing different categories, the various possible interpretations of incomplete information arising in facts and rules can be catered for. The effects of this on queries is discussed and the need for two types of query described. The strategy for query evaluation and the answer to queries in such a system have also been described. The implementation of query evaluation is now under investigation and some details have been given in [7].

Acknowledgements

The authors would like to thank Chris Miller and Gang Chen for helpful discussions and suggestions on earlier drafts of this paper. We also would like to thank the referees and Professor Dov Gabbay of Imperial College for their helpful comments.

References

[1] C.L. Chang and R.C.T. Lee, Symbolic Logic and Mechanical Theorem Proving (Academic Press, New York, N.Y., 1973).
[2] E.F. Codd, A relational model of data for large shared data banks, Communication ACM 13 (1970) 377–387.
[3] E.F. Codd, Extending the database relational model to capture more meaning, ACM Trans. on Database Systems 4 (1979) 397–434.
[4] W. Lipski, Informational systems with incomplete information, in: S. Michaelson and R. Milner (ed.), Proc. 3rd Int. Symp. on Automata, Languages and Programming (Edinburgh University Press, Edinburgh, 1976) 120–130.
[5] W. Lipski, On semantic issues connected with incomplete information databases, ACM Trans. on Database Systems 4 (1979) 262–296.
[6] J.A. Robinson, A machine-oriented logic based on the resolution principle, Journal of ACM 12 (1965) 23–41.
[7] M.H. Williams, Q. Kong, and G. Chen, Handling incomplete information in a logic database, Proceedings of the 1988 Alvey Technical Conference, Swansea, U.K. (1988), 224–227.

RHODES UNIVERSITY LIBRARY

48. M.H.WILLIAMS, Q.KONG and G.CHEN, Handling incomplete information in a logic database, Proceedings of the UK Information Technology '88 Conference, IEE/BCS, Swansea, 224-227 (1988).

HANDLING INCOMPLETE INFORMATION IN A LOGIC DATABASE

M.H. Williams, Q. Kong and G. Chen

Computer Science Department, Heriot-Watt University, Edinburgh.

ABSTRACT

By extending Horn clause format to handle incomplete information, a form of clause suitable for deductive databases has been formulated which allows for certain restricted disjunctive clauses to be represented and manipulated. This paper describes this approach to incomplete information for a deductive database system.

1. INTRODUCTION

A logic database is an extension of the idea of a relational database in which provision is made for the storage, retrieval and manipulation of logical rules as well as facts. The general form of such rules is taken to be clausal form which is equivalent to first order predicate calculus and in which rules have the form

$$P_1 \vee \cdots \vee P_n \leftarrow Q_1 \wedge \cdots \wedge Q_m$$

Unfortunately, owing to the computational complexity of satisfiability of predicate calculus, this general form is unacceptable and a restricted form of clause, known as a Horn clause, is usually adopted. For such a clause $n = 1$ (i.e. the clause has a single positive literal as conclusion). Although it has less power than full clausal form, it is both simpler and safer to use.

However, this form is incapable of expressing any notion of incompleteness. As a result, a representation has been sought which is intermediate between Horn clause form and full clausal form which will permit certain restricted forms of disjunctive clauses. One of the problems with such an endeavour is that of efficiency and an essential criterion for any representation is that the resulting query evaluation strategy should be efficient.

In order to meet this criterion, a representation is proposed which stores incomplete information as a special kind of data type, while retaining the framework of a Horn clause evaluation system. In this representation the value of an attribute may be a set having the form $\{a_1,a_2,\ldots,a_n\}$. This is interpreted as a value which is unknown but known to be one of the values a_1, a_2,... or a_n. Furthermore a clause containing incomplete information, $p(X,\{a,b\},Y)$, will be interpreted as

$$p(X,\{a,b\},Y) \equiv \forall_X \forall_Y \exists_{U \in \{a,b\}} p(X,U,Y) \equiv p(X,a,Y) \vee p(X,b,Y).$$

For example, suppose that the blood group of patient Jones is known to be either A or AB but it is not known for certain which; to express this one might write

bloodgroup (jones ,{a ,ab }).

which is equivalent to

bloodgroup (jones ,a) ∨ bloodgroup (jones ,ab).

When a query is applied to a database containing incomplete information, three different responses are possible, viz. T (i.e. definitely true in the database), F (definitely false in the database) and U (unknown - can neither be proved to be true nor proved to be false). For example, given the database

bloodgroup (smith ,o).
bloodgroup (jones ,{a ,ab }).

the query

? – bloodgroup (smith ,o)

will definitely succeed (i.e. have value T), the query

? – bloodgroup (smith ,a)

will definitely fail (i.e. have value F), whereas the query

? – bloodgroup (jones ,a)

has value U. From the user's point of view there are two conditions which are of interest:

- definitely true (i.e. T), for which the function *def* is provided,
- possibly true (i.e. T or U), for which the function *pos* is provided.

If a query is preceded by the function *def* ,this is interpreted as the user wanting answers which are definitely true, e.g.

? – def (bloodgroup (jones ,a))

asks whether the blood group of John is definitely A (to which the answer is "no"), while a query preceded by *pos* will yield all possible answers, e.g.

? – def (bloodgroup (jones ,a))

asks whether the blood group of John could possibly be A (to which the answer will be "yes").

A more complete discussion of the theory is given in reference (Williams and Kong (3)). This paper focuses on the unification rules used to construct a query evaluation system for this special form of incomplete information.

2. HANDLING INCOMPLETE INFORMATION

Given a set of clauses S, if a clause C is to be proved in the context of S, then following the process of *proof by refutation*, the query C is negated and added to S, and an attempt is made to establish inconsistency of the new clause set. If the empty clause can be deduced (i.e. inconsistency

established) from the set $\{S\} \cup \{\neg C\}$, then the clause C is proved following set S.

Suppose that one has a clause $p(Y_1,\ldots,Y_n)$ and a query $?-p(X_1,\ldots,X_n)$, which, when negated, becomes $\neg p(X_1,\ldots,X_n)$. According to Robinson (2) and Chang and Lee (1), if a most general unifier θ can be found such that when it is applied to the literals $\neg p(X_1,\ldots,X_n)$ and $p(Y_1,\ldots,Y_n)$, it yields the complementary pair $\neg p(A_1,\ldots,A_n)$ and $p(A_1,\ldots,A_n)$ $(p(A_1,\ldots,A_n) = p(X_1,\ldots,X_n)\theta = p(Y_1,\ldots,Y_n)\theta)$ then from (1) and (2) one can deduce the empty clause.

This is the situation for a database containing no incomplete information. If one allows constants representing incomplete information, the task becomes more complicated. Some extension of the notion of unification is required to handle incomplete information. To this end consider the following definitions:

Definition 1
Substitution: A substitution θ is a set of pairs with the form $\theta = \{t_1/v_1, t_2/v_2, \ldots, t_m/v_m\}$, in which each t_i is a term (complete datum, incomplete datum or variable), and each v_i a variable, and for any $i \neq j$, $v_i \neq v_j$.

Definition 2
Definitely unifiable: If there is a substitution $\theta = \{t_1/v_1, t_2/v_2, \ldots, t_n/v_n\}$, such that an empty clause can be deduced from two literals, $\neg p(X_1,\ldots,X_n)$ and $p(Y_1,\ldots,Y_n)$, i.e.

$$\{\neg(p(X_1,\ldots,X_n)\theta), p(Y_1,\ldots,Y_n)\theta\} \vdash \square.$$

then these two literals are said to be *definitely unifiable*, and the substitution θ is called a unifier.

A weaker condition is:

Definition 3
Possibly unifiable: Two literals $\neg p(X_1,\ldots,X_n)$ and $p(Y_1,\ldots,Y_n)$ are said to be *possibly unifiable* if for each pair X_i and Y_i $(1 \leq i \leq n)$,
(1) at least one of them is an uninstantiated variable, or
(2) both are ground items (or instantiated variables) and
$X_i \in Y_i$ if X_i is a complete item and Y_i is an incomplete one;
$Y_i \in X_i$ if Y_i is a complete item and X_i is an incomplete one;
$X_i \cap Y_i \neq \emptyset$ if both X_i and Y_i are incomplete items.

If a query is preceded by the function *def*, then the notion of *definitely unifiable* will be applied during query evaluation, which will follow the normal process of logical deduction and generate an answer which is definitely true in the database. If a query is preceded by the function *pos*, then the definition of *possibly unifiable* will be applied during query evaluation. The answer generated is possibly true in the database.

3. EXAMPLES

The following examples will demonstrate different cases of unification in detail. In what follows, if a variable is instantiated then the value to which it is instantiated will be used. The term variable will refer to an uninstantiated variable.

3.1. Single argument unification

Case 1. Suppose one has a database containing one clause $p(A)$, and one poses the query $?-def(p(B))$. Because the query is preceded by the *def* function, the definite answer is required. By adding the negated query into the database, one has the following two clauses:

(1) $\neg p(B)$
(2) $p(A)$

using the definition of definitely unifiable, one of the following cases apply:

(a) If either A or B is a variable, or if $A = B$ then the two clauses can be unified. The process of unification is straightforward.

(b) If A is either a single value or a set of possible values, represented by $\{a_1,\ldots,a_n\}, n \geq 1$, and B is either a single value or a set of possible values, represented by $\{b_1,\ldots,b_m\}, m \geq 1$, and $B \supseteq A$ then the two clauses can be unified.

(c) For any other case the two clauses do not unify.

The reason for (b) can be seen if $C = \{c \mid c \in B \wedge c \notin A\}$. Then $A \cap C = \emptyset$, $A \cup C = B$ and clause (2) is equivalent to

(3) $$\neg p(B) = \neg p(A \cup C) = \neg(p(A) \vee p(C)) = \neg p(A) \wedge \neg p(C)$$

This is equivalent to the following two clauses

(4) $\neg p(A)$
(5) $\neg p(C)$

Since $\neg p(A)$ unifies with $p(A)$ (as seen in (a)), clauses (2) and (4) will yield $\square$

Hence, if B is a ground term with incomplete data in the negative literal, and A is a ground term with incomplete data in the positive literal, and the relation $B \supseteq A$ is satisfied (B is a superset of A), then the empty clause can be obtained even when $p(B)\theta \neq p(A)\theta$. For the definite query the answer is true if and only if there exists a substitution such that $\{\neg p(B)\theta, p(A)\theta\} \vdash \square$.

Case 2. Suppose one has the same database as above containing one clause $p(A)$, and one poses the query $?-pos(p(B))$. Because the query is preceded by the *pos* function, the possible answer is required. By adding the negated query into the database, one has the following two clauses:

(1) $\neg p(B)$
(2) $p(A)$

Now the definition of possibly unifiable will be used for the evaluation, and one of the following cases will apply.

(a) If either A or B is a variable, then the two clauses can be unified. The process of unification is straightforward.

(b) If A is either a single value or a set of possible values, represented by $\{a_1,\ldots,a_n\}, n \geq 1$, and B is either a single value or a set of possible values, represented by $\{b_1,\ldots,b_m\}, m \geq 1$, and $B \cap A \neq \emptyset$ then the two clauses can be unified.

(c) For any other case the two clauses do not unify.

The reason for (b) is that if $\{a_1,\ldots,a_n\} \cap \{b_1,\ldots,b_m\} \neq \emptyset$, then there exists an element a_i, such that $a_i \in B$ and $a_i \in A$. Now if $B' = B \setminus \{a\}$ and $A' = A \setminus \{a\}$, then $\neg p(B) = \neg(p(a) \vee p(B'))$ and $p(A) = p(a) \vee p(A')$. In this case it is not certain that these two clauses can be resolved to an empty clause. However, it is the case that if $p(a)$ is true, the two can be resolved. Hence these two clauses are possibly unifiable.

3.2. Unification with multiple arguments

The conditions of unification become more complicated if there is more than one argument. Consider the following example.

Case 3. Suppose one has a database containing one clause $p(\{a,b\},Y)$, in which Y is a variable, and one asks the query $?-def(p(A,B))$. By adding the negated query into the database one has the following two clauses:

(1) $\neg p(A,B)$
(2) $p(\{a,b\},Y)$

Because the query is preceded by the *def* function, the normal process of logical deduction will be applied to evaluation, and one of the following cases will apply:

(a) If A is a variable and B is a constant (either complete or incomplete), or both A and B are variables and $A \neq B$, then the two clauses can be unified. The process of unification is straightforward.

(b) If A is an incomplete constant represented by $\{a_1,\ldots,a_n\}, n \geq 1$, and satisfies the condition $A \supseteq \{a,b\}$, and B is either a variable or a constant (complete or incomplete), then the two clauses can be unified. The reason for this is the same as the reason for case 1 (b).

(c) If both A and B are variables and $A = B$, then the two clauses can not be unified.

(d) For any other case the two clauses do not unify.

(c) illustrates the one case when it is not true that a variable can unify with anything if incomplete data are involved. The reason for this is as follows. Since both A and B are variables and $A = B$, the two clauses can be rewritten as:

(1) $\neg p(A,A)$
(2) $p(\{a,b\},Y)$

According to the interpretation of incomplete data given in section 2, clause (2) is equivalent to

(3) $p(a,Y) \vee p(b,Y)$

By instantiating the variables in (1) and (3) with each element of the Herbrand base (i.e. a and b) one obtains:

(4) $\neg p(a,a)$
(5) $\neg p(b,b)$
(6) $p(a,a) \vee p(b,a)$
(7) $p(a,b) \vee p(b,b)$

This clause set is not inconsistent since one possible interpretation is that $p(a,a)$ and $p(b,b)$ are false while $p(a,b)$ and $p(b,a)$ are true. In other words, the empty clause cannot be deduced from this clause set. Hence $\neg p(X,X)$ and $p(\{a,b\},Y)$ are not unifiable.

It must be emphasised that the substitution of an incomplete data value for a variable is more complex than the usual approach followed in first order logic. For example, if the substitution $\theta = <\{a,b\}/X>$ is applied to $p(X)$ then $p(X)\theta = p(X)\{\{a,b\}/X\} = p(\{a,b\}) = p(a) \vee p(b)$, but if it is applied to $p(X,X)$ then $p(X,X)\theta = p(X,X)\{\{a,b\}/X\} = p(a,a) \vee p(b,b) \neq p(\{a,b\},\{a,b\})$.

Case 4. In case 3, when the query $?-def(p(A,A))$ is applied to the database $p(\{a,b\},Y)$ the answer obtained is "no". But if the query is preceded by the *pos* function, $?-pos(p(A,A))$, then the same database should yield the answer "yes" for the same reason as that given in case 2 for single argument unification. As shown in the explanation for case 3 (c), the query and the clause in database are equivalent to

(4) $\neg p(a,a)$
(5) $\neg p(b,b)$
(6) $p(a,a) \vee p(b,a)$
(7) $p(a,b) \vee p(b,b)$

and clauses (4) and (6) (or (5) and (7)) are possibly unifiable.

The full set of definite and possible unification rules for this type of incomplete information is given in Tables 1 and 2 respectively, and the corresponding unification algorithms are presented in the Appendices.

References

1. Chang, C.L., and Lee, R.C.T., 1973, *Symbolic logic and mechanical theorem proving*, Academic Press, New York, N.Y.

2. Robinson, J.A., 1965. *Journal of ACM*, **12**, 23-41.

3. Williams, M.H., and Kong, Q., to appear in *Knowledge and Database Engineering*.

Appendix 1: The algorithm for definite unification

Let $\neg p(x_1,x_2,\cdots x_n) p(y_1,y_2,\cdots y_n)$, be a pair of clauses being unified.

For $k = 1$ to n do

1. If both x_k and y_k are variables, then substitute them with a marked variable, add these new elements to the substitution. Go to step 5.

2. If x_k is a variable and

   (a) y_k is a complete data item then add $\{y_k/x_k\}$ to the substitution. Go to step 5.

   (b) y_k is an incomplete data item with the form B then add $\{B/x_k\}$ to the substitution. Go to step 5.

   (c) y_k is an unmarked incomplete data item and x_k is an unmarked variable then add $\{y_k^*/x_k\}$ to the substitution. Go to step 5.

   (d) y_k is an unmarked incomplete data item and x_k is a

marked variable then stop, the unification fails.

3. If y_k is a variable and

   (a) x_k is a complete data item then add (x_k/y_k) to the substitution. Go to step 5.

   (b) x_k is an unmarked incomplete data item, then add $(x_k^{\cdot}/y_k)$ to the substitution. Go to step 5.

   (c) x_k is an incomplete data item having the form $A^{\cdot}$, then add $(A^{\cdot}/y_k)$ to the substitution. Go to step 5.

   (d) x_k is an incomplete data item having the form A^{*}, then stop, the unification fails.

4. If both x_k and y_k are ground terms and

   (a) both of them are complete data items or unmarked incomplete data items, and $x_k \supseteq y_k$ or $y_k \in x_k$, then go to step 5 (no new element for the substitution required).

   (b) x_k is an incomplete data item having the form A^{*}, then stop, the unification fails.

   (c) y_k is an incomplete data item having the form $B^{\cdot}$, and $B \cap x_k \neq \emptyset$ or $x_k \in B$ (for x_k is a complete data item), then change the elements of the substitution, rebinding all $B^{\cdot}/v$ and x_k/u to $(B^{\cdot} \cap x_k)/u$ and $(B^{\cdot} \cap x_k)/v$. Go to step 5.

   (d) x_k is an incomplete data item having the form $A^{\cdot}$, if y_k is a complete data item and $y_k \in A$, then rebinding all $A^{\cdot}/u$ to y_k/u, go to step 5; otherwise if y_k is an incomplete data item then stop, the unification fails.

5. Continue.

Appendix 2: The algorithm for possible unification

Let $\neg p(x_1,x_2,\cdots,x_n) p(y_1,y_2,\cdots,y_n)$, be a pair of clauses being unified.

For $k = 1$ to n do

1. If x_k is a variable and

   (a) If y_k is an incomplete data item, then add y_k^{p}/x_k to the substitution. Go to step 4.

   (b) Otherwise add y_k/x_k to the substitution. Go to step 4.

2. If y_k is a variable and

   (a) If x_k is an incomplete data item, then add x_k^{p}/y_k to the substitution. Go to step 4.

   (b) Otherwise add x_k/y_k to the substitution. Go to step 4.

3. If both x_k and y_k are ground terms and

   (a) both of them are complete data items or unmarked incomplete data items, and $x_k \cap y_k \neq \emptyset$ (complete data item is regarded as a single element set) then go to step 4 (no new element for the substitution required).

   (b) both of them are incomplete data items having the form A^{p} and $x_k \cap y_k \neq \emptyset$ then change the elements of the substitution, rebinding all y_k/v and x_k/u to $(y_k \cap x_k)^{p}/u$ and $(y_k \cap x_k)^{p}/v$. Go to step 4.

   (c) x_k is an incomplete data item having the form A^{p} and $x_k \cap y_k \neq \emptyset$ then change the elements of the substitution, rebinding all x_k/u to $(y_k \cap x_k)^{p}/u$. Go to step 4.

   (d) y_k is an incomplete data item having the form A^{p} and $x_k \cap y_k \neq \emptyset$ then change the elements of the substitution, rebinding all y_k/u to $(y_k \cap x_k)^{p}/u$. Go to step 4.

   (e) otherwise stop, the unification fails.

4. Continue.

TABLE 1. Rules for definite unification

query args \ clause args	b	y or y^{*}	B	$B^{\cdot}$
a	$a = b$	a/y	$\{a\} = B$	$a \in B^{\cdot}$ †
x	b/x	y^{*}/x	B^{+}/x	$B^{\cdot}/x$
A	$b \in A$	$A^{\cdot}/y$	$A \supseteq B$	$A \cap B^{\cdot} \neq \emptyset$ †
x^{*}	b/x^{*}	y^{*}/x^{*}	$\perp$	$B^{\cdot}/x$
A^{+}	$\perp$	$\perp$	$\perp$	$\perp$
$A^{\cdot}$	$b \in A^{\cdot}$ †	$A^{\cdot}/y$	$\perp$	$A^{\cdot} \cap B^{\cdot} \neq \emptyset$ †

TABLE 2. Rules for possible unification

query args \ clause args	b	y	B	B^{p}
a	$a = b$	a/y	$a \in B$	$a \in B^{p}$ †
x	b/x	y/x	B^{p}/x	B^{p}/x
A	$b \in A$	A^{p}/y	$A \cap B \neq \emptyset$	$A \cap B^{p} \neq \emptyset$ ‡
A^{p}	$b \in A^{p}$ †	A^{p}/y	$A^{p} \cap B \neq \emptyset$ ‡	$A^{p} \cap B^{p} \neq \emptyset$ ‡

Explanation of notation

a, b	complete data items
x, y	variables
x^{*}, y^{*}	marked variables
A, B	incomplete constants
$A^{\cdot}, B^{\cdot}$	incomplete constants arising from negative literals
A^{*}, B^{*}	incomplete constants arising from positive literals
A^{p}, B^{p}	incomplete constants arising from possible query
A/x	substitute A for x
$\perp$	fail
$a = b, a \in A$	succeed conditions
$A \supseteq B$	succeed condition
$A \cap B \neq \emptyset$	succeed condition
$A \cap B \neq \emptyset$ †	rebinding $A/x, B/y$ to $(A \cap B)^{\cdot}/x$ and $(A \cap B)^{\cdot}/y$
$A \cap B \neq \emptyset$ ‡	rebinding $A/x, B/y$ to $(A \cap B)^{p}/x$ and $(A \cap B)^{p}/y$
$a \in A$ †	rebinding A/x to a/x

49. Q.KONG and M.H.WILLIAMS, Evaluating different strategies for handling incomplete information in a logic database, in 'Logic Programming Expanding the Horizons', Eds. T.Dodd, R.Owens and S.Torrance, Intellect Books, Oxford, 124-146 (1990).

Evaluating Different Strategies for Handling Incomplete Information in a Logic Database

Qinzheng Kong M. H. Williams

Abstract

The problem of handling incomplete information in a relational database is well understood and various researchers have investigated strategies for solving it. By contrast, the problem of incomplete information in a logic database is much more difficult and poses considerable challenges.

A formalism for incomplete information in a logic database has been put forward based on Lipski's work on relational databases, and a model has been developed which accounts for the behaviour of a logic database using such a formalism. To implement this formalism two different approaches have been investigated. These are described and their performance when applied to a given set of test examples, is compared.

1 Introduction

In many applications of databases, instances arise where the value of an attribute is not known exactly[2],[3],[4],[13]. Such situations are usually handled by using a null value as the value of the attribute concerned, provided that such a facility is available [7],[8],[10],[11],[12],[18]. However, if one has some information about an attribute value, even though the exact value is not known, then by simply using a null value, one is discarding what information one does have.

Two obvious forms of incomplete information which have been identified are sets and ranges. An extension of the relational model which caters for attribute values of these two types has been developed by Lipski [9]. These

ideas have been used as a basis for an extended form of QBE which handles incomplete information in the forms of sets, ranges and sets of ranges [16].

To demonstrate the problem, consider a database which is part of an office information system. Within such a database one might have the following relation which contains details about meetings scheduled.

meeting	meeting_no	time	date	group
	1	1000	15_Feb	Research_ctee
	2	1400	16_Feb	Finance_ctee

This can be represented in logic as the following set of clauses:

$$meeting(1, 1000, 15_Feb, research_ctee)$$
$$meeting(2, 1400, 16_Feb, finance_ctee).$$

Now suppose that one wants to add a meeting of the Executive Committee which is due to take place at 10am on 17th, 18th or 19th February (the exact date has not yet been finalised). Rather than represent this as

meeting	meeting_no	time	date	group
	3	1000	NULL	Exec_ctee

one might write

meeting	meeting_no	time	date	group
	3	1000	$\{17_Feb, 18_Feb, 19_Feb\}$	Exec_ctee

or represent this as the clause

$$meeting(3, 1000, \{17_Feb, 18_Feb, 19_Feb\}, Exec_ctee)$$

where $\{17_Feb, 18_Feb, 19_Feb\}$ represents the fact that the date is not known exactly but is known to be one of 17th February, 18th February or 19th February.

Extending these ideas to a logic database is a more difficult task [1], [6], [17], [5]. Not only can incomplete information occur in facts but also in rules. In the latter case, it may occur as:

1. Uncertain consequence.

For example, suppose that one is planning a new rota of duties and either Miss Jones or Mrs Smith is to be assigned to take minutes for the Executive committee. This might be expressed as

$$secretary(X, \{miss_jones, mrs_smith\}) \leftarrow meeting(X, Y, Z, exec_ctee)$$

2. Uncertain condition

   To illustrate this, suppose that one of the secretaries responsible for taking the minutes for the Executive committee or for the Finance committee must also handle the Research committee. One might express this knowledge as a rule as follows:

$$secretary(X, Y) \leftarrow meeting(X, A, B, research_ctee), meeting(Z, C, D, \{exec_ctee, finance_ctee\}), secretary(Z, Y).$$

A model has been put forward by Williams and Kong [14] to account for the behaviour of incomplete information in this form in a logic database. This is described briefly in the next section.

This paper describes two strategies which have been used to implement this model of incomplete information and compares their performance when applied to a given set of test examples.

2 Model of incomplete information

An item of incomplete information represents an uncertain value which may be used as an argument in a database fact or rule. When it appears, it represents a disjunction. For example, let x_i be a complete constant and X be a variable. The database fact

$$p(\{x_1, x_2\})$$

represents the situation where either the fact $p(x_1)$ is true or the fact $p(x_2)$ is true or both. It is equivalent to

$$p(x_1) \vee p(x_2).$$

A database rule containing incomplete information with the form:

$$p(X, \{a, b\}) \leftarrow q(X)$$

represents the knowledge that if $q(X)$ is true then either $p(X, a)$ is true or $p(X, b)$ is true. This is equivalent to a rule with a disjunction in the rule head:

$$p(X, a) \vee p(X, b) \leftarrow q(X).$$

A simple model representation of a single argument fact is given in Fig. 1. If the argument is a definite constant, e.g. x_1, the fact is represented by the model shown in Fig. 1(a). If the argument is a variable, e.g. X, then $p(X)$ is represented as a collection of ground facts, one for each value of the variable, as illustrated in Fig. 1(b). If the argument is an incomplete constant, e.g. $\{x_1, x_2\}$, then $p(\{x_1, x_2\})$ may be viewed as a set of separate models corresponding to each distinct value in the constant, as shown in Fig. 1(c).

A query is handled by negating it and adding it to the database. Thus the query

$$? - p(x_1)$$

will correspond to the addition of $\neg p(x_1)$ to the database. Likewise the query

$$? - p(X)$$

results in the addition of a set of negated facts, one for each value of the variable. The query

$$? - p(\{x_1, x_2\})$$

which asks whether either $p(x_1)$ or $p(x_2)$ is true in the database, i.e.

$$? - p(x_1) \vee p(x_2)$$

results in the two negated facts

$$\neg p(x_1)$$

$$\neg p(x_2)$$

being added to the database. This is illustrated in Fig. 2.

This approach can be used to represent rules, both with and without incomplete information. This is explained in greater detail in [14]. Using this model one can picture clearly the effects of resolution and hence the results of any query which involves both inference and incomplete information.

3 Tuple oriented evaluation strategy

One strategy for handling incomplete information is the so called tuple oriented approach. Under this approach, the system evaluates a user query

and returns one answer at a time. If more than one answer is required by the user query then the system will backtrack to re-search the database to find each successive answer.

3.1 Simple tuple oriented query evaluation

This strategy is similar to the fail and back-tracking strategy used by Prolog to find more than one solution. The difference is that in order to handle incomplete information, an extended resolution mechanism is required, which unifies two literals and produces a group of resolvents in a slightly different way from standard Prolog. The approach can be briefly described as follows:

1. An extended unification mechanism is used to unify the first literal of the query with the head of a clause and a substitution θ is obtained.
2. If the query contains more than one literal, (i.e., a conjunctive query) then the substitution is applied to the remaining literals of the query to produce a group of resolvents.
3. All the resolvents must be satisfied in the database to establish the answer.
4. Once the answer is established (or disproved), backtracking is invoked to find the next solution.

To understand this approach, consider the following example. Suppose a database has the following facts:

$$has_opportunity(\{peter, mary\}).$$
$$has_opportunity(john).$$
$$has_opportunity(tom).$$
$$has_motive(\{tom, john\}).$$
$$has_motive(mary).$$
$$has_motive(peter).$$

and a rule:

$$is_suspect(X) :- has_opportunity(X), has_motive(X).$$

If a user wants to find all suspects, i.e.,

$$? - is_suspect(X).$$

the set of answers to this query will be:

$$\{\langle X\rangle | is_suspect(X)\}$$

The evaluation procedure will be as follows:

1. The unification between the query $? - is_suspect(X)$ and the rule head $is_suspect(X)$ follows the ordinary Prolog unification mechanism and yields the substitution $\theta = \{X/X\}$. Thus the rule body $has_opportunity(X), has_motive(X)$ is picked up as the goal to be further evaluated.

2. An extended unification mechanism is used to unify the first literal of the new goal $has_opportunity(X)$ with the first clause defining $has_opportunity$ and a substitution $\theta_1 = \{(\{peter, mary\}/X)\}$ is obtained.

3. The substitution θ_1 is applied to the remainder of the goal, which is $has_motive(X)$. Since the term value in the substitution is an incomplete data item, it stands for several possibilities. In this example X might be either *peter* or *mary*, and consequently two resolvents $has_motive(peter)$ and $has_motive(mary)$ are produced.

4. To establish the answer that *peter* or *mary* is a suspect, one has to prove that both of them have motive. (If one of them has motive, it may lead to the situation that *peter* has opportunity and *mary* has motive or vice versa—in both cases the answer "one of them is a suspect" cannot be established). In this case the two goals $has_motive(peter)$ and $has_motive(mary)$ are both satisfied and hence $\{peter, mary\}$ is returned as a solution to the query.

5. Backtracking is invoked to find further answers.

3.2 Improved query evaluation

There is a serious problem with the above procedure. When backtracking fails, it is not necessarily the case that there are no alternative answers, but simply that no further answer can be established under this particular order of the literals in the goal. In this example, $\{tom, john\}$ is also an answer to the query, but there is no way to obtain this other than to perform the evaluation in a different order. In other words, if the rule were changed to:

$$is_suspect(X) : -has_motive(X), has_opportunity(X).$$

and evaluated again, it would yield the answer $\{tom, john\}$. It is obvious that when the number of literals in the query (or the rule body) increases, the complexity of the evaluation increases exponentially.

To solve this problem, an improved query evaluation strategy is required. The main point of the improvement is that if a variable unifies with a complete constant t in the evaluation of literal p, it remains possible that the variable can be further unified with an incomplete set, S, containing t as its element, and this incomplete set might be the final substitution of the variable if each remaining element of S can be proved to be a solution of p. In this case a re-evaluation mark is used to indicate that the literal p has to be re-evaluated and succeed with each element in S to establish the answer. If there is more than one variable in literal p, this re-evaluation mark can only be made when each variable is substituted by a complete constant. The evaluation procedure can be described as follows:

1. Unify the negative and positive literals by using the extended unification rules.
2. If the substitution contains an incomplete data item as the term value, then apply it to the remainder of the query to produce a group of resolvents. If the substitution does not contain incomplete data but contains some complete data as term values, then apply it to the remaining literals of the query to get a resolvent and mark the current literal re-evaluable.
3. Prove all the resolvents to establish the answer. If the proof fails, and there is a re-evaluating mark, then try to find an incomplete proof of the resolvent, and re-evaluate the previous literals.
4. Backtrack to find more answers.

To illustrate this approach, consider the same suspect example. The derivation of the first answer *peter or mary is a suspect* is exactly the same as above. When backtracking to step 2 to find the next solution, the algorithm proceeds as follows:

2. An extended unification mechanism is used to unify the first literal of the new goal $has_opportunity(X)$ with the second clause defining $has_opportunity$ and a substitution $\theta_2 = \{(john/X)\}$ is obtained.
3. The substitution θ_2 is applied to the remainder of the rule body, yielding the goal $has_motive(john)$. Since the term value in the substitution does not contain incomplete data, it is marked for re-evaluation.

4. *has_motive*(*john*) cannot be proved from the data in the database, but it can be partly proved because there is a fact *has_motive*({*john*, *tom*}) which has the reading that *john* or *tom* has motive. If it can be proved that Tom also has opportunity, then the answer that *John or Tom is a suspect* can be established. Since the re-evaluating mark exists, the re-evaluation of the literal *has_motive*(*tom*) is carried out and succeeds, the incomplete data item $\{john, tom\}$ constitutes another answer of the query.

5. Mark the clause *has_motive*($\{john, tom\}$) to indicate that the incomplete data item $\{john, tom\}$ has already been generated as one answer.

6. Backtracking is invoked and step 2 and 3 are repeated as before with the substitution $\theta_3 = \{\langle tom/X\rangle\}$ for the variable X in the literal *has_opportunity*(X). Although the literal *has_motive*(*tom*) can also be partly proved by matching the clause *has_motive*($\{john, tom\}$), it does not constitute another answer because the answer has already been generated.

7. Further backtracking is invoked and no more answers obtained.

By applying this approach, all answers can be obtained without change to the order of the literals in the query (or in the rule body). However, it still has many problems.

1. When the first occurrence of variable X is unified with an incomplete constant (e.g. $\{peter, mary\}$), the rest of the query will be split into groups of queries with occurrences of the variable X being replaced in turn by each element in the incomplete constant. The whole query succeeds only when each of the branches succeeds. (In our example when X in *has_opportunity*(X) unifies with $\{peter, mary\}$, both *has_motive*(*peter*) and *has_motive*(*mary*) must succeed). If the query is complicated, the formulation of the splitting operation might not be acceptable. For example, if the first literal contains n variables and each of them unifies with an incomplete data item with m_i elements, then the remaining literals of the query will be split into $m_1 \times m_2 \times \ldots \times m_n$ branches. Each branch has to succeed to commit the answer.

2. When the first occurrence of a variable X unifies with a complete constant, a re-evaluating mark is required to enable this variable to be further unified with an incomplete constant. This greatly increases the complexity of the system control.

3. The system efficiency is reduced by the repeated backtracking and re-evaluation. For example, the literal $has_motive(tom)$ is evaluated twice.

4 Set oriented evaluation strategy

An alternative strategy for query evaluation is the *set oriented approach*. This approach follows the basic idea of query evaluation in relational databases. Each literal in the query is regarded as a table generated by evaluating the literal. If there is only one literal in the query, then the result table is the final answer of the query. If there is more than one literal in the query (i.e. the query is a conjunctive query), then the join of all result tables generates the final answer to the query as a whole, and the shared variables in the literals give the attributes of the join.

The basic strategy of this approach is as follows:

1. Evaluate each literal in the query and generate its result set.
2. Perform the join operation on these result sets if required (i.e. for a conjunctive query).
3. To evaluate each literal, first unify it with each fact or rule head (with the same predicate name) in the database. If a fact is unified, then return the substituted value as one result, if a rule head is unified then recompose the rule body by applying the substitution to it and recursively invoke the whole strategy to evaluate the new rule body, returning all the results.

Consider again the example given in previous section. The evaluation procedure of the query

$$?- is_suspect(X)$$

can be briefly described as follows:

1. Unify the query with the head of the rule in the database

   $$is_suspect(X) :- has_opportunity(X), has_motive(X).$$

   and then treat the rule body $has_opportunity(X), has_motive(X)$ as a new goal to be evaluated.

2. An extended unification mechanism is used to unify the first literal $has_opportunity(X)$ with all possible clauses defining $has_opportunity$ and all substitutions of the variable X are returned as a result set. (In this example, the result set is $\{\{peter, mary\}, john, tom\}$).

3. An extended unification mechanism is used to unify the second literal $has_motive(X)$ with all possible clauses defining has_motive and a result set is obtained. (In this example, it is the set $\{\{john, tom\}, peter, mary\}$).

4. An extended join function is applied to these two result sets using the shared variable X, and a joined set is obtained. (The extended join function will be given in detail in a later section. In this example, the joined result is $\{\{john, tom\}, \{peter, mary\}\}$).

The *set oriented* approach is based on the following main components:

1. The unification mechanism.

2. The extended join operation.

3. The recomposition of the rule body after successful unification of the rule head.

The details of the unification mechanism are described in detail in [15]. The remander of this section is devoted to a discussion of the extended join and the rule body recomposition.

4.1 The extended join operation

In general, a query or a rule body has the following format

$$B_1(\vec{X}_1) \wedge B_2(\vec{X}_2) \wedge \cdots \wedge B_n(\vec{X}_n)$$

in which $n \geq 1$. After the evaluation of each literal $B_i(\vec{X}_i)$ in the rule body, a set of results is obtained. The extended join operation is performed on these results. The ordinary join operation in relational algebra can be used for this purpose if there is no incomplete constant in the results. However, if incomplete constants occur, the ordinary join needs to be extended to cope with the situation.

Let $p(X_1, \ldots, X_k, Y_1, \ldots, Y_m)$ and $q(X_1, \ldots, X_k, Z_1, \ldots, Z_n)$ be two literals in a query (or in a rule body), where X_1 through X_k are shared variables.

Let $S = \{\langle x_1, \ldots, x_k, y_1, \ldots, y_m \rangle\}$ be the result set of $p(X_1, \ldots, X_k, Y_1, \ldots, Y_m)$, $R = \{\langle x_1, \ldots, x_k, z_1, \ldots, z_n \rangle\}$ be the result set of $q(X_1, \ldots, X_k, Z_1, \ldots, Z_n)$ and $T = \{\langle x_1, \ldots, x_k, y_1, \ldots, y_m, z_1, \ldots, z_n \rangle\}$ be the result set of the join of S and R. Let $\otimes$ denote the extended join and $\oplus$ the ordinary join, and a_i and A_i denote the complete data and incomplete data respectively. The extended join between two result sets can be given as follows:

Case 1 If neither set contains incomplete data, the function performs exactly the same as the join operator in relational algebra, i.e., $S \otimes R = S \oplus R$.

$$T = \{\langle x_1, \ldots, x_k, y_1, \ldots, y_m, z_1, \ldots, z_n \rangle | \langle x_1, \ldots, x_k, y_1, \ldots, y_m \rangle \in S \wedge \langle x_1, \ldots, x_k, z_1, \ldots, z_n \rangle \in R\}$$

Case 2 If no variable sharing occurs between p and q, (i.e. $k = 0$), even in the presence of incomplete data, the join performs as the ordinary relational join $\oplus$, i.e., $S \otimes R = S \oplus R$.

$$T = \{\langle y_1, \ldots, y_m, z_1, \ldots, z_n \rangle | \langle y_1, \ldots, y_m \rangle \in S \wedge \langle z_1, \ldots, z_n \rangle \in R\}$$

Case 3 If there is a single shared variable with incomplete data as its value, i.e. both p and q contain only one argument which is a shared variable, (i.e. $k = 1$, $n = 0$ and $m = 0$), the result $T = S \otimes R$ can be given as:

$$a \in S \wedge a \in R \rightarrow a \in T.$$
$$A \in S \wedge \forall_{a \in A}(a \in R) \rightarrow A \in T.$$
$$A \in R \wedge \forall_{a \in A}(a \in S) \rightarrow A \in T.$$

As an example consider the join in the evaluation of $is_suspect(X)$. The evaluation of the literal $has_opportunity(X)$ has the result

$$S = \{\langle peter \rangle, \langle mary \rangle, \langle \{john, tom\} \rangle\},$$

whereas the result set of literal $has_motive(X)$ is

$$R = \{\langle \{john, tom\} \rangle, \langle peter \rangle, \langle mary \rangle\}$$

Since

$$\langle \{john, tom\} \rangle \in S \wedge \langle john \rangle \in R \wedge \langle tom \rangle \in R \rightarrow \langle \{john, tom\} \rangle \in T$$
$$\langle \{peter, mary\} \rangle \in R \wedge \langle peter \rangle \in S \wedge \langle mary \rangle \in S \rightarrow \langle \{peter, mary\} \rangle \in T$$

the joined result is

$$T = \{\langle \{john, tom\} \rangle, \langle \{peter, mary\} \rangle\}.$$

Case 4 The most general case. In this case both p and q may contain any number of arguments, some of which are shared variables, and incomplete data items may occur as the values of the shared variables. To simplify the explanation a complete data item a can be regarded as a single element set $\{a\}$. Let S be a result set of $p(X_1, \ldots, X_k, Y_1, \ldots, Y_m)$ and ES_i be an element of S, i.e., $S = \{ES_1, \ldots, ES_s\}$. If ES_i contains only complete data items, it has the standard form

$$ES_i = \langle x_1, \ldots, x_k, y_1, \ldots, y_m \rangle.$$

If ES_i contains some incomplete data items as values, then ES_i can be re-expressed as a set of alternative values each of which contains only complete data, i.e.,

$$ES_i = \{Es_{i,1}, \ldots, Es_{i,j}\}$$

and each $Es_{i,j}$ has the standard format

$$Es_{i,j} = \langle x_1, \ldots, x_k, y_1, \ldots, y_m \rangle.$$

Similarly let R be a result set of $q(X_1, \ldots, X_k, Z_1, \ldots, Z_n)$ and ER_i be an element of R, i.e., $R = \{ER_1, \ldots, ER_r\}$. If ER_i contains only complete data it has the standard form

$$ER_i = \langle x_1, \ldots, x_k, z_1, \ldots, z_n \rangle,$$

If ER_i contains some incomplete data, then re-express it as a set of alternative complete values as follows:

$$ER_i = \{Er_{i,1}, \ldots, Er_{i,l}\}$$

where each $Er_{i,j}$ has the standard format

$$Er_{i,j} = \langle x_1, \ldots, x_k, z_1, \ldots, z_n \rangle.$$

The joined result $T = S \bigcirc R$ can be evaluated as follows:

$$\begin{aligned}
ES_i \;=\; & \langle x_1, \ldots, x_k, y_1, \ldots, y_m \rangle \in S \\
& \wedge ER_j = \langle x_1, \ldots, x_k, z_1, \ldots, z_n \rangle \in R \\
& \rightarrow ET_h = \langle x_1, \ldots, x_k, y_1, \ldots, y_m, z_1, \ldots, z_n \rangle \in T. \quad (1) \\
ES_i \;=\; & \{Es_{i,1}, \ldots, Es_{i,l}\} \in S \\
& \wedge \forall_{Es_{i,j} \in ES_i} (Es_{i,j} = \langle x_{j_1}, \ldots, x_{j_k}, y_{j_1}, \ldots, y_{j_m} \rangle \wedge \\
& \exists_{ER_h \in R} (ER_h = \langle x_{h_1}, \ldots, x_{h_k}, z_{h_1}, \ldots, z_{h_n} \rangle \wedge \\
& x_{j_1} = x_{h_1} \wedge \cdots \wedge x_{j_k} = x_{h_k} \\
& \rightarrow Et_g = \{\langle x_{j_1}, \ldots, x_{j_k}, y_{j_1}, \ldots, y_{j_m}, z_{h_1}, \ldots, z_{h_n} \}\rangle)
\end{aligned}$$

$$\wedge ET = \bigcup_g Et_g \wedge ET \in T. \tag{2}$$

$$\begin{aligned} ER_i \;=\; & \{Er_{i,1}, \ldots, Er_{i,l}\} \in R \\ & \wedge \forall_{Er_{i,j} \in ER_i} (Er_{i,j} = \langle x_{j_1}, \ldots, x_{j_k}, z_{j_1}, \ldots, z_{j_n} \rangle \wedge \\ & \exists_{ES_h \in S} (ES_h = \langle x_{h_1}, \ldots, x_{h_k}, y_{h_1}, \ldots, y_{h_m} \rangle \wedge \\ & x_{j_1} = x_{h_1} \wedge \cdots \wedge x_{j_k} = x_{h_k} \\ & \rightarrow Et_g = \{\langle x_{j_1}, \ldots, x_{j_k}, y_{h_1}, \ldots, y_{h_m}, z_{j_1}, \ldots, z_{j_n} \rangle\}) \\ & \wedge ET = \bigcup_g Et_g \wedge ET \in T. \end{aligned} \tag{3}$$

To illustrate the above join rules consider the following example. Suppose a database has the following data:

$$p(\{a, b\}, \{c, d\}, e)$$
$$p(a, c, f)$$

$$q(a, c, s)$$
$$q(a, d, t)$$
$$q(b, c, s)$$
$$q(b, d, r)$$

A query has the form

$$? - p(X, Y, U), q(X, Y, V).$$

The result of evaluating $p(X, Y, U)$ is

$$S = \{ES_1, ES_2\} = \{\{\langle a, c, e\rangle, \langle a, d, e\rangle, \langle b, c, e\rangle, \langle b, d, e\rangle\}, \langle a, c, f\rangle\}$$

in which ES_1 is a set of alternative values each element of which has the standard form $\langle x, y, z\rangle$, and ES_2 contains only complete data. The result of evaluating $q(X, Y, V)$ is

$$R = \{ER_1, ER_2, ER_3, ER_4\} = \{\langle a, c, s\rangle, \langle a, d, t\rangle, \langle b, c, s\rangle, \langle b, d, r\rangle\}$$

in which all ER_is contain only complete data. The joined result of $p(X, Y, U)$ and $q(X, Y, V)$ is:

$$T = \{ET_1, ET_2\} = \{\{\langle a, c, e, s\rangle, \langle a, d, e, t\rangle, \langle b, c, e, s\rangle, \langle b, d, e, r\rangle\}, \langle a, c, f, s\rangle\}$$

4.2 Recompose the rule body

Consider the following query and clauses in a database:

$$? - q(V, \{a, b\}, c)$$

$$q(X, Y, Z) : -p(X, Y), r(Y, Z)$$
$$p(1, a).$$
$$p(1, b).$$
$$p(2, s).$$
$$r(\{a, b\}, c).$$
$$r(a, c).$$
$$r(a, t).$$
$$r(s, c).$$

When the query unifies with the rule head, a substitution $\theta = \{V/X, \{a, b\}/Y, c/Z\}$ is obtained. Due to the occurrences of incomplete data, the substituted values cannot be applied directly to the rule body. The simplest strategy is to evaluate the rule body without applying the substitution. Such an evaluation is performed by evaluating $p(X, Y)$ and $r(Y, Z)$ separately, joining the results of p and r, and then performing a selection on these joined results according to the rule head substitution θ. In the above example the joined result of p and r is

$$T = \{\langle 1, \{a, b\}, c\rangle, \langle 1, a, c\rangle, \langle 1, a, t\rangle, \langle 2, s, c\rangle\}$$

and after applying the substitution, the final result should be

$$T = \{\langle 1, \{a, b\}, c\rangle, \langle 1, a, c\rangle\}$$

However, in such a strategy unnecessary intermediate results such as $< 1, a, t >$, $< 2, s, c >$ are generated which not only reduce the efficiency but also occupy unnecessary space. A new rule to recompose the rule body is introduced to handle this problem. This process can be defined as follows:

1. If a variable is substituted by a complete constant, i.e. there is an element in the substitution having the form t/X in which t is a complete constant, then replace all the occurrences of the variable X in the rule body with the complete constant t.

2. If a variable is substituted by another uninstantiated variable, i.e. there is an element in the substitution having the form Y/X in which Y is a variable, then replace all the occurrences of the variable X in the rule body with the variable Y.

3. If a variable is substituted by an incomplete constant, i.e. there is an element in the substitution having the form A/X in which A is an incomplete constant arising from a negative literal, then replace all the occurrences of the variable X in the rule body by a dummy variable X'. The linkage between the incomplete constant A and the dummy variable X' specifies the constraint that the dummy variable X' can only be unified with an uninstantiated variable or with a subset of the incomplete data A. This constraint will be referenced by further evaluation.

Consider the above example after application of the substitution θ. The recomposed rule body has the following form:

$$p(V, Y'), r(Y', c)$$

with the condition that:

$$\{T/Y'\} \rightarrow variable(T) \vee A \supseteq T \vee T \in A.$$

which means that the condition of a valid substitution

$$\{T/Y'\}$$

is:

1. T is an uninstantiated variable or
2. T is an incomplete data item which is a subset of A or
3. T is a complete data item which is an element of A.

4.3 The handling of recursive rules

A special strategy is required for handling recursive rules in the *set oriented* approach. Suppose that one of the rules defining $p(X, Y)$ is a recursive rule (direct or indirect recursion). The algorithm for evaluating the recursive rule can be described as:

1. Assign a dummy predicate name p' to the recursive rule defining p.
2. Evaluate all facts and non-recursive definitions of p and store the results under some new predicate p'.
3. Replace the occurrences of p in the body of the recursive rule by the dummy predicate p' and evaluate the new rule (since each occurrence of p has been replaced, the rule is no longer recursive).

4. If new results are obtained which are not already present in p', then add them to p' and evaluate p again. Repeat this until no new result is obtained. The accumulated results of p' are the results of p.

Consider the ancestor example. Suppose the database has the following information:

$$father(tom, peter).$$
$$father(tom, john).$$
$$father(peter, rob).$$
$$father(peter, harry).$$
$$father(harry, mike).$$

$$ancestor(X, Y) :- father(X, Y).$$
$$ancestor(X, Y) :- father(X, Z) \wedge ancestor(Z, Y).$$

If a query seeks all the ancestors in the database, the evaluation procedure proceeds as follows:

1. A dummy predicate name for *ancestor* is *ancestor'*.
2. The results generated by evaluating the non-recursive rule of ancestor are

$$ancestor'(tom, peter).$$
$$ancestor'(tom, john).$$
$$ancestor'(peter, rob).$$
$$ancestor'(peter, harry).$$
$$ancestor'(harry, mike).$$

3. The recursive rule is updated as:

$$ancestor(X, Y) :- father(X, Z) \wedge ancestor'(Z, Y).$$

and the results of evaluating this rule are:

$$ancestor'(tom, rob).$$
$$ancestor'(tom, harry).$$
$$ancestor'(peter, mike).$$

4. As the evaluation continues, the following clause is added:

$$ancestor'(tom, mike)$$

5. No further clauses are generated and the final results are:

$$ancestor(tom, peter).$$
$$ancestor(tom, john).$$
$$ancestor(tom, rob).$$
$$ancestor(tom, harry).$$
$$ancestor(tom, mike).$$
$$ancestor(peter, rob).$$
$$ancestor(peter, harry).$$
$$ancestor(peter, mike).$$
$$ancestor(harry, mike).$$

This way of handling recursive rules does not depend on the order of the literals defining the recursive rules. For example, if the definition of ancestor is changed to

$$ancestor(X, Y) : -ancestor(X, Z), father(Z, Y).$$

the same answer will be generated. On the other hand, the tuple based approach (and even Prolog itself) cannot handle this kind of rule since it will lead to a non-terminating loop.

4.4 The optimisation

The *set oriented* approach makes the order of the literals in the query irrelevant and is suitable for most database applications since most database queries seek all answers. This approach avoids repeated backtracking and reevaluating sets of resolvents produced each time. However, it needs more space than the *tuple oriented* approach especially when the database size is very large. To alleviate this shortcoming, an optimisation can be introduced in which the evaluation of database facts is delayed until the join operation is performed. Thus, the space required to store the intermediate results generated by evaluating the facts is saved. Under the assumption that in a logic database the number of facts is much greater than the number of rules, the space cost will be reduced greatly. Consider again the "suspect" example. Before the optimisation, space is required to store:

1. The internal format of the evaluating result of $has_opportunity(X)$

$$\{peter, mary\}$$
$$john$$
$$tom$$

2. The internal format of the evaluating result of $has_motive(X)$

$$\{john, tom\}$$
$$peter$$
$$mary$$

3. The internal format of the joined result

$$\{peter, mary\}$$
$$\{john, tom\}$$

After the optimisation, the total space required is the space to store the joined intermediate result

$$\{peter, mary\}$$
$$\{john, tom\}$$

5 Comparison of the two approaches

Both *set oriented* and *tuple oriented* approaches have been implemented in Prolog to handle incomplete information. Some comparisons have been made by testing both systems for a set of given examples. Table 1 gives the execution times of three examples in both systems; Figure 3 shows how the execution time increases as the database size is increased; Table 2 gives the space cost of the *set oriented* system—the third column gives the space required for storing the original data and the fourth column the maximum space required during execution. Although the *set oriented* approach needs more space than the *tuple oriented* approach, it has shorter execution times and is more suitable for database applications which require all answers.

References

[1] P. Chisholm, G. Chen, D. Ferbrache, P. Thanisch, and M.H. Williams. Coping with indefinite and negative data in deductive databases:, a survey. *Data and Knowledge Engineering*, 2:259–284, 1987.

Query	Database	Set-or. system	Tuple-oriented	
			one	all
$? - p(X, Y, U), q(X, Y, V)$.	94 facts of $p(X, Y, U)$ 86 facts of $q(X, Y, V)$	9.06	0.55	296.73
$? - s(X, Y, V), p(X, Y, U)$	rule $s(X, Y, V)$ $s(X, Y, V) : -$ $r(X, Y), q(X, Y, V)$ 85 facts of $r(X, Y)$ 86 facts of $q(X, Y, V)$ 94 facts of $p(X, Y, U)$	17.35	0.2	547.07
$? - teaching(T, D, N, H)$ $marks(D, S, N, M)$	62 facts of $teaching(T, D, N, H)$ 5000 facts of $marks(D, S, N, M)$	179.63	1.93	$> 60min$

Table 1: Execution times for three examples in the two systems (measured in CPU seconds)

Query	Database	Space requirements	
		original data	join
$? - p(X, Y, U), q(X, Y, V)$.	94 facts of $p(X, Y, U)$ 86 facts of $q(X, Y, V)$	8112	4040
$? - s(X, Y, V), p(X, Y, U)$	rule $s(X, Y, V)$ $s(X, Y, V)) : -$ $r(X, Y), q(X, Y, V)$ 85 facts of $r(X, Y)$ 86 facts of $q(X, Y, V)$ 94 facts of $p(X, Y, U)$	11480	4944
$? - teaching(T, D, N, H)$ $marks(D, S, N, M)$	62 facts of $teaching(T, D, N, H)$ 5000 facts of $marks(D, S, N, M)$	222,622	183,512

Table 2: Space (in bytes) required for evaluating queries using the set oriented approach

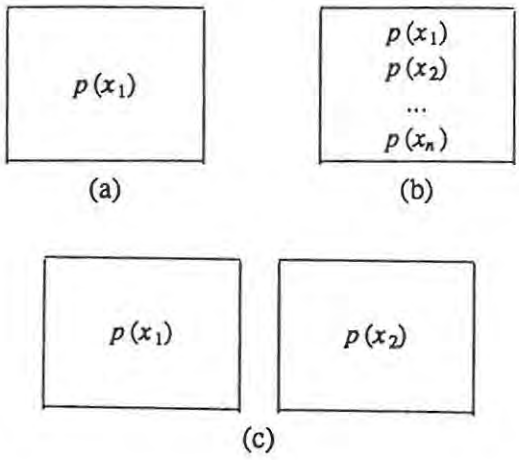


Figure 1: Model representations of (a) $p(x_1)$ (b) $p(X)$ (c) $p(\{x_1, x_2\})$

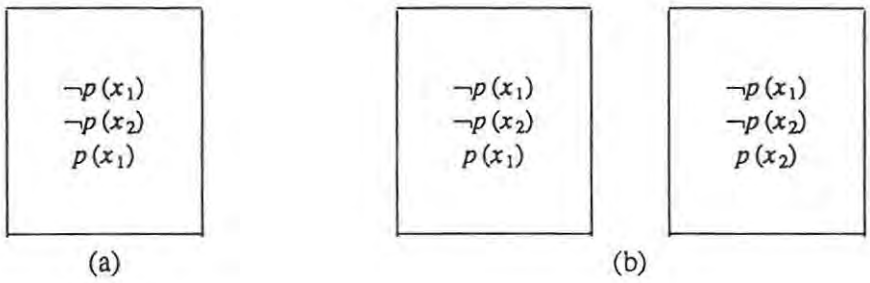


Figure 2: Model representations of query $?-p(\{x_1, x_2\})$ applied to database containing (a) $p(x_1)$ (b) $p(\{x_1, x_2\})$.

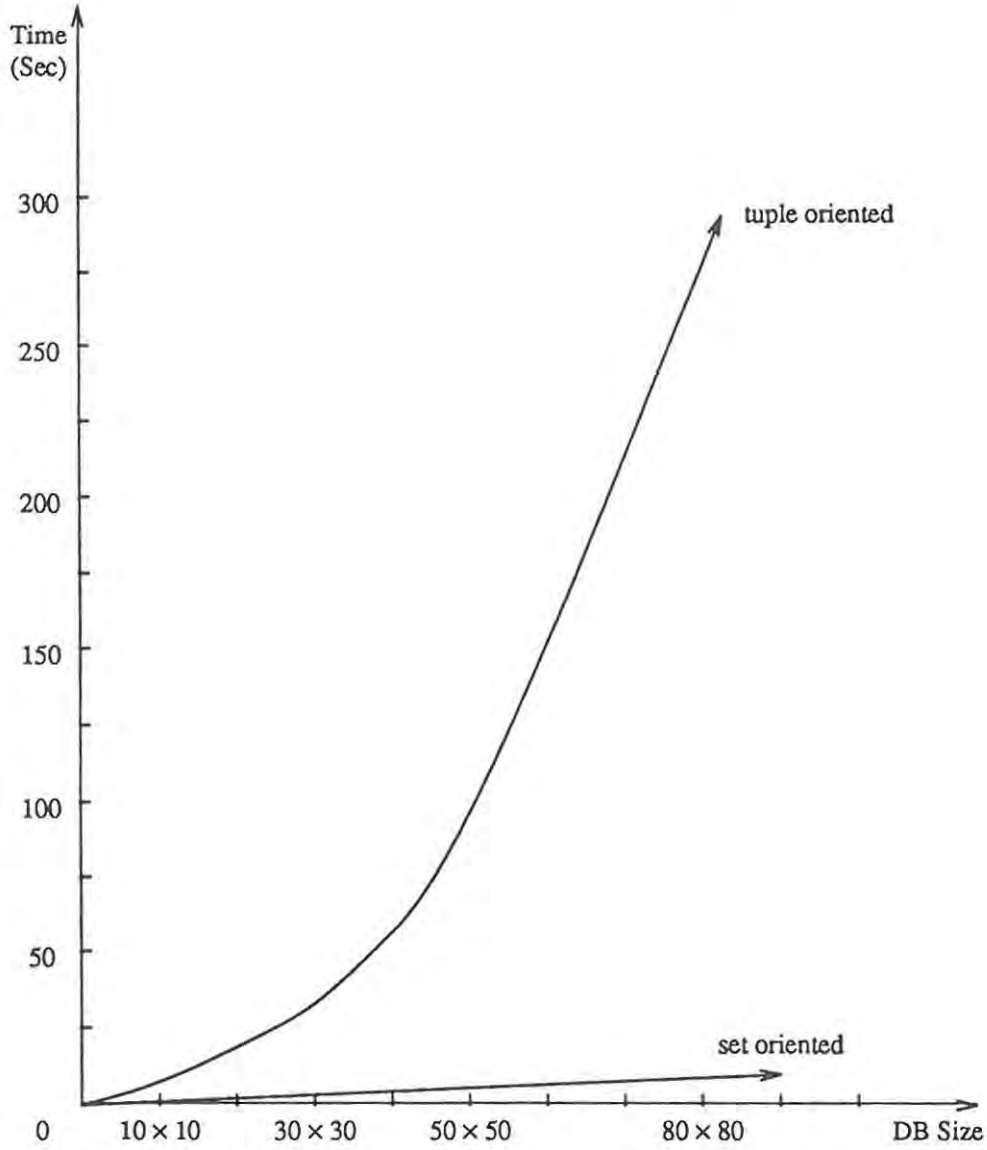


Figure 3: Execution times taken by set oriented and tuple oriented approaches for varying sizes of database

[2] E.F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. on Database Systems*, 4(4):397–434, December 1979.

[3] C.J. Date. Null values in databases. In *Proc. of the Second British National Conference on Databases*, 1982.

[4] J. Grant. Null values in a relational data base. *Information Processing Letters*, 5:156–157, 1977.

[5] J. Grant and J. Minker. Answering queries in indefinite databases and the null, value problem. *Advances in Computer Research*, 3:247–267, 1986.

[6] T. Imielinski. *Query processing in deductive databases with incomplete information*, pages 268–280. October 1984.

[7] T. Imielinski and W. Lipski. On representing incomplete information in a relational databases. In *Proc. of International Conference of VLDB*, pages 388–, 1981.

[8] H.J. Levesque. The logic of incomplete knowledge base. In M.L. Brodie et al., editors, *On Conceptual Modeling*, pages 165–189. Springer, 1984.

[9] W. Lipski. Informational systems with incomplete information. In S. Michaelson and R. Milner, editors, *Proc. 3rd Int. Symp. on Automata, Languages and Programming*, pages 120–130, Edinburgh, 1976. Edinburgh University Press.

[10] R. Reiter. Towards a logical reconstruction of relational database theory. In M.L. Brodie et al., editors, *On conceptual modelling: Perspectives from artificial intelligence,, databases and programming languages*, pages 191–233. Springer, 1984.

[11] R. Reiter. A sound and sometimes complete query evaluation algorithm for, relational database with null values. *Journal of ACM*, 33(2):349–370, 1986.

[12] L. Siklossy. Efficient query evaluation in relational databases with missing values. *Information Processing Letters*, 13(4/5):160–163, 1981.

[13] Y. Vassiliou. Null values in data base management: A denotational semantics approach. *Proc. ACM-SIGMOD International Symposium on Management of Data*, pages 162–169, 1979.

[14] M.H. Williams and Q. Kong. Incomplete information in a deductive database. *Data and Knowledge Engineering*, 3:197-220, 1988.

[15] M.H. Williams, Q. Kong, and G. Chen. Handling incomplete information in a logic database. In *UK IT 88 Conference Publication*, pages 224–227, 1988.

[16] M.H. Williams and K.A. Nicholson. Implementation of incomplete information in databases. *The Computer Journal*, 1985. (to appear).

[17] A. Yahya and L.J. Henschen. Deduction in non-horn databases. *Journal of Automated Reasoning*, 1:141–160, 1985.

[18] C. Zaniolo. Database relations with null values. *Proc. ACM SIGACT-SIGMOD Symposium on principles of Database Systems*, pages 27–33, 1982.

50. M.H.WILLIAMS and Q.KONG, Time and incompleteness in a deductive database, Proceedings of the Third International Conference on Information Processing and the Management of Uncertainty in Knowledge Based Systems, Eds. B.Bouchon-Meunier, R.R.Yager and L.A.Zadeh, Springer Verlag Lecture Notes in Computer Science, No. 521, 443-455 (1991).

TIME AND INCOMPLETENESS IN A DEDUCTIVE DATABASE

M. H. Williams
Department of Computer Science, Heriot-Watt University
79 Grassmarket, Edinburgh EH1 2HJ

Q. Kong
CITR, Ritchie Research Laboratories
University of Queensland, Australia

Abstract

Much attention has been given to the problem of time and how it can be handled within a logic based system. Another problem which has attracted considerable attention is that of incompleteness - a concept which is particularly relevant in the temporal domain. This paper considers how these two ideas may be brought together to produce a powerful system for dealing with data with a significant temporal variation.

Keywords

Deductive database, incomplete information, temporal database.

1. INTRODUCTION

The idea of using programming languages based on logic has been gaining increasing support in recent years. However, while there are considerable benefits to be obtained from using it as a programming language, it is not completely ideal for this in its present form. One major weakness from which it suffers is the assumption of monotonicity. This is one of the basic presuppositions of logic, namely that all propositions which are used in an argument are universally true. Consequently, if additional facts do become known, it will never be necessary to amend or withdraw conclusions as one becomes aware of new facts.

In other words, classical logic takes no account of changes in the state of a system. However, in attempting to apply logic to computer-based problems, one is frequently faced with situations where the information changes with time. Database systems are a case in point - many such systems have to cope with constantly changing information (e.g. on the state of a firm's operations). Expert system are another class of problems which have to deal with change from two quarters. On the one hand, while the knowledge rules are being refined and tested, they may be changed to reflect the expert system builder's understanding of the nature of the expert knowledge; on the other hand, when the system is used, it may have to deal with a situation in which the information available changes with time. An extreme example of this type is a blackboard system such as might be used for air traffic control or military applications, which is constantly processing new data and revising its beliefs on the state of the system it is attempting to monitor.

Modern logicians paid little attention to the problems of time until the middle of this century [2] since when there has been growing interest in finding solutions to these problems. Approaches put forward vary from purely qualitative to quantitative ones. The crudest approach (from the point of view

of coarseness of representation) is that of tense logic, in which four modal tense quantifiers (H - *it will be that*, P - *it has been that*, F - *it has always been that*, and G - *it will always be that*) are added to propositional calculus . [5] A larger set of modal tense quantifiers and temporal operators has been put forward by Seranadas . [10]

At the other extreme one has various approaches based on some form of time stamp as used in databases (e.g. Clifford and Warren). [4] However, even in this case the granularity of temporal values may vary considerably. In a coarse-grained model the precision with which time is represented may be at the level of dates (day, month, year) although for some applications one may only require accuracy at the level of the year. In a fine-grained system one requires a more accurate representation such as second/minute/hour/day/month/year or some equivalent form.

The situation calculus [8] is one such approach in which global states are used as explicit parameters of time-varying relationships. However, it suffers from an excessive computational overhead in carrying over virtually all relationships from one state to the next. To avoid this problem, Kowalski and Sergot [7] have proposed a system based on events which they term *event calculus*.

This paper attempts to combine the notion of time as it is handled in the event calculus with that of incompleteness or granularity to produce an effective system for dealing with databases and similar applications.

2. DEDUCTIVE DATABASES

A deductive database can be regarded as a set of clauses of first order logic. In general each clause has the form:

$$p_1(\vec{A_1})\vee p_2(\vec{A_2})\vee\cdots\vee p_n(\vec{A_n})\leftarrow q_1(\vec{B_1})\wedge q_2(\vec{B_2})\wedge\cdots\wedge q_m(\vec{B_m}) \tag{1}$$

in which $p_i(\vec{A_i})$ and $q_j(\vec{B_j})$ are predicates, $i\geq 0$ and $j\geq 0$. Such a clause represents the knowledge that if $q_1(\vec{B_1})\wedge q_2(\vec{B_2})\wedge\cdots\wedge q_m(\vec{B_m})$ is true then it follows that $p_1(\vec{A_1})\vee p_2(\vec{A_2})\vee\cdots\vee p_n(\vec{A_n})$ is also true. Specifically, if in a clause $n>0$ and $m=0$, i.e. the clause has the form:

$$p_1(\vec{A_1})\vee p_2(\vec{A_2})\vee\cdots\vee p_n(\vec{A_n}) \tag{2}$$

it represents a fact stored in the database. If $n=0$ and $m>0$, i.e. the clause has the form:

$$\leftarrow q_1(\vec{B_1})\wedge q_2(\vec{B_2})\wedge\cdots\wedge q_m(\vec{B_m}) \tag{3}$$

it represents a query applied to the database.

Due to the computational complexity of satisfiability in such system, the type of clause usually used in a deductive database is a restricted form, known as a Horn clause. A Horn clause is a clause which contains only one positive literal, i.e. it gives one definite conclusion. A database which consists entirely of Horn clauses is referred to as a definite deductive database. The clauses in such system have the form:

$$p(\vec{A}) \tag{4}$$

$$p(\vec{A}) \leftarrow q_1(\vec{B_1}) \wedge q_2(\vec{B_2}) \wedge \cdots \wedge q_n(\vec{B_n}) \quad (5)$$

A relation in such a database is defined as a group of clauses in which the head of each clause has the same predicate name and arity. For example, in the clauses given in (4) and (5), $p(\vec{A})$ represents a relational scheme, p is the relation name, $\vec{A}=<A_1, A_2, \ldots, A_n>$ $(n \geq 0)$ is a set of values, each of which comes from an attribute domain. Formula (4) can be regarded as a fact, formula (5) as a rule. $q_i(\vec{B_i})$ are predicates which can be defined either by a fact or by a rule in the database. Such a database can be used not only to search for information explicitly stored in it, but also to deduce the information implied by its rules.

For example, suppose one has a database of family relationships, which contains two relations: *father*(X, Y) which indicates that X is Y's father, and *ancestor*(X, Y) which is interpreted as X is Y's ancestor. The two relations are defined as follows:

father (*henry*, *tom*)
father (*tom*, *bob*)
father (*tom*, *anna*)
father (*john*, *rosa*)

ancestor (*simon*, *rob*)
ancestor$(X, Y) \leftarrow$ *father*(X, Y)
ancestor$(X, Y) \leftarrow$ *father*$(X, Z) \wedge$ *ancestor*(Z, Y).

In this database the relation *father* contains only facts, while *ancestor* contains both fact and rules. Some information, such as *Simon is Rob's ancestor*, is stored in the database directly, and some is not but can be deduced from this database, e.g. *Henry is Anna and Bob's ancestor*.

3. EVENTS AND STATES

In order to take account of the effect of time, a state-transition approach is assumed in which time is represented as a continuous domain which is sampled at discrete points.

In the simplest interpretation, at any given point in time the set of properties describing the system of interest, constitute a *state*. An *event* is a phenomenon which takes place in the universe of discourse at some point in time and which causes a change from one state (corresponding to one set of properties) to the next (corresponding to a slightly different set of properties). In this simple interpretation the transition is assumed to take place instantaneously so that at any moment in time a unique state applies.

Suppose that initially the system is in state S_0 and that with each transition the state number is increased by 1. Then the complete history of the system during the period of interest may be characterised by the sequence of states.

$$(S_0, S_1, \ldots, S_n).$$

Corresponding to each state S_i is a unique time t_i when that state is first entered. Thus the period during which the system is in state S_i corresponds to the half-open interval of time $[t_i, t_{i+1})$. For consistency the total period of interest will be assumed to be the half-open interval $[t_0, t_{n+1})$.

Relating this to logic, the set of properties which constitute a state may be represented as a set of Horn clauses. As a consequence of the above assumptions, each simple fact describing a property of the system will have associated with it a half-open interval of time, $[t_i, t_j)$, during which it is true. One simple way of representing this information is by linking the clause to two separate clauses recording the beginning and end of this interval.

For example, consider a system consisting of a set of employees over a period $[t_0, t_4)$. Suppose that employee number 1, Smith, is present for the entire period, Jones (number 2) was appointed at time t_1 but left at time t_3 whereas Evans (number 3) was appointed at time t_1 and was still employed at time t_4. Green (number 4) was present from time t_2 to t_4. The set of clauses representing this system may be written as follows:

employee (t_a, 1, *smith*)
employee (t_b, 2, *jones*)
employee (t_c, 3, *evans*)
employee (t_d, 4, *green*)
start (t_a, t_0)
start (t_b, t_1)
start (t_c, t_1)
start (t_d, t_2)
end (t_b, t_3).

From this it is clear that at time t_0 there will be one employee (*Smith*) in the system, at time t_1, there will be three (*Smith*, *Jones* and *Evans*), etc. Where the end dates are missing, the end of the interval in which that property holds, has not yet been encountered.

Alternatively the transitions affecting a system may be considered from the point of view of events. In such a case there will generally be a set of rules R which determine the consequences of these events on the properties of the system. Thus if during some period $[t_0, t_n)$ one has a sequence of events E where E_i gives rise to the transition from state S_i to state S_{i+1}, the complete history of the system may be characterised by the initial state, the sequence of events which have caused transitions to occur and the set of rules R which translate these into changes in the state of the system, viz.

$$S_0, (E_0, \cdots, E_n), R.$$

Since in this simple interpretation events are expected to occur instantaneously, it may be assumed that if event E_i occurs at time t_j, the consequence of this event will take effect at time t_j. With this in mind each simple fact which contains information on an event which takes place at time t must be qualified by a parameter which indicates the time of the event.

To illustrate this, consider the example used by Kowalski [6] of engaging and losing academic staff. The basic set of events (E_i) are

hire (*e* 1, *mary*, *lecturer*)
time (*e* 1, 10/5/1970)
leave (*e* 2, *john*, *lecturer*)
time (*e* 2, 1/6/1975)
leave (*e* 3, *mary*, *professor*)

time (*e* 3, 1/10/1980)
promote (*e* 4, *mary*, *lecturer*, *professor*)
time (*e* 4, 1/6/1975).

This tells us that the event in which Mary was hired as a lecturer took place on 10^{th} of May 1970, the event in which John gave up his lecturer position took place on 1^{st} June 1975, etc. In order to relate this to the state of the system, the following set of rules (R) is introduced:

rank (*after* (E), X, Y)←*hire* (E, X, Y)
rank (*before* (E), X, Y)←*leave* (E, X, Y)
rank (*after* (E), X, Y)←*promote* (E, X, Y, Z)
rank (*before* (E), X, Z)←*promote* (E, X, Y, Z).

This tells us that for some period *after(e)* the rank of x is y if x is hired as y at time e, and so on. To define the beginning and end times of these periods the following additional clauses are required:

start (*after* (E), E)
end (*before* (E), E).

For example, *start(after(e1), e1)* states that the period *after(e1)* begins at the time of event $e1$. Finally, we also require

end (*after* (E), E')←*after* (E)=*before* (E')
start (*before* (E'), E)←*after* (E)=*before* (E')
after (E)=*before* (E')←*rank* (*after* (E), X, Y)∧*rank* (*before* (E'), X, Y)∧$E<E'$
∧¬((*rank* (*after* (E^*), X, Y'))∧$E<E^*$∧$E^*<E'$)
∧¬((*rank* (*before* (E^*), X, Y'))∧$E<E^*$∧$E^*<E'$).

Information about an occurrence may usually be represented either as an event or as a property of a set of states. For example, the information about the hiring, promotion and departure of lecturers in the above example could be reexpressed as

rank (t_a, *mary*, *lecturer*)
rank (t_b, *mary*, *professor*)
rank (t_c, *john*, *lecturer*)
start (t_a, 10/5/1970)
end (t_a, 1/6/1975)
start (t_b, 1/6/1975)
end (t_b, 1/10/1980)
end (t_c, 1/6/1975).

The treatment of time as an entity rather than as a property enables missing values to be handled in a natural way. For example, one might rewrite the latter representation to express time as a property as follows:

rank (10/5/1970, 1/6/1975, *mary*, *lecturer*)
rank (1/6/1975, 1/10/1980, *mary*, *professor*)
rank (*?*, 1/6/1975, *john*, *lecturer*).

However, in this case one is faced with the problem of dealing with the missing value for John.

4. PARTIAL INFORMATION

The general problem of incompleteness of information is well-known. Two particular categories of incomplete information are particularly important, viz.

(a) Missing values. In this case the value of an attribute is completely unknown. Kowalski and Sergot have studied this case in detail for the event calculus.

(b) Incomplete values. In this case the value of an attribute is not known exactly but it is known that it ranges over a specified subset of the attribute domain.

An incomplete value can be regarded as a special kind of value which lies part way between an exact value and a missing value - i.e. a value which ranges over a subset of the attribute domain. Conversely, a missing value can be regarded as a special kind of incomplete value in that it represents a value which ranges over the whole attribute domain. The notion of incompleteness corresponds to the notion of granularity of information.

To illustrate this, consider the problem of accuracy of values in any continuous domain. For example, suppose that an attribute has the value 50. Does this mean a value which is somewhere between 45 and 55, or a value between 49.5 and 50.5 or even a value between 49.999995 and 50.000005? Clearly the accuracy of a number is very important in knowing how to interpret the number.

This is certainly true of time. For some events one might record their occurrence to within a small fraction of a second, for others the date (day/month/year) is sufficient. Yet others may only require the month and year or even simply the year. This concept of incompleteness is not restricted to temporal values but could apply to other domains as well, for example that of a student having an age between 18 and 24.

Thus for many applications one may wish to cater for varying degrees of granularity within the system. For example, when an employee is entered into the system, an employee record may be created with a time stamp which is accurate to within a fraction of a second. However, the record may also contain an attribute recording the date of birth of the employee (correct to the nearest day) or his or her age (correct to the nearest year). When querying such a system one may wish to find all employees whose records had been inserted in some particular 1 second interval during a day, or all those whose age is over 60. To deal with such variations in accuracy, the idea of a range of values is introduced.

5. REPRESENTATION OF INCOMPLETE DATA

A range is a new data type representing incomplete information, which can be defined as

Range ::= LowerBound .. UpperBound
LowerBound ::= Numerical value
UpperBound ::= Numerical value.

A **range** represents a particular numerical value which is not known exactly but is known to be not less than the lower bound of the range and not greater than the upper bound of the range. For a discrete ordered domain a range represents some point from a set of points taken from the domain. For example, given the domain $\{t_0, t_1, \ldots, t_n\}$, the range $t_2..t_5$ represents some value from the set $\{t_2, t_3, t_4, t_5\}$. Similarly the fact that *John is in his late thirties* can be described as:

$$age\,(john, 35..39)$$

in which "35..39" stands for a value in the range 35 to 39. It has the same logical meaning as:

$$\exists_{Age(35 \le Age \le 39)}\, age\,(john, Age).$$

The addition of incomplete data necessitates a slight change to the unification mechanism which is used in . [9, 3] Two new definitions of unifiable [12, 13] can be given as follows:

Definition 1 *Unifiable*: If there is a substitution $\theta = \{t_1/v_1, t_2/v_2, \ldots, t_n/v_n\}$, such that an empty clause can be deduced from two literals, $\neg p(X_1, \ldots, X_n)$ and $p(Y_1, \ldots, Y_n)$, i.e.

$$\{\neg(p(X_1, \ldots, X_n)\theta), p(Y_1, \ldots, Y_n)\theta\} \vdash \square.$$

then these two literals are said to be unifiable, and the substitution θ is called a unifier.

A weaker condition is:

Definition 2 *Possibly unifiable*: Two literals $\neg p(X_1, \ldots, X_n)$ and $p(Y_1, \ldots, Y_n)$ are said to be *possibly unifiable* if there exists a substitution θ, a subclause of $p(Y_1, \ldots, Y_n)\theta$ denoted as $p(Y'_1, \ldots, Y'_n)$ and a subclause of $p(X_1, \ldots, X_n)\theta$ denoted as $p(X'_1, \ldots, X'_n)$ such that

$$\{\neg p(X'_1, \ldots, X'_n), p(Y'_1, \ldots, Y'_n)\} \vdash \square$$

In general there two kinds of queries which one might want to put to a database. The first is a *definite query* which seeks to know whether something is definitely true (this corresponds to standard inference or deduction in logic, and the definition of *unifiable* is relevant). The second is a *possible query* which seeks to know whether something could possibly be true (here the definition of *possibly unifiable* is relevant).

To enable users to indicate out what kind of answers they want, two functions *def* and *pos* are provided. If a query is preceded by the function *def*, then the notion of *definitely unifiable* will be applied during query evaluation, which will follow the normal process of logical deduction and generate a *definite answer*. If a query is preceded by the function *pos*, then the definition of *possibly unifiable* will be applied during query evaluation. The answer generated is a *possible answer*.

To illustrate this, consider an example database of information about the engagement and loss of academic staff. Each event in this database has associated with it a unique event identifier and this is used to relate it to the time of the event. For example, the fact that *Tom was hired as a lecturer on the 1st of October 1975* can be expressed in the database as:

hire (*e* 1, *tom*, *lecturer*).
time (*e* 1, 1/10/1975).

the fact that *Tom ceased to be a lecturer in January, 1980 (exact day unknown)* can be expressed as:

leave (*e* 2, *tom* , *lecturer*).
time (*e* 2, 1/1/1980..31/1/1980).

Suppose that the database has a relation *employee_record* (*Start_time* , *End_time* , *Name* , *Rank*) which states that a person *Name* had a *Rank* from *Start_time* to *End_time* . It is defined as:

$$employee_record(T1, T2, Name, Rank) \leftarrow hire(E1, Name, Rank) \wedge time(E1, T1) \wedge leave(E2, Name, Rank) \wedge time(E2, T2).$$

If a query asks whether the date when Tom ceased to be a lecturer was definitely after 15th January 1980, i.e.

$$?- def\ (employee_record(T1, T2, Name, lecturer) \wedge later(T2, 15/1/1980))$$

the answer would be "no". But if the query asks whether the date when Tom ceased to be a lecturer was possibly after 15th January 1980, i.e.

$$?- pos\ (employee_record(T1, T2, Name, lecturer) \wedge later(T2, 15/1/1980))$$

the answer would be "yes", because it is possible that he ceased being a lecturer after the middle of January.

In order to handle queries such as this, one needs to define comparison operators which operate on temporal data (including incomplete temporal data). In the case of the above example these are needed in the definition of the predicate *later* .

6. COMPARISON OF INCOMPLETE VALUES

If the domain of an attribute is a discrete set of values, and the value of an attribute is a single element from this set, then comparison of one attribute value against another is straightforward. However, if the domain of an attribute is ordered, and each attribute value represents a range of domain values, then comparison of two attribute values is more complex. For this purpose a set of operators is introduced.

Let $A \otimes B$ stands for a comparison expression in which A and B are two ranges and $\otimes$ is one of the comparison operators ($<$, $\leq$, $>$, $\geq$, $=$ and $\neq$). Two functions *def* and *pos* can be applied to the comparison expression to specify which kind of inference is required. For example *def* ($A > B$) stands for the fact that whatever the value that A may take within the range specified, it is definitely greater than any value which B may take within the range specified and *pos* ($A > B$) stands for the fact that the value taken by A within the range specified may possibly be greater than the value taken by B within the range specified. The whole set of operators can be given in logic notation as follows:

1) *def* ($A \otimes B$) means that

$$\forall_{a \in A}\ \forall_{b \in B}\ (a \otimes b)$$

2) $pos(A \oplus B)$ means that

$$\exists_{a \in A}\ \exists_{b \in B}\ (a \oplus b)$$

These 12 operators are described in table 1. One thing which should be pointed out is that the *range* here is an incomplete data item which represent a particular unknown value ranging over the *range*. Two ranges having the same lower bound and the same upper bound might represent different values. In other words they are not definitely equivalent to each other.

Other operators including the set of temporal operators given by Allen [1] can be expressed in the same way as above and are shown in table 2.

7. EXAMPLES OF TIME-DEPENDENT INCOMPLETENESS

The combination of *time* and *incomplete information* together can enhance the power of the database. Consider the following examples of engaging and losing academic staff. The database might have such information as:

hire (*e* 1, *mary*, *lecturer*)
time (*e* 1, 10/5/1970)
leave (*e* 2, *john*, *lecturer*)
time (*e* 2, 1/6/1975)
leave (*e* 3, *mary*, *professor*)
time (*e* 3, 1/10/1980)

and the following rules:

employee (*Event*, *Name*, *Rank*) ← *hire* (*Event*, *Name*, *Rank*)

employee (*Event*, *Name*, *Rank*) ← *promote* (*Event*, *Name*, *OldRank*, *Rank*)

employee (*Event*, *Name*, *Rank*) ← *leave* ($Event_1$, *Name*, *Rank*) ∧
(¬*hire* ($Event_2$, *Name*, *Rank*)) ∧
(¬*promote* ($Event_3$, *Name*, *OldRank*, *Rank*)) ∧ *new_event* (*Event*)

employee (*Event*, *Name*, *Rank*) ← *promote* ($Event_1$, *Name*, *Rank*, *NewRank*) ∧
(¬*hire* ($Event_2$, *Name*, *Rank*)) ∧
(¬*promote* ($Event_3$, *Name*, *OldRank*, *Rank*)) ∧ *new_event* (*Event*)

start (*Event*, *Time*) ← *time* (*Event*, *Time*)

start (*Event*, *Start..End*) ← *employee* (*Event*, *Name*, *Rank*) ∧ *end* (*Event*, $Time_1$) ∧
employee ($Event_1$, *Name*, $Rank_1$) ∧

$start(Event_1, Time_2) \wedge earlier(Time_1, Time_2) \wedge$

$(\neg(employee(Event_2, Name, Rank_2) \wedge start(Event_2, Time) \wedge$

$earlier(Time_1, Time) \wedge earlier(Time, Time_2))) \wedge$

$make_range(Time_1, Time_2, Start, End)$

$end(Event, Time) \leftarrow employee(Event, Name, Rank) \wedge$

$promote(Event_1, Name, Rank, NewRank) \wedge time(Event_1, Time)$

$end(Event, Time) \leftarrow employee(Event, Name, Rank) \wedge$

$leave(Event_1, Name, Rank) \wedge time(Event_1, Time)$

$end(Event, Start..End) \leftarrow employee(Event, Name, Rank) \wedge start(Event, Time_2) \wedge$

$employee(Event_1, Name, Rank_1) \wedge$

$end(Event_1, Time_1) \wedge earlier(Time_1, Time_2) \wedge$

$(\neg(employee(Event_2, Name, Rank_2) \wedge end(Event_2, Time) \wedge$

$earlier(Time_1, Time) \wedge earlier(Time, Time_2))) \wedge$

$make_range(Time_1, Time_2, Start, End)$

$employee_record(Time_1, Time_2, Name, Rank) \leftarrow employee(Event, Name, Rank) \wedge$

$start(Event, Time_1) \wedge end(Event, Time_2)$

$employee_record(Time_1, after, Name, Rank) \leftarrow employee(Event, Name, Rank) \wedge$

$start(Event, Time_1) \wedge (\neg end(Event, Time_2))$

$employee_record(before, Time_2, Name, Rank) \leftarrow employee(Event, Name, Rank) \wedge$

$(\neg start(Event, Time_1)) \wedge end(Event, Time_2)$

$earlier(Time_1..Time_2, T3) \leftarrow earlier(Time_2, Time_3)$

$earlier(Time_1, Time_2..Time_3) \leftarrow earlier(Time_1, Time_2)$

$earlier(Day_1/Month_1/Year_1, Day_2/Month_2/Year_2) \leftarrow Year_1 < Year_2$

$earlier(Day_1/Month_1/Year, Day_2/Month_2/Year) \leftarrow Month_1 < Month_2$

$earlier(Day_1/Month/Year, Day_2/Month/Year) \leftarrow Day_1 \leq Day_2.$

Suppose that one then wishes to add the fact that *Mary was promoted from lecturer to professor in June 1975*. Because the system is able to deal with incompleteness of this nature, the above information can be stored into the database as:

promote (*e* 4, *mary* , *lecturer* , *professor*)
time (*e* 4, 1/6/1975..30/6/1975).

If the user asks whether Mary was definitely a lecturer on 31st May 1975, i.e.

? – def (*employee_record* (*T* 1, *T* 2, *mary* , *lecturer*)∧*earlier* (*T* 1, 31/5/1975)∧*earlier* (31/5/1975, *T* 2)).

the answer will be *yes*. If the user asks whether she was definitely a lecturer on 3rd June 1975, i.e.

? – def (*employee_record* (*T* 1, *T* 2, *mary* , *lecturer*)∧*earlier* (*T* 1, 3/6/1975)∧*earlier* (3/6/1975, *T* 2))

the answer will be *no* because from the data available one cannot be certain that Mary was definitely a lecturer on that date. However, if one asks whether she might possibly have been a lecturer on 3rd June 1975, i.e.

? – pos (*employee_record* (*T* 1, *T* 2, *mary* , *lecturer*)∧*earlier* (*T* 1, 3/6/1975)∧*earlier* (3/6/1975, *T* 2))

the answer will be *yes*.

8. CONCLUSION

If time is to be handled adequately in a deductive database, it must be coupled with a mechanism for dealing with incompleteness or uncertainty. The reason for this is partly the fact that the accuracy with which a temporal value is specified, either in a database clause or in a query, defines a level of granularity which is important in finding a solution. This is compounded by the inevitable human difficulties in recording exactly the times of past and future events.

A model is proposed in this paper which combines the notion of incompleteness with that of time, to provide a powerful and consistent approach to solving the problem. The event calculus provides a natural representation of temporal information but on its own does not deal with incomplete information, apart from simple null values. On the other hand, the approach to handling incomplete information described in [12] provides a useful extension to definite clauses in a deductive database. The combination of the two approaches into a single uniform model provides a powerful mechanism for dealing with time.

In order to handle incompleteness, two different kinds of query, definite and possible, have been introduced to enable users to specify their different requirements. This also necessitates two slightly different inference mechanisms, each with its own unification process. These are described in the paper.

The two different forms of query have been realized in a query language, called Squirrel, which is an extension of SQL which caters for rules [11] A database system with Squirrel as its query language has been implemented using the two different inference mechanisms referred to. An overview of the complete system is given in [14]

ACKNOWLEDGEMENTS

This work arises out of work funded under the Alvey programme, and the authors wish to acknowledge the support of the Science and Engineering Research Council and of ICL. Particular thanks are due to Dr. J.M.P.Quinn of ICL and Mr.J.R.Lucking of STC. This work is being taken forward as part of an Esprit project, European Declarative Systems, and once again thanks are due to the EC for their support.

References

1. Allen J.F. (1983), "Maintaining knowledge about temporal intervals," *Communications of the ACM*, **Vol. 26**, (11), pp. 832-843, (1983).

2. Burgess J.P. (1979), "Logic and Time," *Journal of Symbolic Logic*, **Vol. 44**, (4), pp. 566-581, (December 1979).

3. Chang C.L. and Lee R.C.T. (1973), *Symbolic logic and mechanical theorem proving*, Academic Press, New York, N.Y., (1973).

4. Clifford J. and Warren D.S. (1983), "Formal semantics for time in databases," *ACM Transactions on Database Systems*, **Vol. 8**, (2), pp. 214-254, (June 1983).

5. Fiadeiro J. and Sernadas A. (1986), "The infolog linear tense propositional logic of events and transactions," *Information Systems*, **Vol. 11**, (1), pp. 61-85, (1986).

6. Kowalski R.A. (1986), "Database updates in the event calculus," Imperial College, pp. 1-29, (1986).

7. Kowalski R.A. and Sergot M. (1986), "A logic-based calculus of events," *New Generation Computing*, **Vol. 4**, (1), pp. 67-95, OHMSHA, LTD. and Springer-Verlag, (1986).

8. McCarthy J. and Hayes P.J. (1969), "Some philosophical problems from the standpoint of artificial intelligence," in *Machine Intelligence*, ed. B. Meltzer and D. Michie, Edinburgh University Press, Edinburgh, (1969).

9. Robinson J.A. (1965), "A machine-oriented logic based on the resolution principle," *Journal of ACM*, **Vol. 12**, (1), pp. 23-41, (1965).

10. Sernadas A. (1980), "Temporal aspects of logical procedure definition," *Information Systems*, **Vol. 5**, pp. 167-187, (1980).

11. Waugh K.G., Williams M.H., Kong Q., Salvini S., and Chen G. (1990), "Designing SQUIRREL: an extended SQL for a deductive database systems," *The Computer Journal*, **Vol. 33 (6)**, pp. 535-546, (1990).

12. Williams M.H. and Kong Q. (1988), "Incomplete information in a deductive database," *Data and Knowledge Engineering*, **Vol. 3**, pp. 197-220, (1988).

13. Williams M.H., Kong Q., and Chen G. (1988), "Handling incomplete information in a logic database," in *UK IT 88 Conference Publication*, pp. 224-227, (1988).

14. Williams M.H., Chen G., Ferbrache D., Massey P., Salvini S., Taylor H., and Wong K.F. (1988), "Prolog and deductive databases," *Knowledge-Based Systems*, Vol. 1 (3), pp. 188-192, (1988).

Table 1.
Operators for comparing ranges of possible values

operator $\otimes$	full name	explanation $A \otimes B$
pos_l	possibly_less_than	$lb[A] < ub[B]$
pos_g	possibly_great_than	$ub[A] > lb[B]$
pos_e	possibly_equal_to	$ub[A] \geq lb[B] \vee lb[A] \leq ub[B]$
pos_le	possibly_less_or_equal_to	$lb[A] \leq ub[B]$
pos_ge	possibly_great_or_equal_to	$ub[A] \geq lb[B]$
pos_ne	possibly_not_equal_to	$lb[A] \neq lb[B] \vee ub[A] \neq ub[B]$
def_l	definitely_less_than	$ub[A] < lb[B]$
def_g	definitely_great_than	$lb[A] > ub[B]$
def_e	definitely_equal_to	$lb[A] = lb[B] = ub[A] = ub[B]$
def_le	definitely_less_or_equal_to	$ub[A] \leq lb[B]$
def_ge	definitely_great_or_equal_to	$lb[A] \geq ub[B]$
def_ne	definitely_not_equal_to	$ub[A] < lb[B] \vee ub[B] < lb[A]$

Table 2. Allen's operators

operator $\otimes$	explanation $A \otimes B$
before	A def_l B
after	A def_g B
equal	A def_e B
meets	$ub[A] = lb[B]$
overlaps	$lb[A] < lb[B] < ub[A] < ub[B]$
during	$lb[B] < lb[A] < ub[A] < ub[B]$
starts	$(lb[A] = lb[B]) \wedge (ub[A] < ub[B])$
ends	$(lb[B] < lb[A]) \wedge (ub[A] = ub[B])$

51. K.F.WONG and M.H.WILLIAMS, A Superimposed Codeword indexing scheme for handling sets in Prolog databases, in Proceedings of International Symposium on Database Systems for Advanced Applications, Tokyo, Japan, 468-476, 1991.

International Symposium on Database Systems for Advanced Applications *Tokyo, Japan April, 1991*

A Superimposed Codeword Indexing Scheme for Handling Sets in Prolog Databases

Kam-Fai Wong

ECRC GmbH, Arabellastrasse 17, D8000 Munich 81, West Germany.

M.Howard Williams

Dept. of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh, ED1 2HJ, Scotland.

Abstract

While there has been growing interest in the use of Prolog for database applications, the size of these applications is limited by the capabilities of current Prolog systems for handling disk resident clauses. A major impediment is the inordinate amount of time required for retrieval and unification of clauses from a large set stored on disk. Indexing is commonly used in conventional database systems to achieve fast data access to large sets of records. However, the use of variables and non-atomic arguments such as complex structures which may exist in a Prolog clause render conventional indexing strategies unsuitable. To overcome this, several indexing schemes have been developed. The Superimposed Codeword plus Mask Bits (SCW+MB) is one such scheme which is efficient and relatively simple to implement. Sets which are finite lists consisting only of constant terms are useful structures for advanced database systems. Existing indexing schemes for Prolog databases, including SCW+MB, cannot handle sets. In this paper, a modified SCW+MB matching algorithm which can handle set arguments is proposed. Two hardware approaches, based on existing machines - namely CLARE-FS1 and CAFS, for implementing the modified algorithm are suggested.

1. Introduction

The semantics of Prolog has shown to be suitable for implementing various aspects of database systems [3]. However, most current Prolog implementations cannot handle large applications. In a realistic Prolog database application is taken to be of the order of millions of clauses, this considerably exceeds the capabilities of most current Prolog systems. To handle a million clauses by simply extending the main memory would be impractical since the amount of memory required would be excessive. Consequently, secondary disc memory is necessary to store such large data sets. However, handling of large sets of disc resident clauses is very sluggish and the time required to retrieve a million clauses and subsequently to perform unification upon them is enormous. This is a crucial factor which limits the effectiveness of Prolog to small applications.

One way of improving the rate of clause retrieval is to employ some form of indexing strategy. In one class of indexing strategy, each clause in a database is represented by a short codeword. A secondary index file - also known as a surrogate, or a descriptor, file - is formed which associates codewords with their corresponding clause references. In response to a query, the secondary index file is searched and only those clauses whose codewords match the query codeword are extracted. In practice, the size of a secondary index file is much smaller than that of a database file, thereby reducing the amount of information to be examined. In addition the matching process is faster than full unification. For these reasons, the time required to perform unification over the full set of clauses is considerably reduced.

Indexing is commonly used in large relational databases. There are two types of index matching:

(1) *Exact matching* requires the existence of an unique attribute in a database relation. It is referred to as the primary key of the relation. Each record is encoded according to the hashed value of the attribute. A non-overlapping one to one correspondence relationship is formed between codewords and references in the secondary index file. The codewords are matched against a query index, generated from the primary key of the query, during data retrieval. A satisfier in the matching process is a definite answer to a query.

(2) *Potential matching* produces a collection of candidate tuples which are most likely answers to a query. Several or all attributes of a relation are hashed and combined together in some fashion to produce a codeword. There is no requirement of attribute uniqueness. One or more non-unique attributes are chosen as secondary keys. The codewords are maintained in a secondary index file and match against the query index during a query. The answer to a query is a set of records whose keyed attributes matched those specified in a query. Within the solution set, there will be a small proportion of 'false

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the DASFAA copyright notice and the title of the publication and its data appear, and notice is given that copying is by permission of the DASFAA Steering Committee. To copy otherwise, or to republish, requires a fee and/or special permission from the Steering Committee.

drops', i.e. incorrect answers whose codewords successfully match with the query codeword.

In conventional database systems, indexes are formed by encoding one or more attributes selected as a primary key. Thereafter, any retrieval based on that set of attributes can be completed swiftly; on the other hand, if any of the attributes in the primary key is not specified, indexing becomes ineffective. For this reason, the selectivity (or discrimination power) of such schemes is very restrictive and limited to a specific domain. This is difficult to ensure in a Prolog database environment where greater flexibility of attributes in a query is generally required. A suitable indexing strategy for Prolog databases, therefore, should overcome this restriction. Another essential property of a suitable indexing strategy is the ability to cope with database variables and various complex data structures supported in Prolog. Indexing methods based on exact matching cannot fulfill this special requirement. At the present time, several different indexing schemes are available [1,2,5,9,10], all of which are based on potential matching.

One of the indexing policies for Prolog, namely Superimposed Codeword plus Mask Bits (SCW+MB) [9], is being further investigated. Compared with other indexing strategies, the size of SCW+MB indexes is smaller for large applications [3,10] and it is simple to implement. Although, all the Prolog indexing schemes support variables and complex structures, none of them can handle sets. A Set, represented by a finite list of constant terms, is a useful data structure for advanced database applications. (e.g. N^2F databases system). In this paper, a matching algorithm based on SCW+MB for detecting certain set relationships is proposed. This algorithm is best implemented in hardware. Rather than designing a new piece of hardware, the implementation can be simplified by utilising existing hardware. Two approaches are proposed. One is based on an extension on the First Stage Filter of the CLAuse Retrieval Engine (CLARE-FS1) [11]. The extension will be small and could be achieved easily on the existing CLARE-FS1 prototype. An alternative is based on porting the algorithm onto the ICL Content Addressable File Store (CAFS) [4].

The paper is arranged as follows : the next section gives an overview of existing indexing schemes suitable for Prolog databases. In particular, SCW+MB is explained in more detail in section 3. Section 4 describes the modified SCW+MB indexing algorithm. Two possible hardware implementations of this algorithm are proposed in section 5.

2. Indexing Prolog Clauses

A practical large database applications using Prolog (e.g. molecular biology application [6]) is envisaged to have the following properties :

(1) Non-atomic arguments - such as complex structures and embedded variables - exist in database clauses.

(2) For each predicate there is no obvious primary key, each of the arguments of a predicate may be variable in a query.

(3) Predicates with large arities (>10 arguments) may occur.

Currently, there are several indexing schemes which have been specially adopted for Prolog database applications. They are Superimposed Code Word plus Mask Bits (SCW+MB) used in NU-Prolog [5], Superimposed Code Word with Position and internal Variables (SCW+PV) for Prolog-SCX [2], Structural Superimposed Code Word (SSCW) suggested for KBMS-PHI (a logic programming language) [9], Concatenated Code Word (CCW) introduced by Syracuse University [1] and Field Encoded Word (FEW) employed in UNSW-Prolog [10]. Each of the schemes account for the above properties to some extent.

A Prolog database with codeword indexing uses two files : a surrogate (descriptor) file containing the indexes and a data file containing the clauses. Database clauses are converted to indexes and are stored in a surrogate file alongside the corresponding clause references (addresses of clauses within the data file). The execution of a a query begins with a search of the surrogate file, finding the clauses whose indexes match the index of the query. The system then extracts the clause references associated with the matched indexes, retrieves the clauses from the data file using the references and finally subjects the clauses to full unification.

2.1. Formation of Indexing Codewords

Index codeword for database clauses in each of the methods described in [1,2,5,9,10] are produced in a similar fashion. In each case the process is carried out in two stages : (1) encoding and (2) combining. Details of the two stages are different for the different schemes. During the encoding stage, each argument of a clause head is passed through a hashing function and a binary codeword (bcw) is produced. Different schemes employ different hashing functions. Apart from CCW, the FEW, SCW+MB, SCW+PV and SSCW schemes use special n-in-m hashing functions to form bcws. Using such functions, for each argument, the number of bits set (n) in an index of size m is constant.

In the combining stage, bcws of individual clause arguments are integrated. FEW and CCW simply concatenates the bcws to form longer codewords; the other SCW based schemes form their indexes by superimposing - logically or'ing - bcws of all arguments. The three SCW schemes differ from each other in the way they represent positional information and embedded variables. SCW+MB encodes neither positional information nor variables; they are represented in the Mask Bits which is an additional word associated with the index (see section 3). In SCW+PV both positional information and variables are part of the codeword; this is achieved by including them in the bcws during the encoding stage. SSCW is a hybrid of FEW and SCW. Inherited from FEW, a bcw of the SSCW scheme may be divided into fragments to account for embedded arguments in structures; variables are represented by bcws with all 1s. To form the final SSCW, bcws of all clause arguments are superimposed. Intrinsic to the codeword structure of SSCW, FEW and CCW, representation of positional information is explicit. Variables in FEW and SSCW are represented by setting the bits corresponding to the variable argument to 1s. Ordinary CCW can only handle variable in a query and does not cater for variables in database clauses. The enhanced CCW [8] which provides CCW with database vari-

```
/* For simplicity, a SCW indexing codeword is assumed typed long */
typedef unsigned char   SCWtype

#define BOOLEAN       unsigned char
#define BITWISE_AND   &
#define OR            ||
#define RIGHT_SHIFT_BY >>

BOOLEAN SCW_MBmatching(query_argument_index,query_argument_count,database_index,database_mask)
SCWtype   query_argument_index[];
char      query_argument_count;
SCWtype   database_index;
unsigned char database_mask;
{
    BOOLEAN Satisfier;
    char count;

    for (count=1; count!=query_argument_count; count++)
    {
        if((((query_argument_index[count] BITWISE_AND database_index))==database_index) OR
          (((database_mask RIGHT_SHIFT_BY (count-1)) BITWISE_AND 1) == 1))
            Satisfier=1; else { Satisfier=0; break; }
    }
    return(Satisfier);
}
```

Figure 1: Superimposed Codeword plus Mask Bit matching algorithm.

able handling ability, represents variables in the same way as SCW+MB, by using mask bits.

2.2. Index Matching

A similar process is applied to a query to generate a Query Code Word (QCW). The only difference between a QCW and a database index codeword lies in the treatment of variables. In SCW+MB and SCW+PV [2,5], query arguments are encoded and matched against database indexes individually. The criterion for a successful match is : either the bits which are set in a QCW are also set in a database index or the argument (query and/or database) is a variable. A database clause is a HIT if QCWs of all query arguments have satisfied the criterion. SSCW and FEW follow the same matching criterion. However, different from SCW+MB and SCW+PV, individual query arguments are not compared separately. Similarly, no separate comparison is necessary in CCW matching; only one QCW, encoding all query arguments, is required. Under the CCW matching algorithm, a database clause is a satisfier if its index is identical to the QCW. Nevertheless, the enhanced CCW [8] which supports database variables requires query arguments to be compared separately. QCWs of all query arguments are compared with the corresponding bcws of the database clause individually. A query and a database arguments matched with each other if either their codewords are identical or the associated mask bit is set.

Each indexing scheme has its pros and cons. The selection of a suitable scheme is application dependent. In general, CCW performs best if a database consists mainly of ground clauses (i.e. variable free) and the arities of the predicates are small. FEW is suitable for database predicates with simple regular structures and small arities. SCW+MB works best for applications where argument order is not relevant and structures are not too complex (e.g. nesting level not more than 3). The characteristics of SCW+PV are similar to SCW+MB. At the expense of increased codeword size, SCW+PV's discriminatory power is better than that of the SCW+MB scheme for the same set database clauses. The SSCW scheme is best for clauses with nested structures.

The SCW+MB indexing scheme is simple and offers an adequate discriminatory power for large database applications [5]. Furthermore, compared with the other schemes, for the same design parameters, the size of SCW+MB codewords (and hence the surrogate file size) is smallest for the same database [2,13]. For these reasons, it is selected and being employed in [12]. In this paper, the indexing capability of SCW+MB is exploited further to cope with set matching.

3. Superimposed Codeword Plus Mask Bits (SCW+MB)

Originally, superimposed codeword (SCW) indexing was proposed as a means of achieving fast retrieval in a relational database [7]. In implementing database applications in Prolog, a Prolog predicate corresponds to a relation and its argument to the attributes in a relational database. However, the arguments of a Prolog clause are more general than the attributes of a relation and may include variables and complex structures. This makes the original SCW indexing scheme inapplicable. For example, using Prolog to describe a Restaurant relation, a predicate 'restaurant' is employed, i.e. restaurant(Manager,Name,Address,Famous_Food). Suppose there is a Witchery restaurant which is a subsidiary of the Thaumaturgy Department Store and which will therefore move to wherever the Department Store is. The manager post of the restaurant is currently unoccupied and the restaurant is renown

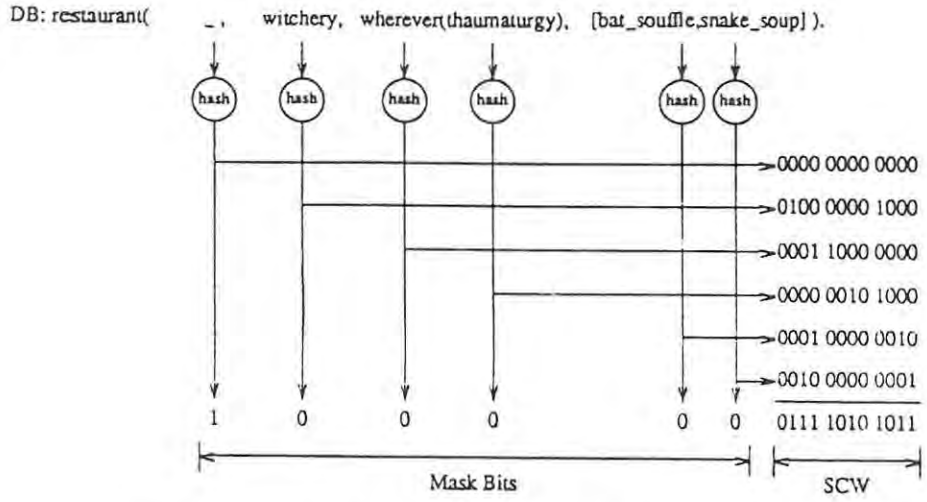


Figure 2: Generation of SCW+MB for a database clause.

Table 1: Intermediate matching outputs

Arg Position	DB codewords	Q codewords	Mask Bits	Outputs	Comments
#1	0111 1010 1011	1100 0000 0000	1	1	Disabled
#2	0111 1010 1011	0100 0000 1000	0	1	Q⊂DB
#3	0111 1010 1011	0001 1000 0000	0	1	Q⊂DB
#4	0111 1010 1011	0000 0010 1000	0	1	Q⊂DB
#5	0111 1010 1011	0001 0000 0010	0	1	Q⊂DB
#6	0111 1010 1011	0000 0000 0000	0	1	Q⊂DB

for its servings of bat_souffle and snake_soup. A Prolog clause describing the Witchery restaurant is:

```
restaurant(_,witchery,wherever(thaumaturgy),[bat_souffle,snake_soup]).
```

In this example, argument one is a variable and arguments three and four are embedded structures.

By itself, the original SCW is incapable of representing either embedded structures or variables in Prolog clauses. A modification to the codeword strategy has been proposed [5], referred to as Superimposed Codeword plus Mask Bits (SCW+MB), which employs an additional field in the codeword to record positional information in a clause. In its simplest form, one mask bit is used for each argument position. If a mask bit is set, the corresponding argument position is a variable and it will match with any query term. Conversely, if a bit is not set, it implies that the corresponding argument position is a constant. Constant arguments are treated as usual - they are hashed individually, forming bcws and these are superimposed to produce the ultimate codeword. Query matching is more complex than for an ordinary SCW. When a query is encountered, hashing is performed upon each of the query arguments to generate binary codewords. Unlike ordinary SCW matching, these query codewords are not or'ed together but instead are compared with database codewords individually. A query argument is a HIT if the pattern of 1s of its codeword is masked off by the database codeword or its corresponding mask bit is set. A database clause is a satisfier if all query arguments are HITs. Figure 1 shows an algorithmic representation of the matching process in C. To account for embedded structures, several mask bits are used (one bit per nested argument). Encoding and matching are extended to functor names and nested arguments of an embedded structure. It is worth noting that mask bits are only required by database clauses and they do not exist in query codewords.

The mechanism for generating the required SCW+MB index for the restaurant/4 clause is shown in figure 2. In the example, it is assumed that bcws of each argument are 12 bits in length and 2 bits are set by hashing. In addition, 6 bits are used for the mask bits. The allocation of the mask bits are 1,1,2 and 2 for the first, second, third and fourth arguments, respectively. Notice that bit 1 is set because the first argument is a variable (_). The same hashing mechanism is applied to each argument of a query. Query codewords are not superimposed; instead, they are compared with the knowledge base superimposed codewords individually.

As an example of index matching, the following query is posed :

```
restaurant(devil,witchery,whatever(thaumaturgy),[bat_souffle|Whateverelse]).
```

Literally, this query attempts to determine what else the 'witchery' restaurant serves apart from bat_souffle. In addition to the restaurant's name, one known famous dish and the infor-

mation relating to the restaurant's address, the name of the manager, 'devil', are specified in the query. Six QCWs are generated, one for each argument term. They are compared with the database indexes individually using the algorithm shown in figure 1. The query is compared with the clause shown in figure 2. Table 1 summarises the intermediate matching results. The database clause is a HIT since all intermediate outputs are positive (column 5 of table 1).

4. Extended Functionality: Set Handling

Sets are useful data structures for representing enumerated items. Current Prolog systems are already inefficient in handling large sets of ordinary facts. The situation will be much worse if sets are included. A set encompasses a finite list of constant terms - terms within a set are unordered.

Example: Suppose a computer supplier Smith in the UK supplies three separate hardware items, PCs, line printers (LPs) and VDUs. This may be expressed as three Prolog clauses.

```
supplier(smith,pc,uk).
supplier(smith,lp,uk).
supplier(smith,vdu,uk).
```

or as a single clause

```
supplier(smith,[pc,lp,vdu],uk).
```

However, if one searches for those suppliers who supply printers, the latter form is difficult to handle. A better approach would be to express this using set, viz

```
supplier(smith,{pc,lp,vdu},uk).
```

In this case one might apply queries involving set operators such as those given in table 2. In figure 3, examples of queries utilising these set operators are shown.

Operator	Semantics
intersect(Q)	$(Q \cup DB) \neq \emptyset$
subset(Q)	$Q \subseteq DB$
superset(Q)	$Q \supseteq DB$
setequal(Q)	$Q = DB$
member(q)	$q \in DB$
NB: Q & DB are sets and q is a constant.	

Table 2: A list of set operators.

At present, existing indexing schemes for Prolog do not support sets. Since the order of the elements within a set is irrelevant and SCW+MB indexing can handle unordered arguments efficiently, SCW+MB is a practical scheme for dealing with set arguments. Nevertheless, as it stands, SCW+MB can only cope with variables, structures and constants. To handle sets, a new matching algorithm is required. Figure 4 shows a modified SCW+MB matching algorithm (in pseudo C) for handling sets.

The difference between the modified and the ordinary SCW+MB matching mechanisms lies in the ways they handle results of the argument comparisons. Under the SCW+MB matching algorithm (see figure 1), a query is a HIT if all its arguments' bcws compared positively with the database index (the 'for' loop in figure 1). As soon as any of the arguments fails the matching process is aborted (the 'break' in the 'else' statement). Effectively, the result of an argument comparison

A Database clause :

```
supplier(smith,{pc,lp,vdu},uk).
```

Queries :

```
?- supplier(smith,intersect({pc,mouse}),uk).
Who = smith

?- supplier(Who,subset({pc,lp}),uk).
Who = smith

?- supplier(Who,superset({pc,lp,vdu,robot}),uk).
Who = smith

?- supplier(Who,setequal({lp,pc,vdu}),uk).
Who = smith

?- supplier(Who,member(pc),uk).
Who = smith
```

Figure 3: Queries involving set operations.

(the 'Satisfier' variable) is monitored once at the end of the comparison; and thereafter it is discarded.

To handle set arguments, the matching algorithm is more complex (figure 4). The results of each argument comparison are not discarded; instead they are stored in a bit array (variable 'task_word' in figure 4) for later use. Modified SCW+MB matching is performed in two stages. In the first stage, SCW+MB matching (SCW_MBmatching()) is performed and the results of all comparisons are collected in the 'task_word'. The 'task_word' is passed to the second stage for set matching (SetMatching()). During the second stage, the 'task_word' is masked with the appropriate bit patterns ('result_mask') according to the set operation (see table 2) specified at compile time. One result_mask is generated for each set argument in a query. It is used to isolate set arguments from the main query. By doing so, the set argument can be processed separately. Conceptually, matching of a query can be considered as being performed in two levels : the top level and the set level. For example, matching of the first query in figure 3 (L0) can be regarded as a composite of the following two sub-matchings (L1 & L2):

```
ORIGINAL QUERY :
L0      supplier(smith,intersect({pc,lp}),uk).
TOP_LEVEL :
L1      supplier(smith,_,uk).
SET_LEVEL :
L2      supplier(_,intersect({pc,lp}),_).
```

```
/* For simplicity, a SCW indexing codeword is assumed typed long */
typedef     unsigned long SCWtype

#define BOOLEAN       unsigned char
#define BITWISE_AND    &
#define BITWISE_OR     |
#define OR             ||
#define RIGHT_SHIFT_BY >>

/*BITS is a predefined type*/
BITS        result_mask[NUMBER_OF_SET_EXPECTED][NUMBER_OF_ARGUMENTS];
BITS        result_word[NUMBER_OF_ARGUMENTS];
char        command[NUMBER_OF_SET_EXPECTED];

SCW_MBmatching(query_argument_index,query_argument_count,database_index,database_mask)
SCWtype     query_argument_index[];
char        query_argument_count;
SCWtype     database_index;
unsigned char  database_mask;
{
    char count;

    for (count=0; count!=query_argument_count; count++)
    {
        if((((query_argument_index[count] BITWISE_AND database_index))==database_index) OR
          (((database_mask RIGHT_SHIFT_BY (count)) BITWISE_AND 1) == 1))
            result_word[count]=1; else { result_word[count]=0; break; }
    }
        return(SetMatching(result_mask,result_test,command));
}
```

Figure 4a: The modified SCW+MB Matching algorithm.

lore than one set argument is permissible - in figure 4, the umber of set argument accountable for is determined by UMBER_OF_SET_EXPECTED. Applying SCW+MB atching to the query LO and the database clause in figure 3, e following result is produced :

Table 3: Result of SCW+MB matching of query example LO

rguments	DB codewords	Q codewords	Mask Bits	Result
smith	0111 1010 1011	0110 0000 0000	0	1
pc	0111 1010 1011	0100 0000 1000	0	1
lp	0111 1010 1011	0001 1000 0000	0	1
vdu	0111 1010 1011	1100 0000 0000	0	0
uk	0111 1010 1011	0001 0000 0010	0	1

lotice that under the normal SCW+MB matching criterion, is query is a failure because a '0' is resulted from the com-rison of 'blonde'.) The resulting bits are concatenated to rm the result_word in the modified scheme; in the example, e result_word is 10111. Different masking operations and atching criteria may be applied to a result_word (the case tement in figure 4) :

se TOP_LEVEL

A result_word is or'ed with the TOP_LEVEL bit pattern. For the example, the bit pattern is kept in result_mask[0]. It is 01100 - bits 0, 1 and 4 correspond to the top level arguments. Or'ing 01100 with 10111 gives 11111. All 1s in top level matching implies that the top level arguments are successfully matched.

Case INTERSECT

A result_word is and'ed with the INTERSECT bit pattern. For the example, the bit pattern is kept in result_mask[1]. It is 01100 - bits 2 and 3 correspond to set arguments. 01100 and'ing with 10111 gives 00100. If the resulting bit pattern contains one or more 1s, this set argument is a HIT.

Case SUBSET

A result_word is or'ed with the SUBSET bit pattern which is a string of 0s patched with 1s in the set argument positions. The set argument is a HIT if a string of 1s is resulted after the 'or' operation.

Case SUPERSET

A result_word is and'ed with the SUPERSET bit pattern which is a string of 1s patched with 0s in the set argument positions. The argument is a HIT if one or more 1s is resulted after the 'and' operation.

Case SETEQUAL

The SCW of a database clause containing a set argument is produced slightly differently. The position and the cardinality of a set are also included in a codeword. For example the SCW of the the database clause in figure 3 is produced by the following :

hash(smith) OR hash((3,2)) OR hash(pc) OR hash(lp) OR hash(uk)

where hash(X) produces an n-in-m hashed value for X. Hash((2,2)) is the hashed value of the set argument {pc,lp}; (2,2) implies second argument in the clause and

```
BOOLEAN SetMatching(result_mask,result_word,command);
BITS result_mask[][];
BITS result_word[];
char command[];
{
        BOOLEAN Sausfier;
        char count;

        for (count=0; count!=NUMBER_OF_SET_EXPECTED; count++)
        {
                switch (command[count])
                {
                        case TOP_LEVEL :
                                if((result_mask[count] BITWISE_OR result_word)==1)
                                        Satisfier=1; else
                                        {Satisfier=0; break;}
                        case INTERSECT :
                                if((result_mask[count] BITWISE_AND result_word)!=0)
                                        Satisfier=1; else
                                        {Sausfier=0; break;}
                        case SUBSET :
                                if((result_mask[count] BITWISE_OR result_word)==1)
                                        Sausfier=1; else
                                        {Sausfier=0; break;}
                        case SUPERSET :
                                if((result_mask[count] BITWISE_AND result_word)!=0)
                                        Satisfier=1; else
                                        {Sausfier=0; break;}
                        case SETEQUAL :
                                if((result_mask[count] BITWISE_AND result_word)!=0)
                                        Satisfier=1; else
                                        {Sausfier=0; break;}
                        case MEMBER :
                                if((result_mask[count] BITWISE_AND result_word)!=0)
                                        Satisfier=1; else
                                        {Sausfier=0; break;}
                }
                if(Sausfier==0) break;
        }
        return(Satisfier);
}
```

Figure 4b: The modified SCW+MB matching algorithm. (cont'd)

it contains 2 elements. For set equal matching, the same procedure as SUPERSET is applied; except that the query bcws for the set argument also include the position information and the cardinality of the set.

Case MEMBER

When a member(X) set operation is posed, the query manager first ensures that the argument involved is a constant before invoking the SCW+MB matching algorithm. Once this condition is ensured, detection of a MEMBER relationship is the same as INTERSECT.

5. Hardware Assisted Set Matching

The modified matching process is more complex than the ordinary one. Software realisation of the ordinary SCW+MB matching algorithm (figure 1) is inefficient. Various specialised hardware devices have been proposed, such as the first stage filter of the CLAre Retrieval Engine (CLARE-FS1) [11] for improving indexing efficiency. With the increased complexity, executing the modified algorithm in software will be more inefficient. Hitherto, there are no known hardware suitable for SCW+MB matching with set handling ability. Rather than designing a new piece of hardware from scratch, one cost-effective solution is to extend existing hardware. CLARE-FS1 serves this purpose nicely because it is specially designed for SCW+MB matching and the existing prototype can be modified easily. CLARE-FS1 basically consists of a set of parallel comparators - one for each query argument (currently 16 of them, see figure 5a). The comparators are made up of PLAs programmed with the SCW+MB matching criteria. The outputs of the comparators are tied together to an and gate. Changes can be easily made at the output stage. Rather than connecting the comparator outputs directly to an and gate, few banks of logic units - such as PLAs or LSI devices - could be placed in parallel between the comparators and the and gate (figure 5b). One bank of logic unit would be associated with one set matching operation. Each bank would also be connected to a register; this register contains the bit pattern for set matching (i.e. result_mask in figure 4). At query time, the logic units would be programmed to perform a set operation or be disabled. A database clause is a HIT, if the outputs of the logic units are all 1s.

Another hardware machine which can be used for implementing the modified SCW+MB matching algorithm is the

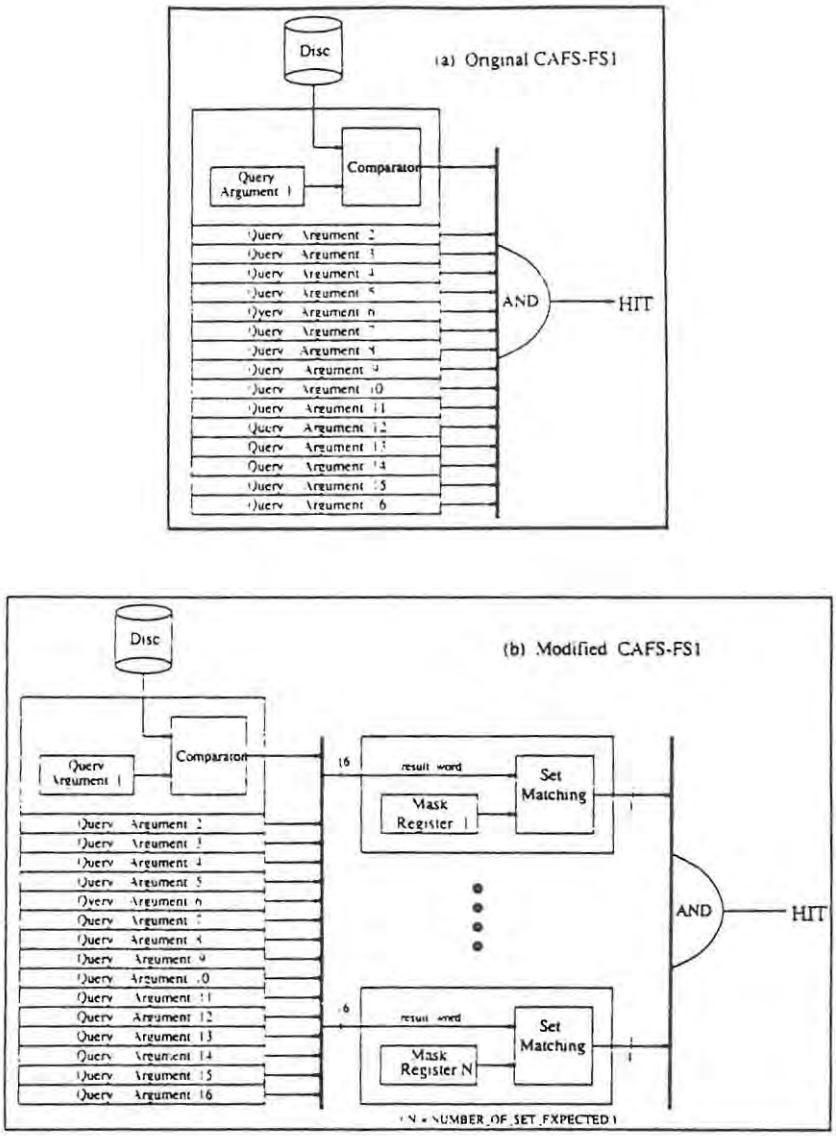


Figure 5: The architectures of (a) the original and (b) the modified CLARE-FS1.

ICL Content Addressable File Store (CAFS) [4]. CAFS has a similar architecture as CLARE-FS1; however, it provides more functionalities. It is a backend database machine designed for ICL mainframes. CAFS consists of 16 parallel comparators (key channels, KC), as shown in figure 6. The Logical Formatter Unit (LFU) directs the data from the disc to the KC. Unlike CLARE-FS1 the comparator (KC) outputs are not tied together. The outputs form a 16-bit 'task word'; this word is streamed to other units for post-processing. The Search Evaluation Unit (SEU) can perform testing, shifting and masking on the 'task-word'. After the SEU, the 'task-word' is passed to the Retrieval Processing Unit (RPU) which consists of a 32 bits general purpose microprocessor. To implement the modified indexing scheme, the RU could be programmed to execute the set matching algorithm. When using CAFS for SCW+MB indexing, 2 key channels are required - one for SCW and one for MB. This means that the number of query arguments accountable for will be 8 which will be insufficient for real Prolog database applications. This restrictions makes the CAFS approach unfavourable.

6. Conclusion

Sets are useful data structures for advanced database applications. Introducing sets as arguments of clauses can make a Prolog database more compact; however, this is achieved at the expense of increased system complexity. In real applications, millions of clauses may exist in a Prolog database. To retrieve database clauses rapidly, indexing techniques are used. One such technique is the SCW+MB indexing scheme. It is a simple and efficient scheme specially for handling large sets of Prolog clauses. Nevertheless, the ordinary SCW+MB policy does not support sets. For this reason, a modified SCW+MB scheme is devised. The modified algorithm is complex and will render software implementation

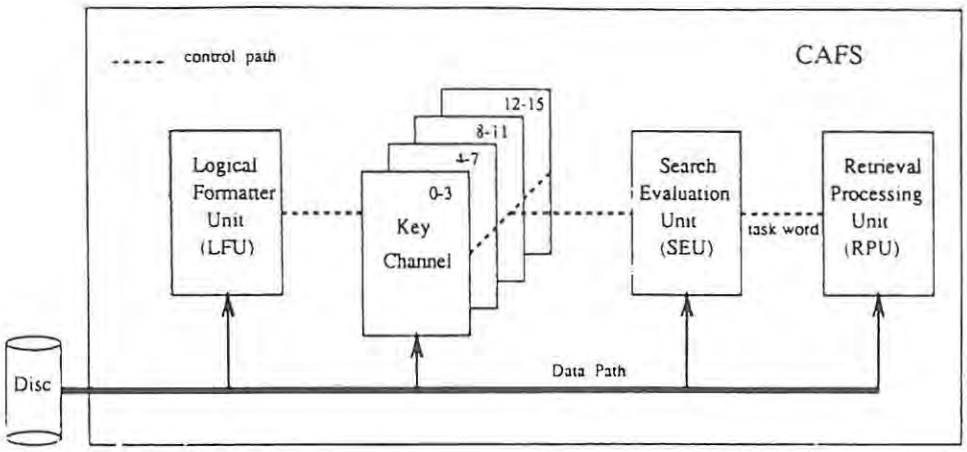


Figure 6: The Architecture of CAFS

inefficient. To improve indexing performance, two hardware approaches are proposed. Both approaches make use of existing hardware machines. The first approach is centred on CLARE-FS1 which is more universal because it was designed with a VMEbus interface. It is reasonably cheap and can be modified easily. The modification will involve adding to the existing hardware, several banks of logic devices and registers - one for each set arguments. The second approach is based on CAFS which is more expensive and is dedicated for ICL mainframes. Implementation of the modified indexing schemes on CAFS will not require any changes. It mainly involves grouping the comparators (key channels) into pairs and programming the Retrieval Unit to post-process the result_word. The disadvantage of the CAFS approach is the limited number (currently 8) of query arguments which can be accounted for. In both cases, the hardware machines provide adequate search rate to ensure clause retrieval in real-time.

Acknowledgements

Thanks are due to the DPS group at ECRC GmbH for supporting the first author to continue the SCW+MB indexing research activity and to the SERC/Alvey and ICL for their support of the Prolog Database Machine project at Heriot-Watt University from which these ideas originated.

Reference

[1] Berra, B.P., Chung, S.M. & Hachem N.I., "Computer Architecture for a Surrogate File to a Very Large Data/Knowledge Base", IEEE Computer, pp25-32, (March 1987).

[2] Colomb, R.M., "Enhancing Unification in Prolog Through Clause Indexing", TR-FB-88-06, CSIRO, Australia, (24 June, 1988).

[3] Gallaire, H., Minker, J. and Nicholas, J-M., "Logic and Database : A Deductive Approach", ACM Surveys, vol.16, no.2, pp153-185, (June 1984).

[4] Macphail, N., "Development of the CAFS-ISP controller product for Series 29 and 39 systems", ICL Technical Journal, vol.4, issue 4, pp393-401, (Nov., 1985).

[5] Ramamohanarao K. and Shepherd J.,"A Superimposed Codeword Indexing Scheme for Very Large Prolog Databases", Proceedings of 3rd International Conference on Logic Programming, London, U.K., pp569-576, (1986).

[6] Rawlings, C.J., Seifert, K. and Saldanha, "A Large Knowledge-based System for Molecular Biology", 1st Workshop on Knowledge Manipulation Engines (SIGKME), University of Reading, England, pp K1-K10,(1987).

[7] Roberts, C.S., "Partial-Match Retrieval via the Method of Superimposed Codes", Proceedings of the IEEE, vol.67, no.12, pp1624-1642, (Dec., 1979).

[8] Shin, D. and Berra, P.B., "An Architecture For Very Large Rule Bases Based on Surrogate Files", *Database Machines and Knowledge Base Machines*, Kitsuregawa, M. and Tanaka, H. (eds), Klumer, pp557-570, (1988).

[9] Wada, M., Morita, Y., Yamazaki, H., Yamashita, S., Miyazaki, N. and Itoh, H., "A Superimposed Code Scheme for Deductive Databases", *Database Machines and Knowledge Base Machines*, Kitsuregawa, M. and Tanaka, H. (eds), Klumer, pp571-584, (1988).

[10] Wise, M.J. & Powers, D.M.W., "Indexing Prolog via Superimposed Code Words and Field Encoded Words", 1984 International Conference on Logic Programming, IEEE Computer Society Press, pp203-210, (Feb., 1984).

[11] Wong, K.F. and Williams, M.H., "Partial Matching Hardware for Selective Clause Retrieval in Large Prolog Knowledge Bases", Technical Report 88/2, Dept. of CS, Heriot-Watt University, Edinburgh, Scotland, (1988).

[12] Wong, K.F. and Williams, M.H., "Design Considerations for a Prolog Database Engine", 3rd International Conference on Data and Knowledge Bases, Jerusalem, Israel, pp111-119, (28-30 June 1988).

[13] Wong, K.F., "Comments on *Comparison of Concatenated and Superimposed Code Word Surrogate Files For Very Large Data-Knowledge Bases*", *Information Processing Letters, vol.33, no.1, pp45-52, (27 Oct., 1989)*

52. S.K.DAS and M.H.WILLIAMS, A path finding method for constraint checking in deductive databases, Data and Knowledge Engineering, vol 4, 223-244 (1989).

Data & Knowledge Engineering 4 (1989) 223–244
North-Holland

A path finding method for constraint checking in deductive databases

Subrata K. DAS and M. Howard WILLIAMS

Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ, Scotland, U.K.

Abstract. A method is presented for checking integrity constraints in a deductive database in which verification of the integrity constraints in the updated database is reduced to the process of constructing paths from update literals to the heads of integrity constraints. In constructing such paths, the method takes advantage of the assumption that the database satisfies the integrity constraints prior to the update. If such a path can be constructed, the integrity of the database has been violated. Correctness of the method has been proved for checking integrity constraints in stratified deductive databases. An explanation of how this method may be realised efficiently in Prolog is given.

Keywords. Integrity constraints, Deductive databases.

1. Introduction

Logic programs have a database interpretation, which generalises the concept of a relational database. A *relational database* [6, 20] is equivalent to a logic program consisting solely of ground unit clauses, whereas a *deductive database* [10, 15, 19] comprises both rules and unit clauses.

In a database an *integrity constraint* [7, 9, 20] (or simply a *constraint*) is a formal representation of a property which the data in the database must satisfy to be consistent with some model of the real world from which it comes. In a logic programming context, a constraint can be defined as a closed first order formula. For example, the constraint 'every worker has a manager' can be expressed as the first order formula

$$\forall x(\text{Worker}(x) \rightarrow \exists y\ \text{Manager}(y, x)) \quad (*)$$

The simplest approach to maintaining consistency involves the evaluation of each constraint whenever the database is updated. However, such an approach is too costly, especially for large databases, and does not make use of the fact that the database satisfies the constraints prior to the update. Different ways of maintaining integrity in deductive databases which do make use of the latter fact have been proposed in several papers, e.g. generalisations of the simplification method [16] by Lloyd and Topor [15], Decker [8] and Bry et al. [2], a general theorem-proving technique by Sadri and Kowalski [17], using Prolog not-predicate by Ling [12], etc.

In this paper a method is presented for constraint checking in deductive databases which will be referred to as the path finding method. This method is an incremental one which does take advantage of the assumption that the database satisfies the constraints prior to the update. In this method, verification of the constraints in the updated database is equivalent to the process of construction of paths from update literals to the heads of constraints.

0169-023X/89/$3.50 © 1989, Elsevier Science Publishers B.V. (North-Holland)

Violation of integrity can be shown by successfully constructing one such path. When a transaction is performed on a database, the method computes in stages the set of ground atoms which are the facts added to the database due to the transaction and another set of partially instantiated atoms which represents the facts deleted from the database due to the transaction. The method avoids reasoning with two database states, i.e. the database before and after update, and can be implemented efficiently in Prolog by taking full advantage of Prolog's computation mechanism.

The paper is organised as follows. The following section describes some basic concepts. In section 3, the path finding method is described with the help of several examples. A proof of the correctness of this method is given in section 4. Section 5 deals with an efficient Prolog implementation of the method and the code for the main algorithm is proposed. Some optimisations which can be applied are also discussed in this section. The implementation of the method has been carried out in a C-Prolog [4] environment. A brief introduction to other constraint checking methods is given in section 6, followed by a comparison of these methods with the path finding method.

2. Basic concepts

This section introduces some of the basic concepts and terminology which will be used in later sections.

Definition. A *definite database clause* (or simply a *clause*) is a formula of the form

$$A \leftarrow L_1, \ldots, L_n \quad n \geq 0,$$

where A is an atom denoting the conclusion and $L_1, \ldots, L_n$ are literals representing conditions. Each L_i is either an atom (a positive condition) or a negated atom (a negated condition). Any variables in $A, L_1, \ldots, L_n$ are assumed to be universally quantified over the whole clause. The commas in the antecedent $L_1, \ldots, L_n$ denote conjunction. A is called the *head* and $L_1, \ldots, L_n$ the *body* of the clause. If $n = 0$ the clause is called a *fact*; otherwise, it is called a *rule*.

A *definite deductive database* D is a finite set of clauses. Throughout the rest of the paper any references to the term 'database' will be taken to imply 'definite deductive database'.

Definition. A *database goal* (or simply a *goal*, or a *query*, or a *denial* [17]) is a formula of the form

$$\leftarrow L_1, \ldots, L_n$$

where $L_1, \ldots, L_n$ are literals. Any variables in $L_1, \ldots, L_n$ are assumed to be universally quantified over the whole goal, the commas in the antecedent $L_1, \ldots, L_n$ denoting conjunction.

Definition. A literal L is said to be an *instance* of a literal L' if there exists a substitution θ such that $L'\theta = L$. If L is ground, then L is called a *ground instance* of L'. Also, L' is called a *generalisation* of L.

Definition. Let G be a goal $\leftarrow A_1, \ldots, A_m, \ldots, A_k$, and C be $A \leftarrow B_1, \ldots, B_q$. If θ is a most general unifier (abbr. mgu) of A_m and A then the goal

$$\leftarrow(A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$$

is a *resolvent* of G and C. If $k = 1$ and $q = 0$ then the resolvent goal always succeeds.

Definition. A computation rule is *safe* if negative literals may only be selected for evaluation if they are ground.

A safe computation rule is required for querying the database correctly. Unless otherwise stated, in the rest of this paper a computation rule has been assumed which selects the leftmost literal from a goal through a safe literal selection strategy. Most Prolog systems do not enforce this safeness condition with the result that they will compute a query correctly only if all positive literals of the query and the program clauses are encountered before all the negative literals. A Prolog system computes answers to a query by using SLDNF-resolution, which is a Linear resolution with Selection function for Definite clauses augmented by Negation as Failure [3].

Definition. Let D be a database, G be a goal. Then the SLDNF-*tree* for $D \cup \{G\}$ is defined as follows:

(a) Each node of the tree is a goal.

(b) The root node is G.

(c) Let $\leftarrow L_1, \ldots, L_k$ $(k \geq 1)$ be a node in the tree and suppose that the selected literal L_m is positive. Then this node has a descendant for each input clause $A \leftarrow M_1, \ldots, M_q$ such that L_m and A are unifiable. The descendant is

$$\leftarrow(L_1, \ldots, L_{m-1}, M_1, \ldots, M_q, L_{m+1}, \ldots, L_k)\theta$$

where θ is an mgu of L_m and A.

(d) Let $\leftarrow L_1, \ldots, L_k$ $(k \geq 1)$ be a node in the tree and suppose that the selected literal L_m is a ground negative literal of the form *Not* A. If for every branch of an SLDNF-tree for $D \cup \{\leftarrow A\}$ the terminating node is a non-empty goal, then the single descendant of the node is

$$\leftarrow(L_1, \ldots, L_{m-1}, L_{m+1}, \ldots, L_k)$$

If there exists a branch of an SLDNF-tree for $D \cup \{\leftarrow A\}$ for which the terminating node is an empty goal then the node has no descendants.

(e) A node which is the empty goal has no descendants.

A branch of an SLDNF-tree for which the terminating node is an empty goal is called a *success branch*, a branch which does not terminate is called an *infinite branch* and a branch for which the terminating node is other than an empty goal is called a *failure branch*. An SLDNF-tree for which every branch is a failure branch is a *finitely failed* SLDNF-*tree*.

Definition. Let D be a database and G a goal. An SLDNF-*derivation* $G_0 = G, G_1, \ldots$ of $D \cup \{G\}$ is defined as follows:

(a) Suppose G_i is $\leftarrow L_1, \ldots, L_k$ and the selected literal L_m is positive. Suppose $A \leftarrow M_1, \ldots, M_q$ is the input clause and L_m and A have mgu θ_{i+1}. Then the derived goal G_{i+1} is

$$\leftarrow(L_1, \ldots, L_{m-1}, M_1, \ldots, M_q, L_{m+1}, \ldots, L_k)\theta_{i+1}$$

(b) Suppose G_i is $\leftarrow L_1, \ldots, L_k$ and the selected literal L_m is a ground negative literal of the form *Not A*. An attempt is made to construct a finitely failed SLDNF-tree with $\leftarrow A$ at the root. If the goal $\leftarrow A$ succeeds, then the subgoal *Not A* fails and so the goal G_i also fails. If $\leftarrow A$ fails finitely, then the subgoal *Not A* succeeds and the derived goal G_{i+1} is

$$\leftarrow (L_1, \ldots, L_{m-1}, L_{m+1}, \ldots, L_k)$$

In the latter case, θ_{i+1} is the identity substitution.

An SLDNF-derivation is *finite* if it consists of a finite sequence of goals; otherwise, it is *infinite*. An SLDNF-derivation is *successful* if it is finite and the last goal is the empty goal. A successful SLDNF-derivation is called an SLDNF-*refutation*. An SLDNF-derivation is *failed* if it is finite and the last goal is not the empty goal. Each branch of an SLDNF-tree corresponds to an SLDNF-derivation.

Definition. Let D be a database and G a goal. A *computation* of $D \cup \{G\}$ represents an attempt to construct an SLDNF-derivation of $D \cup \{G\}$.

Definition. Let D be a database and G a goal. A computation of $D \cup \{G\}$ *flounders* if at some point in the computation a goal is reached which contains only non-ground negative literals.

Definition. Let D be a database and G a goal. A clause $A \leftarrow L_1, \ldots, L_n$ in D is *allowed* [13] if every variable that occurs in the clause occurs in a positive literal of the body $L_1, \ldots, L_n$. The whole database D is *allowed* if each of its clauses is allowed. A goal G is *allowed* if G is $\leftarrow L_1, \ldots, L_n$ and every variable that occurs in G occurs in a positive literal of the body $L_1, \ldots, L_n$.

This restriction of allowedness on a deductive database D is introduced in order to prevent floundering of any computation $D \cup \{G\}$, for some goal G. This restriction corresponds exactly to the notion of *range-restriction* introduced by Decker [8] and Kowalski et al. [11].

Definition. Let D be a deductive database and G be a goal $\leftarrow W$. An *answer substitution* for $D \cup \{G\}$ is a substitution for the variables of G. A *computed answer* θ for $D \cup \{G\}$ is the substitution obtained by restricting the composition $\theta_1 \cdots \theta_n$ to the variables of G, where $\theta_1 \cdots \theta_n$ is the sequence of substitutions used in an SLDNF-refutation of $D \cup \{G\}$.

When a goal G is allowed, no computation of $D \cup \{G\}$ flounders and every computed answer for $D \cup \{G\}$ is a ground substitution for variables in G. A computed answer for $D \cup \{G\}$ will also be referred to as an answer substitution for the SLDNF-refutation of $D \cup \{G\}$.

Throughout the remainder of the paper, any further references to the term 'database' will be taken to refer to an allowed database.

Definition. A *constraint* is a closed first order formula that D is required to satisfy.

The standard definition of constraint satisfiability is as follows. Let D be a database such that comp(D), called the completion of D [3], is consistent. D is said to satisfy W, where W is a constraint, if W is a logical consequence comp(D); otherwise, D violates W. D is said to satisfy I, where I is a set of constraints, if D satisfies each constraint in I; otherwise, D

violates I. To implement reasoning with the completion of the database, the path finding method makes use of negation as failure.

The method deals with constraints in denial form. The denial form must also be allowed. If a constraint has the form

$$A_1 \vee \cdots \vee A_m \leftarrow L_1, \ldots, L_n, \quad m, n \geqslant 0,$$

then this can directly be transformed into the denial form

$$\leftarrow L_1, \ldots, L_n, \mathit{Not}\ A_1, \ldots, \mathit{Not}\ A_m.$$

If a constraint is not in the above form then it can be transformed to denial form as follows. Let W be a constraint which is an arbitrary closed first order formula such that the database D satisfies W. Then this constraint can be replaced by a new constraint $\leftarrow A$, where A is a predicate symbol of zero arity that does not occur elsewhere in the database or constraints, provided that, to the database D is added a set of rules obtained by transforming

$$A \leftarrow \mathit{Not}\ W$$

using the method of transformation as described in [14]. If D' is the resulting database, then D' must be allowed. Also, W is a logical consequence of comp(D) if and only if $\leftarrow A$ is a logical consequence of comp(D'). This justifies the transformation of a constraint to a denial. For the remainder of this paper constraints will be assumed to be in denial form and allowed. For example, the constraint (*) in section 1 can be transformed to a denial of the form

$$\leftarrow \mathrm{A}$$

by adding the rules

$$A \leftarrow \mathit{Worker}(x), \mathit{Not}\ B(x)$$

$$B(x) \leftarrow \mathit{Manager}(y, x)$$

to the database, where the two predicates A and B are not used elsewhere in the database.

Let I be a set of constraints. Then I_d will represent the transformed set of clauses of the form

$$IC(Id) \leftarrow B \quad (*)$$

from I, where $\leftarrow B$ is a constraint from I and Id is a unique identification of the constraint. $IC(Id)$ will be called the *head* of the constraint $\leftarrow B$.

3. The path finding method

This method of integrity checking in an updated database D' with a set of constraints I is based on finding a path in $D' \cup I_d$ from an update literal to the head of a constraint. If such a path is found then the integrity in the updated database is violated; otherwise, if all possible attempts to find a path fail, the integrity is said to be preserved in the updated database.

Definition. Let D be a database. A *path* in D is defined as a chain of literals

$$L_0 \overset{R_1}{\rightarrow} L_1 \overset{R_2}{\rightarrow} \cdots \overset{R_n}{\rightarrow} L_n$$

where L_0 is called the *source* of the path, L_n its *destination*, n its *length* and $R_1, \ldots, R_n$ are clauses from D used to construct the path from L_0 to L_n. If the source L_0 is positive then it is ground and $D \cup \{\leftarrow L_0\}$ has an SLDNF-refutation. For any two consecutive literals L_i and L_{i+1}, L_{i+1} is called the *successor* of L_i in the path, and is obtained from L_i in one of the following ways:

1. If
   (a) L_i is positive, and
   (b) L_i unifies with a positive literal L occurring in the body of the clause R_i: $H \leftarrow B$, and
   (c) α is an mgu of L_i and L, and
   (d) G' is a resolvent of $\leftarrow$B and $L_i \leftarrow$, and
   (e) θ is a computed answer for $D \cup \{G'\}$
   then L_{i+1} is $H\alpha\theta$.
2. If
   (a) L_i is positive, and
   (b) the negative literal *Not* L occurs in the body of the clause R_i: $H \leftarrow B$ such that L_i unifies with L, and
   (c) α is an mgu of L_i and L
   (d) *Not* $H\alpha$ is not an instance of any one of the L_j's, where $0 \leq j \leq i$
   then L_{i+1} is *Not* $H\alpha$.
3. If
   (a) L_i is negative, and
   (b) L_i unifies with a negative literal L occurring in the body of the clause R_i: $H \leftarrow B$, and
   (c) α is an mgu of L_i and L, and
   (d) θ is a computed answer for $D \cup \{G\}$, where G is the goal $\leftarrow B\alpha$
   then L_{i+1} is $H\alpha\theta$.
4. If
   (a) L_i is negative and has the form *Not* M, and
   (b) M unifies with a positive literal L occurring in the body of the clause R_i: $H \leftarrow B$, and
   (c) α is an mgu of M and L
   (d) *Not* $H\alpha$ is not an instance of any one of the L_j's, where $0 \leq j \leq i$
   then L_{i+1} is *Not* $H\alpha$.

From this definition, it is possible to construct more than one path starting from the same literal. For simplicity, a path will sometimes be written without showing the clauses used to construct the path. In the path $L_0 \rightarrow L_1 \rightarrow \cdots \rightarrow L_n$, the *distance* of L_p from L_q ($0 \leq q \leq p \leq n$) is $p - q$ and L_p is said to be *at a distance* $p - q$ from L_q.

Definition. Let D be a database and L a literal such that if L is positive then it is ground and $D \cup \{\leftarrow L\}$ has an SLDNF-refutation. Let S be the set of all paths with L as the source. The *path space* rooted as the literal L is a tree defined as follows:

1. Each node of the tree is a literal.
2. The root node is L.
3. Let N be a node of the tree. Then the set of all the successors of N in the paths of S are the only descendants of N in the tree.

Each branch of the path space corresponds to a path in the database with the root node as the source and vice-versa. A path which ends at the head $IC(Id)$ of a constraint will be called

a *success path*; otherwise, it will be called a *failure path*. A path space containing at least one branch which corresponds to a success path is referred to as a *success path space*.

To check integrity in a database when a new constraint is added, the database is queried directly to make sure that the constraint is a logical consequence of the completed database. If a constraint is deleted then this cannot cause any inconsistency. To check integrity in the updated database D' due to a transaction t applied to D, where a *transaction* is a finite sequence of additions and deletions of clauses to/from a database, the source is taken as an update literal. An *update literal* of the transaction t applied to a database D may be one of the following:

1. A fact of t which is to be added to D.
2. The negation of a fact of t which is to be deleted from D.
3. If a rule $H \leftarrow B$ in t is to be added to D then for a computed answer θ for $D' \cup \{\leftarrow B\}$, the corresponding instance of the head of the rule $H \leftarrow B$, i.e. $H\theta$, which is implicitly added to D due to the transaction.
4. If a rule $H \leftarrow B$ in t is to be deleted from D then the negation of the head of the rule $H \leftarrow B$, i.e. *Not* H, whose instances are likely to be deleted from D due to the transaction.

To preserve integrity one must ensure that a success path space does not exist with source as an update literal in the updated database. The algorithm tries to construct a success path through backtracking.

A transaction is valid if all of its update literals can be computed successfully without going into an infinite derivation and it has been assumed that the transaction does not add and delete the same clause. Also, a transaction is rejected if a success path is not found in the updated database. Modification can be considered as being accomplished by a deletion followed by an addition.

The method is illustrated in the following few examples.

Example 1.

Consider the following database.

Database D1:

Relations:

Mother(x, y): x is the mother of y.
Father(x, y): x is the father of y.
Parent(x, y): x is a parent of y.
Ancestor(x, y): x is an ancestor of y.
Dependent(x, y): x depends on y.
Husband(x, y): x is the husband of y.
Wife(x, y): x is the wife of y.
Married(x, y): x is married to y.
Occupation(x, y): occupation of x is y.
Student(x): x is a student.
Employed(x): x is employed.
Guardian(x, y): x is a guardian of y.
Self(x): x is self-sufficient.
Sponsor(x, y): x sponsors y.

Rules:

R1. $Mother(x, y) \leftarrow Father(z, y), Husband(z, x)$
R2. $Parent(x, y) \leftarrow Father(x, y)$
R3. $Parent(x, y) \leftarrow Mother(x, y)$
R4. $Ancestor(x, y) \leftarrow Parent(x, y)$

R5. $Ancestor(x, y) \leftarrow Parent(x, z), Ancestor(z, y)$
R6. $Wife(x, y) \leftarrow Husband(y, x)$
R7. $Married(x, y) \leftarrow Husband(x, y)$
R8. $Married(x, y) \leftarrow Wife(x, y)$
R9. $Employed(x) \leftarrow Occupation(x, Service)$
R10. $Student(x) \leftarrow Occupation(x, Student)$
R11. $Dependent(x, y) \leftarrow Parent(y, x), Employed(y), Student(x)$
R12. $Dependent(x, y) \leftarrow Married(y, x), Employed(y), Not\ Employed(x)$
R13. $Self(x) \leftarrow Married(y, x), Not\ Employed(y)$
R14. $Guardian(x, y) \leftarrow Dependent(y, x)$

Facts:
$Occupation(2, Service)$
$Occupation(3, Student)$
$Father(1, 3)$

Integrity Constraints I1:
IC1. $Guardian(x, y) \rightarrow Sponsor(x, y)$
IC2. $Married(x, y) \rightarrow Not\ Student(x)$

$I1_d$:
$IC(1) \leftarrow Guardian(x, y), Not\ Sponsor(x, y)$
$IC(2) \leftarrow Married(x, y), Student(x)$

Transaction:
insertfact $Husband(1, 2)$

where x, y and z are variables, and *Student* and *Service* are constants.

Before the transaction is applied, database D1 satisfies the constraints I1. Let D1′ be the updated database after the insertion. The complete path space generated by the insertion is shown in Fig. 1.

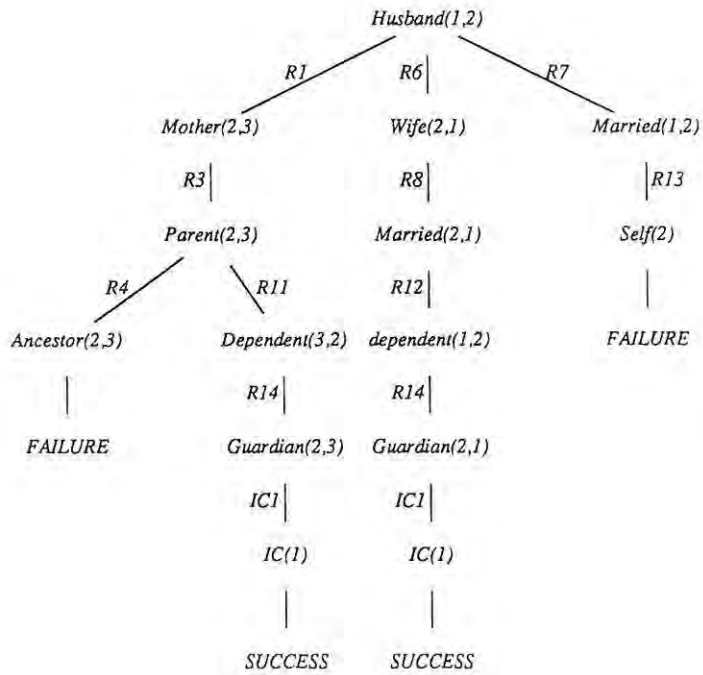


Fig. 1. The complete path space in the case of Example 1.

At the root of the path space is the update literal *Husband*(1, 2). If one attempts to unify *Husband*(1, 2) with a literal on the right hand side of any of the clauses in D1′, the first potential unification is with the literal *Husband*(z, x) occurring in the body of R1 for which $\alpha = \{z/1, x/2\}$ is an mgu. The resolvent of the goal

$$\leftarrow \mathit{Father}(z, y),\ \mathit{Husband}(z, x)$$

with

$$\mathit{Husband}(1, 2) \leftarrow$$

is

$$\leftarrow \mathit{Father}(1, y)\ .$$

Now if $\leftarrow$*Father*(1, y) is treated as a goal and applied to the database, one will arrive at $\theta = \{y/3\}$ as a computed answer for D1′ $\cup$ I1$_d$ $\cup$ $\{\leftarrow \mathit{Father}(1, y)\}$. Thus the next literal of the path is the head of rule R1 with the substitution $\{z/1, x/2, y/3\}$ applied, i.e. *Mother*(x, y)$\alpha\theta$, or *Mother*(2, 3). In a similar way one can derive the literal *Parent*(2, 3) as a successor to *Mother*(2, 3) in the path with the help of R3. The literal *Parent*(2, 3) unifies with the literal of the body of R4 as well as with a literal of the body of R5 and a literal of the body of R11. If one considers the first case, *Ancestor*(2, 3) will be the next literal of the path. Although it unifies with a literal from the body of R5, there is no explicit or implicit fact under the predicate *parent* which is present in D1′ $\cup$ I1$_d$ whose second argument is 2. Hence, this path is a failure path. In the second case, when *Parent*(2, 3) unifies with the first literal from the body of R5, 3 is not an ancestor of anyone in D1′ $\cup$ I1$_d$ which will satisfy the unified second literal and hence this path also fails. In the third case, when *Parent*(2, 3) unifies with the literal *Parent*(y, x) occurring in the body of R11, $\alpha = \{x/3, y/2\}$ would be an mgu of these two literals. The resolvent of the goal

$$\leftarrow \mathit{Parent}(y, x),\ \mathit{Employed}(y),\ \mathit{Student}(x)$$

and

$$\mathit{Parent}(2, 3) \leftarrow$$

is

$$\leftarrow \mathit{Employed}(2),\ \mathit{Student}(3)$$

and there exists a computed answer $\theta = (\ )$ (the identity substitution) for D1′ $\cup$ I1$_d$ $\cup$ $\{\leftarrow \mathit{Employed}(2), \mathit{Student}(3)\}$. Thus the next literal of the path is *Dependent*(x, y)$\alpha\theta$, i.e. *Dependent*(3, 2). Using clauses R14 and IC1, one can construct the following success path

$$\mathit{Husband}(1, 2) \xrightarrow{\text{R1}} \mathit{Mother}(2, 3) \xrightarrow{\text{R3}} \mathit{Parent}(2, 3) \xrightarrow{\text{R11}} \mathit{Dependent}(3, 2) \xrightarrow{\text{R14}}$$

$$\mathit{Guardian}(2, 3) \xrightarrow{\text{IC1}} \text{IC}(1)$$

and hence conclude that the integrity is violated due to the transaction.

Returning to the top-most level, *Husband*(1, 2) may also unify with *Husband*(y, x) from

the body of R6 as well as *Husband*(x, y) from the body of R7. In the first case one can construct the following success path

$$Husband(1,2) \xrightarrow{R6} Wife(2,1) \xrightarrow{R8} married(2,1) \xrightarrow{R12} Dependent(1,2) \xrightarrow{R14}$$
$$Guardian(2,1) \xrightarrow{IC1} IC(1)$$

In the second case *Married*$(1, 2)$ may unify with

(a) *Married*(y, x) on the right hand side of Rule R12. This reduces the right hand side to

$$\leftarrow Employed(1),\ Not\ Employed(2)\ ,$$

If this is applied as a goal to the database D1′ it will fail and hence this rule is ignored.

(b) *Married*(y, x) on the right hand side of rule R13 and $\alpha = \{y/1, x/2\}$ is an mgu. This reduces the right hand side to

$$\leftarrow Not\ Employed(1)\ .$$

If this is applied as a goal to D1′ it will succeed and the computed answer is the identity substitution. Thus the next literal of the path is $Self(x)\alpha$ or *Self*(2). This path is a failure path as the literal *Self*(2) or its negation does not unify with any body literal of the clauses of $D1' \cup I1_d$.

(c) *Married*(x, y) on the right hand side of IC2. This reduces the right hand side to

$$\leftarrow Student(1)\ .$$

If this is applied as a goal to D1′ it will fail and hence this constraint is ignored.

Example 2.

Consider the following database.

Database D2:

Rules:

same as D1

Facts:

Occupation(1, *Service*)

Occupation(2, *Service*)

Husband(1, 2)

Integrity Constraints I2:

same as D1

Transaction:

deletefact *Occupation*(2, *Service*).

Before the transaction is applied, database D2 satisfies the constraints I2. Let D2′ be the updated database after the deletion. Fig. 2 shows the complete path space arising from the deletion.

As the transaction is the deletion of a fact, the only update literal is the negation of this fact and this is taken as a candidate source literal. Clause R9 shows than an implicit deletion of the fact *Employed*(2) may occur due to this transaction. However, to avoid reasoning with two database states, the facts deleted from the database due to this transaction are not determined yet. Instead, it can be said that the fact *Employed*(2) is likely to be deleted from the database. $\theta = \{x/2\}$ is an mgu of *Occupation*(2, Service) and the body literal *Occupation*-

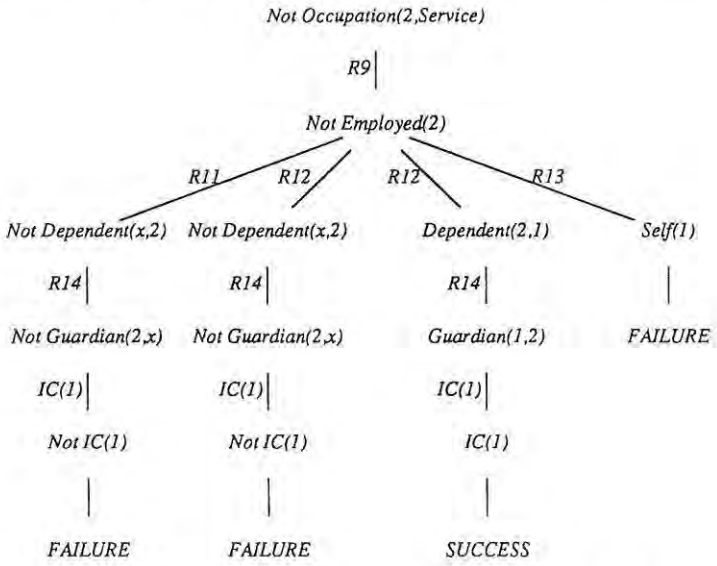


Fig. 2. The complete path space in the case of example 2.

$(x, Service)$ of R9. Thus the next literal of the path is the negation of $Employed(x)\theta$, i.e. *Not Employed*(2). The literal *Employed*(2) unifies with the literal $Employed(y)$ occurring in the body of both R11 and R12 and the negation of *Employed*(2) unifies with a negative literal $Not\ Employed(x)$ occurring in the body of R12 as well as with *Not Employed*(2) in the body of R13. The first and second unification may cause further implicit deletions from the database and $\alpha = \{y/2\}$ is an mgu for both the unifications. The third and fourth unifications may cause implicit additions to the database and an mgu in this case is $\beta = \{x/2\}$. If one considers the first case, the two consecutive literals in this direction of the path will be *Not Dependent*$(x, 2)$ and *Not Guardian*$(2, x)$ and it becomes a failure path as the last literal of this path does not unify with any body literal of any of the clauses in $D2' \cup I2_d$. The second case also generates a failure path.

Now, if one considers the third case, the next two consecutive literals of the path will be *Dependent*$(2, 1)$ and *Guardian*$(1, 2)$ and these facts are implicitly added to the database due to the transaction. The clause IC1 unifies with the last literal of the path yielding a success path

$$Not\ Occupation(2, Service) \xrightarrow{R9} Not\ Employed(2) \xrightarrow{R12} Dependent(2, 1) \xrightarrow{R14}$$

$$Guardian(1, 2) \xrightarrow{IC1} IC(1)$$

and thereby showing that integrity has once again been violated. Considering the last case, *Self*(1) would be the successor of *Not Employed*(2) in another direction. This path eventually fails.

Example 3.
Consider the following database.
Database D3:
Rules:
R1-R10, R12-R14 of D1
Facts:
Occupation(1, *Service*)

Occupation(2, *Service*)
Occupation(3, *Student*)
Husband(1, 2)
Father(1, 3)

Integrity Constrains I3:
same as I1

Transaction:
insertrule R: $Dependent(x, y) \leftarrow Parent(y, x), Employed(y), Student(x)$

Before the transaction is applied database D3 satisfies the constraints I3. Let D3′ be the updated database. To determine the set of instances of the conclusion of R which are likely to be added to the database due to the addition of the rule R, the following set of computed answers must be derived.

$$\{\theta: \theta \text{ is a computed answer for } D3' \cup \{\leftarrow B\}, \text{ where } R \text{ is } H \leftarrow B\}$$

which is equal to

$$\{\{x/3, y/2\}, \{x/3, y/1\}\}.$$

The corresponding set of instances of the conclusion of R (equivalent to $H\theta$), i.e. the set of update literals, is

$$\{Dependent(3, 2), Dependent(3, 1)\}$$

and the facts in this set are added to the database due to the addition of the rule R to the database. Taking one of the update literals $Dependent(3, 2)$ as a source, the success path

$$Dependent(3, 2) \xrightarrow{R14} Guardian(2, 3) \xrightarrow{IC1} IC(1)$$

in $D3' \cup I3_d$ can be constructed. Thus the integrity in the updated database D3′ is violated due to the addition of the rule R to the database.

Example 4.
Consider the following database.
Database D4:
Rules:
same as D1
Facts:
Employed(1)
Occupation(1, *Service*)
Occupation(2, *Service*)
Husband(1, 2)

Integrity Constraints I4:
same as I1

Transaction:
deleterule R: $Employed(x) \leftarrow Occupation(x, Service)$

Before the transaction is applied, database D4 satisfies the constraints I4. Let D4′ be the updated database. The set of ground instances of the conclusion of R which are likely to be deleted from the database due to the deletion of R is not determined. Instead, the negation of the head of R, which represents the set of facts deleted from the database, is taken as a source. From this the following success path can be constructed in the updated database as

the fact *Employed*(2) is implicitly deleted from D4 due to the deletion of R.

$$Not\ Employed(x) \overset{R12}{\rightarrow} Dependent(2,1) \overset{R14}{\rightarrow} Guardian(1,2) \overset{IC1}{\rightarrow} IC(1)\ .$$

Thus the integrity of the database will be violated by the deletion of rule R from the database.

4. Theorems for integrity checking

In this section, the definition of stratified databases [1, 13] is given and the correctness of the path finding method is proved for this class of databases.

Definition. A *level mapping* of a database is a mapping from its set of predicates to the set of nonnegative integers. A database is *stratified* if it has a level mapping such that, for every database clause $H \leftarrow B$, the level of the predicate of every positive condition in B is less than or equal to the level of the predicate of H and the level of the predicate of every negated condition in B is less than the level of the predicate of H.

It has been shown in [1] that if a database D is stratified then the completion of D is consistent and hence will have a model. Based on this fact, the database versions of the following two results [13], which are the basis of the method for determining whether a database satisfies or violates a constraint $\leftarrow B$, are given.

Result 1. Let D be a stratified database, $\leftarrow B$ a constraint. If there exists an SLDNF-refutation of $D \cup \{\leftarrow B\}$, then D violates $\leftarrow B$.

Result 2. Let D be a stratified database, $\leftarrow B$ a constraint. If $D \cup \{\leftarrow B\}$ has a finitely failed SLDNF-tree, then D satisfies $\leftarrow B$.

The proof of the theorem for integrity checking by the path finding method is similar to the one which has been described in [13]. Let D and D' be stratified databases and D' is obtained from D by applying a transaction t to D. Let the transaction t consist of a sequence of deletions followed by a sequence of additions. Suppose that the application of the sequence of deletions to D produces the intermediate database D''. In [13], the two sets of partially instantiated atoms $Pos_{D'',D'}$ and $Neg_{D'',D'}$ have been calculated inductively by the following formulae (similar formulae for $Pos_{D'',D}$ and $Neg_{D'',D}$).

$Pos^0_{D'',D'} = \{A : A \leftarrow W \in D'/D''\}$

$\mathrm{Neg}^0_{D'',D'} = \{\}$

$Pos^{n+1}_{D'',D'} = \{A\theta$: $A \leftarrow W \in D'$, B is a positive condition of $A \leftarrow W$, $C \in Pos^n_{D'',D'}$, and θ is an mgu of B and $C\} \cup \{A\theta$: $A \leftarrow W \in D'$, B is a negative condition of $A \leftarrow W$, $C \in Neg^n_{D'',D'}$, and θ is an mgu of B and $C\}$

$Neg^{n+1}_{D'',D'} = \{A\theta$: $A \leftarrow W \in D'$, B is a positive condition of $A \leftarrow W$, $C \in Neg^n_{D'',D'}$, and θ is an mgu of B and $C\} \cup \{A\theta$: $A \leftarrow W \in D'$, B is a negative condition of $A \leftarrow W$, $C \in Pos^n_{D'',D'}$, and θ is an mgu of B and $C\}$

$$Pos_{D'',D'} = \bigcup_{n \geq 0} Pos^n_{D'',D'}$$

$$Neg_{D'',D'} = \bigcup_{n \geq 0} Neg^n_{D'',D'}$$

The above two sets $Pos_{D'',D'}$ and $Neg_{D'',D'}$ (resp. $Pos_{D'',D}$ and $Neg_{D'',D}$) can be used to capture the differences between the models of comp(D'') and comp(D') (resp. comp(D'') and comp(D)). Suppose,

$$Pos_{D,D'} = Pos_{D'',D'} \cup Neg_{D'',D}$$

$$Neg_{D,D'} = Neg_{D'',D'} \cup Pos_{D'',D}$$

Then the above two sets $Pos_{D,D'}$ and $Neg_{D,D'}$ can be used to capture the differences between a model for comp(D) and a model for comp(D'). Accordingly one can have the following lemma similar to the one described in [13].

Lemma 1. *Let D, D' be stratified databases. Let t be a transaction which when applied to D produces D' and D'' as described above. Let $\leftarrow B$ be a constraint and $\Theta = \{\theta: \theta$ is an* mgu *of an atom of $Pos_{D,D'}$ ($Neg_{D,D'}$) and a positive (complement of a negative) literal occurring in $B\}$. Then D' satisfies $\leftarrow B$ if and only if D' satisfies $\leftarrow B\theta$ for all $\theta \in \Theta$.*

Proof. Result (a) of the simplification theorem given in [13]. □

Suppose the set $Add_{D,D'}$ (resp. $Del_{D,D'}$) is the set of all positive (resp. complement of negative) literals occurring in all the path spaces generated by the update literals of t. When all the path spaces are generated successfully without going into an infinite loop, one can have the following lemma.

Lemma 2. *Let D be a stratified database, and D' and D'' be obtained from D by applying t to D as described above. Then*

(a) *For each $A \in Add_{D,D'}$ there exists an $A' \in Pos_{D,D'}$ such that A is an instance of A'.*
(b) *If there is a ground instance A of an atom $A' \in Del_{D,D'}$ such that* comp(D)$\vdash A$ *but not* comp(D')$\vdash A$, *then A is an instance of an atom $A'' \in Neg_{D,D'}$.*
(c) *If A is a ground instance of an atom $A' \in Pos_{D,D'}$ such that* comp(D')$\vdash A$ *but not* comp(D)$\vdash A$, *then $A \in Add_{D,D'}$.*
(d) *If there is a ground instance A of an atom $A' \in Neg_{D,D'}$ such that* comp(D)$\vdash A$ *but not* comp(D')$\vdash A$, *then A is an instance of an atom $A'' \in Del_{D,D'}$.*

Proof. Straightforward from the definition of path. □

The above result shows that in the case of the path finding method, the set of ground facts $Add_{D,D'}$ is the portion added to a model of comp(D) and the set of partially instantiated atoms $Del_{D,D'}$ represents the portion deleted from a model of comp(D). The following theorem and its corollary which are based on the above lemma help to reduce the checking of a constraint into none or some of the simplified forms of the constraint.

Theorem 1. *Let D be a stratified database, $\leftarrow B$ be a constraint and D' be obtained from D by applying t to D. Let $\Theta = \{\theta: \theta$ is an* mgu *of an atom of $Pos_{D,D'}$ ($Neg_{D,D'}$) and a positive (complement of a negative) literal occurring in $B\}$. Let $\Phi = \{\phi: \phi$ is an* mgu *of an atom of $Add_{D,D'}$ ($Del_{D,D'}$) and a positive (complement of a negative) literal occurring in $B\}$. Then D' satisfies $\leftarrow B\theta$ for all $\theta \in \Theta$ if and only if D' satisfies $\leftarrow B\phi$ for all $\phi \in \Phi$.*

Proof. The following is the proof of the 'if' part of the theorem. The proof of the 'only if' part is similar.

Suppose D' satisfies $\leftarrow B\theta$ for all $\theta \in \Theta$. If possible, let there exist a $\gamma \in \Phi$ such that D' violates $\leftarrow B\gamma$. Suppose L is a positive (negative) literal occurring in B and it (its complement) unifies an atom A of the set $Add_{D,D'}$ ($Del_{D,D'}$) with the mgu γ. By lemma 2(a) (lemma 2(b)), there exists an atom L' from $Pos_{D,D'}$ ($Neg_{D,D'}$) such that $L\gamma$ is an instance of L' (negation of L'). Since D' satisfies $\leftarrow B\theta$ for all $\theta \in \Theta$, it follows that D' must satisfy $\leftarrow B\beta$, where β is an mgu of L' and L. $L\gamma$ is an instance of both L and L' and hence $\leftarrow B\gamma$ is an instance of $\leftarrow B\beta$. Therefore D' cannot satisfy $\leftarrow B\gamma$. This contradicts the initial assumption. Hence D' satisfies ϕ for all $\phi \in \Phi$. □

Corollary 1. *Let D be a stratified database, $\leftarrow B$ be a constraint and D' be obtained from D by applying t to D. Let $\Phi = \{\phi: \phi$ is an* mgu *of an atom of $Add_{D,D'}$ ($Del_{D,D'}$) and a positive (complement of a negative) literal occurring in $B\}$. Then D' satisfies $\leftarrow B$ if and only if D' satisfies $\leftarrow B\phi$ for all $\phi \in \Phi$.*

Proof. Follows from the above theorem and Lemma 1. □

Theorem 2. (*Theorem for integrity checking by the path finding method*). *Let D be a stratified database and I be a set of constraints such that D satisfies I. Suppose D' is obtained from D by the application of transaction t to D. Then the following properties hold.*

(a) *If there exists a success path in $D' \cup I_d$ taking the source as one of the update literals, then D' violates I.*
(b) *If there exists no success path in $D' \cup I_d$ from any one of the update literals, then D' satisfies I.*

Proof.

(a) Follows from corollary 1 and result 2.
(b) Follows from corollary 1 and result 1. □

Corollary 2. *Let D be a hierarchical database* [3] *and I be a set of constraints such that D satisfies I. Suppose D' is obtained from D with the application of transaction t to D. Then the following properties hold.*

(a) *There exists a success path in $D' \cup I_d$ taking the source as one of the update literals if and only if D' violates I.*
(b) *There exists no success path in $D' \cup I_d$ from any one of the update literals if and only if D' satisfies I.*

Proof. It follows from the above theorem directly as the hierarchical nature of D ensures the termination of any query without going into an infinite loop. □

5. Prolog implementation

Facts and rules of a database are stored directly as Prolog facts and rules. A unique number has been assigned to each newly inserted constraint as an identification. A constraint $\leftarrow D$ numbered *No* is stored directly as a Prolog rule *ic*(*No*):-*D*. A rule or a constraint can be retrieved efficiently by using a reference which uniquely identifies the corresponding rule or constraint.

The path can be computed efficiently by maintaining additional information for each rule and constraint in the database. For each literal L occurring in the body of a rule $H \leftarrow B$, clauses of the form

depend(*Ref*, *H*, *L*)

depend(*Ref*, *not H*, *CompL*)

have been maintained, where *Ref* is the reference which uniquely identifies the clause $H \leftarrow B$ and *CompL*, called the complement of *L*, is defined as follows. When *L* is positive, *CompL* is *not L*; otherwise, when *L* is *not A*, then *CompL* is *A*. For each literal *L* occurring in a constraint $\leftarrow D$, a clause of the form

depend(*Ref*, *ic*(*No*), *L*) .

has been maintained, where *ic*(*No*) is the head of the constraint and *Ref* is the reference which uniquely identifies the clause $ic(No) \leftarrow D$.

The task of finding a path from a source literal *S* to a destination literal *D* can be achieved in Prolog with the help of the following meta-interpreter:

```
path(D, S, _):-
  depend(Ref, D, S) ,
  clause(D, B, Ref) ,
  simplify(B, S, SB) ,
  call(SB) .

path(D, S, NegList):-
  depend(_, Via, S) ,
  Via = (not A) ,
  not instance(A, NegList) ,
  path(D, Via, [A | NegList]) .

path(D, S, NegList):-
  depend(Ref, Via, S) ,
  not Via = (not _) ,
  clause(Via, B, Ref) ,
  simplify(B, S, SB) ,
  call(SB) ,
  path(D, Via, NegList) .
```

path(*D*, *S*, *NegList*) means that a path is constructed from the literal *S* to the literal *D* and the set comprising the negation of each literal of the list *NegList* are the only negative literals occurring in this path. *clause*(*H*, *B*, *Ref*) means *H* and *B* are unified respectively with the head and body of a clause which is uniquely identified by the reference *Ref*. The interpretation of *simplify*(*B*, *S*, *SB*) is that if *S* is a positive ground literal then *SB* is obtained by resolving *B* against *S*; otherwise, *SB* is an instance of *B* obtained by unifying one of *B*'s body literals with *S*. *instance*(*A*, *NegList*) means that *A* is not an instance of any of the atoms

occurring in the list *NegList*. This corresponds to conditions 2(d) and 4(d) in the definition of a path and prevents a possible generation of an infinite path from a negative literal in the presence of recursive rules.

The top level C-Prolog goal for integrity checking (?-*ic_violated*(*Tran*).) can be defined as

ic_violated(*Tran*):-

path(*ic*(*No*), *Tran*, []) , ! .

where *Tran* represents an update literal. With the above convention if the database becomes inconsistent due to the transaction *Tran*, then the first inconsistency is reported at a constraint numbered *No*, i.e. this constraint is not a logical consequence of the completion of the updated database.

The efficiency of the meta-interpreter can be improved (at the expense of storage) by storing the two *depend* clauses

depend(H, L, B') ,

depend(*not* H, *Comp*L,_)

for each literal L occurring in the body of a clause $H \leftarrow B$, where B' is obtained from B by removing the occurrence of L and *Comp*L is the complement of L. It is obvious that in the new definition of *path*, one can eliminate the calls to the predicates *clause* and *simplify*.

Some optimisations could be carried out at the time of specifying certain specific kinds of rules or constraints. These optimisations have been illustrated in the context of the alternative implementation. Three cases have been considered. The first two deal with the generation of *depend* clauses of the first type and the third with the generation of the second kind of *depend* clauses in the implementation.

(1) Functional dependencies are a common form of constraint. An example of this kind of constraint is

$$Father(y, x), Father(z, x) \rightarrow y = z$$

Suppose that a constraint of this kind is present in the database and in the course of a transaction a fact under the predicate *Father* is inserted into the database, either explicitly or implicitly. According to the method described in this paper and its implementation described in this section, it is necessary to evaluate the two simplified instances of the above constraint and they are obtained from the following two clauses under the predicate *depend*:

$$depend(ic(Id), father(Y, X), (father(Z, X) \rightarrow Y = Z)), \quad \text{and}$$

$$depend(ic(Id), father(Z, X), (father(Y, X) \rightarrow Y = Z))$$

where *Id* is a unique identification of the constraint. Evaluation of the simplified instance which arises from the second of the above two is redundant and hence may be excluded. This can be prevented by not generating the second one at the time of specifying the constraint.

(2) The second case deals with rules in which two or more similar instances of a predicate occur in the body. An example of this kind of rule is

$$Result(x, Hons) \leftarrow Exam(x, y, A), Exam(x, z, A), y \neq z$$

If a database contains a rule of this form, the alternative implementation of the method described in this section generates the following two clauses under the predicate *depend*:

$$depend(result(X, hons), exam(X, Y, a), (exam(X, Z, a), Y \neq Z)), \quad \text{and}$$

$$depend(result(X, hons), exam(X, Z, a), (exam(X, Y, a), Y \neq Z)).$$

Now, if a transaction inserts into the database either implicitly or explicitly a fact under the predicate *Exam* unifying with $exam(X, Y, a)$, then an instance of $result(X, hons)$ might be added twice implicitly to the database. Consequently, the same path space rooted at the corresponding instance of $result(X, hons)$ would have to be generated twice. This can be avoided by generating only one of the above two clauses under *depend* at the time of compiling the rule.

The above two cases can be formalised as follows. Let D be a database and I be a set of constraints. Suppose there exist literals $L_1, L_2, \ldots, L_n$ under the same predicate occurring in the body of a clause R in $D \cup I_d$ such that any two of them are instances of each other. Suppose for any two L_i and L_j, R' is obtained from R by interchanging the variables in the same argument position of L_i and L_j. If for all L_i and L_j, R' can be transformed to R with the help of the transformations $M, N \leftrightarrow N, M$, $a = b \leftrightarrow b = a$ and $a \neq b \leftrightarrow b \neq a$, where M, N are literals and a, b are terms, then generate only one *depend* clause corresponding to one of the L_k's, $1 \leq k \leq n$.

(3) It can be seen from Fig. 2 corresponding to Example 2 that the first and second branch of the path space are identical to each other. Generating one of them is redundant. This happened because the implicit or explicit deletion of a fact under the predicate *Employed* can delete the same fact from both rules 11 and 12. Since the method does not compute what facts have been deleted, it will be sufficient if one generates the second kind of *depend* clause corresponding to either rule 11 or rule 12.

The above can be formalised as follows. Suppose there are two depend clauses of the form *depend* (_, *Not* H, L) and *depend*(_, *Not* H', L') such that L' is an instance of L and H' is an instance of H. If $H'\theta$ is an instance of $H\theta$ and $L'\alpha$ is an instance of L, where θ is an mgu of L and L', and α is an mgu of H and H', then the latter depend clause can be excluded.

6. Other constraint checking methods

The simplest way of maintaining consistency is to evaluate each constraint upon each update. However, as noted earlier, this is much too costly when the database size increases and makes no use of the fact that the database satisfies the constraints prior to the update. For example, if one wants to insert into the database $D1$ the fact $Guardian(1, 2)$, then only the first constraint is affected by this insertion and to check the integrity it is sufficient to evaluate this constraint only. Evaluating the second constraint in this case is redundant.

Several authors have proposed different methods for checking integrity in a deductive database. The approaches put forward by Decker [8], Kowalski et al. [11] (extension of the method proposed in [17]), Lloyd et al. [13] (extension of the method proposed in [15]), Ling [12] and Bry et al. [2] are introduced briefly in this section. More detail on each of these methods is given in [5].

The simplification method proposed by Lloyd et al. is based on a standard view of constraint satisfiability. By using only rules of the database and updates, the method computes two sets of atoms, the set of atoms likely to be added to the database and the set of atoms likely to be deleted from the database. With these two sets of atoms, constraints are instantiated appropriately and only the affected constraints are evaluated.

Decker's method is also based on a standard view of constraint satisfiability. Like the method of Lloyd et al., Decker's method simplifies the constraints before evaluation and only affected constraints are evaluated. The main difference between the two is that Decker's method computes in stages the above defined two sets of fully instantiated atoms by using not only rules and updates but facts as well. Also, in Decker's method, the constraints are represented in equivalent, evaluable form. At each stage, after deriving each set, the constraint is considered to determine if it is affected by an atom in the respective set.

The consistency method introduced by Kowalski et al. uses an extension of the SLDNF proof procedure which provides both forward and backward reasoning. The algorithm also incorporates some metalevel rules for reasoning about implicit deletions and a generalised resolution step for reasoning forward from negation as failure. Taking one of the updates as a candidate top clause, the algorithm tries to construct a refutation tree, i.e. it tries to construct a search space in which at least one path ends at the empty clause. If all possible attempts fail to do so then integrity is preserved in the updated database.

The integrity checking method proposed by Bry et al. is divided into two phases. The first phase, called the preparatory phase, determines a set of potential updates which represent possible ground updates induced by an update to the database and then generates expressions called update constraints from potential updates and constraints. The second phase, called the evaluation phase, evaluates all update constraints. Facts are accessed only during evaluation. The method discusses the possibility of getting benefit from the query optimisation technique in the second phase. A meta-interpreter has been proposed for simulating query evaluation in the updated database before any update is performed. A procedure for checking constraint satisfiability has also been proposed by Bry et al.

Ling proposes a special form of closed first order formula, called an IC-formula, which uses normal form (NF) negative formulae to express constraints. Given a constraint as an NF negative formula, say Not(P), which contains some views (a view is a positive literal which appears as the head of some clause), the method constructs a set of NF negative formulae EXTRA(Not (P)) using rules of the database. When an update is performed, the method evaluates only the affected constraints from these sets.

7. Comparison with other constraint checking methods

As pointed out previously, the simplest way of maintaining consistency is by evaluating every constraint for each update; however, this is very inefficient since it ignores the assumption that the database satisfies the constraints prior to the update. On the other hand, the path finding method and all the methods described in the previous section do take advantage of this assumption.

Lloyd et al. define a constraint as any closed first order typed formula and its equivalent prenex conjunctive normal form has been considered for verification. Decker has considered the class of range-restricted formulae for expressing constraints. This class of constraints is an important subset of domain-independent formulae. A formula is domain independent if and only if its truth value does not depend on any domain element other than those occurring in the relations that are explicitly mentioned in the formula. This class of formulae has been found meaningful to represent constraints in a logic database. Both the path finding method and the method proposed by Kowalski et al. deal with the same class of constraints. They take arbitrary closed first order formulae as constraints and transform them into denial form by adding rules to the database so that the resulting database becomes equivalent to the original one. The denial form must be allowed. Bry et al. define constraints as closed first order formulae in restricted quantified form as well as in normalized form. As far as

expressive power is concerned, these forms do not impose significant restrictions. Ling proposes a special form of closed first order formula, called an IC-formula, which uses the nested Prolog not-predicate to express constraints. The IC-formulae are readable and concise.

The path finding method reasons forward from an update until it reaches the head of a constraint. The method derives in stages a set of fully instantiated atoms added to the database and a set of partially instantiated atoms likely to be deleted from the database. The approach is identical to that of Decker when there is no implicit deletion in the database. When an implicit deletion occurs in a stage, instead of deriving a set of atoms deleted from the database, the method follows a similar approach to that of Lloyd et al. and computes a set of partially instantiated atoms likely to be deleted from the database. The main reason for this is to avoid reasoning with two database states, i.e. before and after update. This contrasts with Decker's approach which reasons with two database states corresponding to times before and after update to exclude all induced facts which were provable in the database before update.

To compare the performance of the path finding method against the methods of Lloyd et al. and Decker, the following four different cases have to be considered.

1. None of the constraints is simplified by an atom of *Pos*, where *Pos* is the set of partially instantiated atoms computed by the method of Lloyd et al. which represents the set of facts added to the database due to an update.
2. Some constraints can be simplified by the atoms of *Pos* but no instance of any one of the atoms of *Pos* is true in the updated database.
3. None of the constraints is simplified by an atom of *Neg*, where *Neg* is the set of partially instantiated atoms computed by the method of Lloyd et al. which represents the set of facts deleted from the database due to an update.
4. Some constraints can be simplified by the atoms of *Neg* but no instance of the complement of any one of the atoms of *Neg* is true in the updated database.

In the first case, the method of Lloyd et al. will have an advantage over both the methods of Decker and the path finding method as the latter methods will redundantly construct the set of facts which are implicitly added to the database. In the second case, the method of Lloyd et al. may suffer from a redundant constraint evaluation and the performance of both Decker's method and the path finding method will depend on the number of implicitly added facts. In the third case, both the methods of Lloyd et al. and also the path finding method will have an advantage over the method of Decker as the latter method will redundantly construct the set of facts which are implicitly deleted from the database. In the last case, both the methods of Lloyd et al. and the path finding method may suffer from a redundant constraint evaluation and the performance of Decker's method in this case will depend on the computation of the number of implicitly deleted facts, performing more efficiently than the former two if the number of implicitly deleted facts is small, but less efficiently than the other two if it is not.

In the path finding method finding a path from an update literal to the head of a constraint is similar to the problem of finding a refutation with an update literal as a top clause in Kowalski's method. In the latter method finding a refutation means arriving at an empty head of the denial form of a constraint from an update literal. In the path finding method constructing a success path means reaching the head of a constraint from an update literal. The difference lies in the way in which this is achieved – in the method put forward by Kowalski et al. the computation of positive induced updates can be deferred by a proper literal selection strategy but the computation of negative induced updates is essential, in the path finding method the computation of positive induced updates is essential and the computation of negative induced updates is deferred. Also, the literal selection strategy in

Kowalski et al.'s method can play an important role in the efficiency of the method. Earlier selection of a literal whose instances are not true in the database may reduce the computation time.

A disadvantage of Kowalski et al.'s method is that the degree of *granularity* [18] required for the meta-interpreter is much finer than that required for implementing the path finding method. This is due to the fact that the former models the literal selection strategy as well as the inference mechanism for forward reasoning. This results in a loss of efficiency. On the other hand the path finding method relies mainly on the backward reasoning mechanism built into Prolog.

The path finding method, Decker's method and the method of Kowalski et al. can all handle those recursive rules for which the underlying SLDNF-resolution terminates. The simplification of constraints in both Lloyd et al.'s method and Bry et al.'s method are independent of the evaluation method chosen. Each of these methods will work in the presence of recursive rules if the underlying query evaluator handles recursion. Ling's method deals only with hierarchical databases. Lloyd et al.'s method considers a typed database and converts it to a type free database by some transformations. The other methods mentioned above need the database to be range-restricted to evaluate a query correctly.

A more detailed comparison of the relative performance of each of the methods for specific cases is presented in [5].

8. Conclusion

A method called the path finding method for constraint checking in deductive databases has been described and proved to be correct for stratified databases. An efficient Prolog description of the main algorithm has been proposed together with some optimisations which can be achieved. Several other well-known constraint checking methods have been introduced in brief and compared against the path finding method.

Acknowledgement

This work was conducted with support from the Ministry of Education, Government of India, and was done in conjunction with the Logic Database Group who are supported by the SERC through the Alvey initiative.

References

[1] K.R. Apt, H.A. Blair and A. Walker, Towards a theory of declarative knowledge, in J. Minker (ed.) *Foundation of Deductive Databases and Logic Programming* (Morgan Kaufman Publishers, Inc, 1988) 89–148.

[2] F. Bry, H. Decker and R. Manthey, A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases, *Proceedings of Extending Database Technology*, Venice (1988) 488–505.

[3] K.L. Clark, Negation as failure, in H. Gallaire and J. Minker (eds.) *Logic and Databases* (Plenum Press, New York, 1978) 293–322.

[4] W.F. Clocksin and C.S. Mellish, *Programming in Prolog* (Springer International Student Edition, 1984).

[5] S.K. Das and M.H. Williams, Integrity checking methods in deductive databases: A comparative evaluation, in: M.H. Williams (Ed.) *Proc. Seventh British Nat. Conf. on Databases* (Cambridge University Press, 1989) 85–116.

[6] C.J. Date, *An Introduction to Database Systems, Vol. 1, 4th edition* (Addison Wesley, 1986).

[7] C.J. Date, *An Introduction to Database Systems, Vol. 2* (Addison Wesley, 1985).

[8] H. Decker, Integrity enforcements on deductive databases, in L. Kerschberg (ed.) *Proceedings of the First International Conference on Expert Database Systems*, Charleston, South Carolina (April 1986) 271–285.
[9] E.B. Fernandez, R.C. Summers and C. Wood, *Database Security and Integrity* (Addison Wesley, 1981).
[10] H. Gallaire, J. Minker and J.-M. Nicolas, Logic and databases – a deductive approach, *Computing Surveys* 16, 2 (1984) 153–185.
[11] R.A. Kowalski, F. Sadri and P. Soper, Integrity checking in deductive databases, *Proceedings of the 13th VLDB Conference*, Brighton (1987) 61–69.
[12] T.-W. Ling, Integrity constraint checking in deductive databases using the Prolog not-predicate, *Data & Knowledge Engineering* 2 (1987) 145–168.
[13] J.W. Lloyd, E.A. Sonenberg and R.W. Topor, Integrity constraint checking in stratified databases, Department of Computer Science, University of Melbourne, Technical Report 86/5 (1986).
[14] J.W. Lloyd and R.W. Topor, Making Prolog more expressive, *J. Logic Programming* 1, 3 (1984) 225–240.
[15] J.W. Lloyd and R.W. Topor, A basis for deductive database systems, *J. Logic Programming* 2, 2 (1985) 93–109.
[16] J.-M. Nicolas, Logic for improving integrity checking in relational databases, *Acta Informatica* 18 (1982) 227–253.
[17] F. Sadri and R.A. Kowalski, An application of general purpose theorem-proving to database integrity, in J. Minker (ed.) *Proceedings of the Workshop on Foundations of Deductive Databases and Logic Programming* (1987).
[18] L. Sterling and E. Shapiro, *The Art of Prolog* (The MIT Press, Cambridge, Massachusetts, 1986) 303–329.
[19] R.W. Topor, T. Keddis and D.W. Wright, Deductive database tools, Department of Computer Science, University of Melbourne, Technical Report 84/7.
[20] J.D. Ullman, *Principles of Database Systems, 2nd edition* (Computer Science Press International, Inc, Maryland, USA, 1984).

53. S.K.DAS and M.H.WILLIAMS, Extending the integrity maintenance capability in deductive databases, Logic Programming New Frontiers, Ed. D.R.Brough, Intellect Books, Oxford, 75-111 (1992).

Extending the Integrity Maintenance Capability in Deductive Databases

Subrata Kumar Das M.H.Williams

Abstract

The methods proposed so far for checking integrity in deductive databases suffer from two major drawbacks. The first is that they deal with (integrity) constraints which are restricted closed first-order formulae and confining constraints to this form they exclude some important kinds of constraints from consideration. The second is that the methods are only applicable to definite databases and are unable to check integrity in indefinite databases. To take account of the first problem, extensions of some of the existing methods are proposed in the context of definite databases which allow a set of *aggregate predicates* to be included within formulae representing constraints. To overcome the second problem, the paper proposes extensions of the methods in another direction to handle constraints imposed on indefinite databases. This is based on a new definition of constraint satisfiability in indefinite databases. The constraint verification program, based on this concept of constraint satisfiability, uses a query evaluator which works on the principle of *negation as possible failure* for inferring negative information from indefinite databases.

1 Introduction

In a database an *integrity constraint* [9],[24], [11] (or simply a constraint) is a formal representation of a property which the data in the database must satisfy to be consistent with some model of the real world from which it comes. In a logic programming context, a constraint can be defined as a closed formula. Different ways of maintaining integrity in deductive databases have been proposed in several papers, e.g. generalisations of the simplification method [21] by Lloyd and Topor [17] , Decker [10] , and Bry *et al.* [3] , a general theorem-proving technique by Sadri and Kowalski [22] , using Prolog not-predicate by Ling [14] , consistency proof method and a modified

program method by Asirelli et al. [2] , a path finding method by Das and Williams [7], etc. The following two major shortcomings have been observed [8] in the methods proposed so far:

1. The methods only allow a subset of the set of all closed first-order formulae as constraints [8]. Confining constraints to this form excludes some important kinds of constraints from consideration. For example, constraints of the form

   the total number of employees in a particular department may not exceed 100, or
   a student obtaining the highest marks in a particular course receives an award, or
   a student obtaining an average mark greater than or equal to 50, passes the examination, etc.

   cannot be expressed conveniently as first-order formulae and it is not possible to evaluate them efficiently even if they are expressed as first-order formulae.

2. The underlying database is always *definite*, i.e. the head of each database clause is an atom and the body is a conjunction of literals. When the database becomes *indefinite* by allowing the head of a database clause to be a disjunction of atoms, the methods are unable to handle this.

In response to the first problem, the paper attempts to extend the *simplification method* of Lloyd *et al.* [17] and the *path finding method* [7] in the context of definite databases by allowing a set of aggregate predicates (e.g. *Count*, *Sum*, *Maximum*) to be included within formulae representing constraints. The extended methods can be implemented efficiently in Prolog by using the aggregate predicate `setof`.

In response to the second problem, the paper extends both the simplification and the path finding methods to handle integrity checking in indefinite databases. This is based on a new definition of constraint satisfiability in indefinite databases. This definition is a generalised version of the *theorem-hood* [22] view of constraint satisfiability defined in terms of *completion* [4] in the context of definite databases. Following this definition, a constraint is satisfied by an indefinite database if and only if the constraint is satisfied by each *possible form* of the indefinite database.

Although a constraint verification program based on the above concept of constraint satisfiability can work on any query evaluator for the underlying

database, the paper considers the semantics of the database based on the *negation as possible failure rule* for inferring negative information from a database. In the declarative semantics of negation as possible failure, a ground atom is taken as false if its negation is true in the completion of each of the possible forms of the database. The procedural semantics is based on two mutually recursive resolution schemes, *definite resolution* and *possible resolution*. These two resolution schemes coincide and reduce to the mechanism of simple SLDNF-resolution when the database is definite.

There is a direct link between the introduced notion of constraint satisfiability and the negation as possible failure rule. If a constraint (in *denial form*) has a *finitely failed possible tree* in a database D, then the constraint is said to be satisfied by D.

The Prolog implementation of the method and a meta interpreter of extended nH-Prolog [19] are described. The extended nH-Prolog which is capable of inferring negative information from the database is based on the idea of negation as possible failure rule. The query evaluator which is required for constraint evaluation operates on this extended nH-Prolog.

The organisation of the paper is as follows. The following section describes some basic concepts. Section 3 considers the extension of integrity maintenance for definite databases to include formulae which are not closed first-order, whereas Section 4 looks at constraints which are closed first-order formulae but in indefinite databases. The C-Prolog [5] syntax has been used in the subsections dealing with Prolog implementation.

2 Basic concepts

Definition 1 *A* (first-order) literal *is an atom (referred to as a* positive literal*) or its negation (*negative literal*).*

Definition 2 *A* (database) clause *is a formula of the form*

$$A_1 \vee \cdots \vee A_m \leftarrow L_1 \wedge \cdots \wedge L_n, m \geq 1, n \geq 0$$

where $A_1 \vee \cdots \vee A_m$ *is the* head *(or* conclusion *or* consequent*) of the clause and* $L_1 \wedge \cdots \wedge L_n$ *the* body *(or* condition *or* antecedent*). Each* A_i *is an atom and each* L_j *is either an atom (a positive condition) or a negated atom (a negative condition). The* $\vee$ *symbols in the consequent denote disjunction and the* $\wedge$ *symbols in the antecedent denote conjunction. Any variables in* $A_1, \ldots, A_m, L_1, \ldots, L_n$ *are assumed to be universally quantified over the whole clause.*

The different forms of a clause corresponding to different values of m and n are as follows:

1. $m = 1, n = 0$, i.e. the body is empty and the head is a single atom. The clause is called a (*definite*) *fact.*
2. $m > 1, n = 0$, i.e. the body is empty and the head is a disjunction of atoms. The clause is called an *indefinite fact.*
3. $m = 1, n > 0$, i.e. the head is a single atom and the body is non empty. The clause is called a (*definite*) *rule.*
4. $m > 1, n > 0$, i.e. the head is a disjunction of atoms and the body is non empty. The clause is called an *indefinite rule.*

A *definite clause* is either a fact or a rule. An *indefinite clause* is either an indefinite fact or an indefinite rule.

Definition 3 *A* database *is a finite set of clauses. A* definite database *is a finite set of definite clauses.*

Definition 4 *A* (database) query *(or* goal *or* denial*) is a formula of the form*

$$\leftarrow L_1 \wedge \cdots \wedge L_n, n \geq 1$$

where $L_1, \ldots, L_n$ are literals. Any variables in $L_1 \wedge \ldots \wedge L_n$ are assumed to be universally quantified over the whole goal.

Definition 5 *Let D be a database and G a goal. A clause $A_1 \vee \cdots \vee A_m \leftarrow L_1 \wedge \cdots \wedge L_n$ in D is* allowed *(or* range-restricted *[12]) if every variable that occurs in the clause occurs in a positive literal of the body $L_1 \wedge \cdots \wedge L_n$. The whole database D is* allowed *if each of its clauses is allowed. A goal G is* allowed *if G is $\leftarrow L_1 \wedge \cdots \wedge L_n$ and every variable that occurs in G occurs in a positive literal of the body $\leftarrow L_1 \wedge \cdots \wedge L_n$*

To avoid floundering in the construction of a derivation of $D \cup \{G\}$, where D is a database and G a goal, it has been assumed in the rest of the paper that both D and G are *allowed* [15].

Definition 6 *A* computation rule *is a function which maps a goal to a literal, called the* selected literal, *in that goal. A* safe computation rule *is a function from a set of goals, none of which consists entirely of non-ground negative literals, to a set of literals such that the value of the function for such a goal is either a positive literal or a ground negative literal.*

Unless otherwise stated, computation rules have been assumed to be safe to evaluate a query correctly. Most Prolog systems do not enforce this safeness condition with the result that they will compute a query correctly only if the literals of each query and database clause are ordered in such a way as to meet this condition.

3 Extension using higher order predicates

In this section, the class of databases considered is definite and hence any usage of the terms 'database' will be taken as 'definite database'.

3.1 General formulae and integrity constraints

In extending clauses to include aggregate predicates, the following five have been chosen as an initial set:

Count

Average

Sum

Maximum

Minimum

although this set can clearly be enlarged to include other such functions.

Definition 7 *An* aggregate atom *is one of the following forms :*

$Count(W, n)$

$Average(x, W, r)$

$Sum(x, W, r)$

$Maximum(x, W, r)$

$Minimum(x, W, r)$

where W is a conjunction of literals (a goal written without $\leftarrow$) in which some of the variables are free and the remaining variables are assumed to be existentially quantified in front of W. The variable x is one of the bound variables of W, and variables n and r represent the result of the function (as an integer or real number, respectively). The variable x can be replaced by an identifier (e.g. an integer) which will identify uniquely the occurrence of x in W. For the sake of readability, it has been considered as a variable. Thus $Count(W, n)$ is interpreted as counting the number of different answers to the query $\leftarrow W$ which are true in D, and returning this value as n; $Average(x, W, r)$ computes the average of all the values of the variable x obtained from different answers of the query $\leftarrow W$ which are true in D, and return this value as r; and so on. In each case the query $\leftarrow W$ is assumed to be allowed.

An aggregate literal *is either an aggregate atom or the negation of an aggregate atom. A* general atom *(resp.* general literal*) is either an atom (resp. literal) or an aggregate atom (resp. aggregate literal). In each of the above aggregate atoms the formula W will be called the* key formula.

Definition 8 *A* general formula *may now be defined as follows :*

1. *A general literal is a general formula.*
2. *If A and B are general formulae, then so are $\neg A$, $A \vee B$, $A \wedge B$, $A \rightarrow B$ and $A \leftrightarrow B$.*
3. *If A is a general formula and x is a variable, then $\forall x A$ and $\exists x A$ are general formulae.*
4. *Nothing else is a general formula.*

Definition 9 *Given a formula consisting of the single general atom α, the general atom α is said to* occur positively *in the general formula α. If a general atom α occurs positively (resp. negatively) in a general formula A, then α* occurs positively *(resp.* negatively*) in $\exists A$ and $A \wedge B$ and $A \vee B$ and $B \rightarrow A$. If a general atom α occurs positively (resp. negatively) in a general formula A, then α* occurs negatively *(resp.* positively*) in $\neg A$ and $A \rightarrow B$.*

Definition 10 *Let λ be an aggregate literal occurring in a general formula A. Then all the bound variables of the key formula W of λ are said to be* local *to the aggregate literal λ. Any other variables occurring in λ are said to be* global *variables of A.*

Definition 11 *A* closed general formula Γ *is a general formula with no free occurrences of any global variables.*

Definition 12 *A* first-order constraint *is a closed first-order formula.*

The standard view of constraint satisfiability of a first- order constraint C imposed on a database D, where $comp(D)$ is assumed to be consistent, is defined as follows. D is said to satisfy C if C is a logical consequence of $comp(D)$; otherwise D violates C.

A first-order constraint imposed on a database D can always be assumed to be in the form of a denial. If a first-order constraint is not in the form of a denial then it can be transformed to denial form by suitably extending the database [22]. Let C be a constraint which is an arbitrary closed first order formula such that the database D satisfies C. Then this constraint can be replaced by a new constraint $\leftarrow A$, where A is a predicate symbol of zero arity that does not occur elsewhere in the database or constraints, provided that, to the database D is added a set of rules obtained by transforming

$$A \leftarrow \neg C$$

using the transformation as described in [16]. If D_T is the resulting database, then D_T must be allowed.

Definition 13 *An* aggregate constraint *is a closed general formula* Γ *of the form*

$$\lambda_1 \wedge \cdots \wedge \lambda_n \rightarrow \alpha_1 \vee \cdots \vee \alpha_m$$

where $\alpha_1 \vee \cdots \vee \alpha_m$ *is the* conclusion *of* Γ *and* $\lambda_1 \wedge \cdots \wedge \lambda_n$ *is the* condition. *Each* α_i *is a general atom and each* λ_j *is a general literal and at least one of* $\alpha_1, \ldots, \alpha_m, \lambda_1, \ldots \lambda_n$ *is an aggregate literal. Variables occurring in the condition are assumed to be universally quantified at the front of* Γ. *Any variables in the conclusion which do not occur in the condition are assumed to be existentially quantified at the front of the conclusion.*

Thus the example of constraints given in the introduction can be expressed using this notation, as follows:

$$Dept(x) \wedge Count(EmpDept(y, x), z) \rightarrow z \leq 100,$$

$$Paper(x) \wedge Maximum(v, Marks(u, x, v), y) \wedge Marks(z, x, y) \rightarrow Award(z, x),$$

and

$$Student(x) \wedge Average(v, Marks(x, u, v), y) \wedge y \geq 50 \rightarrow Pass(x),$$

where x, y, z, u, v are variables and the schema for the predicates $Dept$, $EmpDept$, $Paper$, $Marks$, $Award$ are respectively $Dept(department)$, $EmpDept(employee, department)$, $Paper(paper)$, $Marks(student, paper, marks)$, $Award(student, paper)$. According to the definition, the variables x and z in the first constraint are assumed to be universally quantfied in front of the whole constraint, whereas the variable y is assumed to be existentially quantified in front of $EmpDept(y, x)$.

Definition 14 *A* constraint *is either a first-order constraint or an aggregate constraint.*

In following two subsections, the only aggregate predicate considered will be the predicate $Count$. In Subsection 3.4 this will be generalised to other aggregate predicates from the initial set.

3.2 Aggregate constraint satisfiability

Let D be a database and Q_D be the set of all queries. Consider the second-order function $\chi_D : Q_D \rightarrow N$, where N is the set of all non-negative integers, defined in the following way:

$$\text{for } \leftarrow W \in Q_D, \chi_D(\leftarrow W) = card\{W\theta : comp(D) \vdash W\theta\}$$

where $cardX$ denotes the cardinality of the set X. Clearly the mapping is well defined, i.e. every query $\leftarrow W$ is mapped into a unique integer which is equal to the number of instances of W which can be derived from $comp(D)$. This asserts that the number of different instances of W which are true in D is $\chi_D(\leftarrow W)$. Thus if a constraint contains the aggregate atom $Count(W, n)$ and all global variables occurring in W are instantiated, $\chi_D(\leftarrow W)$ represents n. In other words, $\chi_D(\leftarrow W)$ is the number of different computed answres of the SLDNF-derivation of $D \cup \{\leftarrow W\}$, provided that it is capable of returning all answres without going into an infinite loop.

Lemma 1 *Let D be a database and $\leftarrow W$ a member of Q_D . Let $x_1, \ldots, x_n$ be all the variables of W and D_T be $D \cup \{\forall x_1 \cdots \forall x_n(P(x_1, \ldots, x_n) \leftarrow W)\}$, where P is a predicate symbol not occurring elsewhere in the database D. If $\chi_D : Q_D \rightarrow N$ and $\chi_{D_T} : Q_{D_T} \rightarrow N$ are the two mappings defined as above, then $\chi_D(\leftarrow W) = \chi_{D_T}(\leftarrow P(x_1, \ldots, x_n))$.*

Proof : This follows from the fact that $comp(D) \vdash W\theta$ if and only if $comp(D) \vdash P(x_1, \ldots, x_n)\theta$. □

In view of the above lemma, the key formula of an aggregate atom under the predicate *Count* occurring in a constraint will be assumed to be an atom.

Definition 15 *The* first-order equivalent of an aggregate atom $Count(A, n)$ *is the first-order atom* $Count_A(n, y_1, \ldots, y_g)$ *(or simply* $Count(n, y_1, \ldots, y_g)$*), where* n *is a variable and* $y_1, \ldots, y_g$ *are the other attributes of* A*. The* first-order equivalent form of a general formula Γ *is obtained from* Γ *by replacing its aggregate atoms with their first-order equivalent form.*

Definition 16 *Let* D *be a database and* $Count(A, n)$ *be an aggregate atom. Suppose,* $z_1, \ldots, z_l$ *are the local variables of* A *and* $y_1, \ldots, y_g$ *are the other attributes of* A *(some of which may be constants). One can always assume that* A *has the form* $P(z_1, \ldots, z_l, y_1, \ldots, y_g)$*, where* P *is a first-order order predicate. Suppose there are a finite number of constant terms* $a_1, \ldots, a_m$ *in the language underlying the database. In the context of database* D*, the* expansion *of* $Count(A, n)$ *defining its first-order equivalent, denoted by* $Exp_D(Count(A, n))$ *(or simply* $Exp(Count(A, n))$*, when* D *is clear from the context), is the following set of* $m^g + m^l + m^l(m^l - 1)/2 + m^l(m^l - 1)(m^l - 2)/3.2 + \cdots + 1$ *first-order clauses:*

m^g *clauses:*

$$\begin{aligned} Count(0, y_1, \ldots, y_g) \leftarrow\ & y_1 = a_1^1 \wedge \cdots \wedge y_g = a_g^1 \wedge \\ & \neg Count(1, y_1, \ldots, y_g) \wedge \cdots \\ & \wedge \neg Count(m^l, y_1, \ldots, y_g) \end{aligned}$$

where each a_i^1 *is equal to* a_p *for some* p*.*

m^l *clauses:*

$$\begin{aligned} Count(1, y_1, \ldots, y_g) \leftarrow\ & P(a_1^1, \ldots, a_l^1, y_1, \ldots, y_g) \wedge \\ & \neg Count(2, y_1, \ldots, y_g) \wedge \cdots \\ & \wedge \neg Count(m^l, y_1, \ldots, y_g) \end{aligned}$$

where each a_i^1 *is equal to some* a_p *for some* p*.*

$m^l(m^l - 1)/2$ *clauses:*

$$\begin{aligned} Count(2, y_1, \ldots, y_g) \leftarrow\ & P(a_1^1, \ldots, a_l^1, y_1, \ldots, y_g) \wedge \\ & P(a_1^2, \ldots, a_l^2, y_1, \ldots, y_g) \wedge \\ & \neg Count(3, y_1, \ldots, y_g) \wedge \cdots \\ & \wedge \neg Count(m^l, y_1, \ldots, y_g) \end{aligned}$$

where each a_i^k*,* $k = 1, 2$ *is equal to* a_p *for some* p *and for* $k \neq k'$*,* $1 \leq k, k' \leq 2$*, there exists a* p *such that* $a_p^k \neq a_p^{k'}$*.*

$m^l(m^l-1)(m^l-2)/3, 2$ *clauses:*

$$\begin{aligned} Count(3, y_1, \ldots, y_g) \leftarrow\ & P(a_1^1, \ldots, a_l^1, y_1, \ldots, y_g) \wedge \\ & P(a_1^2, \ldots, a_l^2, y_1, \ldots, y_g) \wedge \\ & P(a_1^3, \ldots, a_l^3, y_1, \ldots, y_g) \wedge \\ & \neg Count(4, y_1, \ldots, y_g) \wedge \cdots \\ & \wedge \neg Count(m^l, y_1, \ldots, y_g) \end{aligned}$$

where each a_i^k, $k = 1, 2, 3$ *is equal to* a_p *for some* p *and for* $k \neq k'$, $1 \leq k, k' \leq 3$, *there exists a* p *such that* $a_p^k \neq a_p^{k'}$.

$\vdots$

1 clause:

$$\begin{aligned} Count(m^l, y_1, \ldots, y_g) \leftarrow\ & P(a_1^1, \ldots, a_l^1, y_1, \ldots, y_g) \wedge \cdots \\ & \wedge P(a_1^{m^l}, \ldots, a_l^{m^l}, y_1, \ldots, y_g) \end{aligned}$$

where each a_i^k, $k = 1, 2, \ldots, m^l$ *is equal to some* a_p *for some* p *and for* $k \neq k'$, $1 \leq k, k' \leq m^l$ *there exists a* p *such that* $a_p^k \neq a_p^{k'}$.

When the above set of clauses has been added to the database, it would be said that the *database is expanded wrt the aggregate literal Count(A, n)* and the expanded database would be denoted as $Exp(D, Count(A, n))$.

Lemma 2 *For all ground substitutions* θ *of the variables in* $\{y_1, \ldots, y_g\}$, *at most one of* $Count(1, y_1, \ldots, y_g)\theta, \ldots, Count(m^l, y_1, \ldots, y_g)\theta$ *can be derived at a time from* $comp(Exp(D, Count(A, gn)))$.

Proof : Assume that both $Count(M, y_1, \ldots, y_g)\theta$ and $Count(N, y_1, \ldots, y_g)\theta$ can be derived from $comp(Exp(D, Count(A, n)))$, for $N \neq M$. Without any loss of generality one can assume $0 \leq M < N \leq m^l$. Since $Count(M, y_1, \ldots, y_g)\theta$ is derivable from $comp(Exp(D, Count(A, n)))$, from the clauses defining $Count(M, y_1, \ldots, y_g)$ one can say that none of $Count(M+1, y_1, \ldots, y_g)\theta, \ldots, Count(N, y_1, \ldots, y_g)\theta$, $\ldots, Count(m^l, y_1, \ldots, y_g)\theta$ is derivable from $comp(Exp(D, Count(A, n)))$. This contradicts the initial assumption. Hence the lemma. □

Lemma 3 *For all ground substitutions* θ *of the variables in* $\{y_1, \ldots, y_g\}$, $\chi_D(\leftarrow A\theta) = N$, *where* $Count(N, y_1, \ldots, y_g)\theta$ *is derivable from* $comp(Exp(D, Count(A, n)))$.

Proof : Suppose that only M different instances of θ are derivable from $comp(D)$ (i.e. $\chi_D(\leftarrow A\theta) = M$). The instances of $A\theta$ which are derivable from $comp(D)$ occur in the body of $U\theta$, where U is a clause in $Exp(D, Count(A, n))$. The head of U is $Count(M, y_1, \ldots, y_g))$ and $Count(M, y_1, \ldots, y_g)\theta$ is derivable from $comp(Exp(D, Count(A, n)))$. Hence, by Lemma 2, $M = N$. □

Definition 17 *Let D be a database and Γ an aggregate constraint and suppose $Count(A_1, n_1), \ldots, Count(A_p, n_p)$ are the only aggregate literals occurring in Γ. The* expansion of D wrt Γ, *denoted by $Exp(D, \Gamma)$, is the set $D \cup Exp(Count(A_1, n_1)) \cup \cdots \cup Exp(Count(A_p, n_p))$. Let I be a set of constraints and $\Gamma_1, \ldots, \Gamma_p$ be the only aggregate constraints in I. The* expansion of D wrt I, *denoted by $Exp(D, I)$, is the set $D \cup Exp(D, \Gamma_1) \cup \cdots \cup Exp(D, \Gamma_p)$.*

Lemma 4 *Let D be a database and I be a set of constraints with occurrences of aggregate literals $Count(A_1, n_1), \ldots, Count(A_p, n_p)$. If $comp(D)$ is consistent then $comp(Exp(D, I))$ is also consistent.*

Proof : For each integer n, suppose that $Count(n, y_1, \ldots, y_g)$ corresponding to $Count(P_i, n_i)$ for some $i = 1, 2, \ldots, p$ is replaced by another atom $Countn_i(y_1, \ldots, y_g)$, where $Countn_i$ does not appear elsewhere in the database. Then the clauses added to D to obtain $comp(Exp(D, I))$ can be taken as satisfying the hierarchical constraint. Because of the hierarchical nature of the clauses added to D and because added clauses define predicates which are not already defined in D, $comp(D)$ is consistent. □

Suppose Γ is an aggregate constraint and G is its first-order equivalent. In view of Lemma 3, D can be said to satisfy Γ if G is a logical consequence of $comp(Exp(D, \Gamma))$; otherwise D violates Γ. D is said to satisfy I, where I is a set of constraints, if D satisfies each constraint in I; otherwise D violates I.

An aggregate constraint Γ is transformed to its *first-order equivalent denial form* $\leftarrow G_d$ through the following steps:

1. Each first-order atom A which occurs in the conclusion and which contains at least one existentially quantified variable of the conclusion, is replaced by $P(x_1, \ldots, x_n)$. D is expanded by adding the clause $P(x_1, \ldots, x_n) \leftarrow A$ to it, where $x_1, \ldots, x_n$ are the variables of A also occurring in the condition and the predicate symbol P does not occur elsewhere in the database. Let D_T be the transformed database and G_T be the transformed aggregate constraint.

2. Convert G_T to its first-order equivalent form G_f .

3. G_f is transformed to its equivalent denial form $\leftarrow G_d$ by simply adding the negated form of each atom A, occurring in the conclusion, as a conjunct of the condition.

When a first-order constraint $\leftarrow W$ in denial form is considered, the standard method of determining whether a database satisfies or violates the constraint is by evaluating the query $\leftarrow W$ in the context of the database. If there is an SLDNF-refutation of $D \cup \{\leftarrow W\}$ via a safe selection then D violates $\leftarrow W$. If $D \cup \{\leftarrow W\}$ has a finitely failed SLDNF-tree, then D satisfies W. When an aggregate constraint Γ is considered, it is transformed to its equivalent first-order denial form $\leftarrow G_d$ in the above way by transforming the database D to $Exp(D_T, \Gamma)$. Then one can have the following syntactic definition of aggregate constraint satisfiability of C.

Lemma 5 *Let D be a database, Γ an aggregate constraint and R a safe selection rule. The terms D_T and $\leftarrow G_d$ are defined as above. Suppose $comp(D)$ is consistent. If there is an SLDNF-refutation of $Exp(D_T, \Gamma) \cup \{\leftarrow G_d\}$ via R then D violates Γ. If $Exp(D_T, \Gamma) \cup \{\leftarrow G_d\}$ has a T-finitely failed SLDNF-tree via R, then D satisfies Γ.*

Proof : Similar to the case of first-order constraint. □

3.3 The extended simplification theorem

An important task of the simplification method proposed by Lloyd *et al.* in [17] is to capture the difference between a model for $comp(D')$ and a model for $comp(D)$, where D and D' are databases and D' is obtained from D by the application of a transaction t to D. This is done by treating the transaction t as a sequence of deletions followed by a sequence of insertions where the application of the sequence of deletions to D produce the intermediate database D''. By using only rules of the database and updates, the method computes in stages the four sets of partially instantiated atoms $Pos_{D'',D'}$, $Neg_{D'',D'}$, $Pos_{D'',D}$ and $Neg_{D'',D}$, where the two sets $Pos_{D'',D'} \cup Neg_{D'',D}$ and $Neg_{D'',D} \cup Pos_{D'',D}$ represent respectively the part that is added to the model for $comp(D)$ when passing from D to D' due to a transaction and the part that is deleted. To preserve the consistency of the updated database, constraints are instantiated appropriately with these two sets of atoms and only affected constraints are evaluated. Instantiated constraints are evaluated using the SLDNF [4] proof procedure. The formulae for deriving the

above two sets of atoms are inductively defined as follows :

$$
\begin{aligned}
Pos^0_{D,D'} &= \{A : A \leftarrow W \in D' \setminus D\} \\
Neg^0_{D,D'} &= \{\} \\
Pos^{n+1}_{D,D'} &= \{A\theta : A \leftarrow W \in D, B \text{ occurs positively in } W, \\
&\quad C \in Pos^n_{D,D'}, \text{ and } \theta \text{ is an mgu of } B \text{ and } C\} \cup \\
&\quad \{A\theta : A \leftarrow W \in D, B \text{ occurs negatively in } W, \\
&\quad C \in Neg^n_{D,D'}, \text{ and } \theta \text{ is an mgu of } B \text{ and } C\} \\
Neg^{n+1}_{D,D'} &= \{A\theta : A \leftarrow W \in D, B \text{ occurs positively in } W, \\
&\quad C \in Neg^n_{D,D'}, \text{ and } \theta \text{ is an mgu of } B \text{ and } C\} \cup \\
&\quad \{A\theta : A \leftarrow W \in D, B \text{ occurs negatively in } W, \\
&\quad C \in Pos^n_{D,D'}, \text{ and } \theta \text{ is an mgu of } B \text{ and } C\} \\
Pos_{D,D'} &= \cup_{n \geq 0} Pos^n_{D,D'} \\
Neg_{D,D'} &= \cup_{n \geq 0} Neg^n_{D,D'}
\end{aligned}
$$

The generalised simplification method for the case of an aggregate constraint is as follows:

Theorem 1 *Let D be a databases and Γ an aggregate constraint such that D satisifies Γ. Suppose that t is a transaction consisting of a sequence of deletions followed by a sequence of insertions such that when the sequence of deletions is applied to D, it produces the intermediate database D'', and when the sequence of insertions is applied to this, it produces the database D'. Suppose that the completion of each of D and D' are consistent (this is guaranteed when each of D and D' is stratified [1]). If $x_1, \ldots, x_m$ are the global variables of Γ, the following three sets can be defined:*

$$
\begin{aligned}
\Theta &= \{\theta : \theta \textit{ is the restriction to } x_1, \ldots, x_m \textit{ of an mgu of the atom } A, \\
&\quad \textit{where } Count(A, n) \textit{ is an aggregate atom occuring in } \Gamma, \\
&\quad \textit{and an atom in} Pos_{D'',D'} \cup Neg_{D'',D'} \cup Pos_{D'',D} \cup Neg_{D'',D}\} \\
\Phi &= \{\phi : \phi \textit{ is the restriction to } x_1, \ldots, x_m \textit{ of an mgu of a first-order} \\
&\quad \textit{atom occurring negatively in } \Gamma \textit{(i.e. in the condition of } \Gamma) \\
&\quad \textit{and an atom in } Pos_{D'',D'} \cup Neg_{D'',D}\} \\
\Psi &= \{\psi : \psi \textit{ is the restriction to } x_1, \ldots, x_m \textit{ of an mgu of a first-order} \\
&\quad \textit{atom occurring positively in } \Gamma \textit{(i.e. in the conclusion of } \Gamma) \\
&\quad \textit{and an atom in } Neg_{D'',D'} \cup Pos_{D'',D}\}.
\end{aligned}
$$

Then D' satisfies Γ if and only if D' satisfies $\Gamma\mu$, for all $\mu \in \Theta \cup \Phi \cup \Psi$.

Proof: Let E denote $Exp(D, \Gamma)$. If $comp(D)$ is consistent, then by Lemma 4, $comp(E)$ is also consistent. Suppose that the transaction t produces E' when it is applied to E and produces the intermediate database E''. Recall that $Count(n, y_1, \ldots, y_g)$ is the first-order equivalent of an aggregate atom $Count(A, n)$, where $y_1, \ldots, y_g$ are the attributes other than the bound variables of A. Suppose G is the first-order equivalent of Γ. Consider the following two sets:

$$\begin{aligned}\Omega_1 = \{&\omega : \omega \text{ is the restriction to } y_1, \ldots, y_g \text{ of an mgu of an atom} \\ &Count(n, y_1, \ldots, y_g) \text{ occurring in the condition of } G \\ &\text{and an atom in } Pos_{E'',E'} \cup Neg_{E'',E}\},\end{aligned}$$

and

$$\begin{aligned}\Omega_2 = \{&\omega : \omega \text{ is the restriction to } y_1, \ldots, y_g \text{ of an mgu of an atom} \\ &Count(n, y_1, \ldots, y_g) \text{ occurring in the conclusion of } G \\ &\text{and an atom in } Neg_{E'',E'} \cup Pos_{E'',E}\}.\end{aligned}$$

Suppose $\theta \in \Theta$ and $A' \in Pos_{D'',D'} \cup Neg_{D'',D}$ such that A' unifies A, where A has the form $P(z_1, \ldots, z_l, y_1, \ldots, y_g)$. Each y_j is equal to x_k, for some k, $1 \leq k \leq m$. Whatever the bindings to the variables $z_1, \ldots, z_l$ corresponding to θ, $Count(1, y_1, \ldots, y_g)\theta, \ldots, Count(m', y_1, \ldots, y_g)\theta$ are members of $Pos_{E'',E'} \cup Neg_{E'',E}$, and $Count(0, y_1, \ldots, y_g)\theta, \ldots, Count(m'-1, y_1, \ldots, y_g)\theta$ are members of $Neg_{E'',E'} \cup Pos_{E'',E}$.

Again, suppose $\omega \in \Omega_1$. This implies that $P(a_1^i, \ldots, a_l^i, y_1, \ldots, y_g)\omega$ is a member of $Pos_{E'',E'} \cup Neg_{E'',E}$, i.e. $P(a_1^i, \ldots, a_l^i, y_1, \ldots, y_g)\omega$ is a member of $Pos_{D'',D'} \cup Neg_{D'',D}$.

The above two paragraphs prove that, in the case of the addition of a fact, the satisfiability of $\Gamma\theta$ in D' is equivalent to the satisfiability of $\Gamma\omega$ in E. Similarly, in the case of deletion. Hence,

$$\begin{aligned}&E' \text{ satisfies } \Gamma\omega, \text{for all } \omega \in \Omega_1 \cup \Omega_2 \cup \Phi \cup \Psi \\ \Leftrightarrow\ &E' \text{ satisfies } \Gamma\mu, \text{for all } \mu \in \Theta \cup \Phi \cup \Psi.\end{aligned}$$

Again,

$$\begin{aligned}&D' \text{ satisfies } \Gamma \\ \Leftrightarrow\ &E' \text{ satisfies } G \text{(by definition)} \\ \Leftrightarrow\ &E' \text{ satisfies } G\omega, \text{for all } \omega \in \Omega_1 \cup \Omega_2 \cup \Phi \cup \Psi \\ &\text{(by simplification theorem for first-order constraints)} \\ \Leftrightarrow\ &E' \text{ satisfies } G\mu, \text{for all } \mu \in \Theta \cup \Phi \cup \Psi \\ \Leftrightarrow\ &D' \text{ satisfies } G\theta, \text{for all } \theta \in \Theta \cup \Phi \cup \Psi.\end{aligned}$$

□

3.4 Generalisation to other aggregate predicates

If an aggregate atom involving the predicate *Sum* occurs in a constraint, then the lemmas in the previous section as well as the above simplification theorem still hold since an occurrence of an aggregate atom of the form $Sum(z, A, r)$ in a constraint can be replaced by $Sum(r, y_1, \ldots, y_g)$ by adding the following clauses to the database (assuming that A has the form $P(z_1, \ldots, z_l, y_1, \ldots, y_g)$ and without any loss of generality z has been taken as z_1) :

1 clause

$$Sum(x, y_1, \ldots, y_g) \leftarrow Sum(_, x, y_1, \ldots, y_g)$$

$m^l(m^l-1)\cdots(m^l-k+1)/k(k-1)\cdots 2.1$ clauses, for each $k = 1, \ldots, m^l$:

$$\begin{aligned} Sum(k, x, y_1, \ldots, y_g) \leftarrow\ & P(a_1^1, \ldots, a_l^1, y_1, \ldots, y_g) \wedge \cdots \\ & \wedge P(a_1^k, \ldots, a_l^k, y_1, \ldots, y_g) \wedge \\ & x = a_1^1 + \cdots + a_1^k \wedge \\ & \neg Sum(k+1, _, y_1, \ldots, y_g) \wedge \ldots \\ & \wedge \neg Sum(m^l, _, y_1, \ldots, y_g) \end{aligned}$$

where each a_i^p, $p = 1, \ldots, k$ is equal to a_q, for some q' and for $q \neq q'$, $1 \leq q, q' \leq k$, there exists an i such that $a_i^q \neq a_i^{q'}$.

For an occurrence of the aggregate predicate *Maximum*, the set of clauses added is the following:

1 clause :

$$Maximum(x, y_1, \ldots, y_g) \leftarrow Maximum(_, x, y_1, \ldots, y_g)$$

For each $k = 1, \ldots, m^l$, $km^l(m^l-1)\cdots(m^l-k+1)/k(k-1)\cdots 2.1$ clauses:

$$\begin{aligned} Maximum(k, x, y_1, \ldots, y_g) \leftarrow\ & P(a_1^1, \ldots, a_l^1, y_1, \ldots, y_g) \wedge \cdots \\ & \wedge P(a_1^k, \ldots, a_l^k, y_1, \ldots, y_g) \wedge \\ & a_1^j > a_1^1 \wedge a_1^j > a_1^{j-1} \wedge \cdots \\ & \wedge a_1^j > a_1^{j+1} \wedge \cdots \wedge a_1^j > a_l^k \\ & \wedge \neg Maximum(k+1, _, y_1, \ldots, y_g) \wedge \\ & \cdots \wedge \neg Maximum(m^l, _, y_1, \ldots, y_g) \end{aligned}$$

where each a_i^p, $p = 1, \ldots, k$ is equal to a_q for some q, and for $q \neq q'$, $1 \leq q, q' \leq k$, there exists an i such that $a_i^q \neq a_i^{q'}$.

The case of an aggregate atom involving the predicate *Minimum* is similar to the above case for *Maximum*. The aggregate predicate *Average* can be defined as an extension of the predicate *Sum*.

3.5 Prolog implementation

Let D be a database and Γ an aggregate constraint. Lemma 5 and Theorem 1 state that the simplified first-order equivalent denial form of Γ would have to be evaluated in the updated database E' to preserve consistency, where E' is obtained by applying the transaction to $E(= Exp(D, \Gamma)$, assuming that the existentially quantified variables have already been resolved). However, in the Prolog implementation, the effect of the resolution in E' of a selected literal $Count(x, y_1, \ldots, y_g)$ from a goal which is the first-order equivalent of an aggregate literal $Count(P(z_1, \ldots, z_l, y_1, \ldots, y_g), n)$, can be achieved by evaluating the Prolog goal $? - setof([Z_1, \ldots, Z_l], p(Z_1, \ldots, Z_l, Y_1, \ldots, Y_g), N)$ in D'. Hence, one does not need to expand the underlying database D to E. In Prolog, the aggregate predicate $Count(A, N)$ is represented by $count(p([Z_1, \ldots, Z_l], [Y_1, \ldots, Y_g]), N)$, and defined as

```
count(P, N):-
    P=..[_|[Z|_]],
    setof(Z, P, ZL),
    length(ZL, N).
```

where the procedure `length(ZL, N)` returns the length of the list `ZL` into `N`. To get the correct intended meaning of an aggregate constraint Γ, it is necessary to follow a generalised computation rule, called *generalised safe*, which selects

1. a first-order negative literal, only when it is ground, and
2. an aggregate positive literal, only when its key formula is ground wrt to its free variables, and
3. an aggregate negative literal, only when it is ground wrt its global variables.

In Prolog's leftmost literal selection strategy, to get the effect of a generalised safe computation rule an aggregate literal λ (resp. negative literal L) can be placed after all the positive literals containing all the occurrences of global variables of λ (resp. variables of L). The other aggregate predicates can be implemented efficiently using a similar mechanism.

4 Extension to indefinite databases

In this section, the class of databases considered is indefinite and hence any usage of the term 'database' will be interpreted as 'indefinite database'.

4.1 Possible forms of a database

The definition of an indefinite database given in Section 2 is slightly different from that given in [12]. The definition adopted in this paper does not allow a negative fact to be explicitly represented in the database. Instead, a special rule is introduced to infer negative facts from the database. This is somewhat similar to the closed world assumption or negation as failure in the case of definite databases.

Consider the following example of an indefinite database.

Example 1 *Database D1:*

Rules: Definite rules:

$$Postgraduate(x) \leftarrow MSc(x)$$

$$Postgraduate(x) \leftarrow PhD(x)$$

$$Student(x) \leftarrow Undergraduate(x)$$

Indefinite rule:

$$MSc(x) \vee PhD(x) \leftarrow Student(x) \wedge \neg Undergraduate(x)$$

Facts: Definite facts:

$$Student(Subrata)$$

$$Supervisor(Howard, Subrata)$$

Indefinite fact:

$$Undergraduate(Choux) \vee MSc(Choux)$$

Definition 18 *Let D be a database and R be a clause in D. Let A be an atom occurring in the head of R. The* possible form *of R with respect to A denoted by $pos(R, A)$ is the clause obtained from R by replacing its head by A.*

According to this definition, the possible form of a definite rule R with respect to the only atom in its head is the rule R and the possible form of a definite or indefinite fact with respect to the atom A appearing in it, is the fact A. The set of clauses comprising at least one possible form of each of the clauses of D is called a *possible form* of the database D. For example, database $D1$ has nine possible forms, each of which is the union of the set of all definite clauses of $D1$ with different possible forms arising from the two indefinite clauses of $D1$.

A query relating to the database $D1$ is

Who are the postgraduate students ?

which can be expressed as

$$\leftarrow Postgraduate(x).$$

In the context of both relational and definite databases, a substitution for the variables of a query makes the bound body of the query either true or false in the database. If the bound body is true in the database then the corresponding substitution is called an answer substitution. By contrast, an indefinite database may represent several possible worlds and in the context of this kind of database, a substitution for the variables of a query can make the bound body of the query neither true nor false in the database. A substitution which makes the query true in every possible form of the database is called a *definite answer* (e.g. for the query $\leftarrow Postgraduate(x)$ the substitution $\{x/Das\}$ is a definite answer) whereas one which makes the query true in at least one possible form of the database (e.g. $\{x/Choux\}$) is called a *possible answer.*

4.2 The declarative semantics for negative information

Let D be a database and $HB(D)$ its Herband base. Let $\{D_1, \ldots, D_n\}$ be the set of all possible forms of D such that the completion of each D_i is consistent. Then the Herband base $HB(D)$ can be partitioned into three subsets:

$$\begin{aligned} Def_true(D) &= \cap_{i=1}^{n}\{A : A \in HB(D) \text{ and } comp(D_i) \vdash A\} \\ Def_false(D) &= \cap_{i=1}^{n}\{A : A \in HB(D) \text{ and } comp(D_i) \vdash \neg A\} \\ Unknown(D) &= HB(D) \setminus (Def_true(D) \cup Def_false(D)) \end{aligned}$$

where the symbol \ denotes set difference. The set of facts in $Def_true(D)$ can be taken as definitely true in D whereas the set of facts in $Def_false(D)$ are definitely false in D. The facts in $Unknown(D)$ are possibly true in D.

Clearly, when the database is definite the above definition of semantics reduces to Clark's idea of a completed database. When the database is definite and hierarchical the set *Unknown* is empty and the Herband base is divided between the two sets Def_true and Def_false. The SLDNF-resolution mechanism suffices to determine any element of the set Def_true and hence any element of the set Def_false. When all three subsets are non-empty, the situation is more complex. In this case the definite resolution mechanism determines the elements of the set Def_true. The possible resolution mechanism will be able to determine the set of elements which are either in the set *Unknown* or in the set Def_true. An element of the Herband base which is not generated by the possible resolution mechanism can be taken to be false and hence is a member of Def_false.

4.3 The procedural semantics for negative information

The procedural semantics is based on two mutually recursive resolution schemes, one for possible resolution and the other for definite resolution.

4.3.1 Possible Resolution

Let D be a database, G a goal, and R a computation rule. Initially, none of the clauses in D is marked as used possibly with respect to atoms occurring in their heads. A *possible derivation* for $D \cup \{G\}$ via R consists of a sequence $D_0^+ = D, D_1^+, D_2^+, \ldots$ of positive databases, a sequence $D_0^- = \{\}, D_1^-, D_2^-, \ldots$ of negative databases, a sequence $G_0 = G, G_1, G_2, \ldots$ of goals, a sequence $C_1, C_2, \ldots$ of variants of ground negative literals or program clauses from the possible forms of D, and a sequence $\theta_1, \theta_2, \ldots$ of substitutions satisfying the following:

1. For each i, G_i is $\leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_p$, the selected literal L_m in G_i is
   (a) a positive literal and L_m unifies with
      i. A definite fact A in D_i^+ with an mgu θ_{i+1}. Then C_{i+1} is A, D_{i+1}^+ and D_{i+1}^- are respectively D_i^+ and D_i^-, and G_{i+1} is $\leftarrow (L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_p)\theta_{i+1}$.

ii. The head of a definite rule $H \leftarrow B$ in D_i^+ with an mgu θ_{i+1}. Then C_{i+1} is $H \leftarrow B$, D_{i+1}^+ and D_{i+1}^- are respectively D_i^+ and D_i^-, and G_{i+1} is $\leftarrow (L_1 \wedge \cdots \wedge L_{m-1} \wedge B \wedge L_{m+1} \wedge \cdots \wedge L_p)\theta_{i+1}$.

iii. An atom A occurring in an indefinite fact F of D_i^+ with an mgu θ_{i+1} and F has not been used possibly with respect to A. If none of the literals of D_i^- has a definite refutation in $D_i^+ \cup \{pos(F, A)\}$ then C_{i+1} is $pos(F, A)$, D_{i+1}^+ is $D_i^+ \cup \{A\}$, D_{i+1}^- is D_i^-, and G_{i+1} is $\leftarrow (L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_p)\theta_{i+1}$. Finally, F is marked as used possibly with respect to A.

iv. An atom A occurring in the head of an indefinite rule $R : F \leftarrow B$ of D_i^+ with an mgu θ_{i+1} and R has not been used possibly with respect to A. If none of the literals of D_i^- has a definite refutation in $D_i^+ \cup \{pos(R, A)\}$ then C_{i+1} is $pos(R, A)$, D_{i+1}^+ and D_{i+1}^- are respectively $D_i^+ \cup \{pos(R, A)\}$ and D_i^-, and G_{i+1} is $\leftarrow (L_1 \wedge \cdots \wedge L_{m-1} \wedge B \wedge L_{m+1} \wedge \cdots \wedge L_p)\theta_{i+1}$. Again, R is marked as used possibly with respect to A.

(b) a ground negative literal $\neg A$ and there is no definite refutation (introduced in the next section) for $D_i^+ \cup \{\leftarrow A\}$. In this case, θ_{i+1} is an identity substitution, C_i is $\neg A$, D_{i+1}^+ and D_{i+1}^- are respectively D_i^+ and $D_i^- \cup \{A\}$, and G_{i+1} is $\leftarrow L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_p$.

2. If the sequence $G_0, G_1, \ldots$ of goals is finite, then the last goal G_n of the sequence is either empty or has the form $\leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_p$, where L_p is selected and

(a) L_m is an atom and there is no program clause in D_n^+ for which any of the literals occurring in its head unifies with L_m , or

(b) L_m is an atom and there are program clauses $\{R^1, \ldots, R^l\}$ in D_n^+ such that for each j, one of the atoms A occurring in the head H^j of R^j unifies with L_m . But, for each j, there exists a literal A' in D_n^- such that $D_n^+ \cup \{pos(R^j, A)\} \cup \{\leftarrow A'\}$ has a definite refutation, or

(c) L_m is a ground negative literal of the form $\neg A$ and $D_n^+ \cup \{\leftarrow A\}$ has a definite refutation.

Each derivation of a possible resolution constructs a sequence of positive databases and a sequence of negative databases. When a goal is refuted by a possible resolution with an answer substitution θ, it can be said that the goal bound with the answer substitution θ is true in the last database of the sequence of positive databases. The sequence of negative databases keeps track of the ground negative atoms occurring in goals and which have been inferred false by step 1.1b. Thus, a member of D_i^- is false in D_i^+, for every i.

Steps 1(a)i and 1(a)ii formalise the SLD-resolution principle for definite clauses and hence in these two cases both positive and negative database remain fixed. Step 1.1b is a negation as failure which is concerned with keeping track of the inferred negative literal by adding it to the literal to the latest negative database. The process of marking clauses in steps 1(a)iii and 1(a)iv is necessary to avoid redundant answers.

A possible derivation is *finite* if it consists of a finite sequence of goals; otherwise, it is *infinite*. A possible derivation is *successful* if it is finite and the last goal is the empty goal. A successful possible derivation is called a *possible refutation*. A possible derivation is *failed* if it is finite and the last goal is not the empty goal.

From the definition of possible derivation, a *possible tree* for $D \cup \{G\}$ via R is defined in the usual way where each branch of the possible tree corresponds to a possible derivation. A branch of a possible tree for which the terminating node is an empty goal is called a *success branch*, a branch which does not terminate is called an *infinite branch* and a branch for which the terminating node is other than an empty goal is called a *failure branch*. A possible tree for which every branch is a failure branch is a *finitely failed possible tree*.

If each possible form of a database D is hierarchical, then the possible resolution in D terminates, provided that the definite resolution of step 1.1b for resolving negative literals terminates.

Definition 19 *Let D be a database and G be a goal $\leftarrow W$. An* answer substitution *for $D \cup \{G\}$ is a substitution for the variables of G. A* possible computation *of $D \cup \{G\}$ is an attempt to construct a possible derivation of $D \cup \{G\}$. A* possible computed answer *θ for $D \cup \{G\}$ is the substitution obtained by restricting the composition $\theta_1 \cdots \theta_n$ to the variables of G, where $\theta_1, \ldots, \theta_n$ is the sequence of substitutions used in a possible-refutation of $D \cup \{G\}$. When G is allowed, every possible computed answer for $D \cup \{G\}$ is a ground substitution for variables in G. A possible computed answer for $D \cup \{G\}$ will also be referred to as a* possible answer substitution *for the possible-refutation of $D \cup \{G\}$.*

4.3.2 Definite resolution

The formal definition of a resolution principle to obtain a definite answer substitution of a query can be adopted from either the definite resolution introduced in [6] or the proof procedure like SLGNF [18] or nH-Prolog [19],[23] (by extending the capability of inferring negative information). In this paper, an nH-Prolog-like resolution has been considered to define a *definite derivation*,

definite refutation, etc. From the implementation considered these terms can be defined easily.

4.3.3 Negation as possible failure

Based on the two resolution strategies discussed in the previous two sections, the rule (procedural semantics) for inferring negative information denoted by *negation as possible failure*, is defined as follows. The negation of a fact A is taken as true in a database D if $D \cup \{\leftarrow A\}$ has a finitely failed possible tree.

See [6] for more details about the two resolution schemes and the equivalence between declarative and procedural concepts of negative information semantics.

4.4 Constraint satisfiability in indefinite databases

In this subsection and the remainder of Section 4, a constraint is taken to be a closed first order formula. In the context of database $D1$ the constraint

every postgraduate student has a supervisor

can be expressed as the first order formula

$$\forall x(Postgraduate(x) \rightarrow \exists y Supervisor(y, x)).$$

Definition 20 *(Declarative definition of constraint satisfiability) A database D is said to* satisfy a constraint *C if C is a logical consequence of the completion of each of the possible forms of D. The database D is said to satisfy a set of constraints I if D satisfies each of the constraints of I; otherwise, D violates I.*

From the above definition of constraint satisfiability if one wants to impose the above constraint on the database $D1$ the integrity of the database would be violated as $Choux$ can be proved to be a postgraduate student in one of the possible forms of $D1$ without having a supervisor in that possible form.

For the method introduced in the next section for checking integrity in a database, each constraint is an allowed formula expressed in denial form. If a constraint W is not in denial form then it can be transformed to denial form by the same process described in Section 3.1. When the database D

is indefinite, the constraint W is a logical consequence of the completion of each of the possible forms of D if and only if the transformed constraint $\leftarrow A$ is a logical consequence of the completion of each of the possible forms of D_T. This is true because the difference between D_T and D is only a set of definite clauses and they are part of each and every possible form of D_T . This justifies the transformation of a constraint to a denial. As an example, the $Postgraduate - Supervisor$ constraint can be transformed to a denial of the form $\leftarrow A$ by adding the rules

$$\begin{aligned} A &\leftarrow Postgraduate(x) \wedge \neg B(x) \\ B(x) &\leftarrow Supervisor(y, x) \end{aligned}$$

to the database, where the two predicates A and B are not used elsewhere in the database.

Let I be a set of constraints. Then I_d will represent the transformed set of clauses of the form

$$IC(No) \leftarrow B$$

from I, where $\leftarrow B$ is a constraint from I and No is a unique identification of the constraint. $IC(No)$ will be called the *head* of the constraint $\leftarrow B$.

The declarative definition of constraint satisfiability in the context of a database D can be related to the declarative semantics of D as follows. The constraint $\leftarrow B$ violates the database D if a ground instance of B can be made true in $Def_true(D) \cup Unknown(D)$. Accordingly, the syntactic definition of constraint satisfiability, which is equivalent to the declarative definition, is as follows:

Definition 21 *(Syntactic definition of constraint satisfiability) Let D be a database and $\leftarrow B$ a constraint in denial form.*

1. *If $D \cup \{\leftarrow B\}$ has a finitely failed possible tree then $\leftarrow B$ is said to satisfy D*
2. *If $D \cup \{\leftarrow B\}$ has a possible refutation, then $\leftarrow B$ violates D.*

4.5 The extended path finding method

This method for checking integrity, introduced in [7] for the case of definite databases, is now generalised for checking integrity in indefinite databases. In the case of definite databases the method calculates a set of facts implicitly added to the database and a set of partially instantiated atoms whose

instances are likely to be deleted from the database due to an update. These two sets of atoms are determined through the computation of all possible paths from update literals. If a success path is found then the integrity is said to be violated in the database. In the case of indefinite databases, the method has been generalised by determining the following two sets:

1. the set of facts implicitly added to the database and possibly true in the updated database, and
2. the set of atoms whose instances are likely to be deleted from the database.

These two sets are computed through all possible path calculations. The formal definition of a path is as follows.

Definition 22 *Let D be a database. A* (possible) path *in D is defined as a chain of literals*

$$L_0 \xrightarrow{R_1} L_1 \xrightarrow{R_2} \cdots \xrightarrow{R_n} L_n$$

where L_0 is called the source *of the path, L_n its* destination, *n its* length *and $R_1, \ldots, R_n$ are clauses from the union of the possible forms of D* used *to construct the path from L_0 to L_n . If the source L_0 is positive then it is ground and $D \cup \{\leftarrow L_0\}$ has a possible-refutation. For any two consecutive literals L_i and L_{i+1} , L_{i+1} is called the* successor *of L_i in the path, and is obtained from L_i in one of the following ways:*

1. If
   (a) L_i is positive, and
   (b) L_i unifies with a positive literal L occurring in the body of the clause $R : A_1 \vee \cdots \vee A_m \leftarrow B$, and
   (c) α is an mgu of L_i and L, and
   (d) θ is a possible computed answer for $D \cup \{G\}$, where G is the goal $\leftarrow B\alpha$

   then L_{i+1} is the term $A_j\alpha\theta$, $1 \leq j \leq m$, and R_{i+1} is $pos(R, A_j)$.
2. If
   (a) L_i is positive, and
   (b) the negative literal $\neg L$ occurs in the body of the clause $R_i : A_1 \vee \cdots \vee A_m \leftarrow B$ such that L_i unifies with L, and

(c) α is an mgu of L_i and L, and

(d) $\neg A_j\alpha$, $1 \leq j \leq m$, is not an instance of any one of the L_k 's, where $0 \leq k \leq i$

then L_{i+1} is the term $\neg A_j\alpha$, $1 \leq j \leq m$, and R_{i+1} is $pos(R, A_j)$.

3. If

(a) L_i is negative, and

(b) L_i unifies with a negative literal L occurring in the body of the clause $R : A_1 \vee \cdots \vee A_m \leftarrow B$, and

(c) α is an mgu of L_i and L, and

(d) θ is a possible computed answer for $D \cup \{G\}$, where G is the goal $\leftarrow B\alpha$

then L_{i+1} is the term $A_j\alpha\theta$, $1 \leq j \leq m$, and R_{i+1} is $pos(R, A_j)$.

4. If

(a) L_i is negative and has the form $\neg M$, and

(b) M unifies with a positive literal L occurring in the body of the clause $R : A_1 \vee \cdots \vee A_m \leftarrow B$, and

(c) θ is an mgu of M and L, and

(d) $\neg A_j\alpha$, $1 \leq j \leq m$, is not an instance of any one of the L_k 's, where $0 \leq k \leq i$

then L_{i+1} is the term $\neg A_j\alpha$, $1 \leq j \leq m$, and R_{i+1} is $pos(R, A_j)$.

From this definition, it is possible to construct more than one path starting from the same literal. Furthermore, a positive literal on a path is always ground and possibly true in the database. For simplicity, a path will sometimes be written without showing the clauses used to construct the path. In the path $L_0 \rightarrow L_1 \rightarrow \cdots \rightarrow L_n$, the *distance* of L_p from L_q $(0 \leq q \leq p \leq n)$ is $p - q$ and L_p is said to be *at a distance* $p - q$ from L_q .

Definition 23 *Let D be a database and L a literal such that if L is positive then it is ground and $D \cup \{\leftarrow L\}$ has a possible-refutation. Let S be the set of all paths with L as the source. The* (possible) path space *rooted at the literal L is a tree defined as follows :*

1. Each node of the tree is a literal.

2. The root node is L.
3. If N is a node of the tree, then the set of all successors of N in the paths of S are the only descendants of N in the tree.

Each branch of the path space corresponds to a path in the database with the root node as the source and vice-versa. A path which ends at the head $IC(No)$ of a constraint will be called a *(possible) success path*; otherwise, it will be called a *failure path*. A path space containing at least one branch which corresponds to a success path is referred to as a *(possible) success path space*.

To check integrity in a database when a new constraint is added, the database is queried directly to make sure that the constraint is a logical consequence of the completion of each of the possible forms of the database. If a constraint is deleted then this cannot cause any inconsistency. To check integrity in the updated database D' as a result of the transaction t applied to D, the source is taken as an update literal. An *update literal* of the transaction t applied to a database D may be one of the following :

1. An atom occurring in a fact of t which is to be added to D.
2. The negation of an atom occurring in a fact of t which is to be deleted from D.
3. If an indefinite rule $R : A_1 \vee \cdots \vee A_m \leftarrow B$ in t is to be added to D then for a possible computed answer θ for $D' \cup \{\leftarrow B\}$, the corresponding instance of an atom occurring in the head of the rule R, i.e. $A_i\theta$, $1 \leq i \leq m$, which is possibly implicitly added to D due to the transaction.
4. If a rule $R : A_1 \vee \cdots \vee A_m \leftarrow B$ in t is to be deleted from D then the negation of an atom occurring in the head of the rule R, i.e. $\neg A_i$, $1 \leq i \leq m$, whose instances are likely to be deleted from D due to the transaction.

To preserve integrity one must ensure that a success path space does not exist with source as an update literal in the updated database. The algorithm tries to construct a success path through backtracking.

It has been assumed that the transaction does not add and delete the same clause. A transaction is valid if all of its update literals can be computed successfully without going into an infinite derivation. Equally, a transaction is rejected if a success path is found in the updated database. Modification can be considered as being accomplished by a deletion followed by an addition.

The method is illustrated by means of the following few examples.

Example 2 *Database D2:*

Rules: **R1.** $P(x,y) \leftarrow Q(x,y) \wedge R(x)$

R2. $Q(x,y) \leftarrow S(x,y) \wedge \neg M(x)$

R3. $R(x) \leftarrow U(x)$

R4. $R(x) \leftarrow V(x)$

R5. $U(x) \vee V(x) \leftarrow T(x) \wedge \neg W(x)$

R6. $M(x) \leftarrow R(x) \wedge N(x)$

Facts: $N(A)$

$S(A,C)$

$S(B,C)$

Integrity constraints $I2$: **IC1.** $IC(1) \leftarrow P(x,y) \wedge \neg O(x)$

Update: insertfact $T(A) \vee T(B)$

where x,y are variables and A,B,C are constants. As each of the possible forms of $D2$ is hierarchical in nature, their completions are consistent. Before the update is applied, database $D2$ satisfies the constraint $I2$. Let $D2'$ be the updated database.

As the update is the insertion of an indefinite fact $T(A) \vee T(B)$, the set of update literals is $\{T(A), T(B)\}$. Taking $T(A)$ as a source one can have, with the help of **R5**, either of $U(A)$ or $V(A)$ as the next literal of the source. Considering $U(A)$ first, the next literal of the path is $R(A)$. To find a successor of $R(A)$ with the help of **R1**, one has to find a possible computed answer of $D2 \cup \{\leftarrow Q(A,y) \wedge R(A)\}$. The fact $Q(A,C)$ is an instance of $Q(A,y)$ which is true in one of the possible forms of $D2$ provided that $M(A)$ is false in that possible form. $R(A)$ could be taken to be true in that possible form if $M(A)$ was not provable with the help of **R6**. Hence no instance of $Q(A,y) \wedge R(A)$ is true in the constructed database and **R1** fails to generate a successor for $R(A)$. In the process of backtracking when $V(A)$ is considered, the generated path also becomes failed for the same reason. Backtracking further and considering $T(B)$ as the source, one can construct the success path

$$T(B) \overset{pos(R_5,U(x))}{\rightarrow} U(B) \overset{R_3}{\rightarrow} R(B) \overset{R_1}{\rightarrow} P(B,C) \overset{IC1}{\rightarrow} IC(1).$$

The two complete path spaces generated as a result of the update are shown in Figure 1.

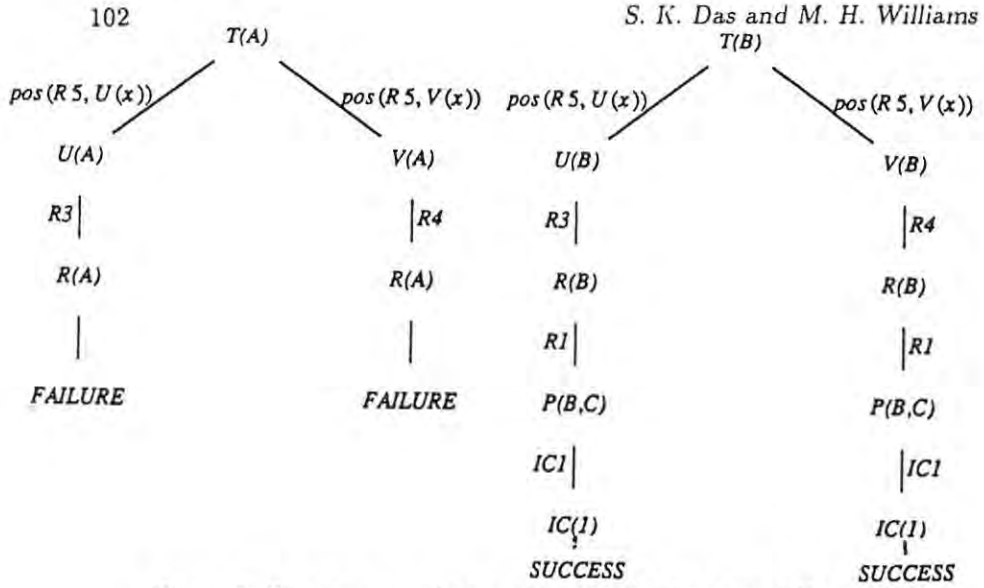


Figure 1: The two complete path spaces in the case of Example 2

4.6 Prolog implementation

This section describes how the integrity checking method described in the previous section and the two resolution strategies described in Section 4 may be implemented in Prolog.

A unique number has been assigned to each newly inserted constraint as an identification. A constraint $\leftarrow B$ numbered No is stored directly as a Prolog rule

$$ic(No) : -B.$$

A rule or a constraint can be retrieved efficiently by using a reference which uniquely identifies the corresponding rule or constraint.

The top level C-Prolog goal for integrity checking ($? - ic_violated(Tran).$) can be defined as

```
ic_violated(Tran):-
    path(ic(No),Tran,[],P,N),!.
```

where $Tran$ represents an update literal and $path(D, S, NegList, P, N)$ means that a path is to be constructed from the literal S to the literal D and the set comprising the negation of each literal of the list $NegList$ are the only negative literals which have occurred so far in this path construction. Each of P and N is an open-ended structure of the form

$$((\ldots((X, C_1), C_2), \ldots), C_n)$$

where X is variable and each C_i is a structure or an atom which will be described later in this section. This helps to keep information regarding the dynamic construction of a possible form of a database. With this convention if the database becomes inconsistent due to the transaction $Tran$, then the first inconsistency is reported at a constraint numbered No.

The path can be computed efficiently by maintaining additional information for each rule and constraint in the database. For each literal L occurring in the body of a rule $R : A_1 \vee \cdots \vee A_m \leftarrow B$ and for each i, $1 \leq i \leq m$, clauses of the form

$$depend(Ref, A_i, L)$$

$$depend(Ref, \neg A_i, CompL)$$

have been maintained, where Ref is the reference which uniquely identifies the clause R and $CompL$, called the complement of L, is defined as follows. When L is positive, $CompL$ is $\neg L$; otherwise, when L is $\neg A$, then $CompL$ is A. For each literal L occurring in a constraint $\leftarrow B$, a clause of the form

$$depend(Ref, ic(No), L).$$

has been maintained, where $ic(No)$ is the head of the constraint and Ref is the reference which uniquely identifies the clause $ic(No) \leftarrow B$.

The task of finding a path from a source literal S to a destination literal D can be achieved in Prolog with the help of the following meta-interpreter:

```
path(D,S,_,P,N):-
    depend(Ref,D,S),
    cl(Ref,Hs,B),
    simplify(B,S,SB),
    simplify(Hs,D,_),
    pos(SB,P,N).

path(D,S,NegList,P,N):-
    depend(_,Via,S),
    Via=(not A),
    not instance_list(A,NegList),
    path(D,Via,[A|NegList],P,N).

path(D,S,NegList,P,N):-
    depend(Ref,Via,S),
    not Via=(not _),
    cl(Ref,Hs,B),
    simplify(B,S,SB),
```

```
simplify(Hs,Via,_),
pos(SB,P,N),
path(D,Via,NegList,P,N).
```

$clause(H, B, Ref)$ means H and B are unified respectively with the head and body of a clause which is uniquely identified by the reference Ref. The interpretation of $simplify(B, S, SB)$ is that SB is an instance of B obtained by unifying one of B's body literals with S. $instance(A, NegList)$ means that A is not an instance of any of the atoms occurring in the list $NegList$. This corresponds to conditions 2(d) and 4(d) in the definition of a path and prevents a possible generation of an infinite path from a negative literal in the presence of recursive rules. The goal $pos(SB)$ is satisfied if there is a possible refutation of the goal SB in the underlying database modified by P and N.

In the implementation of a query evaluation system, nH-Prolog [19],[23] has been selected and extended with the capability of deriving negative information from the database according to the semantics of negation as possible failure. In this case the format used for storing a definite clause [23] using a predicate nH is

$$nH(Head, Body)$$

whereas an indefinite clause is stored under the same predicate as

$$nH(ChosenHead, AuxiliaryHeads, Body, AncestorList).$$

Beside these a separate clause of the form

$$nH_index(Functor, Arity, RefList)$$

has been maintained to store all the references to clauses containing an occurrence of a particular user-defined predicate in their heads, where the predicate is identified by $Functor$ and $Arity$. In a database the number of clauses defining a predicate could be very high and in such a situation maintaining the above set of clauses would be both inefficient and wasteful of storage. Instead, definite clauses have been stored under the predicate cl as

$$cl(StartRef, NextRef, Goal, Body)$$

and an indefinite clause as

$$cl(IndRef, Posn, StartRef, NextRef, Goal, AuxHeads, Body, NewAnc),$$

where $StartRef$ is the unique reference (1,2,...etc) to the clause among the set of clauses containing an occurrence in their heads the user defined

predicate P for *Goal* with specific arity; $NextRef$ is the same for the next clause defining P with the same arity and this is represented by a unique reference (zero) when there is no next clause. For indefinite clauses the two parameters $IndRef$ and $Posn$ denote respectively a unique indefinite clause reference ($i1, i2, \ldots$ etc) and a unique position ($p1, p2, \ldots$ etc) of *Goal* in the head of the indefinite clause. Under the above convention the database $D1$ can be represented as follows :

$cl(1, 2, postgraduate(X), msc(X))$
$cl(2, 0, postgraduate(X), phd(X))$
$cl(1, 2, student(X), undergraduate(X))$
$cl(2, 0, student(X), true)$
$cl(1, 0, supervisor(howard, subrata), true)$
$cl(i1, p1, 1, 2, msc(X), [dh(phd(X), Y)],$
$(student(X), not\ undergraduate(X)), Y)$
$cl(i1, p2, 1, 0, phd(X), [dh(msc(X), Y)],$
$(student(X), not\ undergraduate(X)), Y)$
$cl(i2, p1, 1, 0, undergraduate(choux), [dh(msc(choux), X)], true, X)$
$cl(i2, p2, 2, 0, msc(choux), [dh(undergraduate(choux), X)], true, X)$

The two mutually recursive top-level procedures *def* and *pos* corresponding to the two resolution strategies in Section 2 have been defined in the following way.

```
def(Goal,P,N):-
    copy_query(Goal,OldGoal),
    def_init((X,Goal),OldGoal,P,N),
    var(X).

pos(Goal,_,_):-
    reserve(Goal),
    call(Goal).

pos(not Goal,P,N):-
    not def(Goal,P,N),
    append_left(Goal,N).

pos((A,B),P,N):-
    pos(A,P,N),
    pos(B,P,N).
```

```
pos(Goal,P,N):-
    cl(_,_,Goal,Body),
    pos(Body,P,N).

pos(Goal,P,N):-
    generator(Goal,Gen),
    cl(IndRef,Posn,_,_,Gen,_,Body,_),
    not in_clause(defcl(IndRef,Posn),P),
    in_clause(NegL,N),
    link(NegL,H,[]),
    def(NegL,P,N),
    !,fail.

pos(Goal,P,N):-
    cl(IndRef,Posn,_,_,Goal,_,Body,_),
    append_left(defcl(IndRef,Posn),P),
    pos(Body,P,N).
```

where the procedure *copy_query*$(Goal, OldGoal)$ means $OldGoal$ is a copy of the query $Goal$ and *generator*$(Goal, Gen)$ means the generator of the goal $Goal$ is Gen. Also, *append_left* is a procedure which appends its first argument to its open-ended structure second argument and is defined as follows.

```
append_left(X,Y):-
    var(Y),!,
    Y=(_,X).
append_left(X,(_,X)):-!.
append_left(X,(Y,_)):-
    append_left(X,Y).
```

The following four procedures *def_init*, *def_solve*, *def_restart* and *def_restart_goal* are respectively the modified forms of *nH_init*, *nH_solve*, *nH_restart* and *nH_restart_goal* in [23].

```
def_init(AnsList,OldGoal,P,N):-
    AnsList = (_,Goal),
    def_solve(Goal,1,[anc(q,[])],_,[],DH,init,P,N),
    ( DH = [_|_]
      -> def_restart(AnsList,OldGoal,DH,P,N);
      true).
```

```
def_solve(Goal,_,_,_,DH,DH,_,_,_):-
    reserve(Goal),
    call(Goal).

def_solve(not Goal,_,_,_,DH,DH,_,P,N):-
    not pos(Goal,P,N).

def_solve((A,B),_,Anc,AH,DH1,DH3,Can,P,N):-
    def_solve(A,1,Anc,AH,DH1,DH2,Can,P,N),
    def_solve(B,1,Anc,AH,DH2,DH3,Can,P,N).

def_solve(Goal,_,_,Goal,DH,DH,yes,_,_).

def_solve(Goal,StartRef,Anc,AH,DH1,DH3,Cancel,P,N):-
    get_cl(IndRef,Posn,StartRef,NextRef,Goal,AuxHeads,
        Body,NewAnc),
    NewAnc = [anc(Goal,NextRef)|Anc],
    ( (var(IndRef) ; in_clause(defcl(IndRef,Posn),P))
      -> def_solve(Body,1,NewAnc,AH,DH1,DH3,Cancel,P,N);
      append(AuxHeads,DH1,DH2),
      def_solve(Body,1,NewAnc,AH,DH2,DH3,Cancel,P,N) ).

def_restart(AnsList,OldGoal,DH,P,N):-
    DH = [dh(AH,Anc)|DH1],
    member(anc(ResGoal,StartRef),[anc(not,_)|Anc]),
    def_restart_goal(AnsList,OldGoal,ResGoal,
                     StartRef,Anc,AH,DH1,DH2,Cancel,P,N),
    ( (DH2 = [_|_], Cancel == yes)
      -> def_restart(AnsList,OldGoal,DH2,P,N);
      (DH2 = [], Cancel == yes)
      -> true;
      fail).

def_restart_goal(_,_,not,_,_,AH,DH,DH,yes,_,N):-
    in_clause(AH,N).

def_restart_goal(_,_,Goal,StartRef,Anc,AH,DH1,DH2,
        Cancel,P,N):-
    def_solve(Goal,StartRef,Anc,AH,DH1,DH2,Cancel,P,N).

def_restart_goal(AnsList,OldGoal,q,_,Anc,AH,DH1,DH2,
        Cancel,P,N):-
    copy_query(OldGoal,Query),
```

```
append_left(Query,AnsList),
def_solve(Query,1,Anc,AH,DH1,DH2,Cancel,P,N).
```

The first seven parameters of the procedure *def_solve* are defined in the same way as in [23]. The last two parameters P and N are open-ended structures. Each element in P has the form

$$defcl(IndRef, Posn)$$

where the possible form of an indefinite clause identified by $IndRef$ with respect to the head at the position $Posn$ has been used for a possible resolution and the *def_solve* routine will consider those clauses as definite clauses. Each element in N is an atom representing a negative fact true in the database constructed dynamically.

In the new internal representation of clauses, a clause is retrieved with the help of the following routine *get_cl*

```
get_cl(_,_,0,_,_,_,_,_):-!,fail.   % No more clauses

get_cl(_,_,StartRef,NextRef,Goal,_,Body,_):-
                          % Retrieve a definite clause
    cl(StartRef,NextRef,Goal,Body).

get_cl(IndRef,Posn,StartRef,NextRef,Goal,
        AuxHeads,Body,NewAnc) :-     % Retrieve an
    cl(IndRef,Posn,StartRef,NextRef,Goal,
        AuxHeads,Body,NewAnc).       % ind clause

get_cl(IndRef,Posn,StartRef,NextRef,Goal,
        AuxHeads,Body,NewAnc):-      % Retrieve
    generator(Goal,Gen),             % next clause
    (cl(StartRef,Via,Gen,_);
         cl(_,_,StartRef,Via,Gen,_,_,_)),
    get_cl(IndRef,Posn,Via,NextRef,Goal,
         AuxHeads,Body,NewAnc).
```

5 Conclusion

This paper aims at generalising the approach to handling integrity constraints in deductive databases in two ways. Firstly, closed general formulae have been introduced to represent constraints involving aggregate operations and

Lloyd *et al.*'s simplification method has been extended for checking integrity in definite databases in the presence of such constraints. Secondly, a new definition of constraint satisfiability has been introduced in the case of indefinite databases and the path finding method has been generalised to cope with integrity checking in indefinite databases. An appropriate extension seems possible to maintain integrity in an environment where both the database is indefinite and the imposed constraints are beyond first-order.

The definition of constraint satisfiability in indefinite databases which has been proposed here may seem restricted in the sense that each constraint has to be a theorem of the completion of each of the possible form of the database. An alternative view, which is more relaxed, can be defined as follows. A set of constraints I is said to be satisfied by a database D, if there exists at least one possible form D_i of D such that each constraint in I is a theorem of $comp(D_i)$. While this will permit more flexibility in what is stored, it may produce other problems in terms of inconsistent possible forms unless these can be limited by a more general inference mechanism.

The path finding method for definite databases can also be extended for checking integrity in definite databases in the presence of aggregate constraints as follows. Suppose, at the time of calculating a path P, a fact A' (or a negative literal $\neg A'$) occurring in it, unifies with the key formula A of an aggregate atom $Count(A, n)$ occurring in a constraint Γ. Let θ be an mgu of A and A'. Then, $\Gamma\theta'$ is evaluated in the updated database, where θ' is a unifier obtained from θ by excluding any bindings to the local variables of $Count(A, n)$. For efficiency purpose, all θ's are stored to avoid redundant evaluation of constraints. Lloyd *et al.*'s simplification method can also be extended for checking integrity in indefinite databases, as in the case of definite databases, by calculating a set of partially instantiated atoms whose instances are likely to be deleted from different possible forms of the database.

References

[1] K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In Minker [20], pages 89–148.

[2] P. Asirelli, M. D. Santis, and M. Martelli. Integrity constraint in logic databases. *Journal of Logic Programming*, 3:221–232, 1985.

[3] F. Bry, H. Decker, and R. Manthey. A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. In *Proceedings of Extending Database Technology*, pages 488–505, Venice, 1988.

[4] K. L. Clark. Negation as failure. In H.Gallaire and J.Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

[5] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer International, 1984.

[6] S. K. Das. *Integrity Constraints in Deductive Databases.* PhD thesis, Department of Computer Science, Heriot Watt University, 1990.

[7] S. K. Das and M. H. Williams. A path finding method for checking integrity in deductive databases. *Data and Knowledge Engineering*, 4:223–244, 1988.

[8] S. K. Das and M. H. Williams. Integrity checking methods in deductive databases: A comparative evaluation. In M.H.Williams, editor, *Proceedings of the 7th British National Conference on Databases*, pages 85–116. Cambridge University Press, 1989.

[9] C. J. Date. *An Introduction to Database Systems*, volume 2. Addison Wesley, 1985.

[10] H. Decker. Integrity enforcements on deductive databases. In L.Kerschberg, editor, *Proceedings of the First International Conference on Expert Database Systems*, pages 271–285, Charleston, South Carolina, April 1986.

[11] E. B. Fernandez, R. C. Summers, and C. Wood. *Database Security and Integrity.* Addison Wesley, 1981.

[12] H. Gallaire, J. Minker, and J.-M. Nicolas. Logic and databases — a deductive approach. *ACM Computing Surveys*, 16(2):153–185, 1984.

[13] R. A. Kowalski and K. A. Bowen, editors. *Proceedings of the 5th International Conference and Symposium on Logic Programming*, Seattle, USA, August 1988.

[14] T.-W. Ling. Integrity constraint checking in deductive databases using the Prolog not-predicate. *Data and Knowledge Engineering*, 2:145–168, 1987.

[15] J. W. Lloyd. *Foundations of Logic Programming.* Symbolic Computation Series. Springer Verlag, 2nd, extended edition edition, 1984.

[16] J. W. Lloyd and R. W .Topor. Making Prolog more expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.

[17] J. W. Lloyd and R. W. Topor. A basis for deductive database systems. *Journal of Logic Programming*, 2(2):93–109, 1985.

[18] A. Lobo, J. Minker, and A. Rajasekhar. Weak completion theory for non-Horn logic programs. In Kowalski and Bowen [13].

[19] D. W. Loveland. Near-Horn Prolog. In E.Wada, editor, *Proceedings of the 4th International Conference on Logic Programming*, pages 456–469, Tokyo, Japan, July 1987. Springer Verlag.

[20] J. Minker, editor. *Foundation of Deductive Databases and Logic Programming.* Morgan Kaufman Publishers, Inc, 1988.

[21] J.-M. Nicolas. Logic for improving integrity checking in relational databases. *Acta Informatica*, 18:227–253, 1982.

[22] F. Sadri and R. A. Kowalski. An application of general purpose theorem-proving to database integrity. In Minker [20].

[23] B. T. Smith and D. W. Loveland. A simple near-Horn Prolog interpreter. In Kowalski and Bowen [13], pages 794–809.

[24] J. D. Ullman. *Principles of Database Systems.* Computer Science Press International, Inc, Maryland, USA, 2nd edition edition, 1984.

54. S.K.DAS and M.H.WILLIAMS, Integrity checking methods in deductive databases: a comparative evaluation, Proceedings of the Seventh British National Conference on Databases, Cambridge University Press, 85-116 (1989).

Integrity Checking Methods in Deductive Databases: A Comparative Evaluation

S K Das and M H Williams
Department of Computer Science, Heriot-Watt University, 79, Grassmarket, Edinburgh EH1 2HJ

The major objective of this paper is to evaluate the merits and demerits of a number of integrity checking methods for deductive databases that have been proposed by different authors in the last few years. To achieve this, a brief introduction to the methods is given and their performances compared against each other in the context of different databases. The different formats for representing constraints used by these different methods and the expressiveness of each is discussed. Two views of constraint satisfiability and the differences between them are discussed in detail.

1. Introduction

In a database an *integrity constraint* [8,11,27,28] (or simply a *constraint*) is a formal representation of a property which the data in the database must satisfy to be consistent with some model of the real world from which it comes. A constraint can specify either some property governing the correct states of the database or the allowable transitions from one database state into another. The former is referred to as a *static* constraint and the latter a *transition* constraint [22]. Some examples of static constraints (with their type in parentheses) are as follows:

every child has only one father (functional dependency),
the age of a person is less than 100 (range),
the sex of a person is either male or female (domain),
every worker has a superior (existential),

the number of all employees in the finance department is less than 100 (aggregate), etc.

An example of a transitional constraint is

when updating the age of an individual, the new value must exceed the old one.

This paper is concerned with static constraints. Hence any further references to the term 'constraint' will be taken to refer to 'static constraint'.

A *deductive database* [12,20,24] generalises the concept of a *relational database* [9,27]. A relational database can be viewed as a logic program consisting solely of ground unit clauses, whereas a deductive database is a logic program comprising a finite set of clauses of the form

$$A \leftarrow L_1,...,L_n \qquad n \geq 0,$$

where A is an atom denoting the conclusion and $L_1,...,L_n$ are literals representing conditions. Each L_i is either an atom (a positive condition) or a negated atom (a negative condition). Any variables in A, $L_1,...,L_n$ are assumed to be universally quantified over the whole clause. The commas in the antecedent $L_1,...,L_n$ denote conjunction. A is called the head and $L_1,...,L_n$ the body of the clause. When $n = 0$ the clause is called a *fact*, when $n \neq 0$ it is called a *rule*. Thus a relational database is a deductive database which contains only facts.

In a logic programming context, a constraint can be defined as a closed formula. Normally the type of constraint which is catered for by algorithms for integrity checking in deductive databases, is limited to first order formulae. This assumption has therefore been made here. Thus the last example from the above list of constraints, which normally requires a second order formula, has been excluded from further discussion. The remaining four examples of static constraints can be written as closed first order formulae as follows:

$\forall x \forall y\ (Father(x,z), Father(y,z) \rightarrow x = y)$
$\forall x \forall y\ (Age(x,y) \rightarrow y < 100)$
$\forall x \forall y\ (Sex(x,y) \rightarrow y = Male \vee y = Female)$
$\forall x\ (Worker(x) \rightarrow \exists y\ Superior(y,x))$

where x,y,z are variables and the symbols '$\rightarrow$' and '$\vee$' are respectively the logical 'implication' and logical 'or'. When writing a constraint, all universal quantifiers from the front end of the constraint can be omitted and thus for a constraint written in this way, all free variables occurring in the constraint will be assumed to be universally quantified at the beginning.

It is desirable to check the integrity of a database at the end of each transaction. The simplest approach to checking integrity in a database involves the evaluation of each constraint whenever the database is updated. However, such an approach is too costly, especially for large databases, and does not make use of the fact that the database satisfies the constraints prior to the update. Different ways of maintaining integrity in deductive databases have been proposed in several papers, e.g. generalisations of the simplification method [21] by Lloyd and Topor [20], Decker [10], and Bry et al. [3], a general theorem-proving technique by Sadri and Kowalski [24], using Prolog not-predicate by Ling [16], consistency proof method and a modified program method by Asirelli et al. [2], a path finding method by Das and Williams [7], etc.

In this paper an attempt has been made to produce a comparative evaluation of the main methods proposed in the last few years for checking integrity in deductive databases. The next section of the paper describes two views of constraint satisfiability. The first, which is called the *standard* (or *theoremhood*) view, is followed in [3,7,10,16,20] and the second, which is called the *consistency* view, is followed in [2,24]. The differences between these two views are discussed. Section 3 considers the different forms of constraint representation used by different methods and their expressiveness. Section 4 introduces several methods in brief and illustrates each with the same example to see their behaviour from the same view point. In the last section the methods have been compared against each other and different aspects of the comparison have been verified by practical tests which have been carried out in a C-Prolog [6] environment running on a High Level Hardware Orion computer.

2. Constraint Satisfiability

A *constraint* is a closed first order formula that a database is required to satisfy. The standard definition of constraint satisfiability (theoremhood view) is as follows. Let D be a database such that the *completion* of D [5] , denoted by *comp*(D), is consistent, where *comp*(D) is the collection of completed definitions of predicate symbols in D together with the axioms for equality. Then D is said to satisfy W, where W is a constraint, if W is a logical consequence of *comp*(D); otherwise, D violates W. D is said to satisfy I, where I is a set of constraints, if D satisfies each constraint in I; otherwise, D violates I. This definition is followed in [3,7,10,16,20].

An alternative definition of constraint satisfiability (consistency view) adopted in [2,24] is as follows. A database D satisfies constraints I if and only if the set of formulae comprising the completion of D together with the formulae in I, is consistent.

The above two definitions of constraint satisfiability are equivalent if and only if the completion of the database D is *complete* (A theory T is said to be

complete if for any closed formula W in the language of the theory, either W or its negation is a logical consequence of T). If a constraint W is a theorem of the completion of a database D then *comp*(D) together with W is always consistent. But the converse may not be true. Hence the latter definition of constraint satisfiability deals with a larger class of databases.

The alternative definition of constraint satisfiability may differ from the standard view in its outcome if either of the following two conditions is met:

(a) circular definitions are present in the database, or
(b) a predicate symbol occurs in the constraints but not in the database.

Each of the above points will be clarified with the help of examples.

To illustrate the problem associated with (a), consider the database

D: $P(a) \leftarrow P(a)$

where a is a constant, and the constraint

I: $P(a)$.

The database is recursive and its completion, apart from the equality axiom, is given by

comp(D): $P(x) \leftrightarrow x = a \vee P(a)$,

where x is a variable. *Comp*(D) $\cup$ I is consistent as $\{P(a)\}$ is a model of *comp*(D) $\cup$ I. On the other hand $P(a)$ is not a theorem of *comp*(D).

To clarify point (b), consider the database

D: $P(a)$,

where a is a constant, and a constraint

I: $Q(a)$.

The predicate symbol Q does not occur in the database. The completion of D, apart from the equality axiom, is given by

comp(D): $P(x) \leftrightarrow x = a$

where x is a variable. *Comp*(D) $\cup$ I has a model $\{P(a),Q(a)\}$ and hence *comp*(D) $\cup$ I is consistent. But $Q(a)$ cannot be proved as a theorem from

comp(D).

To implement reasoning with the completion of the database, the methods in [7,10,15,20] make use of the notion of negation as failure [5] proposed by Clark.

Another view of constraint satisfiability, called the *meta* view, has recently been proposed by Kowalski [14]. In this view a constraint is a meta statement which must be true of the database. None of the methods considered in this paper exploits this view of constraint satisfaction.

3. Constraint Representation

This section discusses different ways of representing constraints used by different methods and the expressiveness of each.

The format used for expressing constraints varies according to the method studied. Formats used include closed typed first order formulae, closed first order formulae with restricted quantification, range form, IC-formulae, denials, etc. Each of the expressions is described in turn.

In [20], Lloyd et al. are concerned with a typed database [20,23] and the format which they have adopted for representing a constraint is an arbitrary closed typed first order formula. The requirement that the database is typed ensures that important kinds of integrity constraints are maintained automatically. A formula F is said to be *domain-independent* if and only if its truth value does not depend on any domain element other than those occurring in the relations that are explicitly mentioned in F. A formula is *domain-dependent* if and only if it is not domain-independent. A domain-dependent constraint, such as $\forall x / \tau\, p(x)$, where x is a variable of type τ, can be checked by replacing the formula by $\forall x\ (p(x) \leftarrow \tau(x))$ as a constraint. This means that whenever a new constant a is entered into the domain of type τ, a fact $\tau(a)$ is maintained in the database and according to the constraint a fact $p(a)$ will be inserted into the database. Constraints are converted into a type free formula and evaluated in a type free database. Consider the constraint

$$\forall x\ (Worker(x) \rightarrow \exists y\ Superior(y,x)) \qquad (*)$$

The typed form of this constraint might be expressed as follows :

$$\forall x / Name(Worker(x) \rightarrow \exists y / Name\ Superior(y,x))$$

where *Name* is a unary type predicate. The corresponding type free form is given by

$$\forall x\ (Name(x) \rightarrow (Worker(x) \rightarrow \exists y\ (Name(y) \wedge Superior(y,x))))$$

In [3], Bry et al. have defined constraints as function-free, closed first order

formulae with restricted quantification which are expressed in normalised form [3]. In other words a quantified formula or sub-formula has one of the forms

$$\exists x_1 ... x_n \, [A_1 \wedge ... \wedge A_m \wedge Q]$$
$$\forall x_1 ... x_n \, [Not\, A_1 \vee ... \vee Not\, A_m \vee Q]$$

where the symbol '$\wedge$' is logical 'and', $A_1,..., A_m$ are atoms such that every variable x_i occurs in at least one A_j, and Q is either **true** or **false**, or some formula in which some or all x_i are free. In this notation, the constraint (*) can be written as

$$\forall x \, [Not\, Worker(x) \vee \exists y \, Superior(y,x)]$$

where A_1 is $Worker(x)$ and Q *is* $\exists y \, Superior(y,x)$. The variable x is free in Q.

Decker [10] has considered a subset of domain-independent formulae, called *range-restricted* formulae, for expressing constraints. A formula F in *disjunctive minimised form* [10] (*dmf*) is *range-restricted* with respect to a variable x if and only if the variable x is existentially quantified in the formula, and there is at least one non-negated x-literal in every conjunction in F with an occurrence of x; or if the formula is universally quantified by the variable x, and *dmf*(*Not* F) is range-restricted with respect to x. F is *range-restricted* if and only if it is range restricted with respect to all the quantified variables occurring in F. This class of range-restricted formulae is then represented in their equivalent *range form* [10]. A range form has a range expression, conclusion and remainder. The range form of (*) is given by

$$\forall x \, (Worker(x) \rightarrow Superior(y,x)),$$

where $Worker(x)$ is the range expression and $Superior(y,x)$ is the conclusion. The remainder is empty in this case.

The constraint checking method put forward by Kowalski et al. [15,24] deals with constraints of the form

$$\leftarrow L_1,...,L_n, \quad n > 0,$$

where the L_i are literals and all variables are assumed to be universally quantified in front of the constraint. These formulae are called *denials*. Denials must also be range-restricted. The method does not deal with constraints like $\forall x \, (p(x))$ since it is not possible to convert this kind of constraint to a denial form which is range-restricted. It is also possible to deal with constraints that are in a more general form than denials with the help of ten transformations given in [19] and the process has been described in [24]. For example, the

constraint (*) can be transformed to a denial of the form

$$\leftarrow A$$

with the addition of the rules

$$A \leftarrow Worker(x), Not\ B(x)$$
$$B(x) \leftarrow Superior(y,x)$$

to the database, where the two predicates A and B are not used elsewhere in the database.

The path finding method [7] and the method by Kowalski et al. both deal with the same class of constraint. In the path finding method a constraint is represented in the form of the clause

$$IC(Id) \leftarrow D$$

where $\leftarrow D$ is a constraint in denial form and Id is a unique identification of the constraint. The formula $\leftarrow D$ has to be *allowed* [17] (a formula F is allowed if and only if it is range restricted). $IC(Id)$ is called the *head* of the constraint. Hence, the constraint (*) in this form will be represented by the clause

$$IC(Id) \leftarrow A,$$

together with the two definitions of A and B required for the case of Kowalski et al.

Ling [16] introduces a special form of closed first order formula, called an IC-formula, which uses normal form (NF) negative formulae, to express constraints. An NF negative formula $Not(P)$ has the form

$$Not(A_1,...,A_m, Not(B_1),..., Not(B_n)),$$

where each A_i is either a positive literal or an evaluable predicate and each $Not(B_j)$ is an NF negative formula. Using this, a constraint formula or IC-formula has been defined by Ling to be a closed formula of the following form :

$$A_1,...,A_n, Not(B_1),..., Not(B_m) \rightarrow C_1,...,C_p, \qquad n,m,p \geq 0$$

where each C_i has the same form as B_j. The constraint (*) is thus already in the form of an IC-formula, where A_1 is $Worker(x)$, m is zero, and C_1 is $Superior(y,x)$

Asirelli et al.'s modified program method deals with constraints of the form

$p(x_1,...,x_n) \rightarrow \Psi(x_1,...,x_n)$

such that the left hand side has only distinct variables. A formula with terms different from variables on the left hand side can also be transformed to the above form and has been discussed by the authors in their paper [2]. This form of formula has the problem of representing constraints which involve negative conditions and is therefore more limited than the forms which can be handled by the other methods. The constraint (*) can be written in the above form as

$Worker(x) \rightarrow \psi(x)$

by adding to the database the following definition of ψ

$\psi(x) \leftarrow Superior(y,x)$

4. Constraint Checking Methods

In this section a number of methods for checking integrity in deductive databases are described briefly and the operation of each of the methods is expressed with the help of the following example.

Example 1

Database D1 :

Relations :

Mother(x,y) : x is the mother of y.
Father(x,y) : x is the father of y.
Parent(x,y) : x is a parent of y.
Ancestor(x,y) : x is an ancestor of y.
Dependent(x,y) : x depends on y.
Husband(x,y) : x is the husband of y.
Wife(x,y) : x is the wife of y.
Married(x,y) : x is married to y.
Occupation(x,y) : occupation of x is y.
Student(x) : x is a student.
Employed(x) : x is employed.
Guardian(x,y) : x is a guardian of y.
Self(x) : x is self-sufficient.
Sponsor(x,y) : x sponsors y.

Rules :

*R*1. *Mother(x,y)* ← *Father(z,y), Husband(z,x)*
*R*2. *Parent(x,y)* ← *Father(x,y)*
*R*3. *Parent(x,y)* ← *Mother(x,y)*
*R*4. *Ancestor(x,y)* ← *Parent(x,y)*
*R*5. *Ancestor(x,y)* ← *Parent(x,z), Ancestor(z,y)*
*R*6. *Wife(x,y)* ← *Husband(y,x)*
*R*7. *Married(x,y)* ← *Husband(x,y)*
*R*8. *Married(x,y)* ← *Wife(x,y)*
*R*9. *Employed(x)* ← *Occupation(x,Service)*
*R*10. *Student(x)* ← *Occupation(x,Student)*
*R*11. *Dependent(x,y)* ← *Parent(y,x), Employed(y), Student(x)*
*R*12. *Dependent(x,y)* ← *Married(y,x), Employed(y), Not Employed(x)*
*R*13. *Self(x)* ← *Married(y,x), Not Employed(y)*
*R*14. *Guardian(x,y)* ← *Dependent(y,x)*

Facts :

*F*1. *Occupation(2,Service)*
*F*2. *Occupation(3,Student)*
*F*3. *Husband(1,2)*
*F*4. *Father(1,3)*
*F*5. *Sponsor(2,3)*
*F*6. *Sponsor(2,1)*

Integrity Constraints :

*IC*1. *Guardian(x,y)* → *Sponsor(x,y)*
*IC*2. *Married(x,y)* → *Not Student(x)*

Update :

insertfact *Occupation(1,Service).*

where *x*, *y* and *z* are variables, and *Student* and *Service* are constants.

The integrity checking methods proposed thus far can be classified into the following categories on the basis of their internal mechanism :

1. Each constraint is evaluated whenever the database is updated (this method will be referred to as Simple).

2. The constraint set is simplified using induced updates (the set of facts which are implicitly or explicitly added to or deleted from a database due to an update to the database) and only the relevant constraints are evaluated. This idea of simplification of constraints was introduced by Nicolas [21] for checking integrity in relational databases. The methods in [3,7,10,20] are based on this idea of simplification of constraints, and they differ in the way these induced updates are computed and the constraints are evaluated.
3. A generalised proof procedure is used to evaluate the constraints at the end of a transaction. The method described in [15] is based on a proof procedure which generalises the concept of SLDNF-resolution by reasoning forward from an update. The proof procedure of [25] is used to evaluate constraints in the context of extended deductive databases (extended with default rules).
4. Derivation rules are modified to produce a modified database. This technique has been followed in [2,13].
5. Constraints are evaluated only against the fact base by transforming each constraint into a form which is defined over the base facts only. This idea was first introduced in [4] in the context of relational databases and later in [16] in the context of deductive databases.

4.1. Method proposed by Lloyd et al.

The simplification method proposed by Lloyd et al. in [20] is based on a standard view of constraint satisfiability. The correctness of the method was first proved for the class of definite databases in [20] and later for the class of stratified databases [1] in [18]. An important task of the simplification method is to capture the difference between a model for *comp*(D′) and a model for *comp*(D), where D and D′ are databases and D′ is obtained from D by the application of a transaction T on D. By using only rules of the database and updates, the method computes in stages the two sets of partially instantiated atoms $pos_{D,D'}$ and $neg_{D,D'}$. These two sets of atoms represent respectively the part that is added to the model for *comp*(D) when passing from D to D′ due to a transaction and the part that is deleted. To preserve the consistency of the updated database, constraints are instantiated appropriately with the two sets of atoms $pos_{D,D'}$ and $neg_{D,D'}$, and only affected constraints are evaluated. Instantiated constraints are evaluated using the SLDNF proof procedure.

To apply the algorithm to example 1, the following two sets of atoms $pos_{D,D'}$ and $neg_{D,D'}$ are determined.

$$pos_{D,D'} = \{Occupation(1, Service), Employed(1), Dependent(x,1), Guardian(1,x)\}$$
$$neg_{D,D'} = \{Self(x), Dependent(1,y)\}.$$

The formulae for deriving the above two sets of atoms is inductively defined as follows :

$pos^0_{D,D'} = \{A : A \leftarrow W \in D'/D\}$
$neg^0_{D,D'} = \{\}$

$pos^{n+1}_{D,D'} = \{A\theta : A \leftarrow W \in D',$ B is a positive condition of $A \leftarrow W, C \in pos^n_{D,D'},$ and θ is an mgu of B and C$\}$
$\cup \{A\theta : A \leftarrow W \in D',$ B is a negative condition of $A \leftarrow W, C \in neg^n_{D,D'},$ and θ is an mgu of B and C$\}$

$neg^{n+1}_{D,D'} = \{A\theta : A \leftarrow W \in D',$ B is a positive condition of $A \leftarrow W, C \in neg^n_{D,D'},$ and θ is an mgu of B and C$\}$
$\cup \{A\theta : A \leftarrow W \in D',$ B is a negative condition of $A \leftarrow W, C \in pos^n_{D,D'},$ and θ is an mgu of B and C$\}$

$pos_{D,D'} = \cup_{n \geq 0} pos^n_{D,D'}$
$neg_{D,D'} = \cup_{n \geq 0} neg^n_{D,D'}$

The following shows the different stages of determining the sets $pos_{D,D'}$ and $neg_{D,D'}$.

$pos^0_{D,D'} = \{Occupation(1,Service)\}$
$neg^0_{D,D'} = \{\}$

$pos^1_{D,D'} = \{Employed(1)\}$
$neg^1_{D,D'} = \{\}$

$pos^2_{D,D'} = \{Dependent(x,1)\}, Dependent(x,1)\}$
$neg^2_{D,D'} = \{Dependent(1,y), Self(x)\}$

$pos^3_{D,D'} = \{Guardian(1,x)\}$
$neg^3_{D,D'} = \{\}$

$pos^4_{D,D'} = \{\}$
$neg^4_{D,D'} = \{\}$

At this point, no more new atoms can be generated. In $pos^2_{D,D'}$, the second atom *Dependent(x,1)* is an instance of the first one and can be excluded while computing the atoms of the next stage. The number of stages for calculating the two sets of atoms is always finite, even in the presence of recursive rules, as at any stage a generated atom is excluded if it is an instance of an atom which is already generated.

Using the two sets of atoms $pos_{D,D'}$ and $neg_{D,D'}$, constraints are instantiated appropriately. The only constraint affected is IC1 and an instance of this that has to be evaluated in the database is the formula

$$Guardian(1,x) \rightarrow Sponsor(1,x).$$

This is evaluated using the SLDNF [5] proof procedure in the updated database. The above instance is negated and chosen as a top clause. Fig 1 shows the partial search space in which at each step a leftmost literal is selected from a goal under safe literal selection strategy. In Fig 2, the tree diagram illustrates the process of deriving the above two sets of atoms by a depth-first approach. All positive literals belong to the set $pos_{D,D'}$ and all negative literals belong to the set $neg_{D,D'}$.

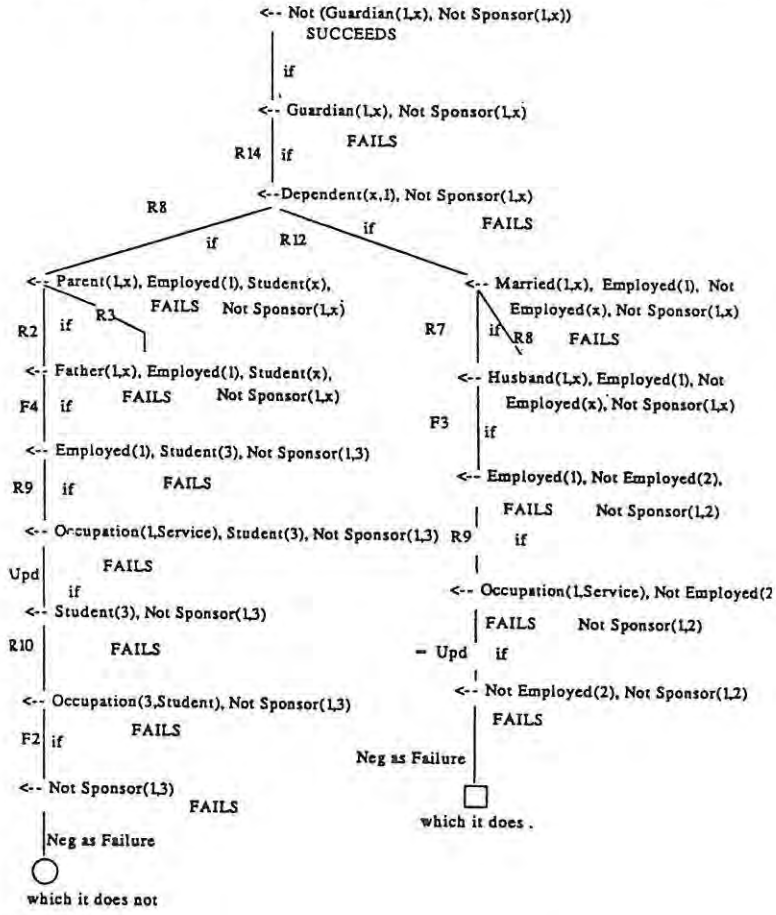


Fig 1. Partial search space for evaluating the constraint in Lloyd et al.'s method by SLDNF-proof procedure

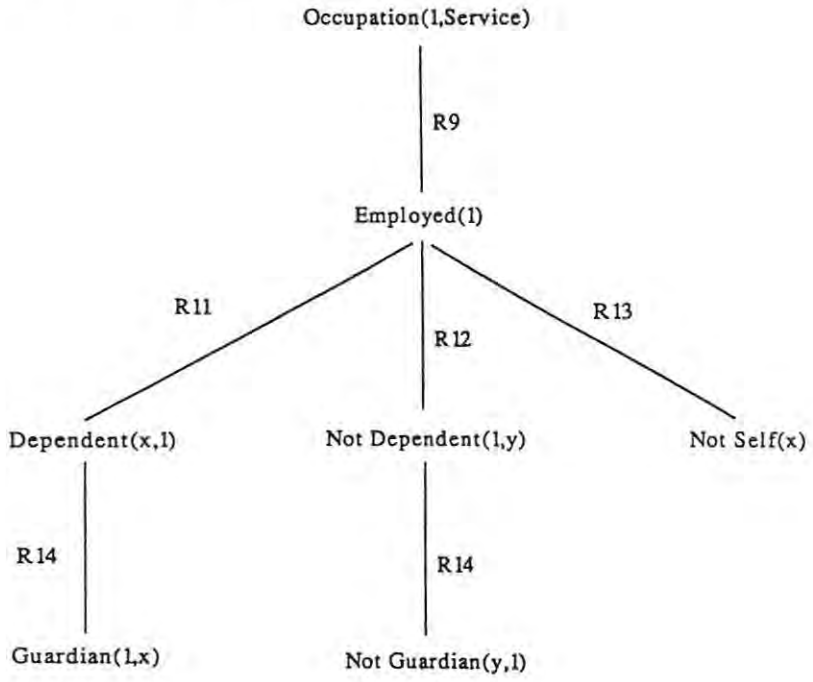


Fig 2. Illustration of the process of deriving the two sets of partially instantiated atoms $pos_{D,D'}$ and $neg_{D,D'}$ in Lloyd et al.'s method.

4.2. Method proposed by Decker

Like the previous method, the method proposed by Decker in [10] is also based on a standard view of constraint satisfiability. The method is a generalisation of the simplification method put forward by Nicolas [21] for relational databases. Decker's method simplifies the constraints with induced updates and only affected constraints are evaluated. Each constraint F is represented in the database as a set uc(F) of *update constraints*, each of which is either an *insert constraint* or a *delete constraint.* An insert constraint has the form

insert UL only if UC

and a delete constraint has the form

delete UL only if UC,

where *UL* is an *update literal* and UC is an *update condition* which is in range form. When a transaction T is performed on a database D the set of all update literals due to T are divided into two sets of ground atoms, D^T and D_T, defined as follows. Let DT denote the set of clauses obtained after execution of the transaction T on the database D. Then, $D^T = DT^*/D^*$ and $D_T = D^*/DT^*$, where / denotes the set difference and D^* denotes the set of facts derivable from D. Each element of D^T has been called an *include-fact* and each element in D_T has been called a *remove-fact*. Intuitively, D^T (D_T) is the set of facts added to (deleted from) the database due to the transaction.

Decker has proposed an algorithm which will evaluate all relevant simplified constraints defined above for each fact in D^T and D_T. The elements of D^T and D_T are computed in stages for each unit transaction of T. The algorithm is called with argument *include(C)* (*remove(C)*) for each element *include(C)* (*remove(C)*) in T. The three-step algorithm essentially does the following. In the first step the relevant insert (delete) constraints simplified by each include (remove) fact of D^C (D_C) are evaluated, where D^C (D_C) is the set of facts added to (deleted from) the database by the clause C itself. In fact, the set of all include (remove) facts D^T (D_T) is the union of all D^C (D_C), where C is in DT (D). If a simplified constraint is falsified then an inconsistency occurs and the algorithm stops. In the second and third stages, the algorithm is called recursively with arguments of the form *include(R)* or *remove(R)*, where R is an instantiated rule in D which could possibly add to or delete some facts from the database D due to the facts of D^C or D_C.

To apply the algorithm to example 1, the following two sets of facts are determined.

The facts added to the database as a result of the update are

D^T = {*Occupation*(1,*Service*), *Employed*(1), *Dependent*(3,1), *Guardian*(1,3)}.

The facts deleted from the database as a result of the update are

D_T = {*Self*(2)}.

The following shows the different stages of determining the sets D^T and D_T. D^T (D_T) is the union of all D^{Ti} (D_{Ti}).

Initial Stage :

D^{T0} = {*Occupation*(1,*Service*)}

D_{T0} = { }

No constraint is affected by an atom in D^{T0} or D_{T0}.

First Stage :

D^{T1} = {*Employed*(1)}

D_{T1} = {*Self*(2)}

No constraint is affected by an atom in D^{T1} or D_{T1}.

Second Stage :
$D^{T2} = \{Dependent(3,1)\}$
$D_{T2} = \{\}$
No constraint is affected by an atom in D^{T2} or D_{T2}.

Third Stage :
$D^{T3} = \{Guardian(1,3)\}$.

At this stage the fact *Guardian*(1,3) in D^{T3} matches a condition of the first constraint. Therefore the simplified constraint *Sponsor*(1,3) has to be evaluated in the updated database. Clearly this is not evaluable in the updated database and hence the integrity is violated.

The tree diagram in Fig 3 illustrates the process of deriving the above two sets of atoms and evaluation of constraints. All positive literals belong to D^T and all negative literals belong to the set D_T.

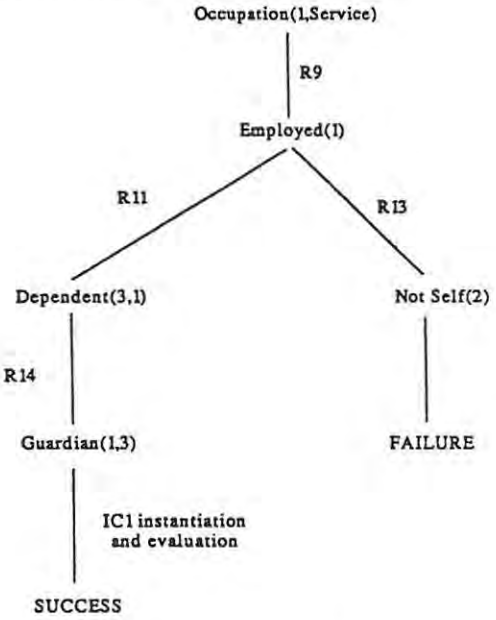


Fig 3. Illustration of the process of deriving the two sets of ground atoms D^T and D_T in Decker's method

4.3. Method proposed by Kowalski et al.

The consistency method which was introduced by Sadri and Kowalski in [24]

and later formalised in [15] by Kowalski et al., is based on the consistency view of constraint satisfaction. The method is essentially an attempt to construct a *refutation tree* (a refutation tree is a search space in which at least one path ends at the empty clause) taking each of the updates in turn as a candidate top clause. If all possible attempts fail to do so then integrity is preserved in the updated database. The method uses a proof procedure which is an extension of the SLDNF proof procedure and provides both forward and backward reasoning. In SLDNF the clause can only be a denial and the set of formulae which appear in the derivation are either denials or the empty clause. But to reason forward, the new proof procedure allows as top clause any clause, denial or negated atom and the set of formulae appearing in the derivation may be any of these, the empty clause or a formula of the form

$$Not(A) \leftarrow L_1,...,L_n \qquad n \geq 1,$$

where A is an atom and the L_i are literals. The method defines some metalevel rules for reasoning about implicit deletions and a generalised resolution step for reasoning forward from negation as failure.

In the formalisation of the proof procedure a relation *Refute(s,c)* has been defined which means that there is a refutation with s as input set and c as top clause. The relation has been defined by five different rules. The first three formalise the process of resolution and negation as failure [5]. The last two cater for the cases of implicit deletions resulting from additions and deletions.

To apply the algorithm to example 1, the update has been taken as a candidate top clause and a refutation tree is constructed. Part of the search space for this is shown in Fig 4. According to the order of clauses and constraints of database D1, only the leftmost branch will be followed. The rightmost branch of the search space shown in the figure implies an implicit deletion of the fact *Self(2)* from the database. The method selects the leftmost literal through a safe literal selection strategy. For ground negated atoms, the negation as failure rule has been applied. A clause number, labelling an arc, is used for resolution purposes with the selected literal.

4.4. *Method proposed by Bry et al.*

The integrity checking method proposed in [3] by Bry et al. can be divided into two phases. The first phase, called the *preparatory phase*, determines a set of potential updates which represent possible ground updates induced by an update to the database and then generates expressions called *update constraints* from potential updates and constraints. The set of potential updates is determined in the same way as for the method described in section 4.1. An update constraint is generated for every potential update which is relevant to a constraint. An update constraint has the form

Update_Constraint(L,(Not Delta(U,L) Or New(U,SI))),

where *Delta(U,L)* holds if and only if L is satisfied in $U(D)$ (U denotes a ground single fact update to a database D and $U(D)$ denotes the updated database) and *New(U,F)* denotes the evaluation of the formula F over the updated database $U(D)$. Another phase of the method, called *evaluation phase*, evaluates all the update constraints generated by the preparatory phase.

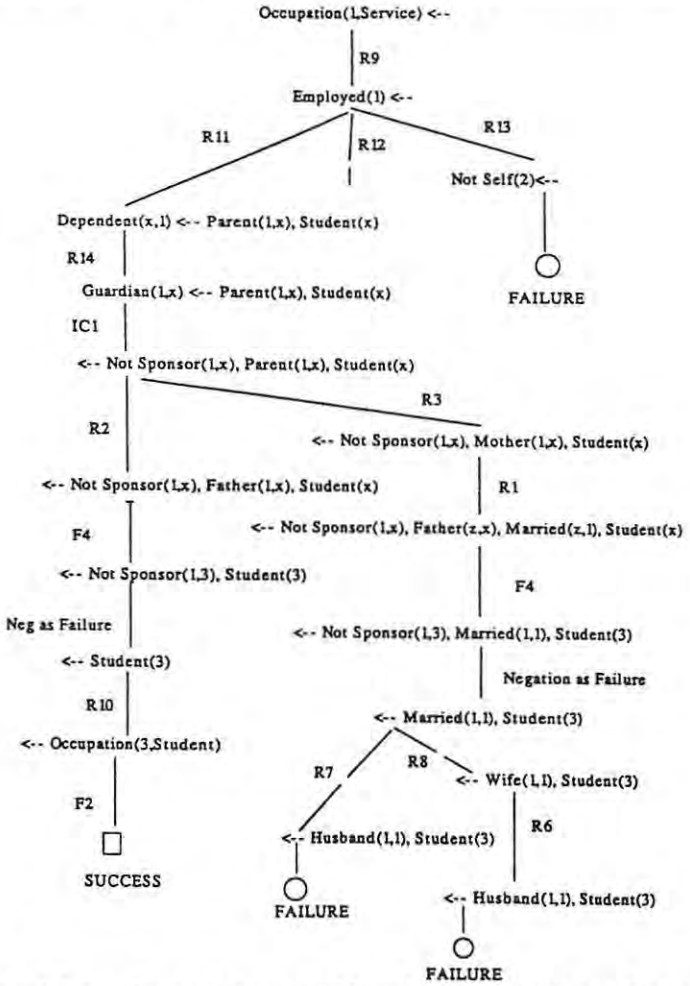


Fig 4. Partial search space generated by taking the update as the top clause in Kowalski et al.'s method

Applying the method to example 1, in the first phase the set of potential updates is determined in the same way as in section 4.1 and consists of the union

of the set of all atoms of $pos_{D,D'}$ and the set of all negated atoms obtained by negating each of the atoms of $neg_{D,D'}$. Also, in this phase the set of queries

$$\{(Not\ Delta(Occupation(1,Service),Occupation(1,Service)) \vee New(Occupation(1,Service),(Not\ Guardian(1,y) \vee Sponsor(1,y)))))\}$$

is obtained from generated update constraints. In the second phase the above set of queries is evaluated and in this case inconsistency will be detected.

4.5. *Method proposed by Asirelli et al.*

Asirelli et al. [2] have proposed two methods . The first, called the *consistency proof method*, is suitable for proving consistency of an existing database but is not very efficient for checking consistency after each transaction. The second method, which will be referred to as the modified program method, handles constraints with respect to a database (or program) D by finding a database, called the modified database (*mod*(D)), whose minimal model is exactly the subset of the minimal model of D which satisfies the constraints. If S is the set of atoms satisfying a set of constraints IC, then it can be proved that the minimal model of *mod*(D) is a subset of S.

To apply the method to example 1, the constraint IC2 has been excluded from the constraint set as the method cannot deal with this constraint. The modified program for the database D1 and constraints IC1 of example 1 can be constructed as follows. First add to the database D1 the following definitions:

$$\Psi(u,v) \leftarrow Not_Unify([u,v],[x,y])$$
$$\Psi(x,y) \leftarrow Sponsor(x,y)$$

where u,v are variables. Then IC1 can be rewritten as

$$Guardian(x,y) \rightarrow \Psi(x,y).$$

Thus the final database, ie the modified version of the database is given by

mod(D1): *R*1-*R*13
$Guardian(x,y) \leftarrow Dependent(y,x), \Psi(x,y)$

together with the above definition of Ψ. *Mod*(D1) is an internal form which generates correct answers to queries of D1, ie *mod*(D1) is equivalent to D1 constrained by IC1.

4.6. *Method proposed by Ling*

Given a constraint *Not(P)* which contains some views (a view is a positive

literal which appears as the head of some clause), the method constructs a set of NF negative formulae *EXTRA(Not (P))* using rules of the database. The two-step algorithm for constructing this set from a given constraint *Not(P)* operates as follows. The algorithm is first called with the set consisting of the constraint *Not(P)*. The first step of the algorithm replaces a negative formula containing a view by a set of negative formulae with the help of the definitions of the view in the underlying database. The second step of the algorithm invokes the whole algorithm itself recursively with the new set of negative formulae, if there is a view occurring in one of the negative formulae of this set; otherwise, it stops. When an update is performed, the method evaluates only the affected constraints from these sets.

The construction of the set of NF negative formulae corresponding to a constraint may go into an infinite loop in presence of recursive rules. However, in the case of example 5 this does not create a problem and the NF negative formulae which represent constraints in this example are the following:

P:Not(Guardian(x,y),Not Sponsor(x,y)),
Q:Not(Married(x,y),Student(x)).

The following two sets can be constructed with the help of deductive rules.

EXTRA(Not(P)) = {Not(Father(x,y), Occupation(x,Service),
Occupation(y,Student), Not Sponsor(x,y)),
Not(Father(z,y), Husband(z,x), Occupation(x,Service),
Occupation(y,Student), Not Sponsor(x,y)),
Not(Husband(x,y),Occupation(x,Service),
Not Occupation(y,Service), Not Sponsor(x,y))},
Not(Husband(y,x),Occupation(x,Service),
Not Occupation(y,Service), Not Sponsor(x,y))},
EXTRA(Not(Q)) ={ Not(Husband(x,y), Occupation(x,Student)),
Not(Husband(y,x), Occupation(x,Student))}.

As a result of insertion of the fact *Occupation(1,Service)*, incremental checking is required for the NF negative formulas from the set *EXTRA(Not(P))*. They are unified appropriately by the update literal and simplified. Since the simplified form of the first one is not true in the database, integrity is violated due to the update.

4.7. Path finding method

The method proposed by Das and Williams [7] is called the *path finding method*. In this method verification of the constraints in the updated database is reduced to the process of constructing paths from update literals to the heads of constraints. An update literal could be either a fact which is to be

added to a database or the negation of a fact which is to be deleted from a database; alternatively it could be an instance of a rule R which is added to (negation of the head of a rule R whose instances are likely to be deleted from) a database, if the rule R is added to (deleted from) the database. If such a path is found then the integrity in the updated database is violated; otherwise, if all possible attempts to find a path fail, the integrity is said to be preserved in the updated database. The method reasons forward from an update until it reaches the head of a constraint. The method derives in stages a set of fully instantiated atoms added to the database and a set of partially instantiated atoms likely to be deleted from the database. Starting from an update literal S the next literal of the path could be a fact which has been implicitly added to the database due to S or the negation of an atom whose instances are likely to be implicitly deleted from the database due to S.

To apply the path finding method to example 1, the update literal *Occupation(1,Service)* is taken as a source. Because of the insertion of this fact into the database, with the help of R9 one can say that the fact *Employed(1)* is implicitly added to the database and so *Employed(1)* is the next literal of the path. Continuing in this way one can construct the following success path (which ends at the head of a constraint)

$$Occupation(1,Service) \rightarrow Employed(1) \rightarrow Dependent(3,1) \rightarrow Guardian(1,3) \rightarrow IC(1)$$

and the complete path space generated by the insertion is shown in Fig 5. The leftmost branch corresponds to the success path and this will be followed according to the order of the clauses. The successor literal of *Employed(1)* along the second branch of the path space is *Not Dependent(1,y)*, by rule R12. This shows that instances of *Dependent(1,y)* are likely to be deleted from the database due to the the implicit addition of *Employed(1)* to the database.

5. Comparison

In this section, methods are compared from both theoretical and practical points of view. To achieve the latter, the Simple method and the methods proposed in sections 4.1, 4.2, 4.3 and 4.7 have been implemented in a simple manner without going into much detail (e.g., if a constraint is violated then the implementation simply reports this fact without showing any detail of the cause of violation). Furthermore, the comparison is limited to those types of database defined in the introductory section. The following paragraph discusses briefly the implementation issue. The other methods have been excluded for various reasons. For example, the reason for excluding Asirelli et al.'s method is its limited constraint expressiveness. Ling's method is excluded because it cannot cope with recursive databases. The reason for excluding Bry et al.'s method is its close similarity with Lloyd et al.'s method with the result

that its performance figures will be similar to those of Lloyd et al. in most cases considered in this paper.

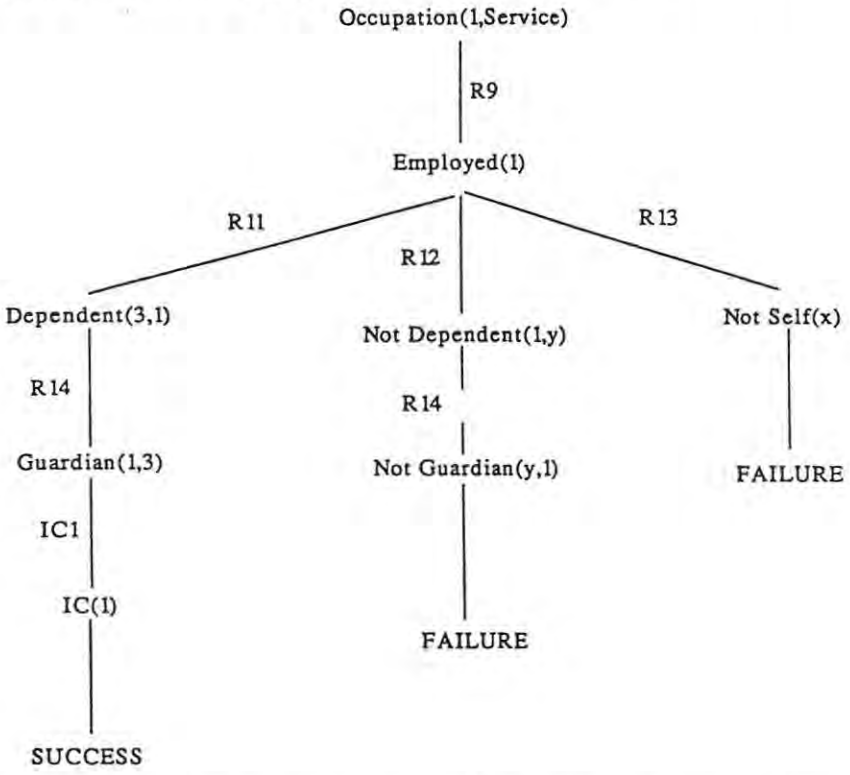


Fig 5. The complete path space traversed by the path finding method, taking the update as the source

In the implementation of each of the methods, facts have been stored directly as prolog facts. Rules are stored directly as prolog rules except in the method by Kowalski et al. In this method rules (and also constraints) have been stored under a ternary predicate *cl* as *cl(Ref,Type,Clause)*, where *Ref* is an unique identification of the clause (rule or constraint), *Type* is either a constraint or a rule, and *Clause* is a list of structures where each structure has the form *(Side,Sign,Atom)* defined as follows. *Side* is either condition or conclusion, *Sign* is either positive or negative and *Atom* is an atom occurring either in the head or the body of the clause identified by *Ref*. Furthermore, a separate clause of the form depend(L,H←*B*) for Decker's method, *depend(H,L)* for Lloyd et al.'s method, *depend(Ref,H,L)* for the path finding method and *depend(Ref,I,S)* for Kowalski et al.'s method has been maintained, where L is a literal occurring in the body of the rule H ← *B* identified

by *Ref* and *S* is a structure of the form defined above corresponding to the literal occurring in the I-th position of the clause H← . Constraints are not distinguished from rules for their representation in both Kowalski et al.'s method and the path finding method. In Decker's method a constraint has been converted to a set of update constraints and each update constraint is stored under the predicate *update_cons*. In Lloyd et al.'s method each constraint has been stored under a separate predicate *ic*. Transactions are stored under the predicate *trans* in an appropriate format to reason with two database states, ie before and after the transaction.

The simplest way of maintaining consistency (ie the Simple method) is by evaluating every constraint for each update; however, this is very inefficient since it ignores the assumption that the database satisfies the constraints prior to the update. All the implemented methods do take advantage of this assumption.To demonstrate the difference which this makes, consider the following example (also, example 5 later in this section).

Example 2

Database D2 :

Rules :

as described in database D1.

Facts :

A list of following 1095 facts not involving constants *1,2,3*:
175 facts under the predicate *Father*,
228 facts under the predicate *Husband*,
620 facts under the predicate *Occupation*,
72 facts under the predicate *Sponsor*.
A list of 1085 facts of example 3.
A list of 1058 facts of example 4.
Occupation(1,Service)
Occupation(2,Service)
Occupation(3,Student)
Father(1,3)
Sponsor(1,3)

Integrity Constraints I2 :

Two constraints of example 1.
$Occupation(x,y), Occupation(x,z) \rightarrow y = z$

Update :

transaction {insertfact *Husband*(1,2), insertfact *Sponsor*(2,3)}.

Database D2 satisfies I2 before the update. Each entry in Table 1 denotes a method and the time taken by that method to report inconsistency due to update. It is observed that the time taken by the simple SLDNF approach is much greater than that for any of the other methods because it does not take advantage of the assumption that the database D2 satisfies I2 prior to the update. For example, it evaluates the last constraint in the updated database even though this is redundant as it is not affected by any of the induced updates.

Method	Time
Simple	336.1
Lloyd et al.	10.0
Decker	31.5(18.9)
Kowalski et al.	133.7(1.1)
Path finding	18.4

Table 1.Times taken (in cpu seconds) by different methods to report inconsistency in the case of example 2

A disadvantage of Kowalski et al.'s method is that the degree of *granularity* [26] required for the meta-interpreter is much finer than that required for the other implemented methods. This is due to the fact that the former models the literal selection strategy. This should result in a loss of efficiency. On the other hand the other implemented methods rely on the strategy built into Prolog. For this reason, in some cases the time taken by Kowalski et al.'s method to report inconsistency is considerably larger than the time taken by any of the other methods for the same transaction. The literal selection strategy in this method can play an important role in the efficiency of the method. The test results in this paper are based on an implementation of the method which selects (under safe selection) a literal from the condition part of the top clause, until no such literal remains. Another example of a literal selection strategy is to select a literal from the conclusion part first, if there is a conclusion. In all the tables the additional entry (within parentheses) corresponding to Kowalski et al.'s method is the time taken by the method with this literal selection strategy (excluding rule R5 to avoid an infinite loop). It has been shown by Kowalski et al. that Decker's method and the method of Lloyd et al. can be approximated by considering respectively the previous two literal selection strategies. Due to this fact, the test results of Kowalski et al.'s method with the former literal selection strategy behave in the same way as

those of Decker's method and test results with the latter literal selection strategy behave in the same way as those of Lloyd et al.'s method.

The path finding method, Decker's method and the method of Kowalski et al. can handle those recursive rules for which the underlying SLDNF-resolution terminates. The simplification of constraints in both Lloyd et al.'s method and Bry et al.'s method are independent of the evaluation method chosen. Each of these methods will work in the presence of recursive rules if the underlying query evaluator handles recursion. Lloyd et al.'s method considers a typed database and converts it to a type free database by some transformations. The other methods need the database to be range restricted to evaluate a query correctly.

When a transaction is performed on the database the path finding method derives in stages a set of fully instantiated atoms added to the database and a set of partially instantiated atoms whose instances are likely to be deleted from the database. By contrast Decker's method derives two sets of fully instantiated atoms. The first set is the set of atoms implicitly added to the database and the second set is the set of atoms deleted from the database. The methods of Lloyd et al. and Bry et al. derive two sets of partially instantiated atoms. These represent respectively the set of atoms likely to be added to the database and the set of atoms likely to be deleted from the database. The approach taken by the path finding method is identical to that of Decker when there is no implicit deletion in the database. When an implicit deletion occurs in a stage, the method follows a similar approach to that of Lloyd et al. It can be observed from Fig 5 that the rightmost two paths starting from the literal *Employed*(1) are similar to the paths in the same position of Fig 2. This is because these two paths do not show any implicit addition to the database. It is also observed from Fig 5 that starting from the root the leftmost branch is similar to the branch in the same position in Fig 3. This is due to the fact that this path does not show any implicit deletion from the database.

The major difference between the path finding approach and the approach by Decker on the one hand and those of Lloyd et al. and Bry et al. on the other is that, in the first kind of approach instantiation and evaluation of constraints is done in stages whereas in the latter approaches instantiation and evaluation of constraints are performed after the derivation of the two sets of partially instantiated atoms.

Decker's method could be inefficient for a complex transaction which requires reasoning with two database states (before and after update) to calculate the two sets of atoms mentioned earlier. For each induced update, ie each fact which is implicitly added or deleted, the method checks whether it is provable in the database before update or not. The method may show better performance than the other methods when constraints are relevant to the induced update due to an update but some of the induced updates are already provable in the database before update. In all the tables the extra entry within

parentheses corresponding to Decker is the time taken by the method when this kind of checking (in the considered examples this is redundant) has been excluded from the implementation of the method.

In both the approaches by Lloyd et al. and Bry et al., two sets of partially instantiated atoms are always derived even if no ground instance of these atoms is true in the updated database and hence may cause redundant partial evaluation of the constraints in the case of Lloyd et al.'s method (Bry et al.'s method discards this possibility by using the Delta predicate). Consider the following example.

Example 3

Database D3 :

Rules :

as described in Database D1.

Facts :

A list of following 1085 facts not involving constants *1,2*:
177 facts under the predicate *Father*,
229 facts under the predicate *Husband*,
620 facts under the predicate *Occupation*,
59 facts under the predicate *Sponsor*.
Occupation(1,Service)
Occupation(2,Service)

Integrity Constraints I3:

$Guardian(x,y) \rightarrow Sponsor(x,y)$
$Sponsor(x,y), Guardian(z,y) \rightarrow Parent(x,y)$

Update :
insertfact *Husband(1,2)*.

Database D3 satisfies the constraints I3. Now if one wants to insert the fact *Married(1,2)* to D3, these two methods will derive the two sets of atoms

{Husband(1,2), Mother(2,y), Parent(2,y), Ancestor(x,y), Dependent(y,2), Guardian(2,y), Wife(2,1), Married(2,1), Self(1), Married(1,2), Dependent(2,1), Guardian(1,2), Self(2)} and *{ }*.

Derivation of these two sets of atoms is highly redundant as no ground instance of these atoms under the predicate *Guardian*, which are simplifying the constraints, is true in the updated database and hence will cause redundant evaluations of the instantiated constraints corresponding to the atoms *Guardian(2,y),Guardian(1,2)*. As no fact is implicitly added to the database corresponding to the insertion of *Married(1,2)*, in both the approaches by Decker and the path finding method, no constraint would be evaluated. The time taken by different methods to check consistency when the above insertion is made to the database D3 in the presence of constraints I3 is given in Table 2.

Method	Time
Lloyd et al.	3.4
Decker	0.8(0.2)
Kowalski et al.	0.4(3.3)
Path finding	0.2

Table 2.Times taken (in cpu seconds) by different methods to report consistency in the case of example 3

In the approach proposed by Lloyd et al. (and also that proposed by Bry et al.), it is sometimes possible to have the evaluation of unsimplified constraints which may cause inefficiency. For example, if one wants to delete (resp. insert) a ground instance of *Occupation(y,Service)* from (to) a database D containing rules described in Database D1, then *Self(x)* would be a member of $pos_{D,D'}$ ($neg_{D,D'}$) described in section 4.1. In that case if there is a constraint C relevant to the atom *Self(x)* (*Not Self(x)*), no simplification for C is possible before its evaluation. In Decker's method constraints are always simplified before evaluation as the method always calculates two sets of ground atoms implicitly added to or deleted from the database. In the above case of addition, the path finding method will follow Decker's approach and hence C will be simplified accordingly. But, in the above case of deletion, the method will follow the approach of Lloyd et al. and hence will cause the evaluation of unsimplified C.

The path finding method and Decker's method suffer from the drawback that all induced updates are computed, including those for which no constraints are relevant. Kowalski et al.'s method can avoid this by proper literal selection strategy. On the other hand the methods of Lloyd et al. and Bry et al. do not suffer from this drawback as they compute only potential updates that represent possible ground induced updates. Consider the following example.

Example 4

Database D4 :

Rules :

as described in Database D1.

Facts :

A list of following 1058 facts not involving constants *1,3*:
184 facts under the predicate *Father*,
226 facts under the predicate *Husband*,
600 facts under the predicate *Occupation*,
48 facts under the predicate *Sponsor*.
A list of 10 facts under the predicate *Father* with *1* as first argument.
Occupation(2,Service)
Occupation(3,Student)
Father(1,3)

Integrity Constraints I4:

$Father(x,z), Father(y,z) \rightarrow x = y$

Update :

insertfact *Husband*(1,2).

Database D4 satisfies the constraints I4. Now if one wants to insert the fact *Husband(1,2)* to D4, these two methods will derive in stages the following set of atoms

{Wife(1,2), Married(1,2), Married(2,1), Mother(2,C), Parent(2,C), Ancestor(2,C), Dependent(C,2), Self(2), Guardian(2,C) :
C is a constant and *Father(1,C)* is true in the updated database}

and, possibly, some more under the predicate *Ancestor*. It is clear that no constraint in I4 is relevant to the above set of atoms and will cause redundant evaluation. The evaluation time in this case depends mainly on the number of facts present in the database under the predicate *Father* whose first argument is the constant *1*. The time taken by the different methods to check consistency when the above insertion is made to the database D4 in the presence of constraints I4 is given in the column corresponding to 'add' of Table 3.

Method	Operation	
	add	del
Lloyd et al.	0.7	0.7
Decker	62.9(38.5)	63.0(43.9)
Kowalski et al.	>>100.0(0.5)	>>100.0(31.8)
Path finding	37.4	0.3

Table 3.Times taken (in cpu seconds) by different methods to report consistency in the case of example 4

The entries in the column headed 'del' in Table 3 are the times taken by different methods to check consistency when the fact *Husband(1,2)* is deleted from the updated database of example 4. In this case Decker's method calculates all the facts implicitly deleted from the database due to the deletion of *Husband(1,2)* and these are precisely those which were implicitly added to D4 by the addition of *Husband(1,2)*. This calculation is redundant and the path finding method avoids this by following Lloyd et al.'s approach.

In Lloyd et al.'s approach, an instance of a literal from any one of the two sets of partially instantiated atoms may be evaluated more than once if that particular literal simplifies more than one constraint. This will cause inefficiency. The path finding method also suffers from this drawback but only when a negative literal simplifies a constraint. On the other hand Decker's method does not have this drawback at all as the literal which simplifies the constraint is always true in the database and hence it is removed from the unified constraint.

In the path finding method finding a path from an update to the head of a constraint is similar to the problem of finding a refutation with an update literal as a top clause in Kowalski's method. In the latter method finding a refutation means arriving at an empty head of the denial form of a constraint from an update literal. In the path finding method constructing a success path means reaching the head of a constraint from an update literal. The difference lies in the way in which this is achieved - in the method put forward by Kowalski et al. the computation of positive induced updates can be deferred by a proper literal selection strategy but the computation of negative induced updates is essential, in the path finding method the computation of positive induced updates is essential and the computation of negative induced updates is deferred.

As the database size increases, the times taken by some operations (eg. search, unification) employed with constraint checking increase accordingly and in this kind of situation the time taken by each of the methods to check consistency varies almost directly as the number of relevant clauses present in the database. Table 4 shows the test results obtained by performing the update

of example 2 on the database of example 2 varying the number of facts in the following way. Initially the database contains the 1095 facts of example 2. After performing the tests another 1085 facts of example 3 are added and the tests are performed. Final tests are performed by adding 1058 facts of example 4 to the database. No facts in the database involve constants 1,2,3. It can be observed that the time taken by each method to perform the transaction increases as the number of facts in the database increases.

Method	No of Facts		
	1095	2180	3238
Lloyd et al.	1.9	2.9	4.0
Decker	4.9(2.5)	15.1(8.6)	31.3(18.7)
Kowalski et al.	31.1(0.7)	74.5(0.8)	133.6(0.9)
Path finding	2.4	8.3	18.3

Table 4.Times taken (in cpu seconds) by different methods in the case of example 2 varying the number of facts

Finally, from a more practical point of view, a test has been carried out using a typical example (example 5). The example describes three different states of a database by varying the number of facts but keeping the set of rules the same. Furthermore the set of constraints imposed on each of these three states of the database is the same. A set of transactions is performed on the three different states of the database. Appendix 1 contains the details of the three states of the database of example 5, the set of constraints imposed on it and the set of transactions to be performed. Table 5 contains the results obtained from applying the set of transactions to each of the three different states of the database using the different constraint checking methods.

Method	No of Facts		
	1010	3062	5119
Simple	37.0	311.5	860.6
Lloyd et al.	2.9	11.7	28.3
Decker	16.6(2.5)	54.0(12.3)	99.4(29.8)
Kowalski et al.	14.2(17.0)	58.6(67.8)	129.1(147.9)
Path finding	2.5	11.9	29.0

Table 5.Times taken (in cpu seconds) by different methods in the case of example 5 varying the number of facts

6. Conclusion

In the past few years a number of methods have been proposed by different

authors for checking integrity in deductive databases. Several of these are introduced in brief and their performances are evaluated by performing transactions on different states of a database using a number of test examples. The ideas of constraint satisfiability and constraint expressibility employed by different methods are also discussed.

Acknowledgements

This work was conducted with support from the Ministry of Education, Government of India and was done in conjunction with the Logic Database Group who are currently supported by the SERC through the Alvey initiative.

References

1. K.R.Apt, H.A.Blair, and A.Walker, Towards a theory of declarative knowledge, J.Minker (eds.): *Foundation of Deductive Databases and Logic Programming*, pp. 89-148, Morgan Kaufman Publishers, Inc, (1988).
2. P.Asirelli, M.D.Santis, and M.Martelli, Integrity constraint in logic databases, *Journal of Logic Programming*, Vol. 3, pp. 221-232, (1985).
3. F.Bry, H.Decker, and R.Manthey, A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases, *Proceedings of Extending Database Technology*, pp. 488-505, Venice, (1988).
4. C.L.Chang, DEDUCE 2: Further investigations of deduction in relational databases, H.Gallaire,J.Minker (eds.):*Logic and Databases*, New York, New York, (1978).
5. K.L.Clark, Negation as failure, H.Gallaire,J.Minker (eds.):*Logic and Databases*, pp. 293-322, Plenum Press, New York, (1978).
6. W.F.Clocksin and C.S.Mellish, *Programming in Prolog*, Springer International Student Edition, (1984).
7. S.K.Das and M.H.Williams, A path finding method for checking integrity in deductive databases, Submitted to *Data and Knowledge Engineering*.
8. C.J.Date, *An Introduction to Database Systems*, Vol 2, Addison Wesley, (1985).
9. C.J.Date, *An Introduction to Database Systems*, Vol 1, 4th edition, Addison Wesley, (1986).
10. H.Decker, Integrity enforcements on deductive databases, L.Kerschberg (ed.): *Proceedings of the First International Conference on Expert Database Systems*, pp. 271-285, Charleston, South Carolina, (April 1986).
11. E.B.Fernandez, R.C.Summers, and C.Wood, *Database Security and Integrity*, Addison Wesley, (1981).
12. H.Gallaire, J.Minker, and J.-M.Nicolas, Logic and databases - a deductive approach, *Computing Surveys*, Vol. 16, No.2, pp. 153-185, (1984).
13. R.Goebel, K.Furukawa, and D.Poole, Using definite clauses and integrity constraints as the basis for a theory formation approach to diagnostic reasoning, E.Shapiro (ed.): *Proceedings of the 3rd Conference on Logic Programming*, pp. 211-222, Springer-Verlag, London, U.K, (July 1986).
14. R.A.Kowalski, Logic programming with integrity constraints, Workshop on Logic Programming, Imperial College, London, (1989).
15. R.A.Kowalski, F.Sadri, and P.Soper, Integrity checking in deductive databases, *Proceedings of the 13th VLDB Conference*, pp. 61-69, Brighton, (1987).
16. T.-W.Ling, Integrity constraint checking in deductive databases using the Prolog not-predicate, *Data & Knowledge Engineering*, Vol. 2, pp. 145-168, (1987).
17. J.W.Lloyd, *Foundations of Logic Programming*, 2nd, Extended Edition, Springer Verlag, Symbolic Computation Series, (1984).

18. J.W.Lloyd, E.A.Sonenberg, and R.W.Topor, *Integrity constraint checking in stratified databases*, Department of Computer Science, University of Melbourne, Technical Report 86/5.
19. J.W.Lloyd and R.W.Topor, Making Prolog more expressive, *Journal of Logic Programming*, Vol. 1, No.3, pp. 225-240, (1984).
20. J.W.Lloyd and R.W.Topor, A basis for deductive database systems, *Journal of Logic Programming*, Vol. 2, No.2, pp. 93-109, (1985).
21. J.-M.Nicolas, Logic for improving integrity checking in relational databases, *Acta Informatica*, Vol. 18, pp. 227- 253, (1982).
22. J.M.Nicolas and K.Yazdanian, Integrity checking in deductive databases, H.Gallaire, J.Minker (eds.):*Logic and Databases*, pp. 325-344, Plenum Press, New York, (1978).
23. R.Reiter, On the integrity of typed first order databases, H.Gallaire,J.Minker,J.-M.Nicolas (eds.):*Logic and Databases*, Vol. 1, pp. 137-157, Plenum Press, New York, London, (1981).
24. F.Sadri and R.A.Kowalski, An application of general purpose theorem-proving to database integrity, J.Minker (ed.): *Proceedings of the Workshop on Foundations of Deductive Databases and Logic Programming*, (1987).
25. C.Small, Guarded default databases: a prototype implementation, *Prolog and Databases:implementations and new directions*, pp. 121-134, Ellis Horwood Limited, England, (1988).
26. L.Sterling and E.Shapiro, *The Art of Prolog*, pp. 303-329, The MIT Press, Cambridge, Massachusetts, (1986).
27. J.D.Ullman, *Principles of Database Systems*, 2nd edition, Computer Science Press International, Inc, Maryland, USA, (1984).
28. M.H.Williams, J.C.Neves, and S.O.Anderson, Security and integrity in logic databases using Query-by-Example, *Proceedings of Logic Programming Workshop*, pp. 304-340, Algarve, Portugal, (1983).

Appendix 1

The following is a summary of the contents of the three states of the database, integrity constraints and transaction used in example 5.

DATABASE

Relations :

Civil Status (Social Security Number (SS No), Age, Sex, Occupation)
Age (SS No, Age)
Sex (SS No, Sex)
Occupation (SS No, Occupation)
Father (SS No of Father, SS No of Son)
Mother (SS No of Mother, SS No of Son)
Parent (SS No of Parent, SS No of Son)
Husband (SS No of Husband, SS No of Wife)
Dependent (SS No of Dependent, SS No of Provider)
Tax (SS No of Tax Payer, SS No of Tax Paid For)

Rules :

$Age(x,y) \leftarrow Civil\ Status(x,y,p,q)$

Sex(x,y) ← *Civil Status(x,p,y,q)*
Occupation(x,y) ← *Civil Status(x,p,q,y)*
Mother(x,y) ← *Father(z,y), Husband(z,x)*
Parent(x,y) ← *Father(x,y)*
Parent(x,y) ← *Mother(x,y)*
Dependent(x,y) ← *Parent(y,x), Occupation(y,Service), Occupation(x,Student)*

Facts :
The following facts not involving the constants *1,2,3* in the domain of *SS No* :
600 (1800 in state2, 3020 in state3) facts under the predicate *Civil Status*,
169 (518 in state2, 860 in state3) facts under the predicate *Father*,
220 (669 in state2, 1120 in state3) facts under the predicate *Husband*,
21 (75 in state2, 119 in state3) facts under the predicate *Tax*.

INTEGRITY CONSTRAINTS

Civil Status(x,y_1,z_1,t_1), *Civil Status*$(x,y_2,z_2,t_2) \rightarrow y_1=y_2, z_1=z_2, t_1=t_2$
Father$(x_1,y), Father(x_2,y) \rightarrow x_1=x_2$
Husband$(x,y_1), Husband(x,y_2) \rightarrow y_1=y_2$
Husband$(x_1,y), Husband(x_2,y) \rightarrow x_1=x_2$
Civil Status$(x,y,z,t) \rightarrow x > 0, x < 100000, y > 0, y < 125,$
$z \in$ *{Male,Female}*, $t \in$ *{Student,Retired,Business,Service}*
Civil Status(x,y,z,Student) $\rightarrow y < 25$
Civil Status(x,y,z,Retired) $\rightarrow y > 60$
Father(x,y) → *Sex(x,Male), sex(y,Male)*
Husband(x,y) → *Sex(x,Male),sex(y,Female)*
Husband$(x,y), Age(x,p), Age(y,q) \rightarrow p \geq 20, q \geq 20$
Civil Status(x,y,z,t), $y < 20 \rightarrow t =$ *Student*
Dependent(x,y) → *Tax(y,x)*

TRANSACTION

insertfact *Civil Status(1,50,Male,Service)*,
insertfact *Civil Status(2,45,Female,Business)*,
insertfact *Civil Status(3,19,Male,Student)*,
insertfact *Husband(1,2)*,
insertfact *Father(1,3)*,
insertfact *Tax(1,3)*.

where x, y, z, p, q, t, x_1, x_2, y_1, y_2, y_3, z_1, z_2, z_3 are variables, and *Male, Female, Student, Retired, Business, Service* are constants. The above set of constraints satisfies the database consisting of above rules and facts.

55. K.WICOVSKY, P.F.THANISCH and M.H.WILLIAMS, Improved recursion handling through integrity constraints, The Computer Journal, vol. 34, 282-285 (1991).

Table 2. Overall timings (seconds)

F	2^n	N_F	*Perc*	COMB	*CRIBLE*	*KL*
1	128	124	0.97	4.7	—	585.3
2	1024	664	0.65	122.9	13.7	358.1
3	32	13	0.41	1.7	0.7	93.4
4	128	60	0.47	6.5	7.6	314.5
5	1024	1024	1.00	21.6	105.9	370.2
6	64	37	0.58	2.3	1.6	199.9
7	64	22	0.34	2.2	0.8	143.8
8	128	92	0.72	3.9	7.8	243.2
9	1048576	524288	0.50	—	—	1768.0
10	33554432	16777216	0.50	—	1125.6	992.7
11	33554432	4194304	0.12	—	—	1357.1
12	32768	1149	0.30	—	1.5	345.6
13	4096	6	0.00	359.6	0.5	203.1
14	1048576	581	0.00	—	1.7	539.9
15	33554432	13472	0.00	—	4.4	1008.0

Table 3(*f*). Results for F_9

ε	δ	N	N_s	N_F^*	*Err*	*CPU*
0.1	0.2	20724	5739	534886.8	+0.02	1384.8
0.1	0.3	17075	4723	534264.6	+0.02	1132.1
0.1	0.4	14485	4047	539652.1	+0.03	948.2
0.1	0.5	12477	3457	535165.9	+0.02	808.6
0.2	0.1	6741	1894	542693.8	+0.03	434.9
0.3	0.1	2996	849	547350.0	+0.04	196.0
0.4	0.1	1686	457	523549.6	−0.00	108.9
0.5	0.1	1079	288	515549.4	−0.02	69.3

Table 3(*g*). Results for F_{10}

ε	δ	N	N_s	N_F^*	*Err*	*CPU*
0.1	0.2	11398	2959	16877492.2	+0.01	772.3
0.1	0.3	9391	2331	16136971.6	−0.04	640.9
0.1	0.4	7967	2135	17421865.8	+0.04	512.4
0.1	0.5	6863	1868	17695159.3	+0.05	446.5
0.2	0.1	3708	1021	17901013.5	+0.07	248.7
0.3	0.1	1648	457	18028126.4	+0.07	111.4
0.4	0.1	927	245	17182167.7	+0.02	63.1
0.5	0.1	594	149	16307651.7	−0.03	39.8

Improved Recursion Handling through Integrity Constraints

A new form of database integrity constraint is introduced which describes the structure present in data when that structure takes the form of a graph, tree or list. This type of integrity constraint is of particular benefit in reducing the overhead when detecting termination in the case of recursive queries.

Received September 1987

1. Introduction

Some database integrity constraints control the values which an attribute can adopt (e.g. age < 21) or the existence of values in other relations, e.g.

> if there is an entry for supplier n in the supplier-parts relation, then there must be a corresponding entry for supplier n in the supplier relation.

Other constraints (data dependencies) are concerned with the relationship between the sets of values in one set of attributes with the sets of values in some other, e.g. functional dependencies, multivalued dependencies, etc.

However, there are other relationships which may be present in the data in a database which are not readily captured by existing constraints. For example, the data stored in a relation may represent some form of structure such as a graph, tree or list. Apart from some limited cases, it is not immediately obvious how one can represent information such as this using existing integrity constraints.

If the data in a relation (or set of relations) do satisfy some structural form, one may wish to impart this information to a database management system so that it can check that this form is maintained just as it checks the integrity of data as specified by other integrity constraints. For this purpose a new form of constraint, called a *structure constraint*, is proposed.

Besides the obvious use of such a constraint in maintaining the integrity of data in a database, this type of constraint carried with it another significant advantage. It can be used to produce a more efficient system for handling recursion in databases by reducing the overhead incurred in run-time detection of termination.

The following section describes the basic structural forms used in this constraint. Section 3 details the structure constraints and gives examples while Section 4 discusses their role in improving the efficiency of dealing with recursive relations. Section 5 looks at some theorems relating to these constraints.

2. Structural forms

This section considers some basic structural forms which may exist in a set of data stored in a relation. Before defining these structures, consider briefly some terminology which will be used. Let I be a set of constraints. SAT(I) denotes the set of relations that satisfy each of the constraints in I. Let I_1 and I_2 be constraints or sets of constraints.

$$I_1 \Rightarrow I_2 \text{ if } \mathrm{SAT}(I_1) \subseteq \mathrm{SAT}(I_2)$$

A tuple is a set of mappings from attributes to domain values. A relation is a set of tuples. Let R be a relation over a set, U, of attributes and let $t \in R$. For $W \subseteq U$, we let $t(W)$ denote the restriction of t to the set W. Let R be a relation over a set, U, of attributes, and let A, $B \subseteq U$. R obeys the *functional dependency* $A \rightarrow B$ if for any pair of tuples $t_1, t_2 \in R$, $t_1(A) = t_2(A)$ implies that $t_1(B) = t_2(B)$.

suppose that R is a relation with the relational scheme $R(A, B, C)$ where A and B are attribute sets and C is an abbreviation for the rest of the attributes. Then:

Definition

A *chain* over a pair of attribute sets A and B of $R(A, B, C)$ is a sequence of tuples (a_1, b_1, c_1), $(a_2, b_2, c_2) \ldots (a_n, b_n, c_n)$ where $b_i = a_{i+1}$ for every , $0 < i < n$.

Definition

The *length* of a chain is the number of tuples, n, in the sequence.

2.1 Directed acyclic graph

A typical example of this type of structure occurs in the part of relation, where an object is described in terms of its component parts. An illustration of this is given in Table 1.

Since this graph is intended to be acyclic, no part should be a sub-part of itself. Thus the addition of a tuple such as

part-of (5, 1, 1)

would be invalid as it would violate the condition of acyclicity.

A formal definition of this type of structure is as follows:

Definition

A *directed acyclic graph structure* (*dag structure*) is present in attribute sets A and B of a relation R with relation scheme $R(A, B, C)$, written as *graph* (A, B), if there is no chain of the form $(a_1, b_1, c_1), (a_2, b_2, c_2) \ldots (a_n, b_n, c_n)$ in which $a_1 = b_n$.

2.2 Directed tree structures

A special case of a dag structure is the tree structure. This is identical to the tree structure in QBE [1, 2]. An example of this is shown in Table 2.

Table 1. Example of acyclic directed graph structure

part	part 1-no	part 2-no	no-of-pts
	1	2	1
	1	3	2
	2	4	4
	2	5	2
	3	5	7
	51	55	2
	100	101	2
	101	102	4
	101	103	2
	103	104	2
	100	104	2

Here one wants to guarantee that nobody is his own ancestor and that every child has only one father. Thus the following additions would violate this type of structure:

parent (klaus, peter) since Peter would then have two fathers

parent (klaus, paul) since this would create a cycle.

A directed tree structure may be defined formally as follows:

Definition

A *directed tree structure* is present in attribute sets A and B of a relation R with relation scheme $R(A, B, D)$, written as tree (A, B), if

(a) A dag structure is present in attribute sets A and B (i.e. $R \in$ SAT (graph (A, B)), and

(b) if some tuple t occurs in the ith position of some chain C_1 ($i > 1$) and in the jth position of some chain C_2 ($j > 1$) then the values of A and B for the tuple in the $(i-1)$th position of C_1 must be the same as those in the $(j-1)$th position of C_2.

2.3 U-list structure

A U-list is an acyclic list in which each element is unique within the list. For example, consider the relation given in Table 3. In this relation nobody can be his own ancestor; in addition, every father has only one eldest son and every son only one father.

Possible additions which would give rise to violations of this structure include:

inheritor (klaus, peter) since Klaus may have only one eldest son,

inheritor (kevin, klaus) since Klaus may have only one father,

inheritor (jim, paul) since this would lead to a cycle.

A formal definition of a U-list structure is as follows:

Definition

A *U-list structure* is present in attribute sets A and B of a relation R with relation scheme R (A, B, D), written as *U-list* (A, B), if

(a) a tree structure is present in attribute sets A and B, i.e. $R \in$ SAT (tree (A, B)), and

Table 2. Example of directed tree structure

parent	father	child
	paul	peter
	paul	john
	peter	kevin
	peter	jim
	john	klaus
	andrew	charles

Table 3. Example of U-list

inheritor	father	eldest-son
	paul	peter
	peter	kevin
	kevin	jim
	john	klaus
	klaus	andrew

(b) if some tuple t occurs in the ith position of some chain C_1 (other than the last) and in the jth position of some chain C_2 (other than the last) then the values of A and B for the tuple in the $(i+1)$th position of C_1 must be the same as those in the $(j+1)$th position of C_2.

2.4 Special cases

A number of special cases can be derived from these three basic structures. For example, one might define a connection condition as follows:

Definition

If for every pair of tuples (a_x, b_x, c_x) and (a_y, b_y, c_y), there is a sequence of tuples $(a_x, b_x, c_x) = (a_1, b_1, c_1)$, $(a_2, b_2, c_2) \ldots (a_n, b_n, c_n) = (a_y, b_y, c_y)$ such that for each successive pair (a_i, b_i, c_i), $(a_{i+1}, b_{i+1}, c_{i+1})$ either $a_i = b_{i+1}$ or $b_i = a_{i+1}$ then the set of tuples is *connected* in A and B.

If this condition is satisfied and one of the three basic structures is present then one has a connected directed acyclic graph, a connected directed tree or a connected U-list.

One may also place restriction on the in-degree and out-degree of nodes in the structure. For example, one might define a b-tree as follows:

Definition

If for every tuple t in a directed tree structure, either there is no chain starting at t or there are exactly two different chains of length two starting at t, then the structure is a *b-tree*.

Likewise one might define a binary tree as follows:

Definition

If for every tuple t in a directed tree structure there are at most two different chains of length two starting at t, then the structure is a *binary tree*.

Similarly if one wanted a balanced tree, one might define:

Definition

A chain is *maximal* if there is no tuple in the relation that can be added to the end of it.

Definition

If for every tuple t in a directed tree structure, every maximal chain starting with t has the same length (± 1) then the structure is a *balanced tree*.

3. Specification of structure constraints

The following predicates can be used for specifying structure constraints:

(a) *chain* $(R\ (A, B, C),\ A, B, a, b)$ is true if there exists a chain in the relation R over the attribute sets A and B that starts with $a = a_1$ and ends with $b = b_n$.

(b) *indegree* $(R\ (A, B, C), A, B, a, b, n)$ is true if the in-degree of the node (tuple) $R(a, b, C)$ is exactly n.

(c) *outdegree* $(R\ (A, B, C),\ A, B, a, b, n)$ is true if the out-degree of the node (tuple) $R(a, b, C)$ is exactly n.

(d) *maxlength* $(R(A, B, C),\ A, B, a, b, max)$ is true if the longest distance between the tuple $R(a, b, c)$ and a leaf of the graph is *max*.

(e) *minlength* $(R(A, B, C), A, B, a, b, min)$ is true if the shortest distance between the tuple $R(a, b, c)$ and a leaf of the graph is *min*.

Note that to specify that an indegree (or outdegree) is less than a given value x, one can use numerical dependencies [3]. For example, the dependency denoted by $a \rightarrow (^x) b$ states that for every value of a there at most x different values for b. However, these cannot be used for the specification of exact indegrees (or outdegrees).

Example

(1) If one wishes to specify that the relation part-of illustrated in Table 1 represents a dag, one might write

part-of $(x, y, z) \rightarrow$ not chain (part-of (part 1-no, part 2-no, no-of-pts), part 1-no, part 2-no, y, x)

This means that a tuple part-of (5, 1, 5) can only be added if there is no chain from 1 to 5. In the example in Table 1 there is a chain (1, 3, 2), (3, 5, 7) and hence adding the tuple would violate the constraint.

(2) If one wished the part-of relation to have the structure of a balance tree, one might add the following constraints:

(a) part-of $(A1, B, -)$ & part-of $(A2, B, -) \Rightarrow (A1 = A2)$

(b) part-of $(A, B, -)$ & maxlength (part-of (part 1-no, part 2-no, no-of-pts), part 1-no, part 2-no, A, B, N) & minlength (part-of (part 1-no, part 2-no, no-of-pts), part 1-no, part 2-no, $A, B, M) \Rightarrow (M = N) \vee (M = N+1) \vee (M = N-1)$

(c) part-of $(B, B1, -)$ & part-of $(A, B, -)$ & outdegree (part-of (part 1-no, part 2-no, no-of-pts), part 1-no, part 2-no, A, B, N) & $(N > 0) \Rightarrow (N = 2)$

The first constraint guarantees a tree structure, the second ensures that every path starting at a given node has the same length while the third makes the tree a binary tree.

Acyclic structures need not be confined to a single relation. It is possible to have such structures existing in pairs of attribute sets located in different relations. Thus one may define:

Definition

A directed acyclic graph structure (graph, tree or U-list) is present in attribute sets A_1 of relation R_1 (with scheme R_1 (A_1, B_1, C_1)) and

Table 4. Example of acyclic graph structure over two relations

part	no 1	name	weight	mat
	1	p1	44	ff
	2	p2	2	ww
	3	p3	12	ee
	4	p4	7	ww
	5	p5	23	er
	6	p6	12	op
	7	p7	5	ww

part_sup	no 2	needed-for	no-of-pts
	2	p1	7
	3	p1	2
	5	p2	1
	6	p2	2
	6	p3	1

A_2 of relation R_2 (with scheme $R_2(A_2, B_2, C_2)$) if a dag structure (graph, tree or U-list) is present in attribute sets A_1 and A_2 of R_1 [$B_1 = B_2$) R_2.

Example

Given the two relations part and part-sup, illustrated in Table 4, one can specify the presence of a dag structure in attribute sets 1 and 2 in the relation obtained by joining part and part-sup on the condition name = needed-for. The joined relation is shown in Fig. 1.

part [name = needed-for] part-sup					
no 1	no 2	name	weight	mat	no-of-pts
1	2	p1	44	ff	7
1	3	p1	44	ff	2
2	5	p2	2	ww	1
2	6	p2	2	ww	w
3	6	p3	12	ee	1

Figure 1. Joined relations part and part-sup.

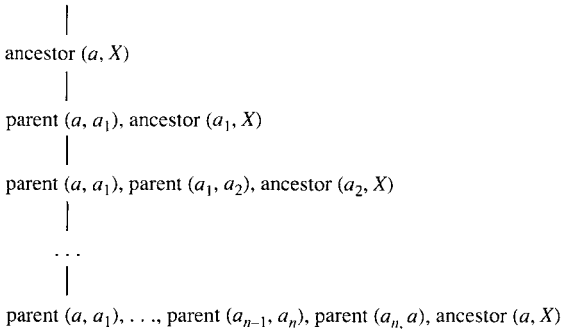


Figure 2. Branch of resolution tree with two identical subqueries ancestor.

4. Termination of recursion

One of the problems with dealing with recursion in databases is that of detecting termination. In programming languages such as Prolog it is up to the programmer to ensure that infinite loops do not occur – if they do arise it is his responsibility to deal with them. However, in a database system, the system cannot permit a user to get into an infinite loop – if he should do so he might tie up the whole system and affect all other users.

During recent years many approaches have been proposed for solving the problem of detecting termination of recursive axioms.[6,8] All of these carry a considerable overhead and are expansive in terms of both computer time and main memory. Some particular disadvantages of these methods include:

(*a*) a user-defined axiom is compiled into an internal representation and must be administered separately.[4,5]

(*b*) all subqueries and subtuples must be stored temporarily, even if they are not an answer to the original query,[9]

(*c*) different proof procedures must be used for recursive and non-recursive axioms.[9]

The main assumption which most of these methods make in handling recursion is that they use only the rules and facts in the database and do not consider any additional knowledge which could be gained from integrity constraints. One exception to this is Mendelzon[7] but he is concerned only with the case of backtracking. Another assumption which is generally made is that the database is function-free and comprises only finite relations.[10] In this section the benefit to structure constraints in improving the handling of recursion will be demonstrated in relation to Vieille's Query/Subquery Approach (QSQ).

To illustrate the technique, consider the following example:

Query: $q(Y) \leftarrow$ ancestor (a, Y).

Rules: ancestor $(X, Y) \leftarrow$ parent (X, Y).

ancestor $(X, Y) \leftarrow$ parent (X, Z) & ancestor (Z, Y).

IC: parent $(X, Y) \rightarrow$ not chain (parent (father, son), father, son, Y, X).

A simplified description of the QSQ is as follows. The QSQ can be seen as a special theorem prover that has the capability to decide whether or not a possible resolution step is necessary in order to yield a complete answer to a query. This decision is made on the basis of the current state of processing. This state is defined by the pair (Q, R). Q being a set of subqueries to be answered and R being the set of answers found for these subqueries so far. In an axiom $a(X, Y) \leftarrow b(X, Z)$ & $a(Z, Y)$, $b(X, Z)$ is a subquery and $a(Z, Y)$ a recursive subquery of the query $a(X, Y)$, which itself can be the subquery of another query. Two subqueries are equal, if they have the same name and equal constants in equal positions.

The recursion is terminated, i.e. the special theorem prover decides not to make a possible resolution step, if the state of processing will not be changed by it.

In order to maintain this state of processing throughout the processing of the whole query, it is necessary to store temporarily the set of subqueries Q and the set of answers R. In addition, it is necessary to search Q and R every time a tuple or a subquery is generated in order to determine if the tuple or subquery is new, i.e. if the state of processing has changed.

These two disadvantages can be avoided for the subclass of recursive axioms that has the character of graph traversal algorithms over acyclic graphs. A more precise definition can be found below.

This whole process can be avoided if one can be sure that a query will not get itself into an infinite loop. This is certainly true in the case of acyclic graphs. For instance, in the above example one knows that the parent relation represents an acyclic graph structure. With this knowledge, one can guarantee that:

(*a*) every recursive subquery is unique in its branch of the resolution tree and every recursion step produces a new recursive subquery, i.e. the set Q of subqueries changes with every recursion step and hence the state of processing changes with every recursion step, and;

(*b*) the answer to a query will be complete, even if one does not use the set R at all.

Obviously, (*b*) allows one to reduce the state of processing from (Q, R) *to* (Q) and since (*a*) guarantees that Q changes with every recursion step, it is unnecessary to check whether the reduced state (Q) has changed. Thus (*a*) and (*b*) together state that a theorem prover does not need to use the state of processing in order to ensure a complete termination of the recursive axioms in an example such as this.

Proof

(*a*) Consider an arbitrary branch of the resolution tree and assume that a subquery ancestor (a, X) appears twice in this branch, as shown in Fig. 2.

This can only happen if there is a chain of parents: parent $(a, a_1) \ldots$ parent (a_{n-1}, a_n) parent (a_n, a). But this situation violates the integrity constraint that ensures acyclicity for the parent relation.

Since every recursion step produces a subquery and since every subquery is unique within its branch of the resolution tree, one can deduce that every subquery is new.

(*b*) Consider the part of a resolution tree which one is able to generate with the reduced state of processing (Q) alone, as illustrated in Fig. 3.

If there is no tuple parent (a_n, X) in the database, the subquery ancestor (a_n, X) will fail (since both rules for ancestor will fail) on the next recursion level and thus the recursion comes to a natural end at this level, i.e. the answer is complete.

In order to incorporate this into the query evaluation mechanism of the database system, use is made of the hierarchical structure of

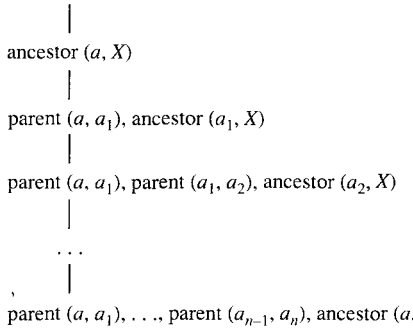


Figure 3. Branch of resolution tree over a relation with acyclic structure.

such a system in which the proof of a predicate may depend on the proof of a number of subsidiary predicates. Under certain circumstances, if the property of acyclicity holds for some attributes of predicate A, one may deduce that it will also hold for some attributes of predicate B. for example, given the database

parent $(X, Y) \leftarrow$ father (X, Y),
married (X, Z).

ancestor $(X, Y) \leftarrow$ parent (X, Y).

ancestor $(X, Y) \leftarrow$ parent (X, Z),
ancestor (Z, Y).

If the predicate father is known to be an acyclic structure, one can deduce that parent is acyclic (since there is no other rule generating parent) and hence one can conclude that ancestor is acyclic. In general, it is possible to propagate the acyclic structure over attributes A and B of a relation $R(A, B, C)$ to some other relation $S(A, B, D)$ where C and D are arbitrary sets of attributes, if the relation S is generated from the relation R by selection, projection (on a set of attributes which includes both A and B) and join. Equally, if S is the transitive closure of R then S is of acyclic structure if R is of acyclic structure. The more general operation of union is more difficult to deal with.

In our implementation structure constraints are invoked whenever rules in the database are updated. They are then used to determine whether or not the rules need to have termination checking associated with them. If they do, the appropriate calls are compiled into the rules before they are stored in the database. This provides an efficient mechanism for query evaluation.

5. Some theorems

Some simple results which apply to structure constraints are as follows:

(1) graph $(A, B), B \rightarrow A \Rightarrow$ tree (A, B)

This states that if a dag structure is present in attribute sets A and B of a relation R and if B functionally determines A then there is a directed tree structure present in attribute sets A and B.

Proof

Let $r \in$ SAT ({graph (A, B), $B \rightarrow A$}).
We prove that $r \in$ SAT (tree (A, B)).
Condition (a) of a tree structure is implied by the fact that

$$r \in \text{SAT}(\text{graph}(A, B)).$$

Let

$$c_1 = \langle t_1, t_2, \ldots, t_m \rangle$$

and

$$c_2 = \langle s_1, s_2, \ldots, s_n \rangle$$

be two chains in R such that

$$t_i = s_j \, (i > 1 \text{ and } j > 1),$$

From the definition of a chain, $t_{i-1}(B) = t_i(A)$ and $s_{j-1}(B) = s_j(A)$. Thus $t_{i-1}(B) = s_{j-1}(B) R \in$ SAT $(B \rightarrow A)$, so $t_{i-1}(A) = s_{j-1}(A)$.
Thus condition (b) of a tree structure holds and $R \in$ SAT (tree(A, B)).
Thus SAT (graph (A, B), $B \rightarrow A) \subseteq$ SAT (tree (A, B)),
so {graph $(A, B), B \rightarrow A$} implies tree (A, B).

(2) tree (A, B), $A \rightarrow B \Rightarrow$ U-list (A, B).

This states that if a directed tree structure is present in attribute sets A and B of a relation R and if A functionally determines B then there is a Ulist structure present in attribute sets A and B.

Proof

Let $R \in$ SAT (tree (A, B), $A \rightarrow B$). We prove that $R \in$ SAT (Ulist (A, B))
It is given that $R \in$ SAT (tree (A, B)) so condition (a) of the U-list structure is satisfied.
Let

$$c_1 = \langle t_1, t_2, \ldots, t_m \rangle$$

and

$$c_2 = \langle s_1, s_2, \ldots, s_n \rangle$$

be two chains in R such that $t_i = s_j$ $(i > 1$ and $j > 1)$. From the definition of a chain, $t_i(B) = t_{i+1}(A)$ and $s_j(B) = s_{j+1}(A)$. Thus $t_{i+1}(A) = s_{j+1}(A)$. But it is given that $R \in$ SAT $(A \rightarrow B)$, so $t_{i+1}(B) = s_{j+1}(B)$.
Thus R conforms to condition (b) in the definition of a U-list structure.

(3) Ulist $(A, B) \Rightarrow A \rightarrow B$, $B \rightarrow A$

If a U-list structure is present in attribute sets A and B of a relation R then A functionally determines B and B functionally determines A. This follows from (1) and (2).

(4) Graph $(A, B) \Rightarrow$ Graph (B, A)

(5) Ulist $(A, B) \Rightarrow$ Ulist (B, A)

6. Conclusion

The idea of a graph structure present in the attributes of a relation is a common one. Codd[11] discusses *diagraph relations* involving two attributes of a relation and introduces an operator for computing the transitive closure of such a digraph. Zloof[1,2] too has catered for the transitive closure operation in the query language QBE. Clemons[12] has extended SQL. by introducing the construct STRUCTURE which enables one to specify a hierarchical tree structure over a relation, and provides a transitive closure operation similar to that of Zloof.

This paper proposes a mechanism for specifying the presence of graph structures in a relation as a form of integrity constraint. This enables updates to be checked to ensure that the graph properties of the attributes are maintained. These properties can then be used to improve the efficiency of dealing with a recursive query by reducing the termination checks required by it.

K. WICOVSKY, P. F. THANISCH and M. H. WILLIAMS*

Computer Science Department, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ.

References

1. D. Maier, *The Theory of Relational Databases*. Computer Science Press (1983).
2. M. M. Zloof, *Query by Example: Operations on the Transitive Closure*. IBM Research Report, RC5526 (1976).
3. J. Grant and J. Minker, Normalization and axiomatization for numerical dependencies. *Information and Control* **65** (April 1985).
4. F. Bancilhon, D. Maier, Y. Sagiv and J. Ullman, Magic sets and other strange ways to implement logic programs. *Proc. 5th ACM SIGMOD SIGACT Symposium on Principles of Database Systems, Cambridge, Massachusetts* (1986).
5. J. Rohmer, R. Lescoeur and J. M. Kerisit, The Alexander method – a technique for the processing of recursive axioms in deductive databases. *New Generation Computing* **4**, 273–285 (1986).
6. F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. *ACM/SIGMOD, Washington D.C.* (1986).
7. A. O. Mendelzon, Functional dependencies in logic programs, I. *Proc. VLDB 85, Stockholm*, pp. 324–330 (1985).
8. L. Vieille, *Recursive Axioms in Deductive Databases: Various Solutions*. ECRC, Technical Report KB-6 (May 1985).
9. L. Vieille, Recursive axioms in deductive databases: the query/subquery approach. *Proc. of the 1st International Conference on Expert Systems, (4)* (1986).
10. D. E. Smith, M. R. Genesereth and M. L. Ginsburg, controlling recursive inference. *Artificial Intelligence*, **30**, 343–369 (1986).
11. E. Codd, extending the database relational model to capture more meaning. *ACM TODS* **1**, 397–433 (1979).
12. E. K. Clemons, Design of an external schema facility to define and process recursive structures. *ACM TODS*, **6**, (1981).

* To whom correspondence should be addressed.

56. K.G.WAUGH, M.H.WILLIAMS, Q.KONG, S.SALVINI and G.CHEN, Designing SQUIRREL: an extended SQL for a deductive database system, The Computer Journal, vol. 33, 535-546 (1990).

Designing SQUIRREL: an extended SQL for a deductive database system

K. G. WAUGH, M. H. WILLIAMS, Q. KONG, S. SALVINI AND G. CHEN
Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ

One of the problems facing the designers of a deductive database is that of the choice of query language. The deductive database model is seen as a natural progression from the relational model; the query language should reflect this. This paper reports the decisions taken while designing an extended form of SQL, called SQUIRREL, as the query language for a Prolog-based deductive database. The extensions relate to the inclusion of both rules and incomplete information in the query language and result in changes to the data definition, data manipulation and query languages of SQL. The extensions were constrained by the desire to retain the existing SQL texture in the new language while introducing concepts such as rules, rule manipulation and incomplete information which are alien to the relational philosophy. The language we describe is being used as the interface language for an implementation of a deductive database which will run on a version of Prolog developed to handle database applications.

Received May 1989, revised December 1989

1. INTRODUCTION

The relational model which was introduced by Codd,[1] and underlies relational databases, is now well established, and a number of different query languages have been developed for relational systems. More recently the logic database model has been proposed[2-4] as a natural successor to the relational model. This model is an upwardly compatible extension of the relational model. However, as yet there has been little work done on designing query languages for deductive or logic database systems based on the logic database model.

One significant difference between the approaches lies in the basic unit used by each. In the case of the relational model the basic unit is the **tuple**, sets of which are grouped together to form **relations**; in the logic database model the unit is the **clause**, sets of which make up **procedures**. A clause may be a ground fact which corresponds to the notion of a tuple, or it may be some form of rule. This difference is a fundamental one as it affects the basic strategy for query evaluation in the two models. In the relational model all tuples are stored explicitly and require no evaluation; by contrast the clauses stored in a deductive database are formulae which generate explicit tuples when they are evaluated. Thus the query language for a deductive database must permit a distinction to be made between the clauses (rules and facts) as objects stored in the database and the objects which are produced when these clauses are evaluated.

The functional characteristics of a logic database system make it an appropriate vehicle for realising some aspects of knowledge-based systems. The distinction between a logic database and a knowledge base is a fine one. Database systems have been developed for the storage and retrieval of large volumes of data in a safe and efficient manner. Interfaces to such systems are generally based on a simple (preferably high-level) approach which permits different applications to access data simultaneously, returning the data in the form in which each application needs it. An important function of a Database Management System (DBMS) is to ensure the security and integrity of data stored in the database through appropriate authorization and concurrency controls. A logic database system is a natural successor to the relational database system, based firmly on the theory of first-order logic as opposed to the relational theory typically found in DBMSs. Traditional relational databases permit only data (information) to be manipulated; by using first-order logic it becomes possible for more complex units of information (the foundation for a knowledge base) to be manipulated.

A Knowledge Base Management System (KBMS) is similar in concept to the DBMS. Just as a DBMS is responsible for the storage and management of data (in one or more internal databases), a KBMS is a repository for knowledge which is stored in one or more knowledge bases. In the case of a KBMS, different users and application programs should be able to access the internal knowledge bases it maintains in order to retrieve, store and update knowledge. The KBMS contrasts with the idea of a Knowledge Base (KB), which frequently refers only to a single user system for storing and retrieving knowledge. Such KBs often use rigid knowledge formats appropriate only to the application for which they were designed. If the KBMS is to parallel the DBMS approach the knowledge base should be representation (format)-independent, capable of returning 'knowledge' in a format usable directly by the application that requests it. This suggests that the applications using such a system would be able to access the contents of a KBMS in a high-level, representation-independent manner, and specify the representations to be used for the knowledge returned by the KBMS. Clearly, such a requirement will depend upon agreement as to the useful encoding of knowledge units. At this stage the ideas of KBMSs are still being formulated, and there is no clear theoretical framework on which the representation formats should be based.

Returning to the idea of a logic or deductive database, one obvious factor in designing a query language for such a database is the desirability for it to be an upwardly compatible extension of a relational database query language. In particular, the one relational query language which has emerged as a standard for communicating with relational database systems is SQL,[5]

and compatibility with SQL would clearly be advantageous.

This paper describes some of the design decisions taken in producing a query language for a deductive database system. The language, SQUIRREL, is a natural extension of SQL. The paper examines how the addition of rules to the SQL language requires a number of significant alterations to the SQL philosophy in order to permit a compact, manageable language without compromising the deductive database capabilities. Alternative views of the database contents are considered and the dual nature (termed syntactic and semantic) of a relation introduced. The problem of indexing and cursor use is examined briefly in the light of this dual nature.

Finally, this paper shows how incomplete information has been incorporated into the SQUIRREL language. The underlying database implementation will permit the manipulation of incomplete values under different evaluation strategies; the language has been designed to facilitate these.

2. THE DEDUCTIVE DATABASE POPULATION

In a relational database each tuple consists of a set of explicit values, such as strings, characters, integers, reals, etc. A relation may be represented by a view, which is a projection of values found in other relations and views. As an example consider the database consisting of family details (Table 1) in which a relation **person** is defined by a table and a relation **parent** by a view. A reference to values in a view or a table is always to some value stored explicitly in some table.

Table 1. Relational database population

Relation: person

name	child	age	address
Mary	John	103	Newcastle
Milly	Billy	33	Sunderland
John	Jack	72	Hartlepool
Bill	NULL	22	Newcastle
Jack	Paula	45	Sunderland
Paul	Karen	34	Sunderland

```
VIEW definition
 CREATE VIEW parent(adult, child)
 AS SELECT name, child
 FROM person
 WHERE child IS NOT NULL
```

In a deductive database a relation may be populated with explicit facts and with rules which express the logical relation that holds between values in the database, therefore defining the condition that a tuple must satisfy for its inclusion in that relation.[2] For example, consider a database of family details in which the relation **person** is defined by a set of tuples and the relation **grandparent** by a set of rules and facts (Table 2). For this section the rules are represented as Prolog clauses, while later the SQUIRREL syntax for rules will be introduced. Notice that the relation grandparent is stored as a set of rules and facts, but represents a set of tuples.

Table 2. Deductive database population

Relation: person(name,child,age,address)

```
person(Mary, John, 103, Newcastle).
person(Milly, Billy, 33, Sunderland).
person(John, Jack, 72, Hartlepool).
person (Bill, NULL, 22, Newcastle).
person(Jack, Paula, 45, Sunderland).
person(Paul, Karen, 34, Sunderland).
```

Relation: grandparent(elder,younger)

```
grandparent(Old, Young):-
 person(Old, Middle, _, _),
 person(Middle, Young, _, _).
grandparent(George, Michael).
grandparent(Susan, Michael).
```

A distinction is drawn between the two types of objects that are involved in query evaluation: syntactic and semantic objects. A relation is populated with syntactic objects (rules, facts and views) which define the semantic objects under some resolution strategy. The semantic objects are those tuples produced by the evaluation of the syntactic objects in the database under some inference mechanism. This is illustrated in Table 3. This categorisation is somewhat different from that of Extensional and Intensional databases (EDB and IDB). In the latter case[6,7] a set of closed formulas can be viewed as having two components, the extensional part which contains all the ground unit formulas and the intensional part which contains all the nonground or non-unit formulas. The combined EDB and IDB are equivalent to the set of syntactic objects of the database. It is essential for the database query language to be able to distinguish between the internal storage of the objects and the objects they represent in addition to being able to distinguish between internal objects.

Table 3. Syntactic vs. semantic objects in a deductive database

Syntactic objects

```
person(Mary, John, 103, Newcastle).
person(Milly, Billy, 33, Sunderland).
person(John, Jack, 72, Hartlepool).
person(Bill, NULL, 22, Newcastle).
person(Jack, Paula, 45, Sunderland).
person(Paul, Karen, 34, Sunderland).

grandparent(Old, Young):-
 person(Old, Middle, _, _),
 person(Middle, Young, _, _).
grandparent(George, Michael).
grandparent(Susan, Michael).
```

Semantic objects

```
person(Mary, John, 103, Newcastle).
person(Milly, Billy, 33, Sunderland).
person(John, Jack, 72, Hartlepool).
person(Bill, NULL, 22, Newcastle).
person(Jack, Paula, 45, Sunderland).
person(Paul, Karen, 34, Sunderland).

grandparent(George, Michael).
grandparent(Susan, Michael).
grandparent(John, Paula).
grandparent(Mary, Jack).
```

In a relational database the distinction between syntactic and semantic objects vanishes, and reduces to two different interpretations of the same underlying object. On the other hand, a deductive database management system must provide the facilities whereby a user can manipulate both the syntactic objects stored in the database, and the semantic objects produced by evaluating those stored objects. However, the ways in which these two types of object are manipulated are not fundamentally different: one still wants to select, insert, delete and update syntactic objects in the same way that semantic objects are manipulated in SQL. For example, one may select for display the rules stored in a named relation (excluding any facts stored). One may even wish to combine syntactic and semantic objects in a query. A restriction may be imposed on the syntactic objects (as in the selection above) and, rather than display the resulting rules, one may require the constrained relation to be evaluated in a further subquery.

2.1 Indexing and cursor use in a deductive database

The dual nature of the deductive database population complicates the relational concepts of indexing and cursor use. In a relational database it is the explicit storage of base values that permits indexing and cursor operations. The base representation permits the designation of an ordering of tuples in a relation and the selection of tuples based on positions relative to that ordering. When all tuples in the relations are explicitly present (or referenced through views) this ordering can be achieved, and each tuple has an identifiable next tuple and previous tuple.

However, a deductive database permits the combination of rules and facts to represent a relation. Given a rule which generates a set of tuples, this raises questions such as whether one indexes the rule bodies or the tuples. If one imposes a cursor on a relation containing rules, how does one insert, delete or retrieve tuples relative to that cursor when the cursor currently identifies a tuple generated by a rule? In the light of these difficulties it was felt that an alternative to cursors should be investigated.

2.2 Clause and domain variables

When selecting objects, either for display (`SELECT`) or for update (`DELETE`, `UPDATE`), one must be able to distinguish clearly between the selection of a syntactic or a semantic object as target of the manipulation. To facilitate this, two different types of variables are introduced. These are referred to as clause variables and domain variables, respectively. A clause variable ranges over the syntactic objects stored in a relation, whereas a domain variable ranges over semantic objects and as such is identical to the traditional SQL table and column name reference. The use of these variables will be shown in later sections of this paper.

The domain variable is represented by the SQL column reference of [*table name.*]*column name* or (for the complete relation) *table name*.

The clause variable is formed by adding @ to the front of the table name. Thus, using the database of Table 2, the domain variable **grandparent** will range over the **tuple** set:

```
{(George, Michael), (Susan, Michael),
 (John, Paula), (Mary, Jack)}.
```

whereas the clause variable @**grandparent** ranges over the **clause** set:

```
{grandparent(George,Michael).,
 grandparent(Susan,Michael).,
 grandparent(Old,Young):-
 person(Old,Middle,_,_),
 person(Middle,Young,_,_).}
```

These two sets can be displayed by the statements

```
SELECT*
FROM grandparent
```

and

```
SELECT @grandparent
FROM grandparent
```

3. INSERTING SYNTACTIC OBJECTS INTO THE DATABASE

All objects entered into the database, whether facts or rules, are entered as syntactic objects. This means that the existing SQL insertion of values is seen as inserting facts which are syntactic objects rather than tuples which are semantic objects. Apart from this reclassification of the inserted values, no other alternation is made to the existing insertion statement of SQL.

The major extension to the data manipulation language lies in the statements for the insertion of rules into the database. A number of alternative representations for the rule syntax were examined. They fall broadly into two categories, those that retain the SQL reference by name to manipulate values and those that introduce variables for use in the rule body.

3.1. Reference-by-name approach

SQL is a reference-by-name language, that is, a language in which each data reference has an explicit (and universal) name. For example, the **student** relation (Table 5) contains a column **age** which is referenced as *student.age* or simply as *age* when no ambiguity can arise.

If this reference-by-name philosophy is maintained, it leads to unwieldy rule bodies. To illustrate this, consider the following examples.

Table 4. Examples of possible clause restriction representations

RESTRICT Statement

```
SELECT elder, age
FROM person, grandparent
RESTRICT rules (@grandparent)
WHERE age > 40
```

Clause Restriction Statement

```
SELECT elder, age
FROM person, grandparent: rules
WHERE age > 40
```

Using the relations in Table 2, restrict the grandparent relation to rules only and select the names and ages of the grandparents with ages over 40.

Table 5. Database schema

Relation	Column names				
staff	name	age	sex	dept	grade
student	name	sex	age	year	dept
marks	dept	name	num	mark	
record	dept	name	total_courses	avg_mark	
result	dept	name	grade		
teaching_staff	name	dept			
research_staff	name	dept			
ancestor	anc	man			
father	name	dad			

Table 6. Extended cartesian product of parent and benefit relations

parent		benefit	
name	**child**	**name**	**amount**
mark	william	william	10
mark	william	simon	20
mark	william	tom	30
mark	william	mary	40
mary	simon	william	10
mary	simon	simon	20
mary	simon	tom	30
mary	simon	mary	40
mark	tom	william	10
mark	tom	simon	20
mark	tom	tom	30
mark	tom	mary	40

(1) Assuming relations **ancestor** and **father** are defined as *ancestor*(*person*,*elder*) and *father*(*dad*,*son*), the ancestor rules can be inserted as:

```
INSERT INTO ancestor
USING father, ancestor
DEFINE ancestor IF SELECT son, dad
 FROM father
DEFINE ancestor IF SELECT son, elder
 FROM father, ancestor WHERE dad=person
```

The rule body has the form of a select statement to allow multiple assignment to columns in the target relation. The alternative is to have explicit assignments to *ancestor.person* and *ancestor.elder* (*ancestor.person* = *father.son* and *ancestor.elder* = *father.dad* in the first rule body). This would inhibit the reading of a rule as an assignment to a tuple.

(2) Using the **research_staff, teaching_staff** and **staff** definitions in Table 5, the definition of **research_staff** as a rule can be inserted as:

```
INSERT INTO research_staff(name, dept)
USING staff, teaching_staff
DEFINE research_staff IF
  SELECT staff.name, staff.dept
  FROM staff
  WHERE staff.grade <> 'technician' AND
  NOT EXISTS (SELECT name
    FROM teaching_staff
    WHERE staff.name = teaching_
     staff.name AND staff.dept =
     teaching_staff.dept
  )
```

3.2 Prolog-like approach

An alternative approach for the rule formats is to base these on the logic programming language Prolog. Prolog is a reference-by-position language, that is, one which permits a variable to be declared which takes its value from some positional information supplied when it is used. For example, *student*(_,_,*Age*,_) would associate the variable *Age* with the values found in the third argument position of the **student** tuples.

Using this approach, the above examples can be coded in Prolog in the following manner.

(1) The Prolog definition of **ancestor,** assuming *ancestor*(*elder*,*younger*) and *father*(*dad*,*son*) are defined as for Example 1 above, is as follows:

```
ancestor(X, Y):- father(X, Y).
ancestor(X, Y):- father(X, Z),
 ancestor(Z, Y).
```

(2) The Prolog definition of **research_staff,** assuming the schema defined in Table 5, is

```
research_staff(N, D):-
 staff(N, _, _, D, G),
 not (G = 'technician'),
 not (teaching_Staff(N, D)).
```

However, this approach is possible only when the number of columns in a relation and their ordering is known. Clearly in a relational database language this cannot be guaranteed. A user may not have authorized access to all columns of a relation, may not know in what order the columns were defined or what the names of all the columns are; or a relation may be unwieldy in this form, having a large number of columns of which only a few are of interest.

3.3 Intermediate approach

A solution to the difficulty of using positional representations in a relational language which takes advantage of the shorthand offered by the use of variables and unification can be found in SQL. The existing SQL INSERT statement permits the use of an insert column list to define the target columns and the order in which

they will be referenced in the query. There is then no need to use all columns or to depend on some default ordering. The insert column list can be used to override the default information.

Example:

```
INSERT INTO staff(name,age,grade)
VALUES ('clare',23,'technician')
```

Note that columns not referenced in the INTO statement default to the NULL value provided that the relation permits this; the column names that are referenced can be in any order. Thus if the relation actually contains two further fields, *sex* and *dept*, this would be interpreted as if it were the statement:

```
INSERT INTO
 staff(name,age,sex,dept,grade)
VALUES
 ('clare',23,NULL,NULL,'technician')
```

In SQUIRREL the two reference systems have been combined by permitting the declaration of source and target columns in a reference-by-name manner; this declares the order of those columns for the reference by position employed in the rule bodies. In order to allow the definition of source tables for the rules, a USING clause is added to the INSERT statement. The following examples show how the INSERT USING statement is employed to insert rules into a relation using both reference by name and reference by position. The first example shows how the declaration of source and target relations can be used to override the default order of the columns of the relations used. The second shows how the number of columns used in the rule definition can be restricted to those actually used in the rule bodies. The final example shows how correlation names can be employed, in this instance for the insertion of a recursive rule. (The schema description of the tables used in these examples can be found in Table 5.)

Examples:

(1) Insert rules into the **result** relation.

```
INSERT INTO result(grade,name,dept)
USING   record(dept,name,total_courses,
         avg_mark)
        DEFINE result('excellent',D,S,)
         IF record(D,S,N,A) and N>3
         and A>=80
        DEFINE result('good',D,S)
         IF record(D,S,N,A) and N>3
         and A<80 and A>=60
        DEFINE result('pass',D,S)
         IF record(D,S,N,A) and N>3
         and A<60 and A>=40
        DEFINE result('fail',D,S,)
         IF record(D,S,N,A) and N>3
         and A<40
        DEFINE result ('incomplete',D,S)
         IF record(D,S,N,A) and N<=3
```

(2) Insert a rule defining the **research_staff** relation.

```
INSERT INTO research_staff(name,dept)
USING  staff(name,dept,grade),
        teaching_staff
       DEFINE research_staff(N,D) IF
        staff(N,D,G) and G<>'technician'
        and not teaching_staff(N,D)
```

(3) Insert the recursive rule pair for the **ancestor** relation.

```
INSERT INTO ancestor(anc,man)
USING  father(name,dad),
        ancestor(man,anc) anc1
       DEFINE ancestor(person,elder)
        IF father(person,elder)
       DEFINE ancestor(person,elder)
        IF father(person,parent) and
        anc1(parent,elder)
```

A number of constraints are imposed on the rule bodies, including some on the use of variables in the INSERT and USING statements. The following restrictions apply:

(1) Variables must agree both in position and type with the lists of columns appearing in the relation descriptions in the INSERT and USING statements.

(2) All variables must have a defining use, that is one in which the value of a variable can be bound to a constant. A defining use declares the variable name and (due to its positional use) a type. A defining use of a variable occurs in some primitives and under certain conditions. The primitives permitting a defining use are assignment and use in an argument position in a table reference. Terms of the form, *table_name*(*argument list*), offer defining use for all variables used in the argument list (unless already defined). All variables must agree in type and number with the column types declared in the USING list. Use of a variable in the head of a rule is not a defining use as the head of a rule is considered to be the target of the variables. This prevents unbound variables appearing in the head of a rule and hence in a column of the relation.

(3) Assignment (⟨*var*⟩? ⟨*value expr*⟩) is non-destructive; that is, it assigns the result of the ⟨*value expr*⟩ to the variable on the left of the operator (?) which should not have previously been used in a defining position.

(4) A comparison predicate does not offer a defining use. For example, the comparison expression, $Var < 10$, would offer a set of values for *Var* if it were not already bound to a value.

(5) In a rule body containing a disjunction the defining use of a variable is global if and only if it occurs in both branches of the disjunction and the defining use agrees with respect to type. For example, in '*parent*(*X*,*Y*) *OR grandparent*(*X*,*Z*)', *X* is a defining use if the types of the first columns in parent and grandparent agree. *Y* and *Z* are not global defining uses.

(6) No variables can be defined globally under negation. If a defining use does occur within a negation, this is treated as a local defining use. For example, in '*NOT*(*parent*(*X*,*Y*) *AND* $Y > 10$)', *X* and *Y* are treated as local defining uses. Furthermore, if the negation appears in '*grandparent*(*X*,*Z*) *AND NOT* (*parent*(*X*,*Y*) *AND* $Y > 10$)', *Y* cannot be used elsewhere in the rule, whereas *X* may be used, as its defining use occurs in the argument list of grandparent, which is not negated.

4. QUERY LANGUAGE

Since the query language in SQUIRREL must permit the retrieval of both syntactic and semantic objects, this requires the ability to distinguish between these two aspects of a relation. Furthermore, it is useful to be able

to constrain syntactic objects before a relation is evaluated. This is achieved with the clause restriction.

Clause and domain variables are used to indicate the type of result required from the SELECT statement. However, a few constraints must be imposed on the use of clause and domain variables. In the select list of the SELECT statement clause and domain values cannot be mixed, since it does not make sense to permit the mixing of rules and the objects they generate. Similarly, it would be difficult to select clause–variable objects if the table expression produced domain-variable values, so that the use of a clause variable in the select list is not permitted when a table expression (WHERE, GROUP BY and HAVING clause) is present in the query.

In this section of the paper consideration is given to the syntax for manipulation of the syntactic population of the database, before showing how to form queries which return the syntactic objects. The evaluation of a query over a constrained relation is shown and an explanation given of how syntactic constraints and semantic conditions can be combined in a single query. It remains possible to pose typical relational queries, which select semantic objects from the database as if it were a standard relational database.

4.1 Restricting the syntactic objects

Just as the SQL WHERE clause selects semantic objects on the basis of some specified condition, one needs to be able to select syntactic objects on the basis of syntactic properties. This can be achieved by a clause restriction; this either takes the form of a separate clause, the RESTRICT clause (which is similar to WHERE, GROUP BY, etc.) or by adding the restriction to the FROM clause.

The two forms for restricting objects on the basis of their syntactic structure are illustrated in Table 4. While the first (RESTRICT clause) approach appears most SQL-like there are a number of interpretation problems which arise if one needs to restrict more than one relation from within a single clause. The second form removes these problems while strengthening the concept of the FROM clause as the source of the syntactic objects used in the evaluation of the user query.

There are two basic problems with the RESTRICT clause: inter-relation dependencies and conditions involving independent variables.

(*a*) *Inter-relation dependencies.* The following description of the dependency problem gives some background on the SQL-evaluation of inter-relation dependencies and shows how this would result in problems if it were carried into the clause restriction.

The SQL-processing of the FROM list involves the production of the extended cartesian product of the relations. So that for each condition in the table expression the values in the extended cartesian product can be used to examine dependency information. For example, consider the relations *parent*(*name*,*child*) and *benefit*(*name*,*amount*) with the values:

```
parent(mark, william).
parent(mary, simon).
parent(mark, tom).

benefit(william, 10).
benefit(simon, 20).
benefit(tom, 30).
benefit(mary, 40).
```

and the query:

```
SELECT   parent.name, amount
FROM     parent, benefit
WHERE    child = benefit.name
```

which lists the benefit amounts for each child and the person to whom it is payable.

The FROM list is evaluated to give the extended cartesian product of the two relations; this is shown in Table 6. The evaluation of the WHERE condition is then performed by examining each row in the cartesian product table to produce the constrained result (Table 7). The first and fourth columns of Table 7 are projected to give the result of the query.

The effect of the inter-relation dependencies is best demonstrated by examining the result of the (badly formed) query:

```
SELECT   parent.name
FROM     parent, employee
WHERE    parent.child IS NOT NULL
```

which produces the results shown in Table 8.

If this were paralleled in the clause restriction it would greatly increase the complexity of the subsequent evaluation of a constrained relation. Consider that the RESTRICT clause is intended to constrain the syntactic objects in the database to allow subsequent evaluation of the constrained rule set. A clause condition statement modelled on the above evaluation strategy would not result in a single set of syntactic objects for each relation, but a collection of sets of syntactic objects.

Consider the following query fragment:

```
FROM       P, Q
RESTRICT   rules(P) and rules(Q) and
                @P.a = @Q.x
```

where P is defined as having columns a and b; Q as having columns x and y.

The intention of the RESTRICT statement is to apply restrictions to each **set** of clauses (P and Q). However, the above fragment appears to impose a dependency condition between the sets (that of equality between two columns). Thus, given the initial clauses $P(1,2)$:–$A(1,2)$, $P(2,3)$:–$A(2,3)$, $P(3,4)$:–$A(3,4)$, $Q(1,2)$:–$B(1,2)$, $Q(2,3)$:–$B(2,3)$ and $Q(3,4)$:–$B(3,4)$ the FROM statement would produce the cartesian product shown in Table 9.

The RESTRICT statement would then eliminate the clauses not satisfying the dependency condition and would produce the full six clauses, but paired into the sets:

```
{P(1,2):–A(1,2), Q(1,2):–B(1,2)}
{P(2,3):–A(2,3), Q(2,3):–B(2,3)}
{P(3,4):–A(3,4), Q(3,4):–B(3,4)}
```

Table 7. Result of applying the WHERE condition to Table 6

parent		benefit	
name	**child**	**name**	**amount**
mark	william	william	10
mary	simon	simon	20
mark	tom	tom	30

Table 8. Result of the badly formed query

parent name
mark
mark
mark
mark
mary
mary
mary
mary
mark
mark
mark
mark

Table 9. Cartesian product produced by RESTRICT clause

P	Q
P(1,2):–A(1,2)	Q(1,2):–B(1,2)
P(1,2):–A(1,2)	Q(2,3):–B(2,3)
P(1,2):–A(1,2)	Q(3,4):–B(3,4)
P(2,3):–A(2,3)	Q(1,2):–B(1,2)
P(2,3):–A(2,3)	Q(2,3):–B(2,3)
P(2,3):–A(2,3)	Q(3,4):–B(3,4)
P(3,4):–A(3,4)	Q(1,2):–B(1,2)
P(3,4):–A(3,4)	Q(2,3):–B(2,3)
P(3,4):–A(3,4)	Q(3,4):–B(3,4)

This would require evaluation of 'each' set of clauses to produce the exhaustive set of semantic objects for the clauses satisfying the dependency. The effect of this is not easy to predict, nor is it easily implementable. Thus, any inter-relation dependency information does not introduce a useful restriction on the clause sets. For this reason the clause restriction operates on clauses taken one at a time from the source clauses. This prevents dependency information such as explicit equality being used in the clause restriction.

(*b*) *Independent variables.* If the RESTRICT clause is constrained to conditions with independent variables (that is, inter-relation dependencies are prevented), the RESTRICT clause can be written as either

```
FROM P,Q
RESTRICT F(P) or G(Q)
```

or

```
FROM P,Q
RESTRICT F(P) and G(Q)
```

where F and G are conditions over P and Q respectively. Since F and G are independent and are evaluated for each single-source clause taken from P and then Q, the conjunction of $F(P)$ and $G(Q)$ cannot be satisfied. For this to occur the source clause would need to be a syntactic object in relation P and in relation Q. With independent conditions only the disjunction can be permitted.

For these reasons the RESTRICT clause would require a complex formulation of syntax rules to ensure the above code fragments did not occur. Notice in Table 4 that in the augmented FROM statement it is impossible to contravene the above requirements. The source relation and its restriction are bound tightly, which prevents interdependencies and forces the separation of independent conditions.

The clause restriction is used in the delete and update statements as well as in selection for display purposes. Examples of these statements are given in a later section.

4.2 Clause restriction predicates and functions

A new set of functions and predicates applicable to syntactic objects in the database is required. The following gives an indication of the form that these might take.

4.2.1. Predicates

A useful set of predicates might include:

facts: true for those clauses which are facts.

rules: true for those clauses which are rules.

recursive: true for those clauses which can lead to a recursive call of the relation, under a static search of the call tree.

uses(relation name): true for those clauses which include a call to the named relation.

match(string): true for those clauses that contain the named string in the head or body.

var(column name): true for those clauses with a variable in the named column position in the object head.

null(column name) or **column name is [not] null**: true for those clauses with null in the named column position in the head of the rule.

4.2.2. Functions

The function set should include:

head(clause variable): returns the head structure of the clauses selected by the clause variable.

body(clause variable): returns the body of the clauses selected by the clause variable.

count_clauses(clause variable): returns a count of all syntactic objects represented by the clause variable.

count_rules(clause variable): returns a count of the rules represented by the clause variable.

count_facts(clause variable): returns a count of the facts represented by the clause variable.

4.3 Selecting syntactic objects

Consider how syntactic objects can be retrieved from the database. The clause variable and the clause restriction can be combined to constrain a relation population and then display the constrained set of syntactic objects.

The simplest retrievals are those without any clause restriction imposed on the selection. This is comparable with the 'select*from relation' of SQL. The objects returned are the representations of the syntactic objects stored under a given relation name. An example is:

```
SELECT   @student
FROM     student
```

Similarly, one can apply the functions mentioned earlier to the result of the SELECT statement, e.g.

```
SELECT   count_rules(@student)
FROM     student
```

A retrieval may restrict the clauses chosen before display. An example of this is

```
SELECT   @result
FROM     result: grade = 'good'
```

which selects the objects in which the named column position (*grade*) in the head of the clause has the specified value.

The restriction may be more complex than this, e.g.

```
SELECT   @result
FROM     result: grade = 'good' and
          var(dept)
```

in which the *dept* column position in the head must be occupied by a variable and the *grade* position must have the value '*good*'.

Notice (as stated earlier) that since the target of the above queries is a clause variable one cannot have a semantic condition (table expression) as this would result in the evaluation of the syntactic objects.

4.4 Evaluation under the clause restriction

As mentioned previously, it is useful to be able to evaluate a relation which is constrained to syntactic objects satisfying a clause restriction condition. That is, one may constrain a relation (as shown above) and then require that the constrained relation is evaluated to produce semantic objects to be used in a table expression.

In implementing the clause restriction one must ensure that the restriction does not lead to side-effects in other aspects of the query. Thus one must ask how restricting the set of syntactic objects for a particular evaluation of a relation affects other relations that depend on the values produced by the restricted relation. Two alternative approaches are the following.

(1) Temporarily remove from the database all clauses that do not satisfy the restriction. Consider a database DB and a query $Q(p,e)$ (i.e. the query is over relations p and e). If Q restricts the relation p to p' then $Q(p,e)$ is evaluated over the database $DB-p+p'$. Hence, if e contains a procedure using relation p, the evaluation of e is restricted to those clauses that use the clauses in p'.

(2) Create temporary relations and fill these with the restricted clauses. Consider a database DB and the query $Q(p,e)$. If Q restricts p to p', then copy p' into the database in a marked form so that $DB' = DB+p'$. Now rename the explicit use of relation p in Q to p', i.e. $Q(p,e) \rightarrow Q(p',e)$ and execute this query over the database DB'. After the execution of the query, remove p' from the database, i.e. $DB' \rightarrow DB$. Now, if e contains a rule which uses relation p it is able to use all of relation p contained in the database.

For example, if the initial database consists of Prolog clauses:

```
ancestor(X,Y) if father(X,Y).
ancestor(X,Z) if father(X,Y),
 ancestor(Y,Z).
father(....etc.
```

Under the first implementation a query *ancestor*(A,B) in which the relation *ancestor* is restricted to only recursive clauses, would result in the evaluation of the query *ancestor*(A,B) over the database:

```
ancestor(X,Z) if father(X,Y),
 ancestor(Y,Z).
father(....etc.
```

Under the second implementation the same query would result in the evaluation of *ancestor_marked*(A,B) over the database:

```
ancestor_marked(X,Z) if
 father(X,Y), ancestor(Y,Z).
ancestor(X,Y) if father(X,Y).
ancestor(X,Z) if father(X,Y),
 ancestor(Y,Z).
father(....etc.
```

The second implementation strategy has been employed in this project for the following reasons.

(*a*) Recursion remains safe under this interpretation. Suppose that under the Prolog resolution strategy the example database above will execute the query *ancestor*(A,B) correctly and will terminate correctly. If the first implementation were used to constrain the database, the *ancestor* query would not terminate. Under the second implementation the query will be executed and terminate correctly.

(*b*) In the database one does not want the effects of a restriction on the syntactic objects of one relation to alter the semantic objects in other relations. That is, in the example query above one does not want a restriction on p to affect the semantic objects produced by evaluating e. This is similar to an SQL query in which a view over a base table and the base table itself are included in a query. If a *where* statement imposes a condition on the base table the view is not affected.

4.5 Selecting semantic objects

Having seen how a constrained relation can be evaluated, consider how domain variables, clause restriction and the SQL table expression can be combined in the retrieval of semantic objects (tuples).

In these cases one wants the tuple sets of traditional relational database queries to be returned by a query. The simplest example is that of a normal SQL query such as

```
SELECT     name, mark, dept
FROM       marks
GROUP BY   name,dept,mark
HAVING     max(mark) = mark
```

No distinction is necessary when using tables which are defined by rules, for example

```
SELECT   name,depth
FROM     result
WHERE    grade = 'excellent'
```

From the preceding section it has been seen that SQUIRREL will permit a query to be evaluated over a set of clauses taken from a relation definition by using a clause restriction. An example is

```
SELECT   name
FROM     result: facts
WHERE    grade = 'excellent'
```

This restricts the evaluation of the query to those syntactic objects which are facts (it excludes any rules used to define the relation **result**).

This can be used to calculate the restricted **ancestor** relation described in the database population section above.

```
SELECT   anc
FROM     ancestor: uses(ancestor)
```

or

```
SELECT   anc
FROM     ancestor: recursive
```

The use of the clause restriction means that some queries can be expressed in two ways, using the clause restriction or using the WHERE clause. For example, the query

```
SELECT   name, dept
FROM     result: grade = 'incomplete'
```

has the same result as the query

```
SELECT   name, dept
FROM     result
WHERE    grade = 'incomplete'
```

However, in the second case the entire relation for **result** must be generated and examined, tuple at a time, in order to compare the grade with the constant '*incomplete*'. In the first case, only those objects that will generate the constant '*incomplete*' in the *grade* position will be used, hence no extra tuples are generated then excluded. (Obviously this depends on the *grade* position being filled with constants, a slightly safer but less efficient approach would be to include the rules which have variables in the *grade* position. That is, `SELECT name, dept FROM result: grade = 'incomplete' or var(grade).`)

5. OTHER OPERATIONS

The changes to the data definition language and the delete and update statements are introduced in this section. The problems of the update of semantic objects are considered, and parallels are drawn with the difficulty of update through relational views.

5.1 Data definition language

The CREATE TABLE statement is used to define a table. The only change to this statement is the addition of an optional restriction which permits a table to be defined WITHOUT RULES. An example (taken from the database schema in Table 5) is:

```
CREATE TABLE staff(
  name char(20) NOT NULL UNIQUE,
  age integer NOT NULL,
  sex char(1),
  dept char(5),
  grade char(20) NOT NULL
  ) WITHOUT RULES.
```

The main reason for this is for use in optimization processes which are applicable during query evaluation.

5.2 Deletion

There are two possible forms for the deletion statement, either deleting a syntactic object or deleting a semantic object.

(*a*) For the purposes of deleting a syntactic object from a relation, the relation is identified using the clause variable and then the optional clause restriction is applied to restrict the deletion to those clauses one actually wants to remove. To illustrate this, consider the following examples.

(1) Delete all syntactic objects in the **result** relation,

```
DELETE FROM @result
```

(2) Delete all *facts* from the **result** relation,

```
DELETE FROM @result: facts
```

(3) Delete all *rules* from the **research_staff** relation that use the **student** relation and the *facts* that have NULL in the *dept* position.

```
DELETE FROM @research_staff:
 (rules and uses(student)) or
 (facts and null(dept))
```

A semantic condition (WHERE clause) cannot be applied if the target of the deletion is a clause variable. The reason for this is the same as the reason that the table expression statement is not permitted if the target of the select statement is a clause variable.

(*b*) The semantic objects for deletion can be identified using a combination of domain variable, clause restriction and semantic condition (WHERE clause). The syntax for SQUIRREL permits the expression of statements to delete semantic objects. Some examples are considered below.

(1) Delete all semantic objects from relation **result**.

```
DELETE FROM result
```

(2) Delete the semantic objects from relation **result** which have a *grade* column value of '*fail*'.

```
DELETE FROM result
WHERE  grade = 'fail'
```

(3) Delete all semantic objects that are generated by the *facts* for the relation **result**.

```
DELETE FROM result: facts
```

(4) Delete the semantic objects that are generated by *rules* and have a *grade* column value of '*pass*'.

```
DELETE FROM result: rules
WHERE  grade = 'pass'
```

5.3 Update

Updates to the contents of a relation can similarly be divided into those with syntactic or semantic targets.

(*a*) The update of syntactic objects is identified by the use of the clause variable as target for the update. As with select and delete, one cannot have a semantic condition if the target of the update request is a syntactic object. Rule update is currently performed using a simple string-replacement operation. Thus, for a given clause the operation **replace(string1,string2)** will replace all occurrences of *string1* in the clause body with *string2*. Examples include,

(1) Update the *facts* in relation **student** replacing the *age* of all '*kevin's* with the *null value*,

```
UPDATE   @student: facts and
          name = 'kevin'
SET      age = NULL
```

(2) Update the *rules* in **research_student** that use the relation **student**, replacing the '*student*' string with '*staff*',

```
UPDATE   @research_student:
          rules and uses(student)
SET      replace ('student', 'staff')
```

(*b*) Semantic updates are identified by the use of the domain variable as target. The SQUIRREL syntax permits the statement of queries that update semantic objects. The problems of implementing such updates are considered in the next section. Examples include,

(1) Update all semantic objects in relation **student** replacing the *age* field with *NULL*.

```
UPDATE   student
SET      age = NULL
```

(2) Update the semantic objects in relation **student** which have the *name* field equal to '*kevin*' setting the *age* field to 23.

```
UPDATE   student
SET      age = 23
WHERE    name = 'kevin'
```

(3) Update the semantic objects generated by the *facts* in the syntactic object set for the relation **student** setting the *dept* field to '*remedial*'.

```
UPDATE   student: facts
SET      dept = 'remedial'
```

(4) Update the semantic objects generated by *rules* for the **student** relation that have the *name* field equal to *kevin*, setting the *age* value to 23.

```
UPDATE   student: rules
SET      age = 23
WHERE    name = 'kevin'
```

5.4 Update and deletion of semantic objects

The use of views in relation databases leads to problems in updating and deleting tuples in relations represented as views. In a deductive database this problem is exaggerated by the generality permitted in the rule bodies.[8] This problem is partly ascribable to the fact that in logic the world is static.[9] That is, once a rule is defined there can be no exceptions to that rule at a later instant.

Consider the following syntactically valid request to delete a semantic object generated by a relation which contains rules:

```
DELETE FROM grandparent
WHERE elder = 'kevin'
```

There are two possible ways in which tuples containing '*kevin*' can be generated: either the tuple appears in the set of facts in the syntactic set, or it appears in the set of semantic objects generated by a rule in the syntactic set. In the first instance an implementation can delete the corresponding facts from the syntactic set. In the second case (as has already been identified in respect of the indexing problem), the rule generating the tuple cannot be removed as this action will have the side-effect of removing all other semantic tuples generated by the rule. The same arguments apply to update operations.

The question then arises as to the interpretation of a request to delete a semantic object. Should it result in an alteration to the syntactic objects so that the rules when evaluated cannot generate the deleted tuple, or should this form of deletion be forbidden (allowing only deletion or update of syntactic objects)? The first of these suggestions is unsatisfactory, as any implementation that achieves this effect would need to be free from side-effects and would lead to the possibility of inefficient rules being created automatically without the database user being aware of this. The second is also unsatisfactory from the user's perspective, since if one wants to delete a tuple one should not need to be aware of how that tuple is generated internally.

In fact a preferred reading of such a deletion request would be to create facts in the set of syntactic objects which are interpreted as positive exceptions to the relation, that is, as tuples that the rules cannot generate but which should be included in the relation. The request to delete a tuple would then be interpreted as creating an exception to the relation, that is, a tuple which can be generated by the syntactic set but which should not be included in the relation.

This suggests that the problem of a general update to deductive databases parallels the negative information problem. Much research has been conducted on the representation of negative information in databases. Gallaire[9] summarizes this research, but the results reported are generally difficult or expensive to implement.

6. INCOMPLETE INFORMATION IN SQUIRREL

In many database applications a value of an attribute might be unknown. A special value called **null** is introduced to represent the missing value.[10] In some situations one may have some partial information about an attribute even though its exact value is unknown. If the null value is used for this case the information one does have will be discarded. To handle this problem the concept of **incomplete information** is introduced. This kind of value falls between exact values and the null value. The handling of incomplete information in relational databases has been studied by many researchers.[11,12,14,17]

The manipulation of incomplete information in a logic database is much more difficult than it is in a relational database, because the complexity of deductive operations is substantially increased.[18] A model has been put forward by Williams and Kong[15] to account for the behaviour of incomplete information in a logic database.

In this model, a student called *John Smith*, whose age is not known exactly but is known to be one of 18, 19 or 20, can be expressed as:

```
student('John Smith', {18, 19, 20}).
```

which is equivalent to the following expression

```
student('John Smith', 18) OR
student('John Smith', 19) OR
student('John Smith', 20).
```

The implementation problems associated with incomplete information in a logic database have been discussed in Refs 13 and 16.

6.1 Inserting incomplete constants

The values that can be stored in a deductive database are expanded to include incomplete values, and the manipulation strategies applicable to the incomplete values are similarly expanded. The evaluation strategy employed for incomplete information can be expensive in both memory usage and execution time. For this reason the algorithm is capable of selectively applying the evaluation strategy. If, for instance, no incomplete information is employed, the normal deductive database evaluation strategy is applied. To permit maximum use of this conditional strategy the definition of a column in a relation will default to complete information. A database designer will be required to specify that a column in a relation may include incomplete information when defining the relation. The following example shows this.

Define a table for relation **staff** which is declared to contain no rules but may have incomplete information in attributes *age* and *dept*.

```
CREATE TABLE staff(
  name char(2) NOT NULL UNIQUE,
  age integer NOT NULL WITH INCOMPLETE,
  sex char(1) NOT NULL,
  dept char(20) WITH INCOMPLETE,
  grade char(20) NOT NULL
  ) WITHOUT RULES
```

The insertion of incomplete values into the above table is demonstrated in the following example:

```
INSERT INTO staff(name, age, sex, dept, grade)
VALUES ('Bill Stone',{31, 32, 33},
'm',{'Physics', 'Chemistry'}, 'Lecturer')
```

A rule may contain incomplete information, for example: a statement that *a child has blood group A or O if one of his parents has blood group A and the other has blood group O* can be expressed as:

```
INSERT   blood_group(name, type)
USING    father(name, child),
 mother(name, child),
 blood_group(name, type) bg
DEFINE   blood_group(NameC,{'A', 'O'}) IF
             father(NameF, NameC) AND
             mother(NameM, NameC) AND
             bg(NameF, 'A') AND
             bg(NameM, 'O')
DEFINE   blood_group(NameC,{'A', 'O'}) IF
             father(NameF, NameC) AND
             mother(NameM, NameC) AND
             bg(NameF, 'O') AND
             bg(NameM, 'A')
```

6.2 Manipulating incomplete values

The evaluation strategies defined in Ref. 16 allow two types of interpretation to be employed in query evaluation. These are *definite* and *possible* evaluation. To these modes is added a default evaluation strategy which treats all incomplete values as if they were the null value. The user is able to control the type of evaluation strategy employed in a similar way to the ALL and DISTINCT selections of SQL which either take all values in a result or restrict the result to unique values. The query and subquery specifications are given an optional DEF or POS clause to specify the form of evaluation strategy to be employed. Examples include:

(1) Select tuples in the **staff** relation which definitely have the value 21 in the *age* column,

```
SELECT DEF*
FROM staff
WHERE age = 21
```

(2) Select tuples which possibly have the value 21 in the *age* column,

```
SELECT POS*
FROM staff
WHERE age = 21
```

(3) Select the tuples with 21 in the *age* column, treating incomplete information in the *age* column as if it were a *NULL* value,

```
SELECT*
FROM staff
WHERE age = 21
```

A new predicate is added for use in the where clause of the table expression: the incomplete data predicate. This is true if the column referenced contains a value that is in the incomplete data item or is a subset of that data item.

(4) Select tuples where the *age* value is wholly contained in the incomplete constant {21,22,23,24},

```
SELECT DEF*
FROM staff
WHERE age IS {21, 22, 23, 24}
```

Since for update and deletion queries the contents of the database are evaluated, it is necessary for the user to specify which evaluation strategy is employed in these statements. This is achieved by including the option DEF or POS (or default) in the update and delete statements. The update statement also permits an incomplete constant to be used in the set clause. Examples show how this is used:

(1) Delete all possible staff members whose *age* may be above 65:

```
DELETE FROM POS staff
WHERE age > 65
```

(2) Update the staff members who are definitely in the Physics department making them all technical *grade*,

```
UPDATE DEF staff
SET grade = 'technical'
WHERE dept = 'Physics'
```

(3) Update all staff members making those with *ages* possibly below 21 have the *grade* {'*junior*','*apprentice*'},

```
UPDATE POS staff
SET grade = {'junior', 'apprentice'}
WHERE age < 21
```

7. SUMMARY

The SQUIRREL language presented in this paper is a variant of SQL, designed for access to a deductive database. It permits the definition and manipulation of relations containing logic statements. The distinction

between the two interpretations of the database populations (syntactic and semantic) is made, and language features to distinguish between the database populations described. Examples of the definition, insertion and manipulation of syntactic objects are given. A brief description is given of the implementation strategy that permits syntactic and semantic manipulations to be combined in a query. The logic database implementation permits the expression and manipulation of incomplete information. Examples are given which show how such information is incorporated in the SQUIRREL language.

Acknowledgement

The authors wish to acknowledge the support received from the Science and Engineering Research Council, who are funding the project under the Logic Programming Initiative of the Alvey Programme, and from Mr J. R. Lucking of STC, who is industrial uncle to the project.

REFERENCES

1. E. F. Codd, A relational model of data for large shared data banks. *Communications of the ACM* **13**, 377–387 (1970)
2. H. Gallaire, J. Minker and J. M. Nicholas, An overview and introduction to logic and databases. In *Logic and Databases*, edited H. Gallaire and J. Minker, pp. 3–30. Plenum, New York (1978).
3. J. W. Lloyd and R. W. Topor, A basis for deductive database systems. *Journal of Logic Programming*, no. 2, pp. 93–109 (1985).
4. J. W. Lloyd and R. W. Topor, A basis for deductive database systems. II. *Journal of Logic Programming*, no. 1, pp. 55–67 (1986).
5. ISO, Final Draft ISO 9075-1987(F), *Database Language SQL*. International Standards Organization (1986).
6. R. Reiter, Deductive Q-A on relational databases. In *Logic and Databases*, edited H. Gallaire and J. Minker, pp. – . Plenum, New York (1978)
7. R. Reiter, On closed world databases. In *Logic and Databases*, edited H. Gallaire and J. Minker. Plenum, New York (1978).
8. S. Manchanda and D. S. Warren, A logic based language for database update. In *Foundations of Deductive Databases and Logic Programming*, edited J. Minker, pp. 363–394. Morgan Kaufmann, New York (1988).
9. H. Gallaire, Bridging the gap between AI and databases: logic approach. Proceedings of the IFIP TC2 *Working Conference on Knowledge and Data (DS-2), Aldeia das Acoteias, Portugal* (1986).
10. E. F. Codd, Extending the database relational model to capture more meaning. *ACM Trans. on Database Systems* 4 (4), 397–434 (1979).
11. J. Grant, Partial values in a tabular database. *Information Processing Letters* 9 (2), 97–99 (1979).
12. M. Gray, Views and imprecise information in databases. *Ph.D. Thesis, Cambridge* (1982).
13. Q. Kong and M. H. Williams, Evaluating different strategies for handling incomplete information in a logic database. *Workshop on Logic Programming – Expanding the Horizons.* Imperial College (1989).
14. W. Lipski, Informational systems with incomplete information. In *Proc. 3rd Int. Symp. on Automata, Languages and Programming*, edited S. Michaelson and R. Milner, pp. 120–130. Edinburgh University Press, Edinburgh (1976).
15. M. H. Williams and Q. Kong, Incomplete information in a deductive database. *Data and Knowledge Engineering* **3**, 197–220 (1988).
16. M. H. Williams, Q. Kong and G. Chen, Handling incomplete information in a logic database. *Proceedings of UK IT 88 Conference*, pp. 224–227 (1988).
17. M. H. Williams and K. A. Nicholson, Implementation of incomplete information in databases. *The Computer Journal* **31** (2), 133–140 (1988).
18. A. Yahya and L. J. Henschen, Deduction in non-Horn databases. *Journal of Automated Reasoning* **1**, 141–160 (1985).

Announcements

Applied Intelligence: The international journal of artificial intelligence, neural networks, and complex problem-solving technologies

Call for Papers

Editor-in-Chief: **Moonis Ali**, The University of Tennessee Space Institute, Tullahoma, TN 37388, U.S.A.

This journal will address issues involving solutions of real-life manufacturing, defence, management and industrial problems which are too complex to be solved through conventional approaches and which require the simulation of intelligent thought processes, heuristics, applications of knowledge, and distributed and parallel processing. The integration of these multiple approaches in solving complex problems will be of particular importance.

The objective of *Applied Intelligence* is to provide a medium for exchanging scientific research and technological achievements accomplished by the international community – academia, industry and government – in the areas associated with intelligent systems.

The emphasis of the reported work will be on new and original research and technological developments covering methodologies as well as applications. Special topic issues will be published from time to time.

Applied Intelligence is a new international journal, published four times a year, beginning in January 1991.

Instructions for Authors

Potential authors should write to the following address for 'instructions for authors':

Karen S. Cullen, Kluwer Academic Publishers, 101 Philip Drive, Norwell, MA 02061, U.S.A.

57. S.SALVINI and M.H.WILLIAMS, Knowledge management for expert systems, IEE Digest No. 1990/077, 3/1-3/3 (1990).

Knowledge Management for Expert Systems

Steven Salvini
M. Howard Williams

1. Introduction

Before the advent of the Database Management System (DBMS), it was observed that in many an organisation there were a number of different application programs operating on the same or similar data, with the result that multiple copies of such data were being kept in different files and in different formats. Not only was this inefficient but it also led to various problems as a consequence of inconsistent updating of the different data files. DBMS were developed to solve this problem, providing a single uniform repository for the data required by different applications.

This same philosophy can be extended to the handling of knowledge. As more and more knowledge based applications (such as expert systems) are developed so the problem of duplication of knowledge will become apparent. For example, in an office where a number of different expert systems are used, any change to the legislation regarding tax or to the knowledge concerning the company and its operation might affect a number of different expert systems and might need to be encoded in the appropriate form for each different expert system.

The solution to this problem lies in the concept of the Knowledge Base Management System (KBMS), which is a repository for knowledge in the same way that a DBMS is a repository for data. Such a system would store the knowledge required by different knowledge based applications in a single base.

The idea of creating a knowledge store which can be accessed simultaneously by multiple applications, may at first sight appear to be straight forward. However, one important property which is essential for a KBMS is that it should provide knowledge in the form that the knowledge base application requires it. This makes the design of a KBMS considerably more complicated.

This paper describes a system developed as a first step towards a full KBMS. The system is based on a Logic Database Management System (LDBMS) or Deductive Database Management System (DDBMS) with appropriate extensions.

2. Proposed Architecture

Just as a Relational Database Management System (RDBMS) is based on relational calculus, a LDBMS or DDBMS is based on a subset of first-order predicate calculus. In the latter case the objects stored are clauses, each of which is either a fact (which corresponds to a tuple in a RDBMS) or a rule.

The fact that logic has proved an excellent vehicle for the expression of knowledge has led to its adoption as an internal format for many systems and to the widespread use of the logic programming language Prolog for developing knowledge based applications. For this reason a LDBMS provides a convenient repository for storing knowledge.

However, one major problem is that each different expert system shell has its own particular format in which knowledge is expressed. Even if two different users were to use the same expert system shell to develop an expert system for the same purpose, the way in which knowledge is represented in each might be very different.

To allow different expert systems using different external knowledge representation formats to share knowledge held in a LDBMS, an interface may be developed, referred to as the Expert Systems Interface (ESI), through which expert systems communicate with the LDBMS via a set of special commands. These commands enable the user to manage and manipulate rules and facts stored in the

Steven Salvini Department of Computer Science
Howard Williams Heriot-Watt University

LDBMS using the external representation of these rules and facts as used by the user expert system.

The ESI is responsible for mapping different external knowledge representation formats to a common internal format, which will be referred to as the Expert Systems Knowledge Internal Representation (ESKIR). Rules and facts from the knowledge base of a user expert system must be converted to ESKIR format before storing them in the LDBMS. Subsequent queries aimed at retrieving knowledge from the LDBMS must also be converted to ESKIR format. Furthermore, when the results of such a retrieval are returned to the user expert system, they must be translated back from ESKIR to the original format of the user expert system.

3. Implementation

The system which has been developed, is based on a LDBMS implemented in Prolog which provides concurrent access by multiple users to sets of clauses. rules). The version of Prolog used, Prolog-x+, has been developed for large database applications and can use the CRS/CLARE system (Clause Retrieval Server/CLAuse Retrieval Engine) [Won88a, Won88b] for accessing large clause sets held on secondary storage. The query language used by the LDBMS is called *SQUIRREL* [Wau89a], an extension of ISO standard *SQL* [ISO87a]. In addition to pure relational concepts, the "tables" in SQUIRREL may contain any collection of clauses - both facts and rules - determining that relation.

On top of this has been developed an ESI which provides its own set of commands thus the expert systems never see the SQUIRREL query language statements generated on their behalf by the ESI. These commands take as parameters concepts and objects which are part of the expert system's own external knowledge representation language thus reducing the number of new concepts must be grasped by a knowledge engineer before using the system.

The requirement for two-way translation of knowledge formats is met by providing a semi-automatic translator-generator system driven by a BNF description of the knowledge base given by the knowledge engineer.

In the literature on KBMSs it is usually assumed that "knowledge sharing" is permitted in that any knowledge held in a KBMS will be available to any user [Man86a]. In this system, a slightly different approach is taken. For example, in the case of a company-wide knowledge store it should be possible to avoid allowing a vdu operator to see the rules that determine the salary of his or her superior. Thus it could be said that this system implements "knowledge hiding" rather than knowledge sharing. Thus the issue of authorisation is important in a KBMS.

This is handled by each expert system having an associated schema in which its rules and facts are stored thus each expert system is treated as a separate user. By granting the appropriate access privileges to an expert system it is possible to implement knowledge sharing in a secure manner.

4. Conclusions

The concept of a KBMS is regarded as important for the future. The main reason for this is the increasing size and complexity of the knowledge bases of current knowledge-based systems [Len86a] and the proliferation of knowledge-based and expert systems themselves. For example, Stachowitz reports that by 1995 it is expected that KBMSs will handle 10**6 rules although the Japanese are discussing rulebases of 10**9 rules (R. Stachowitz in [Bro86a]). It formed one of the key elements in the architecture of the Fifth Generation Computer Systems presented by the Japanese [Mur84a] because of this importance.

It is expected that KBMSs will make use of some of the ideas and techniques of DBMSs. However, they may also raise new problems such as those arising from the security requirement for knowledge stored in large shared knowledge bases. Whilst it is important that applications be able to share knowledge, it is equally important that certain knowledge can be hidden from certain applications or

users. This is a similar problem to that encountered in DBMS but with the additional difficulties posed by constraining access to knowledge generated through rules.

References

Bro86a. M.L. Brodie, R. Balzer, G. Wiederhold, R. Brachman, and J. Mylopoulos, "Knowledge base management systems: discussions from the working group," in *Expert Database Systems*, ed. L. Kerschberg, pp. 19-33, Benjamin/Cummings Publishing Company Inc., (1986).

ISO87a. ISO, "Database Language SQL," ISO 9075-1987(E), (1987).

Len86a. Lenat, D., Prakash, M., and Shepherd, M., "CYC: using common sense knowledge to overcome brittleness and knowledge acquisition bottlenecks," *AI Magazine*, **Vol. 6**, (4), pp. 65-85, (1986).

Man86a. F. Manola and M. L. Brodie, "On knowledge based systems architectures," in *On knowledge base management systems*, ed. M. L. Brodie and J. Mylopoulos, pp. 87 - 92, Springer-Verlag, (1986).

Mur84a. K. Murakami, T. Kakuta, and R. Onai, "Architectures and hardware systems: parallel inference machine and knowledge base machine," in *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, (1984).

Wau89a. K.G. Waugh, G. Chen, Q. Kong, S. Salvini, and M.H. Williams, "SQUIRREL : A BNF Grammar," Technical report 89/5, Dept. of Computer Science, Heriot-Watt University, (April 1989).

Won88b. K-F. Wong and M.H. Williams, "Design Considerations for a Prolog Database Engine," in *Proceedings of the Third International Conference on Data and Knowledge Bases*, Jerusalem, Israel, (June 1988).

Won88a. K-F. Wong and M.H. Williams, "Limiting a Resolution Set in a Large Prolog Database Using Special Hardware," in *Proceedings of 1988 ACMSIGSMALL/PC Symposium on small and personal computers*, pp. 133-145, Cannes, France, (4-6 May, 1988).

58. S.SALVINI and M.H.WILLIAMS, Central knowledge management for expert systems, Mathematical and Computer Modelling, vol. 16, nos. 6/7, 137-144 (1992).

Mathl. Comput. Modelling Vol. 16, No. 6/7, pp. 137–144, 1992
Printed in Great Britain. All rights reserved

0895-7177/92 $5.00 + 0.00
Copyright© 1992 Pergamon Press Ltd

CENTRAL KNOWLEDGE MANAGEMENT FOR EXPERT SYSTEMS

STEVEN SALVINI AND M. HOWARD WILLIAMS
Department of Computer Science, Electrical and Electronic Engineering
Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ, Scotland

Abstract—The position of the knowledge base management system in relation to expert systems is analogous to that of the database management system in relation to standard applications. Just as database management systems provide facilities to store and manipulate data for user applications, so knowledge base management systems provide facilities to store and manipulate knowledge for user knowledge-based or expert systems. A number of different approaches have been taken in trying to build prototypes of knowledge base management systems. One such approach is to extend an existing deductive database, and one such prototype system is described. An associated demonstration application from the domain of dentistry is examined, to show the practical use of the system and its utility in knowledge sharing.

1. INTRODUCTION

Just as the database management system (DBMS) was a product of the increased use of data–intensive applications in the 70's, so the idea of a knowledge base management system (KBMS) is becoming more and more important as the investment in expert systems continues to rise. Not only is the number of expert systems in regular use continuing to grow at a rapid pace, but also the size of the knowledge base is increasing. This leads to the need for reliable and efficient facilities for the storage and manipulation of knowledge.

A number of strategies for building KBMSs are being prototyped. This includes work being done on deductive databases [1,2], object oriented databases [3,4], and on providing specialised hardware for knowledge bases [5]. This paper is concerned with work developed on deductive databases, as these are well-suited to providing a starting point for building KBMSs.

A deductive database is a database system based on first order logic, in which is stored a collection of clauses. These clauses include facts (which correspond to the explicit tuples in a relational database) and rules (which can be used to generate implicit tuples). Thus, if a retrieval query selects a set of tuples satisfying some condition, the tuples which are produced may be any combination of explicit and implicit tuples.

Since a deductive database stores as objects a set of clauses, it must provide facilities to insert, delete and update these clauses. This facility provides the basis for building a KBMS in that the rules of an expert system may be expressed as clauses and stored in the deductive database, and then retrieved as objects when required.

While a deductive database system has an inbuilt inference mechanism for deriving implicit tuples from rules, this is not sufficient to replace the inference mechanism within an expert system, particularly since each expert system has its own characteristic variations in this regard. Thus, the deductive database merely acts as a repository for the knowledge required by the expert systems, storing and retrieving it as required.

The dental knowledge bases used in this project were developed in conjunction with Dr. Denis F. Kinane of the Department of Oral Medicine and Pathology, University of Glasgow, to whom we are greatly indebted for his time and patience. This work was partly funded under the Alvey initiative (R-LPI 085), and the authors wish to acknowledge both this support and that from our industrial "uncle," ICL. Finally, we should like to express our gratitude to the other members of the Heriot-Watt University Logic Database Group for their help in the preparation of this paper.

Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX

MCM 16:6/7-J

However, one important property of a database is that it facilitates sharing of data among applications. This means that several different applications may wish to use the stored set of data, and the system will deliver to each the data it requires in the form in which it needs it. This avoids duplication of data (and the problems of maintaining several copies of the same data) and the attendant problem of inconsistency which may arise if all copies of the same data are not updated correctly and at the same time.

This property is important in a knowledge base management system too. If one has one or more expert systems which make use of the same knowledge, one would like this knowledge to be stored as a single copy, so that any changes to it will be consistently transmitted to all expert systems which make use of it. Thus if one expert system has rules which determine whether or not a particular patient is likely to be vulnerable to a given infection, and another expert system has rules for determining whether or not a particular patient is a compromised host, it can readily be seen that a number of these rules will overlap. It would seem very worthwhile to be able to store all the rules in a central knowledge store, with only a single instance of each of the overlapping rules being stored and with the rules being made available to the user expert systems in whatever format they require.

This paper describes a system which has been developed to provide a shared knowledge capability within the Logic Database Management System (LDBMS) [6].

2. ARCHITECTURE

The system is based on the LDBMS, which is a deductive database with a query language, SQUIRREL [7], which can handle rules and facts. SQUIRREL is a superset of ISO standard SQL [8], and treats the SQL tables as comprising tuples derived either from facts explicitly stored in the database, or generated by the application of rules describing the tuples. It can be used to retrieve either the tuples themselves in a manner identical to traditional relational systems, or the facts and rules describing relations. Rules stored via SQUIRREL may be treated as "active" or "passive" rules. Rules are considered active when used to generate tuples in answer to a query; rules are passive when they are retrieved as objects in their own right. Normally, any rule in the logic database will be addressable in both modes; however, for the purposes of a KBMS, some rules can be treated as being passive-only rules. This means that they are stored in the normal way but are never fired.

The LDBMS is written in Prolog-X+ [9], a Prolog optimised for database applications, and closely linked to the CRS/CLARE (Clause Retrieval Server/CLAuse Retrieval Engine) system for efficient clause manipulation and storage.

Although SQUIRREL provides facilities for the manipulation of rules and facts, this is only possible providing that those rules and facts are in the standard format required by this query language. To overcome this restriction, a special interface was developed, which is referred to as the Expert System Interface (ESI) [2].

The ESI makes available two groups of functions. On one hand, it provides knowledge engineers with facilities to manipulate the schemata associated with particular expert systems, and to enter knowledge bases into the LDBMS. On the other, it provides commands which can be directly incorporated in the code of user expert systems; for example, to retrieve a set of rules.

A knowledge engineer must set up schemata for expert systems because different expert system acting as users of the KBMS need not actually see all the knowledge contained in the virtual knowledge base. Whilst in the literature on KBMSs, it is usually assumed that all the knowledge in the KBMS should be available to all user expert systems by "knowledge sharing" (for example, [10]), it would seem likely that such an approach may not be ideal in practical situations. Just as in the DBMS situation where, for example, it is usual to be able to restrict a video display unit (vdu) operator from seeing the salary of his or her superiors, so it would seem necessary to be able to also restrict that vdu operator. Perhaps a better description of the approach taken is "knowledge restriction," rather than knowledge sharing.

This effect is achieved by each expert system having an associated schema in which its rules and facts are stored. In other words, each expert system is treated as a "user" in the usual database terminology. By granting the appropriate access privileges to an expert system (or

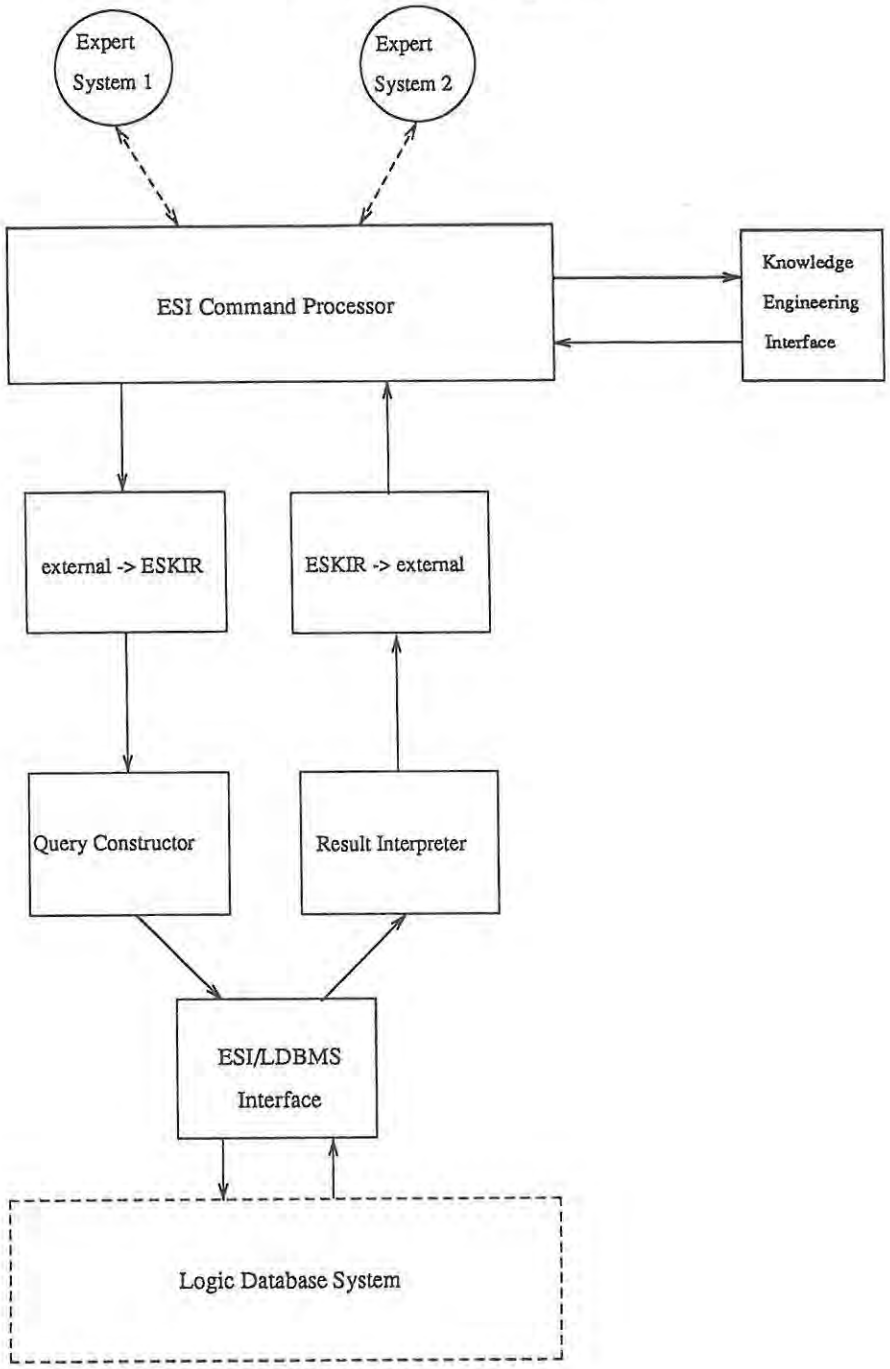


Figure 1. ESI architecture.

group of expert systems) on relations owned by other expert systems, it is possible to implement knowledge sharing in a secure manner. Three categories of knowledge can be identified in the KBMS:

(a) **common knowledge** which is available to any user expert system;
(b) **shared knowledge** which is available to a group of expert systems; and
(c) **knowledge private** to the owner expert system.

This simple concept is illustrated in Figure 2.

Once a knowledge engineer has set up the appropriate schema or schemata, he must then alter the code of the expert system which is to use the KBMS for knowledge storage. The

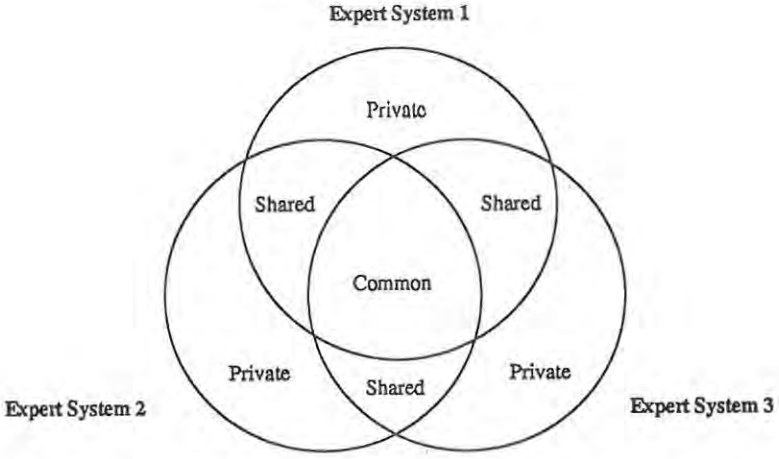


Figure 2. An illustration of the possible overlap of knowledge from three different expert systems.

alterations consist of changing the program, so that ESI commands are used for accessing and manipulating knowledge. Commands are available to insert, delete and retrieve rules and facts, and for other purposes such as to identify a user expert system to the ESI. One such command is "`insert_rule`". This takes one parameter, an expert system rule, and inserts this into the appropriate schema for the owner expert system, creating any required tables both for the rule itself and any sub-parts of the rule, and automatically sets up the appropriate permissions on these for any other systems which have been granted use of this system's knowledge.

These commands are designed to be as simple as possible, and to impose as little overhead on the knowledge engineer as possible. They greatly simplify the interface to the knowledge store by removing the necessity for any direct interaction between knowledge engineer or user expert system and the underlying SQUIRREL query language, as well as insulating from the internal knowledge representation format, (ESKIR [11]) used in the system. They take as parameters objects and concepts from the user expert system's own knowledge representation format and automatically transform these into the required knowledge representation format for the LDBMS. This transformation is a major part of the ESI's functionality, and is considered fundamental to the acceptability of any KBMS, since it removes the need for a knowledge engineer to restrict himself to a given system for reasons of compatibility of knowledge representation format and also allows a wide range of pre-existing systems to be used with the KBMS.

The transformation of rules and facts and their sub-parts (for example, as selection criteria in retrieval queries) is carried out semi-automatically by means of a translator-generator system which is initially configured by the knowledge engineer providing a BNF description of his expert system's knowledge representation format.

3. DENTAL APPLICATION

To demonstrate the potential of the KBMS, a system was developed for the domain of dentistry, building on previous experience gained in this area [12].

The system was intended to form the first step towards developing a complete expert system competent in all the specialties of dentistry. It was decided to begin by concentrating on a subset of Periodontology, namely, the diagnosis of dental pain, as this had already been investigated in part [11] in earlier work, and then to construct a new expert system for use in Restorative Dentistry. This second system is concerned with the selection of therapy in the form of the correct bridge, partial denture or crown for a particular patient. Each expert system was built using a different expert system shell with different knowledge representation formats.

Whilst constructing the knowledge base for the periodontal expert system, rules were formed with the help of an appropriate dental expert clinician, concerning the development of a number of conditions including, for example, dental abscesses. Many of these were dependent on information about the anatomical structure of the mouth. For example, the rules might describe

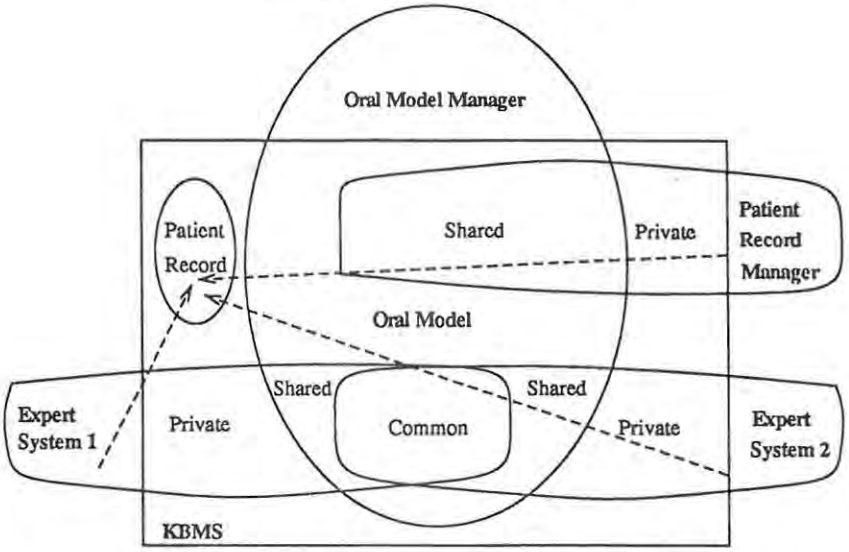


Figure 3. Schematic showing the relative overlapping of the knowledge for the dental application. The dotted lines represent the creation of the patient record from the information gathered by the patient record manager and the user interfaces of the two expert systems.

the hypothesis that pain due to a certain type of dental abscess is most common in or near the molars (i.e., the rear-most teeth). In a similar way, when working on the knowledge base for the restorative dentistry system, it became clear that there were a number of rules which required rules and facts describing the oral cavity itself. For example, in forming rules to represent the criteria for the selection of suitable teeth to act as abutments to hold bridges, the anatomical relationships must be very clearly described.

There were many similar areas of overlap, for example, rules concerned with the description of pain were common to both rule sets, and two strategies were used to handle this feature. By careful study of the two emerging rule bases, it became clear that to simplify the construction of the two expert systems, it would be useful to utilise the marked overlap of knowledge required about the positions of anatomical structures and lesions by developing a third system, the Oral Model (OM).

The Oral Model is a component-based central knowledge base whose knowledge is available to either user expert system. It contains deep knowledge in the form of facts and rules about the anatomy and physiology of the teeth and soft tissues of the oral cavity and about their inter-relationships. It also contains those shallow rules which are common to more than one user expert system, and rules to carry out translations such as from exact data values to more general statements or from the terminology of one speciality to that of another. For example, as noted above, the description of the position of the teeth is important in both expert systems although the fineness of description differs. In the restorative system, the positions must be more accurately described, as the clinician is trying to physically fit prostheses onto the teeth themselves, whereas in the periodontal system the position is often only important to locate the source of the pain in general terms in the history. Looking ahead to possible future additional expert systems required to develop a knowledge based system for a wide range of dental specialties, in Dental Prosthetics the description of the orientation and spatial relationships of the dentition must be even more accurate still. This means that if a patient has already consulted the prosthetics department the case records may already contain very detailed reports of the dentition, for example, from measurements taken from dental x-rays. If this information was held in the KBMS, it would seem futile to then ask a clinician to re-examine the patient to obtain what is in essence a less-detailed description of the patient's mouth on consulting the restorative department. Rather than provide completely different rule sets for each system, it would appear that a cleaner approach is to provide a translation ruleset to carry out the conversion from the information already held about the patient to the level of detail required by a calling expert system.

The second strategy employed was to rely solely on the facilities of the KBMS to deal with the sharing of the many other rules which were common to both of these expert systems, but which would be unlikely to overlap with future components of the system quite so markedly. Thus, these rules were retained in the original expert systems' knowledge bases and, when these knowledge bases were added to the central knowledge store, the KBMS facilities were used to allow the sharing of a single representation of each of these rules by the two systems involved.

Whilst this sharing of knowledge was to some extent expected, the fact that certain rules would require to be stored so that they could not be shared had been less apparent initially. These rules tended to involve terminology for which there was no common definition in the different specialties.

A separate module, the Patient Record Manager (PRM), performs a basic clerking function common to most dental specialties to produce a patient record with reference to the data in the OM.

The Oral Model Manager allows the user to inspect the oral model just as the PRM allows interrogation of the state of the current patient record.

Some example rules from the expert systems and the oral model showing the various degrees of sharing possible are as follows:

1. Common Rule—held in the oral model and used by the oral model and both expert systems:

> **If** total width of teeth in arch is greater than the total width of the arch,
> **then** *there is crowding present in this arch.*

Example use by a rule in the restorative expert system:

> **If** *there is crowding present in this arch,*
> **then** teeth can act as abutments for missing teeth, having up to a maximum of twice the number of roots they themselves have.

Example use by a rule in the pain diagnosis expert system:

> **If** *there is crowding present in this arch,* and the tooth in question is not virgin and it has a polished high-spot,
> **then** there is strong evidence that the tooth may be pulpitic.

2. Shared Rule—held in the oral model and used by the oral model and the restorative expert system only:

> **If** the 7 is missing and the 8 on the same side has erupted and is present,
> **then** *this 8 may function as a pseudo-7.*

Example use by a rule in the restorative expert system:

> **If** *the 8 was functioning as a pseudo-7,* and the 8 is extracted and the 6 on the same side is present,
> **then** the appliance of choice is a Free Fixed Bridge for the 7 and 8.

3. Oral Model Fact—held in the oral model and used by the oral model and, thus, indirectly by both expert systems:

> **Fact:** the 8 on either side and on either arch is a *molar.*

Example use by a rule in the pain diagnosis expert system:

> **If** the tooth is tender to percussion and it is either a *molar* or a pre-molar,
> **then** there is strong evidence that there may be an associated abscess present.

The system operates as follows. Independent of which expert system the user opts to use, the PRM is run first to gather the basic information about the patient (i.e., name, age, sex, etc.), and create a patient record which it stores in the LDBMS. The appropriate expert system then uses the information in the patient record, the oral model and any additional information it may request from the user, which is also added to the patient record via the PRM. The knowledge bases of both expert systems, the oral model, and the patient record are all stored in the LDBMS. As noted above, for each expert system, some of the information is Private (i.e., can be accessed from the LDBMS only by that expert system), some is Shared (i.e., is held as part of the OM in

the LDBMS and is available only to expert systems or the OM), and some is Common (i.e., is held as part of the OM in the LDBMS and is available to either expert system and the OM).

All the interactions described here between the expert systems and OM on the one hand, and the LDBMS on the other, are handled by the ESI. Thus the user expert systems see the ESI/LDBMS combination as a single KBMS and at no time are required to use the LDBMS SQL-based interface directly.

4. CONCLUSIONS

The concept of a KBMS is regarded as important for the future. The main reason for this is the increasing size and complexity of the knowledge bases of current knowledge-based systems [13] and the proliferation of knowledge-based and expert systems themselves. For example, Stachowitz reports that by 1995 it is expected that KBMSs will handle 10^6 rules although the Japanese are discussing rulebases of 10^9 rules [14]. Recognising its importance, it formed one of the key elements in the architecture of the Fifth Generation Computer Systems presented by the Japanese [15,16].

It is expected that KBMSs will make use of some of the ideas and techniques of DBMSs. However, they may also raise new problems such as those arising from the authorisation requirements for knowledge stored in large shared knowledge bases. Whilst it is important that applications be able to share knowledge, it is equally important that certain knowledge can be hidden from certain applications or users. This is a similar problem to that encountered in DBMS, but with the additional difficulties posed by constraining access to knowledge generated through rules.

This paper has described one approach to providing some of the services of a full KBMS. This approach concentrates on providing the facilities of an underlying logic database to user expert systems, without requiring these users to make use of one fixed knowledge representation language imposed by the system. Rather, a user expert system retains its original knowledge representation format and any translations required between it and the underlying internal format are carried out automatically by the ESI. This has disadvantages in that it is not yet clear how to handle the representation of different uncertainty-handling mechanisms in a uniform manner, and also in the amount of knowledge format translation required; but it is thought that these will be offset in the long term by the advantages of the knowledge engineer having less to learn, and also by permitting the incorporation of existing expert systems within the KBMS rather than just systems written *de novo* with its use in mind.

The example from the domain of dentistry has shown the application of these ideas in a practical setting, and its construction has helped clarify some of the issues regarding knowledge sharing.

REFERENCES

1. J. Minker, Perspectives in Deductive Databases, *Journal of Logic Programming* 5 (1), 33–60 (1988).
2. S. Salvini and M.H. Williams, Towards a knowledge base management system, Technical Report TR89012, Department of Computer Science, Heriot-Watt University, Edinburgh, (1989).
3. G. Copeland and D. Maier, Making Smalltalk a database system, In *Proceedings SIGMOD Conference*, Boston, 1984, (Edited by P.A. Bernstein), pp. 52–59, ACM, Boston, (1984).
4. J. Banerjee, H.-T. Chou, J.F. Garza, W. Kim, D. Woelk, N. Ballou and H.J. Kim, Data model issues for object-oriented applications, *ACM Transactions on Office Information Systems* 5 (1), 3–26 (1987).
5. D. McGregor, S. McInnes and M. Henning, An architecture for associative processing of Large Knowledge Bases (LKBs), *Computer Journal* 22 (4), 32–37 (1987).
6. M.H. Williams, G. Chen, D. Ferbrache, P. Massey, S. Salvini, H. Taylor and K.F. Wong, Prolog and deductive databases, In *Knowledge-Based Systems*, pp. 188–192, (June 1988).
7. K.G. Waugh, M.H. Williams, Q. Kong, S. Salvini and G. Chen, Designing SQUIRREL: An extended SQL for a deductive database system, *Computer Journal* 33 (6), 534–546 (1990).
8. ISO, Database Language SQL, ISO 9075-1987(E), International Standards Organisation, Geneva, (1987).
9. M.H. Williams, P.A. Massey and J.A. Crammond, Benchmarking Prolog for database applications, In *Prolog and Databases: Implementations and Directions,* (Edited by P.M.O. Gray), pp. 161–187, Ellis-Horwood Limited, Aberdeen, (1988).
10. F. Manola and M.L. Brodie, On knowledge based systems architectures, In *On Knowledge Base Management Systems,* (Edited by M.L. Brodie and J. Mylopoulos), pp. 87–92, Addison-Wesley, Cambridge, (1986).
11. S. Salvini and A.J. Gammerman, PROTEST: A prototyping tool for knowledge engineering, In *Proceedings of Medical Informatics Europe* '87, (Edited by A. Serio, R. O'Moore, A. Tardini, P.H. Roger), pp. 1030–1035, M.I.E. 87 Organizing Secretariat, Roma, (1987).

12. S. Salvini, A prototyping tool for expert systems, M.Sc. Dissertation, Heriot-Watt University, Edinburgh, (1986).
13. D. Lenat, M. Prakash and M. Shepherd, CYC: Using common sense knowledge to overcome brittleness and knowledge acquisition bottlenecks, *AI Magazing* 6 (4), 65–85 (1986).
14. M.L. Brodie, R. Balzer, G. Wiederhold, R. Brachman and J. Mylopoulos, Knowledge base management systems: Discussions from the working group, In *Expert Database Systems,* (Edited by L. Kerschberg), pp. 19–33, Benjamin Cummings, South Carolina, (1986).
15. K. Murakami, T. Kakuta and R. Onai, Architectures and hardware systems: Parallel inference machine and knowledge base machine, In *Proceedings of the International Conference on Fifth Generation Computer Systems,* Tokyo, November 1984, pp. 119–127, ICOT, Tokyo, (1984).
16. K. Furukawa and T. Yokoi, Basic Software System, In *Proceedings of the International Conference on Fifth Generation Computer Systems,* Tokyo, November 1984, pp. 169–177, ICOT, Tokyo, (1984).

59. M.B.IBANEZ-ESPIGA and M.H.WILLIAMS, Data placement strategy for a parallel database system, Proceedings of the International Conference on Database and Expert Systems Applications '92, Eds. A.M.Tjoa and I.Ramos, Springer-Verlag,48-54 (1992).

Data Placement Strategy for a Parallel Database System

M.B. Ibáñez-Espiga
M.H. Williams
Dept. of Computer Science, Heriot-Watt University
79 Grassmarket, Edinburgh EH1 2HJ
United Kingdom

Abstract

The EDS machine is a high performance parallel computer system being developed as part of an ESPRIT II proyect. The database system for EDS is its main application and provides an extended relational system with object-oriented capabilities. One aspect which is crucial to performance is data placement. For several reasons existing data placement strategies are not ideal for the EDS system and a new approach based on two existing strategies will be used. This approach is presented here with some comparisons of its performance.

Keywords Data allocation, data placement, greedy method, parallel database.

1 Introduction

The development of an effective parallel database system requires carefully chosen strategies for solving inter-related issues such as data partitioning, data placement, query processing, data replication, load balancing and concurrency control [8]. For instance, distributed query processing depends on how the data has been allocated to sites and, on the other hand, the optimal method of allocating data depends on the processing strategies used for solving queries. None of the problems can be properly solved in isolation, without taking into account their interdependency with respect to the others.

Furthermore, the complexity of problems such as data placement and query processing is NP complete, and hence their combination is also NP complete. It is for this reason that the majority of research in one area has assumed given solutions for the others [2], [5], [6], [10]. Some attempts to use an integrated methodology have been reported recently [4], [7], [9], but they have not shown any significant performance improvement over the methods which solve the problems independently.

The rest of this paper is organised as follows. In section 2 a brief review of the data allocation problem is given. The relevant characteristics of the EDS machine are outlined in section 3, while section 4 defines the problem of data placement for EDS. The Data Placement strategy proposed is described in section 5.

2 Relevant work relating to the data allocation problem

Apers [2] proposed a solution to the problem of data placement which takes into account the user data access requests and the system processing of those requests. The strategy is based on a heuristic approach coupled with a greedy algorithm. The network is represented as a graph with two different kind of nodes: the physical sites which are machine nodes and the virtual sites which are the fragments of the relations. Each edge represents the cost of data transmission between two nodes. The strategy consists of a re-write process as follows. Each pair of sites is evaluated on the basis of the transmission cost of the allocation, i.e. the sum of the total transmission costs of queries weighted by their execution frequencies. The pair of sites with the highest utilisation is selected and united. This process is repeated until all virtual sites are united with some physical site.

In the paper it is shown that the algorithm gives a good sub-optimal solution to the problem. Apers does not consider load constraints in solving the problem. The strategy uses static schedules. A dynamic heuristic is also proposed but this approach quickly becomes intractable as the problem size increases.

Sacca and Wiederhold present an approach for data allocation in distributed database systems of tightly clustered processors [10]. The strategy consists of selecting the candidate fragments to allocate using Apers' algorithm and then allocating the selected fragments with first-fit bin-packing. The allocating phase of the strategy considers load constraints and the limits of the heterogeneous network used.

The cost evaluation function used in [10] uses capacity constraints but does not take into account

response times. Sacca and Wiederhold claim that their strategy is effective on tightly–coupled processors with small communication delay.

Copeland et al [5] offer a solution to the data placement problem with a strategy which involves declustering, relation assignment, a combination of clustering and indexing, and cacheing.

The records are declustered into segments according to either the key value or its hashed value. When using the key value, the sorted relation is partitioned equally across a set of IRs (Intelligent Repositories) equal in number to the declustering degree of the relation. This allocation is based on balanced *heat* i.e. the access frequency of the object over some period of time, rather than the *size* i.e. the number of bytes of the object.

Relation assignment places the set of relations, each having a set of segments equal in number to the degree of the relation, onto the IRs in such a way that the total heat of each IR is roughly equal.

Heuristic methods are used which contemplate the possibility of reorganizing whenever the actual placement is not ideal.

Mukkamala et al [7] present an integrated methodology which takes into consideration four design issues: relation partitioning, data allocation, distributed query processing, and load balancing. Their approach divides the problem into two phases. In the first phase the fragments are partitioned into groups formed by fragments with a high rate of inter–data communication. In the second phase the groups are allocated among the nodes.

The goal of the first phase is to reduce the size of the allocation problem. The group formation problem is solved by a heuristic algorithm. The second phase is solved by heuristics based on group–to–group interactions, group–to–node interactions, and node–to–node interactions. Neither the heuristic of the first phase nor those of the second phase are presented in the paper.

The approach proposed by Blankinship et al [4] provides a solution to the query optimization and data allocation problems. The initial data allocation is performed using Apers' algorithm. This is followed by an iterative heuristic optimization algorithm which produces a local optimal data allocation. The heuristic alternates between distributed query optimization and distributed data allocation optimization until a solution is reached that cannot be improved.

Pears and Gray [9] present a method which takes into account interacting issues such as query optimisation, access path design and concurrency control. The strategy consists of limiting the number of available nodes for allocating fragments in order to control the relative amounts of task balancing and data parallelism. In general, a small number of available nodes for allocation will induce a small degree of task balancing and a high degree of data parallelism, while a larger number of available nodes will produce the reverse effect.

3 Relevant characteristics of the EDS machine

Consider the main characteristics of the EDS system. The hardware component of the EDS machine is described in [1] and [11] while the information about the storage model is given in [3].

3.1 Hardware

The EDS machine is a homogeneous highly parallel computer system based on a shared–nothing architecture which can scale up to 256 Processing Elements (PEs). The PEs are interconnected by a network which provides equi–distant access. The network is the only shared resource. Each PE consists of four components:

The Store Unit consists of a two level stable store where the top level is non–volatile RAM and the second level is disc storage.

The Processing Unit is based on a SPARC processor.

The System Support Unit runs in parallel with the Processing Unit and its role is to perform low level tasks such as control of message passing.

The Network Interface Unit provides a 20MB/s dual (Tx and Rx) communication channel between the PEs and the network, which in turn has a large communication bandwidth.

3.2 Software Architecture

The EDS system has three sub–systems:

The Request Manager is responsible for three main functions: semantic analysis, optimization of queries and automatic parallelization of those queries.

The Session Manager is responsible for high–level resource management of the database server.

The Data Manager is responsible for the storage and the data held in the relations of the database, the execution of the parallel query pro-

grams and in particular, it is responsible for data placement.

3.3 Storage Model

Each conceptual relation is stored as one direct relation and zero or more indexes. These relations are fragmented horizontally and distributed among virtual PEs according to a function (index, hash, range, etc) applied to one or more attributes. The number of fragments is a function of the size and access frequency of the relation. Each fragment can have zero or more indexes. The indexes are partial, i.e., they might not cover the whole relation. Support for the storage of duplicate copies of relations is planned in order to improve safety and integrity.

3.4 Accessibility of the data

The work to be performed on the data is done where the data resides. Thus data placement has strong repercussions on the load balancing and throughput of the system.

The declustering is supported by a global directory replicated in each PE. The global directory contains information about where each fragment of every relation is stored. Using this information the messages can be routed appropriately.

4 Problem definition

The data placement problem in EDS can be stated as follows.

Given:

$P = \{P_1, \ldots, P_p\}$ is a set of processor nodes (physical sites) of the EDS machine.

$R = \{R_1, \ldots, R_r\}$ is a set of relations.

A set of queries is required to be run on the processor nodes using the relations. This set of queries can be decomposed into a set of *transactions* $T = \{T_1, \ldots, T_t\}$ such that each transaction performs some operations on relations from R with a given frequency. Each transaction can be further decomposed into one or more *associations* which represent the basic unit of work which can be assigned to a set of processor nodes. When an association is assigned to a set of processors, it is copied to each processor concerned, the copy being referred to as a *thread.*

The relations are split into horizontal fragments by the Request Manager. The number of fragments is a function of the size and access frequency of the relation. The number of fragments of a relation defines the number of its *virtual sites* . Thus, every relation R_i in R has associated a set of *fragments* $F(R_i) = \{F_{i1}, \ldots, F_{ih}\}$.

A relation may be part of different associations. The set of physical sites across which an association/relation is distributed is called the *home* of the association/relation. The Request Manager groups the code and the relations that must be allocated into the same home.

Our objective is to find a mapping from virtual to physical sites so that a predefined *cost function* is minimized, while a set of *constraints* is satisfied. To find a mapping from virtual to physical sites means the allocation of the code of the transactions and the fragments of all the relations, i.e.,

$$F = \bigcup_{i=1}^{r} (F(R_i))$$

The cost function must include measures such as communication costs, storage costs etc.

The constraints to take into account include capacity constraints and eventual response time requirements, the home restrictions imposed by the Request Manager, and specific requests from the user for fragments to be assigned to particular physical sites.

5 Data placement strategy

Since the data allocation problem can be computationally very expensive to solve, and due to the dynamic nature of the process no solution can guarantee optimal performance, it is reasonable to opt for a heuristic method which does not require too much overhead and which produces reasonable performance. Depending on the behaviour of the system a re-allocation strategy can be put into practice.

The heuristic methods described in the literature often split the problem into phases to reduce its complexity [7], [4], and [9]. Two pure methods seem to fit our needs:

1. An approach which allocates the fragments by minimizing the total transmission cost (Apers' method).
2. An approach which distributes the fragments according to their heat and size (Bubba method).

Apers' method may perform well at an early stage in the allocation process when some fragments are assigned to the *right* physical sites. However, the algorithm does not by itself give a heuristic for performing such an initial allocation. If the algorithm tries to deal with the complete allocation, it could finish by assigning all fragments to the same physical site.

The Bubba method gives an easy strategy taking into account only quantitative information about the

fragments. It does not take into consideration how the execution of the queries is affected by the placement of the data nor any other semantic information.

One kind of heuristic may be more suitable for one database configuration than for another. Thus three alternatives have been considered :

1. Allocating each fragment according to its heat and size, following the Bubba strategy.
2. Allocating each fragment to minimise total transmission cost, following Apers method.
3. Using a combination of the above two strategies. Which takes account of semantic information contained in the transactions.

The proposed strategy is divided into two main phases. The first phase provides an initial distribution of the PEs based on a round-robin strategy. This initial distribution is performed on the basis of the heat and size of the fragments and it takes into account how the relations are used in the transactions. The second phase assigns the fragmented relations to the PEs by an adapted form of Apers' strategy. In the second phase subsets of fragments are assigned to subsets of PEs, thereby reducing the risk of placing all fragments on the same physical site.

In the following subsections a brief account of the proposed approach is given. For the sake of simplicity the approach is not described in full, but instead the section focusses on giving the guidelines, and omits issues such as : how to deal with the intermediate relations, what constraints are taken into account, and how to deal with replication.

5.1 Proposed Strategy: Phase 1

The Request Manager splits every relation into several fragments in order to allow maximum parallelism when two different fragments of the same relation are held at different physical sites. This phase attempts to take advantage of this information in order to reduce the complexity of the problem and to balance the load (in time and space) of the different PEs.

The fragments of the relations are grouped into units according to the associations in which they appear. The units are distributed on a round-robin basis taking into account the heat of the fragments and the available space in the physical sites. The strategy attempts to allocate different fragments of a relation to different PEs.

5.1.1 Grouping fragments into units

A relation can be involved in more than one association. Assume that relation R must share the same home as relation S in the association A_1 and that R and T belong to the association A_2. This means that R, S and T should share the same home. The strategy will use a transitive closure algorithm to determine which relations should share the same home. From this it is possible to form *clusters* of relations. A *cluster* is the maximal set of relations that share the same home. For every one of these clusters the method determines the maximum fragmentation degree of the relations in it. This number is the number of *units* needed to allocate all the fragments of the relations in the cluster. Each unit will contain one fragment of every relation in the cluster. For those relations having a fragmentation degree smaller than n, assume that they will have fragments with no tuples.

To clarify these ideas, consider the following example.

EXAMPLE 1

Suppose that one has two transactions to run: T1 and T2. T1 has two associations; A_{11} and A_{12}, T2 has one: A_{21}. The relations used by the queries are: A, B, C, D, E, F, with degrees of fragmentation: 4, 2, 2, 2, 4, 2 respectively.

A_{11} uses the relations E and F.
A_{12} uses the relations C and D.
A_{21} uses the relations A, B and E.

As a result of the Transitive Closure algorithm, two clusters are formed:

C1 containing the relations A, B, E and F.
C2 containing the relations C and D.

The maximum degree of fragmentation of the relations in C1 and C2 is 4 and 2 respectively. For the cluster C1, the units contain the following fragments :

$U_{11} : A_1, B_1, E_1, F_1$
$U_{12} : A_2, B_2, E_2, F_2$
$U_{13} : A_3, E_3$
$U_{14} : A_4, E_4$

For the cluster C2, the units contain the following fragments :

$U_{21} : C_1, D_1$
$U_{22} : C_2, D_2$

5.1.2 Distributing units among groups of physical sites

The next step is to pre-assign the units to the phys-

ical PEs. This is referred to as a pre–assignment because the decision as to the final assignment will be taken in phase 2 of the method.

The pre–assignment follows a round–robin approach, taking into account the heat of the fragments and the available space in the physical sites. Thus the idea is to find an appropriate balance among the work load of the processors and an equitable distribution of the space.

In the above example suppose that the number of physical sites is 8 and that all fragments have the same size and the same heat

Under such conditions, U_{11}, U_{12}, U_{13} and U_{14} are pre–assigned to $\langle PE_1, PE_5 \rangle$ [1], $\langle PE_2, PE_6 \rangle$, $\langle PE_3, PE_7 \rangle$ and $\langle PE_4, PE_8 \rangle$ respectively, and U_{21} and U_{22} are pre–assigned to $\langle PE_3, PE_7 \rangle$ and $\langle PE_4, PE_8 \rangle$ respectively.

5.2 Proposed Strategy: Phase 2

After the first phase, the fragments of the relations are grouped into units, each of which is associated with a set of physical sites. The goal of the second phase is to allocate the fragments of each unit to the physical sites assigned to the unit considering constraints such as the memory and load capacity of the physical PEs.

The first step consists of building a cost graph for every unit, the nodes of which represent sets of relation fragments and each arc connects an initial node to a final node. The arc gives a measure of the cost of moving information from that initial node to the final node. The cost is computed taking into account the size of the fragments in the initial node and the frequency with which these fragments are accessed in the database.

The cost graph for a given unit is reduced to a graph with a number of nodes which is less than or equal to the number of physical PEs associated with the unit. The reduction of the graph is done by simple transformations. Once the graph has reached its final state, the fragments are assigned to the physical PEs in such a way that all fragments of a given node are allocated to a single PE.

5.2.1 Building the cost graph

For a given unit the following information is available:

- The set of fragments with their size and frequency of use.
- The set of physical sites with their capacities.

[1] It means that the fragments in the unit U_{11} will be allocated between the physical processors PE_1 and PE_5

For every fragment, the identification of the associations which may use that fragment is also needed [2]. For example, suppose that the set of associations that use the fragments of a particular unit is represented by A = { A_{ij},...,A_{mn} }. For every A_{kl} in A, a node is created to represent the fragments of A_{kl} which are not present in any other association in A. If there exists one or more fragments in A_{kl} which appear in other associations in A, a node is created for each such fragment. Two nodes are linked if and only if they contain fragments appearing in the same association.

To illustrate the method consider the cost graph for the unit U_{11} from the former example. For the sake of simplicity, let us assume that all the fragments have the same size, the same frequency of use, and that the physical sites have infinite capacities.

The fragments A_1 and B_1 can only be used by the association A_{21} and so a node is created to represent A_1 and B_1. Call this node n_1.

The fragment F_1 can only be used by the association A_{11}. The node n_2 is created to represent F_1.

The fragment E_1 can be used by the two associations A_{11} and A_{21}, and thus the node n_3 is created to represent E_1, and is linked to n_1 and n_2 as figure 1 shows.

From the cost graph represented in figure 1, if the fragments A_1, B_1 and E_1 are needed in an association to perform an operation, it will be necessary either to move A_1, B_1 to the physical node where E_1 is allocated (cost of the operation: 2x), or to move E_1 to the PE where n_1 is allocated (cost of the operation: x).

5.2.2 Transforming the cost graph to its final state

When the number of nodes in the cost graph is greater than the number of physical sites, it is necessary to reduce the cost graph. For this purpose the m x m Cost Matrix M will be considered where m is the number of nodes of the cost graph and M_{ij} is the cost of moving all the fragments from n_i to n_j. If M_{kl} is the largest element in the matrix M, the nodes n_k and n_l are unified and a new reduced cost graph is generated taking into account this join. This process stops when the number of nodes remaining in the cost graph equals the number of physical sites. The cost matrix of cost graph 1 is as follows:

$$M = \begin{bmatrix} 0 & 0 & 2x \\ 0 & 0 & x \\ x & x & 0 \end{bmatrix}$$

Figure 2 shows the reduced and final cost graph.

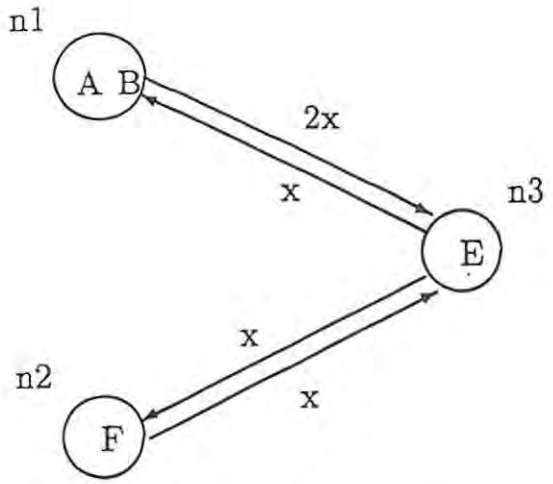


Figure 1: Initial Cost Graph for the unit U_{11}

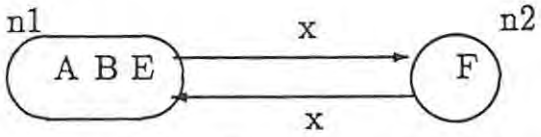


Figure 2: Reduced Cost Graph for the unit U_{11}

The fragments can now be allocated to the physical sites. In the example, fragments A_1, B_1 and E_1 are allocated to PE_1 and the fragment F_1 are allocated to PE_5. Before starting the execution of association A_{11}, fragment E_1 are moved from PE_1 to PE_5 and when the association A_{11} finishes executing, E_1 is returned to PE_1. When association A_{21} is executed, no moves are needed.

6 Performance Comparison

The proposed strategy has been benchmarked using a set of queries and the results have been compared with those of the Apers method and the Round Robin method. Six relations are involved in this set of queries. They are listed as follows:

Relation	Fragment	Fragment size
A	*A1*	30
	A2	10
B	*B1*	10
	B2	20
	B3	30
	B4	40
C	*C1*	10
	C2	30
R	*R1*	10
	R2	20
S	*S1*	10
	S2	10
	S3	10
T	*T1*	20
	T2	10

The set of queries consists of 2 queries of which the information is listed as follows:

Query	Association	Relations involved
Q1	*assoc 1*	R S
	assoc 2	R T
Q2	*assoc 1*	R A
	assoc 2	B C
	assoc 3	C

For simplicity, only four PEs are used in the benchmarking. The memory capacity of these 4 PEs are respectively 100, 100, 120 and 120. The benchmarking results are shown in the following table:

The column under each PE denotes the fragments allocated to this PE using the three data placement strategies. The column of *Moving Cost* represents the minimum cost of moving fragments to the appropriate PE when running the two queries. Assuming that the cost of moving 1 unit of a fragment from one PE to another is 1, the results have shown that the proposed strategy has 0 moving cost, which means that all the fragments are appropriately allocated to 4 PEs so that no movement of fragments is required during execution time.

	PE1	PE2	PE3	PE4	Moving Cost
Round Robin	*A2 B2 B3*	*C1 C2 R1 S3*	*B1 B4 S2 T1*	*A1 R2 S1 T2*	80
Apers	*B4 R1 R2 S1 S2 S3*	*A1 A2 B3 T1 T2*	*C1 C2*	*B1 B2*	80
Proposed Strategy	*A1 R1 S1 T1*	*A2 R2 S2 S3 T2*	*B1 B2 C1 C2*	*B3 B4*	0

[2] Here we do not give the details of how to get this information for the sake of simplicity

7 Conclusions

A data placement strategy for a distributed database system is presented. The strategy is based on ideas contained in two established data placement models: the method used in the Bubba's project and Apers' method. In addition the strategy uses semantic information contained in the transactions to determine the final allocation.

The strategy first assigns the fragments of the relations to subsets of PEs on the basis of their heat and the knowledge of which relations share the same home. This phase guarantees that two or more relations that do not interact are placed on different PEs using an algorithm which is not computational expensive.

The final allocation is done by considering subsets of fragments with high interactivity and allocating them to PEs in such a way as to minimise total transmission cost. Apers' method applied in this way is computationally less expensive and it has less chance of allocationg all the fragments to the same PE.

Some experiments are currently being conducted to compare the performance of the proposed strategy against the other methods. Initial results for one example are given.

Acknowledgments The authors acknowledge the support received from the European Community under the ESPRIT II programme for this work which was conducted as part of project EP2025, European Declarative System. Particular thanks are due to Phil Broughton and Steve Leuning of ICL for their support. Thanks are also due to Norman Paton (Heriot–Watt University) for comments on drafts of this paper and to Patrick Valduriez (INRIA), David Ferbrache (RSRE) and John Sargent (Manchester University) for their valuable comments.

References

[1] F. Andres, B. Bergsten, P. Borla-Salamet, P. Broughton, C. Chachaty, M. Couprie, B. Finance, G. Gardarin, K. Glynn, B. Hart, S. Kellett, S. Leuning, M. Lopez, M. Ward, P. Valduriez, and M. Ziane. Eds-collaborating for a High Performance Parallel Relational Database. *ESPRIT Conference*, November 1990.

[2] P. M. G. Apers. Data Allocation in Distributed Database Systems. *ACM Transactions on Database Systems*, 13(3):263–304, September 1988.

[3] B. Bergsten, M. Couprie, and P. Valduriez. Dbs3, An Implementation of the Eds Database System on a Shared-Memory Multiprocessor. 1991.

[4] R. Blankinship, A. R. Hevner, and S. B. Yao. An Iterative Method for Distributed Database Design. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 389–400, Barcelona (Spain), September 1991.

[5] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data Placement in Bubba. In *Proc. 1988 ACM SIGMOD Conf.*, pages 99–108, Chicago IL, May 1988.

[6] K. A. Huan and C. Lee. An Adaptative Data Placement Schema for Parallel Database Computer Systems. In *Proceedings of the 16th VLDB Conference*, pages 493–506, Brisbane (Australia), 1990.

[7] R. Mukkamala, S. C. Bruell, and R. K. Shultz. Design of Partially Replicated Distributed Database Systems : An Integrated Methodology. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, pages 187–196, Santa Fe, New Mexico (USA), May 1988. ACM.

[8] M. Ozsu and P. Valduriez. *Principles of Distributed Database System.* Prentic-Hall, Inc., 1991.

[9] R. L. Pears and W. A. Gray. Response Time Minimisation in a Parallel Database System. In *Proceedings on the 9th British National Conference on Databases*, pages 148–167, 1991.

[10] D. Sacca and G. Wiederhold. Database Partitioning in a Cluster of Processors. *ACM Transactions on Database Systems*, 10(1):29–56, March 1985.

[11] K. F. Wong, U. C. Baron, H. Benker, S. Bescos, A. B. Cheese, F. J. N. Cosqur, S. A. Delgado-Rannauro, M. Dorochevsky, P. Heuze, M. B. Ibanez, J. Noye, M. Paci, M. J. Ratcliffe, M. J. Reeve, K. Schuerman, A. Sexton, A. Veron, and J. Xu. The European Declarative System (Eds) as Platform for Parallel Logic Programming. *2nd IEEE Symposium on Parallel and Distributed Processing*, December 1990.

60. S.ZHOU and M.H.WILLIAMS,Cache management for retrieved solutions in a Prolog database system,to appear in Proceedings of 14th Colloquium on Information Retrieval, Lancaster University, April 1992, (1993).

Cache Management for Retrieved Solutions in a Prolog Database System

Shaoyu Zhou and M. Howard Williams

Department of Computer Science
Heriot-Watt University
79 Grassmarket
Edinburgh EH1 2HJ

Abstract

This paper describes a cacheing and lookahead strategy for retrieving solutions in a Prolog Database system. The strategy minimizes the number of accesses to the external databases and provides some degree of parallelism among different components of the Prolog Database system. Some benchmarking results are given to evaluate the performance of the strategy.

Keywords

Prolog database, cache management, lookahead strategy.

1. Introduction

Prolog is a logic programming language which is well-suited to the development of knowledge based system applications. However, virtually all existing Prolog implementations suffer from the limitation that they only deal with clauses in main memory and cannot therefore handle large data sets. This shortcoming makes it difficult to use Prolog for a range of applications from deductive databases to large knowledge bases. This problem has motivated the development of Prolog Database systems.

Research into Prolog Database systems has led to the design and development of several systems which combine the functionality of Prolog with database functionalities [8, 11, 12, 14, 16, 17, 20]. Some of them have interfaced Prolog systems to relational databases [1-3, 5]. The Prolog system handles the execution of Prolog programs until disc resident clauses are required, whereupon control is passed to the database system. The database system then deals with the storage and retrieval of clauses from disc. Because the Prolog system returns one tuple at a time to its calling program while the database system returns a complete relation when invoked, this kind of coupling has raised a

problem of buffering the solutions retrieved from external databases. This is a significant problem where the result relation is large or where the database call is part of a recursive query which generates a large quantity of intermediate data. A number of cacheing strategies have been developed to handle this problem.

In some Prolog Database systems, such as the PDBM system [11, 16] which was developed at Heriot-Watt University, a simple approach has been adopted where solutions retrieved are buffered on the *Global* stack, where there is automatic reclamation of space on backtracking. However, the number of solutions buffered must be limited and with deeply recursive procedures the stack might be consumed rapidly.

Another approach is to use a separate area for retrieved solutions, but this has the problem of maintenance of such an area. In addition to this, the user would have to estimate the memory requirements for yet another data area. More importantly a space recovery scheme for this area would have to be devised, since there would be little or no recovery of space upon backtracking / completion of the query.

In this paper, a cacheing and lookahead strategy is described for buffering retrieved solutions, which has been adopted in a Prolog Database system based on a multiprocess architecture[20]. The cacheing and lookahead strategy buffers the retrieved solutions in a specific buffer area and provides the Prolog system with a limited number of solutions whenever a request is received. It also provides for prefetching of solutions so that the Prolog system is not delayed unnecessarily while waiting for solutions to be retrieved from external databases. This enables a higher degree of parallelism to be obtained.

The paper is organized as follows. Section 2 contains a brief description of the architecture of the Prolog Database system in which the cacheing and lookahead strategy has been implemented. Section 3 describes the cacheing and lookahead strategy used and in Section 4, an assessment of its performance is given. The strategy provides a considerable improvement in performance in cases where a large number of solutions are retrieved from external databases.

2. The Prolog Database System Architecture

The Prolog Database system [20], in which the cacheing and lookahead strategy has been implemented, is a multi-user transaction system which allows Prolog to store large sets of clauses on disc and which enables updating of these sets of clauses by multiple users. Functionally the system consists of five main components:

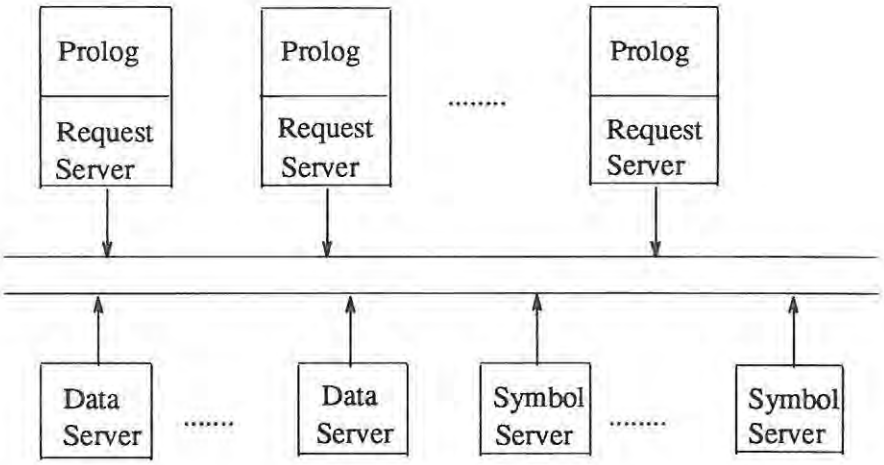
- The Prolog System - providing the user interface and the database query language; currently an extended version of Prolog-X [6] is used.
- The Request Server - providing the functionalities of query transformation, transaction control, resolution, integrity and security control, as well as cacheing and prefetching of solutions.
- The Data Server - handling database operations, such as retrieval, insertion and deletion, and providing the functionalities of lock management, unification, disc block cache and direct look ahead, access method control, and hardware / software device driver control.
- The Symbol Server - dealing with the symbol table, and providing the functionalities of cacheing, index / access control, hardware driver control, etc.
- The Clause Retrieval Engine - providing fast hardware search of a large set of clauses on disc.

The Request Server is closely coupled with the Prolog system and communicates with the Data Server and the Symbol Server through low-level datagram sockets provided by the UNIX operating system. Multiple Data Servers and Symbol Servers can be run on remote machines on request from the Request Server. Each Data Server may access one external Prolog database and return the retrieved data to the Request Server. The Request Server can broadcast its retrieval commands to all Data Servers to allow the Data Servers to retrieve the required data in parallel.

An illustration of the system architecture is given in Figure 1. Figure 2 shows the distribution of different functionalities among the Request Server, the Data Server and the Symbol Server.

Whenever Prolog encounters a database predicate, it invokes the closely coupled Request Server to formulate the appropriate query. The queries are translated into data server format which involves disk file offsets. The disk file offsets are obtained as follows. The symbols in the queries, such as atoms or functors, are first grouped in a packet

Figure 1. The Prolog Database system architecture

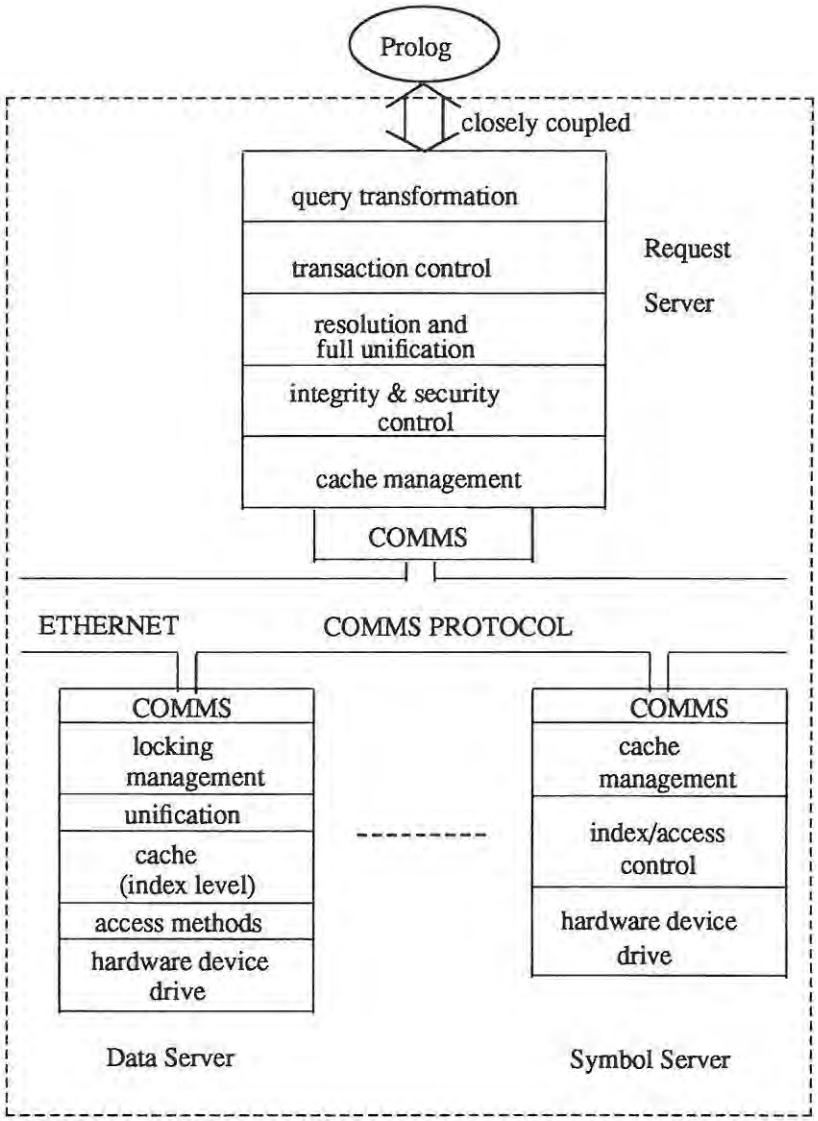


which is sent to the Symbol Server. The new symbols which don't exist in the symbol table will be inserted into the symbol table by the Symbol Server. The Symbol Server then produces a return packet containing disk file offsets of these symbols in the symbol table and sends this to the Request Server. The Request Server also performs security checks and transaction control. If all is correct, the Request Server sends the queries to the Data Server in a packet.

Having received a query packet, the Data Server operates on the appropriate databases to update and/or retrieve the required Prolog clauses. It produces the corresponding index for the query according to the access method the client has chosen. Then it searches the index file to find the matching indexes and retrieves the corresponding Prolog database clauses. If necessary, the hardware engine, CLARE, will be invoked to improve the searching speed. It then sends a packet containing retrieved solutions or error indication to the Request Server. Partial unification and lock management are done in the Data Server.

Since the solutions which the Request Server receives are all in disk-clause format, the Symbol Server may need to be invoked to convert disk file offsets into symbolic form. However, it is not necessary to do this for every clause retrieved from the database as an offset resolution scheme can be incorporated in which the disc file offsets are used while the Prolog system does unification and resolution. Only when the textual forms of atoms and functors are required for I/O routines in the Prolog system is it necessary for the Request Server to send a symbol-retrieval command to the Symbol Server. The latter then accesses the symbol table using the pointers to the textual form of the symbols in the symbol table.

Figure 2. Distribution of different functionalities among the Request Server, the Data Server and the Symbol Server.



The Request Server groups the retrieved clauses and return them in batches for the Prolog system to use. The cacheing and lookahead strategy is used here to buffer the solutions in a specific buffer area and to return a limited number of solutions to the Prolog system whenever a request is made.

A number of cacheing strategies are adopted within the Data Server and the Symbol Server to minimise the overheads incurred due to system paging activity over large virtual memory spaces (thrashing). The Data Server maintains its own cache of active data pages in an attempt to satisfy a continuation or repetition of a query over the knowledge base. Such cacheing is vital if a high degree of backtracking by the client is to be supported efficiently.

The cache management discussed in this paper is built into the Request Server level. It has been developed to handle the retrieved solutions before they are put into the *Global* stack of the Prolog-X system.

3. The Cacheing and Lookahead Strategy

At the Request Server level, the cacheing and lookahead strategy has been adopted to buffer the retrieved solutions, to minimise the number of accesses to the Data Server and to provide correct synchronization among the components so that they are not blocked unnecessarily while waiting for responses from other components.

Due to backtracking which occurs in the execution of Prolog, the same query may be sent to the Data Server many times. For example, consider the following Prolog clauses and query over them:

```
p(X,Y) :- q(X,Z), r(Z,Y).
q(a,1).
q(b,1).
?- p(X,Y).
```

During Prolog execution, the query **r(1,Y)** will be generated after **q(X,Z)** is solved over the first ground clause **q(a,1)**, and again after it is resolved with its second ground clause **q(b,1)**. To avoid sending the same query to the Data Server more than once, the Request Server provides a cacheing mechanism to cache the solutions to queries **q(X,Z)** and **r(Z,Y)** in a buffer area.

Due to differences in speed of different components of the Prolog Database system, Prolog-X processes are often forced to block. Since the Data Server deals with large data sets on disc, it becomes a potential performance bottleneck. As an example, consider the following Prolog program:

```
p(X,Y) :- exdb(X,Z), q(Z,Y).
q(a,1).
q(b,1).
?- p(X,Y).
```

Assuming that **exdb** is a large sized predicate stored in an external database, the size of the set of solutions to the query **exdb(X,Z)** could span a number of pages. After all the clauses of one page have been sent to and examined by Prolog, the next page has to be retrieved. Unless some provision is made, Prolog will have to block waiting for a database fetch to occur. To avoid this, a lookahead strategy is used to prefetch the next page of solutions to the same query into the buffer area. In the cacheing and lookahead strategy, this is called *same query lookahead.*

Consider another set of Prolog program:

```
p(X,Y) :- exdb1(X,Z), exdb2(Z,Y).
?- p(X,Y).
```

Assume that **exdb1,exdb2** are large predicates stored in an external database. After the solutions to the query **exdb1(X,Z)** have been sent to and examined by Prolog, the first page of **exdb2** has to be retrieved. In this case, the Prolog system will again have to block waiting for a database fetch to occur. To avoid this, the lookahead strategy should prefetch the first page of solutions to the next query into the buffer area. This is called *next query lookahead* in our strategy.

3.1. Cacheing strategy

The RDS system stores only a limited number of solutions in the *Global* stack of the Prolog-X system. The rest are kept in a separate buffer area awaiting entry to the *Global* stack. The buffer area for cacheing the retrieved solutions comprises a number of blocks each of which can accommodate solutions to a single external database query. The buffer area is managed in such a way that when all blocks in the area are occupied by the retrieved solutions, some blocks of old queries will be freed in order to accommodate the retrieved solutions to new queries. Before these blocks are freed, they will be copied onto disc. The old queries, along with the disc block addresses of these copied blocks, will be pushed onto a stack so that the solutions to the old queries can be retrieved rather than being re-evaluated. The stack will be referred to as the *Query Trace Table* and is described in Section 3.3. Since each block contains the solutions to a single query, if it is freed, the corresponding copied block on disc will have to be read into the buffer area again when it is needed by the Prolog system on backtracking. Thus, a suitable algorithm should be employed to free the blocks of old queries.

In the cacheing and lookahead strategy introduced in this paper, the buffer area is organized in a tree structure so that the depth and level of queries (subgoals of external predicates) can be found in the structure when deciding the blocks which should be freed. The cacheing strategy is *first in, first out*, in a similar fashion to LRU (Least Recently Used). Consider the following Prolog query as an example,

```
?- p(X,Y).
```

where **p** is an external predicate and the first clause in its procedure is:

```
p(X,Y) :- a(X,Y), b(X,Y), c(X,Y).
```

Assume that **a**, **b** and **c** are external predicates stored in an external database. The first clause in the external procedure **a** is:

```
a(X,Y) :- m(X,Y), n(X,Y).      (m, n are external predicates)
```

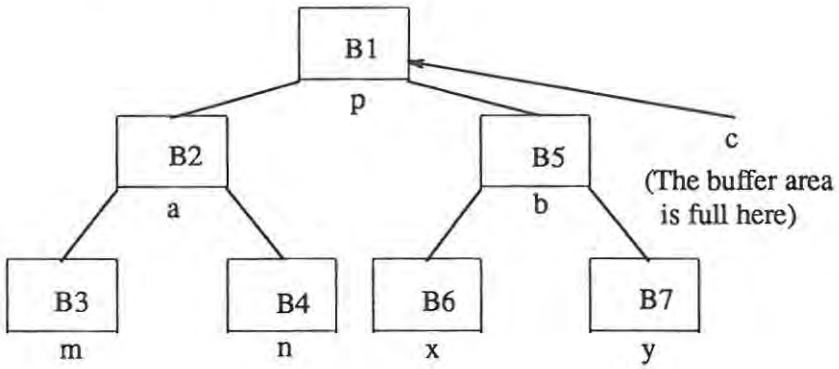
The first clause in the external procedure **b** is:

```
b(X,Y) :- x(X,Y), y(X,Y).      (x, y are external predicates)
```

The first clause in the external procedure **c** is:

```
c(X,Y) :- s(X,Y), t(X,Y).      (s, t are external predicates)
```

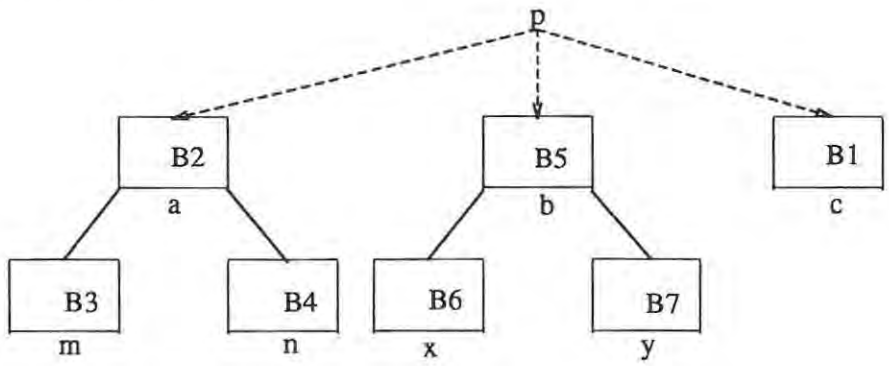
Figure 3. Buffer area structure before B1 is freed



Suppose that at this point, the Prolog system is evaluating the goal **c(X,Y)**. After searching through the external database, the Data Server returns the solutions to the query **c(X,Y)** to the Request Server. The Request Server tries to buffer the solutions in the buffer area, but the buffer area is full so that some blocks have to be freed. Suppose that retrieved clauses of predicate **p** are buffered in cache block **B1**, those of predicate **a** in cache block **B2**, those of predicate **m** in **B3**, those of predicate **n** in **B4**, those of predicate **b** in **B5**, those of predicate **x** in **B6**, and those of predicate **y** in **B7**. In this example, the total number of cache blocks in the buffer area is 7. Figure 3 shows the tree structure of the buffer area.

In the *first in, first out* cacheing strategy, **B1** should be freed in order to accommodate the retrieved solutions of predicate **c**, since **B1** buffers the solutions of predicate **p** which is the earliest retrieved. After block **B1** is freed, the retrieved solutions of predicate **c** can be buffered in **B1**. The solutions of the predicate **p** which used to be buffered in **B1** are saved in a disc block and the query **p(X,Y)**, along with the address of this disc block, is pushed into a stack maintained by the Request Server. Any further retrievals on the query **p(X,Y)** when they are needed by the Prolog system on backtracking will cause the disc block to be read into the buffer area. The structure of the buffer area at this point is shown in Figure 4. After the clauses of predicate **c** are buffered into **B1**, **B2** becomes the cache block containing the earliest retrieved solutions. It will be freed if the solutions to a new query, i.e. **s(x,y)**, are retrieved.

Figure 4. Buffer area structure after B1 is freed



Since the *Global* stack of the Prolog system has limited space, only a fixed number of solutions are retrieved to the Prolog system and stored in the *Global* stack whenever Prolog issues a call to retrieve clauses in an external database. Other solutions are still kept in the appropriate blocks of the buffer area. Each block has a *Current_Clause_Pointer* (CCPTR) pointing to the next clause to be inserted into the Prolog *Global* stack if the same query from Prolog is encountered. If a block is completely consumed by the Prolog system, i.e. the CCPTR of that block reaches the end of the block, the block will be freed. When the earliest retrieved block is copied to disc in order to accommodate a newly retrieved block, some block control information such as the CCPTR will be stored with the block on disc. If the block is re-loaded into the buffer area by the Prolog system on backtracking, the Prolog system can use the stored CCPTR to resume manipulation on this block.

The clauses buffered in the *Global* stack of the Prolog system are in a format which is basically the same as that used by the rest of the Prolog system - a linked list, where each clause contains a pointer to the next clause in the database. There is no need to index these clauses further, since the Request Server and the Data Server will reduce the list to only those which are potential satisfiers. The last pointer in the chain can be an *activity* number. This is then

used on backtracking to fetch the next set of solutions from the Request Server or from the Data Server if the corresponding cache block maintained by the Request Server has been consumed. When backtracking, the previous set of solutions on the stack will then be overwritten since they are no longer needed.

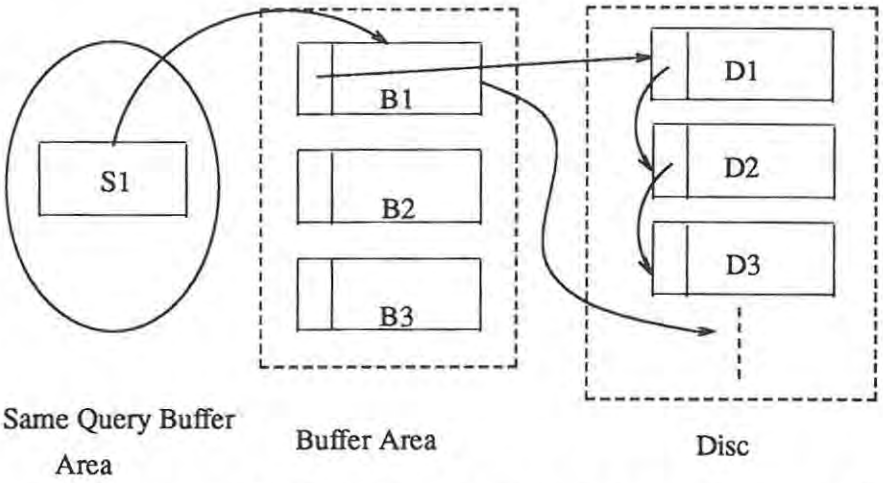
3.2. Look ahead strategy

In each block of the buffer area, there is a lookahead limit to indicate where a *same query lookahead* should start. Whenever the lookahead limit is reached by the CCPTR in the same block, and the solution page which the current block contains is not the last page of the solutions to the same query, a *same query lookahead* is initiated. If it is the last page, no *same query lookahead* is initiated. When the Prolog system starts to manipulate a new sub-goal, a *next query lookahead* starts prefetching of the first block of solutions to the next sub-goal. Therefore, when the Prolog system still has some retrieved clauses in the buffer area to manipulate, the Request Server sends a query to the Data Server to search for either the next block of solutions to the same query or the first block of solutions to the next query. The fetching occurs while the Prolog system manipulates the remaining clauses of the current block. Then, when the first clause in the new block is requested, it will already be present in the buffer area, and the Prolog system does not have to block. This provides some degree of parallelism in the Prolog Database system.

One part of the buffer area is called the *Same Query Buffer Area*, and is used to accommodate the lookahead blocks when *same query lookaheads* occur. The *Same Query Buffer Area* is managed in a similar way to that of the buffer area, but on a much smaller scale (the number of blocks is much smaller). When the current block is completely consumed by the Prolog system, i.e. the CCPTR has reached the end of the block, the corresponding lookahead block in the *Same Query Buffer Area*, which has the same query as the current block, is swapped into the place of the current block. Then, the Prolog system can access the next page of solutions to the same query using the same cache block pointer. When the *Same Query Buffer Area* is full, a similar strategy to that used to manage the buffer area is adopted. The earliest retrieved block will be moved to disc and its query and the disc block address will be pushed onto a stack. These will be used by the Prolog system on backtracking.

When a same query lookahead block is swapped into the place of the current block, the contents of the current block will be copied to a disc block and the address of this disc block will be kept in a chain structure. If the contents of the current block is the first block of solutions to the current query, the disc block address will be kept in the block

Figure 5. The chain structure of solution blocks



control information header of the current cache block as the head of the chain. Otherwise, the disc block address is kept in the block control information header of the disc block which contains the copied contents of the previous block of solutions to the same query. This chain structure of disc block addresses for the copied solution blocks is used to re-load the solution blocks when the same query is received from the Prolog system again so that evaluation of repetitive database calls can be avoided. Figure 5 illustrates such a chain structure of disc block addresses. S1 is swapped into the place of B1 while the contents of B1 are copied to a disc block and this disc block is appended to the end of the chain by keeping its address in the block control information header of D3.

An example of a *same query lookahead* is shown in Figure 6.

Assuming that the lookahead limit in cache block **B3** is reached and the page which **B3** contains is not the last page of solutions to the query, a *same query lookahead* is initiated. While the Prolog system is still manipulating the remaining clauses in **B3**, the next page of solutions to the query is retrieved and stored in a block **S1** in the *Same Query Buffer Area*. When **B3** is completely consumed by the Prolog system, **S1** is swapped into the place of **B3** in the buffer area.

Figure 7 shows an example of a *next query lookahead*, assuming that the Prolog system is manipulating the sub-goals associated with the cache blocks **B3**, **B6** and **B9**. The first pages of the solutions to the next sub-goals, if there are any, are buffered in cache blocks **B4** and **B7** in the buffer area. Thus, in the current structure of the buffer area, **B4** and **B7** are the *next query lookahead* blocks.

Figure 6. Same query lookahead

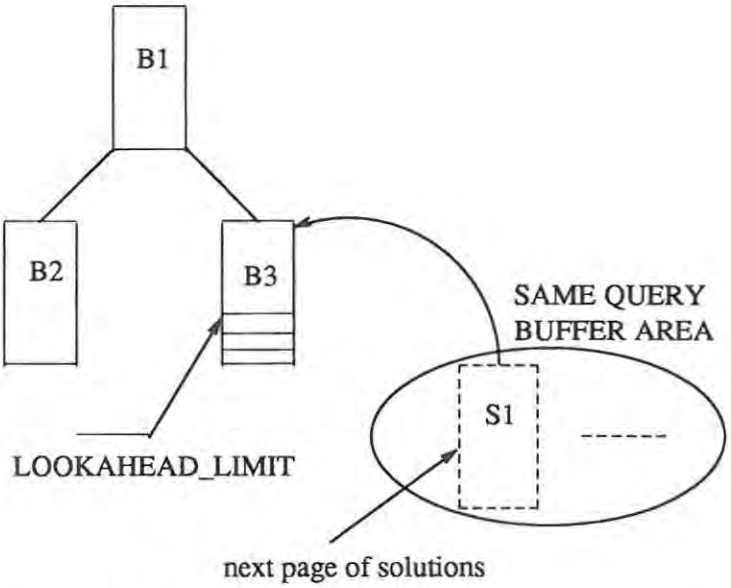


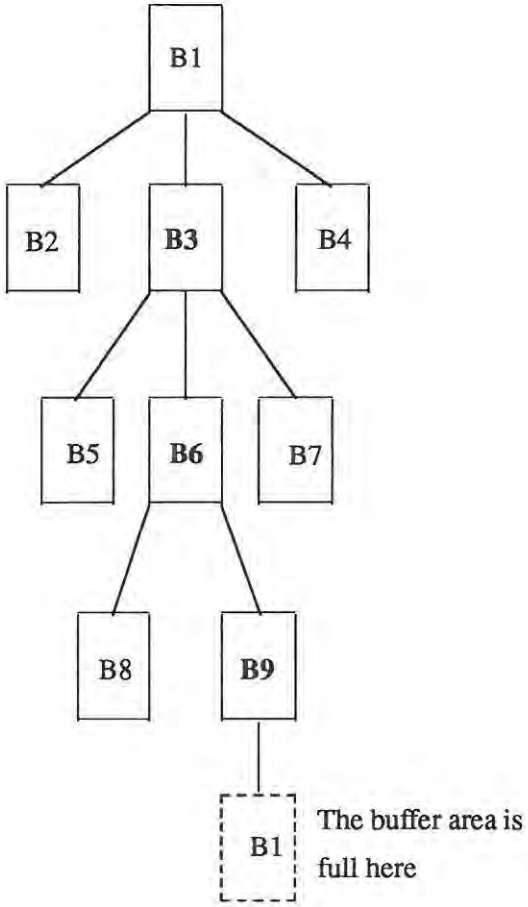
Figure 7 also shows the *first in, first out* cacheing strategy when the *next query lookahead* strategy is adopted. Assuming that the number of cache blocks in the buffer area is 9, as Figure 7 shows, when a page of solutions to a new query is retrieved, **B1** is freed. Then, if another new page is retrieved, **B2** will be freed.

In the implementation of the lookahead strategy, choice points in the Prolog system are checked in order to collect the predicate, arity and arguments of the next sub-goal when a *next query lookahead* is encountered. If the current sub-goal and the next sub-goal share no variable, it is quite simple for the Request Server to generate a query to send to the Data Server to retrieve solutions to the next sub-goal. If the current sub-goal and the next sub-goal do share one or more variables, the problem is more complicated. Consider the following example:

```
p(X,Y) :- q(X, m, Z), r(Z, Y, a). (q, r are external predicates)
```

Assume that **q** is the current sub-goal and **r** the next sub-goal. When the *next query lookahead* for the solutions to sub-goal **r** is initiated, the variable **Z** has not been bound to any ground value. This raises a problem when generating an appropriate query to sent to the Data Server for the *next query lookahead*. Recent research in the fields of parallel execution of logic programs has provided various methods for solving this problem [7, 9, 10, 15]. For the purposes of the current implementation, a simple approach has been adopted which treats the shared variables in the next sub-goal as unbound variables. When the next sub-goal becomes the current sub-goal, the query which Prolog sends to the Request Server subsumes the previous query generated for the *next query lookahead* [4]. The solutions to

Figure 7. Next query lookahead



the current query are a subset of those to the previous query. Because the Prolog system performs unification on the retrieved solutions, the solutions satisfying the current query are selected. After the lookahead limit in the lookahead block is reached by the CCPTR, a *same query lookahead* which uses the current query is initiated and the next block of solutions to the current query (rather than to the previous one) is retrieved.

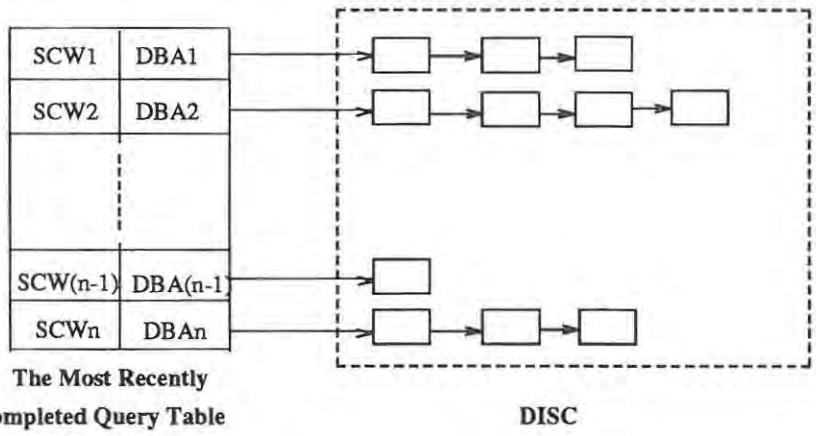
It can be seen from the implementation of the *next query lookahead* strategy that if the current sub-goal and the next sub-goal share no variable, which means that the two sub-goals are independent, the lookahead strategy will give a much better performance.

Consider the case where the current sub-goal and the next sub-goal are independent. Suppose that a *next query lookahead* for the next sub-goal takes place when the Prolog system is manipulating the first solution to the current sub-goal. After a match is found for the current sub-goal, the Prolog system will start to manipulate the lookahead block. When it has consumed the last solution in the last solution block of the next sub-goal, the cache block for the next sub-goal will be released to accommodate the solution block for a new query. This causes the query of the next

sub-goal to be re-evaluated when another match is found for the current sub-goal. Since the current sub-goal and the next sub-goal are independent, the solutions retrieved when re-evaluating the next sub-goal are exactly the same as previously retrieved. Therefore, the tracks of the solution blocks of the next sub-goal should be kept in order to avoid unnecessary re-evaluation.

This is achieved by using the *Most Recently Completed Query Table.* Each entry in this table contains the superimposed codeword of a completed query together with the disc block address of the first solution block of the query. When the Prolog system has completed manipulation on the last solution block of a query, this solution block will be moved to disc and appended to the end of the solution block chain of the query. The superimposed codeword [13, 19, 21] of the query and the disc block address of the solution block chain head will be put into the *Most Recently Completed Query Table*. The *Most Recently Completed Query Table* has a fixed length, which means that only a limited number of most recently completed queries can be kept in the table. The management strategy for the table is also *first in, first out*. Figure 8 illustrates the structure of the *Most Recently Completed Query Table*.

Figure 8. The structure of the Most Recently Completed Query Table

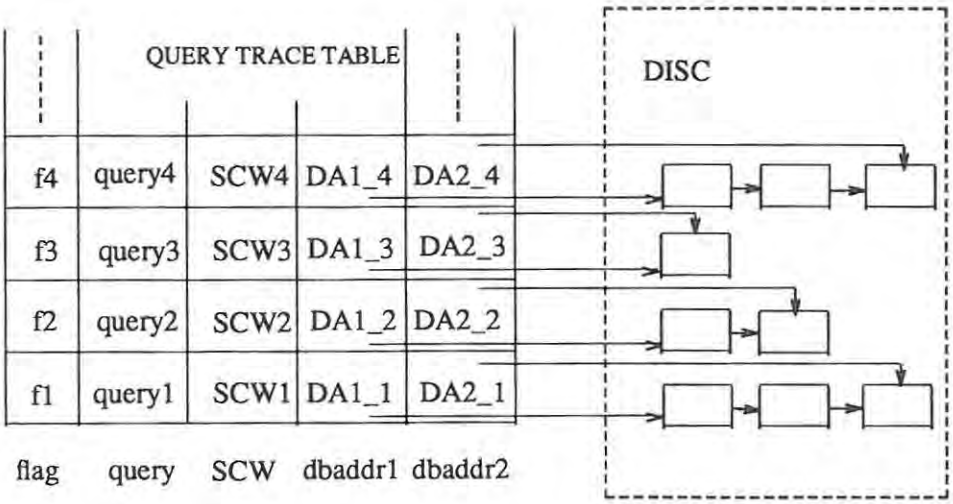


When a query sent by the Prolog system is received, its superimposed codeword will be compared with those in the *Most Recently Completed Query Table.* If an identical superimposed codeword is present in the table, the solution blocks will be re-loaded into the buffer area so that no re-evaluation of the query is needed.

3.3. Query Trace Table

When a current solution block of a query in the buffer area is copied to disc in order to accommodate a newly retrieved block, the query of the block is pushed onto a stack and will be used by the Prolog system on backtracking. Along with the query, the superimposed codeword of this query and the disc block addresses of the first and current solution block of this query are also pushed onto the stack. This stack is called the *Query Trace Table*. It can be used to avoid repetitive calls to the Data Server. Like the *Most Recently Completed Query Table*, the superimposed codewords of the queries in the stack are used to compare an incoming query with the queries already present in the *Query Trace Table*. Since the RDS system supports superimposed codeword indexing, a superimposed codeword is already generated whenever a database query is received from the Prolog system [20]. The disc block address of the first solution block can be obtained from the control information header of the current block. Figure 9 illustrates the structure of the *Query Trace Table*. The first column, *flag*, is used in loop detection. The second column, *query*, contains the queries of the copied cache blocks. The *SCW* column contains the superimposed codewords of the queries. The *dbaddr1* column contains the disc block addresses of the first solution blocks of the queries. The last column, *dbaddr2*, contains the disc block addresses of the current solution blocks. The solution blocks of each query are in a chain structure.

Figure 9. The structure of the Query Trace Table



The *Query Trace Table* is used to avoid sending repetitive database calls to the Data Server in two ways. First, when a paged-out solution block is needed by the Prolog system again on backtracking, the most recent entry of the *Query Trace Table* will be popped out and the *dbaddr2* value in this entry will be used to re-load the current solution

block of the corresponding query. This block will be loaded into the buffer area by the Request Server and the manipulation of this block will be resumed from the place pointed to by the CCPTR stored in the block control information header. Second, when a new query is received from the Prolog system, the Request Server compares this query with each query in the *Most Recently Completed Query Table* and the *Query Trace Table* using the superimposed codewords of both queries. If the same superimposed codeword is found in the *Query Trace Table*, the information in the corresponding entry of the *Query Trace Table* will be used to get the solutions to the new query. The Request Server will get the first solution block of the new query into the buffer area using the *dbaddr1* value in the entry. The chain structure of the solution blocks will be preserved with the CCPTR in each block set to 1.

The *Query Trace Table* can also be used in loop detection, as one of its functions. A simple loop detection algorithm using the *Query Trace Table* is listed as follows.

```
FOR   each database query received from the Prolog system
    look for the query in the Query Trace Table
    IF   the query is in entry K
         set the flag in entry K to 1
         check the flag in entry (K-1)
         IF   (K-1) is 0 or the flag in entry (K-1) is 0
              set Current_Marked_Entry_Length to 1
              set flags of all entries other than entry K to 0
         ELSE   add 1 to Current_Marked_Entry_Length
                IF   Current_Marked_Entry_Length is greater than TL
                     a loop is detected
    ELSE   set Current_Marked_Entry_Length to 0
           set flags of all entries to 0
```

TL in the above algorithm denotes the *tolerable repetition length.* Only when the number of repetitive queries in succession is greater than TL will the repetition be considered to be a loop. Current_Marked_Entry_Length denotes the number of successive entries in which the flags are set to 1. The algorithm accumulates Current_Marked_Entry_Length by checking if the new query is the last member of a repetitive call sequence which can be found in the *Query Trace Table*. If the length of this sequence is greater than TL, a loop is detected.

4. Performance Evaluation of the Cacheing and Lookahead Strategy

The cacheing and lookahead strategy has been implemented in a Prolog database system based on a multi-process architecture on a SUN 3 under UNIX. The Request Server, Data Server and Symbol Server were distributed across the Ethernet, running on a single machine or on several different machines. A set of benchmarks have been

used to assess the performance of the system. The first set of benchmarks uses a database consisting of vertices and edges[18] stored as clauses of the form

```
vertex(VERTEX_NAME, X, Y).
```

which represent the vertices of an $n \times n$ grid of points, and

```
edge(EDGE_NAME, VERTEX_NAME1, VERTEX_NAME2).
```

which represent edges connecting the vertices. The database is set up using a Hilbert's space filling curve algorithm to generate a set of edges corresponding to a Hilbert curve.

Several databases of different sizes (ranging from 16 to 16384 clauses) were generated. These test databases were then exercised to ascertain the answers to the following benchmarks.

(1) A simple external database search query

The format of this query is:

```
benchmark(X,Y) :- vertex(X,Y,0),fail.
```

The query was designed to avoid any other overheads caused by searching through large external databases which might influence the benchmarking results of the cacheing and lookahead strategy. Thus, the performance improvement made by the strategy can be more obvious during benchmarking.

(2) Join on two external database predicates

The format of this benchmark is:

```
benchmark(X,Y) :- vertex(M,N,0),edge(X,M,bound_value).
```

For each particular value of *M*, there is one and only one matching clause in the edge database which leads to a successful join. The *bound_value*s are different in the retrieved edge clauses if the values of *M* are different. Therefore, the *bound_value* can be selected carefully to control the point in the solution blocks of *vertex(M,N,0)* where a successful join is expected. In this benchmark, three values of the *bound_value* were selected respectively so that the first, the middle or the last matching clause retrieved from the vertex database will lead

to a successful join with a clause in the edge database.

(3) A query containing two independent sub-goals

The format of the query is:

```
benchmark(X,Y) :- vertex(X,Y,0),edge(M,N,'00'),fail.
```

The query was designed to benchmark the *next query lookahead* strategy when two sub-goals share no variables. Since the cache and lookahead strategy uses the *Most Recently Completed Query Table* to avoid repetitive database calls sent to the Data Server, the second sub-goal, *edge(M,N,'00')*, is only evaluated once while a number of solutions of the first sub-goal, *vertex(X,Y,0)*, have been retrieved.

Table 1 contains the system performance figures with and without cacheing using the first set of benchmarks. For each test, the percentage improvement made by the system with cacheing over that of the system without cacheing is also given. Benchmark test 1.1 uses a simple external database search query. Test 2 performs a join on *vertex* and *edge* predicates. Within this, test 1.2.1 selects the *bound_value* so that the first clause retrieved from the vertex database will lead to a successful search in the edge database. In this case, very little cache and lookahead facility is exploited. Therefore, the performance with cacheing is very similar to that without cacheing. Benchmark tests 1.2.2 and 1.2.3 select the middle or the last matching clause from the vertex database respectively as the one leading to a successful join. Benchmark test 1.3 uses a query containing two independent sub-goals.

The second set of benchmarks uses a database containing a flight timetable stored as clauses of the form

```
flight(City1,City2,Dep_hour,Dep_min,Arr_hour,Arr_min,Flight_num).
```

This set of benchmarks contains the following two simple application programs which retrieve on the flight timetable database.

The first application program is designed to get a list of cities which can be reached within a given flight time from a certain city, using the following set of Prolog rules

```
within(P1,P2,Len,Time) :-
        flight(P1,P2,Dep_hour,Dep_min,Arr_hour,Arr_min,Fnum),
        duration(Len,Dep_hour,Dep_min,Arr_hour,Arr_min),
        Len < Time.
```

Table 1. Comparative study of performances of the system with and without cacheing using the first set of benchmarks (all times given in msec.)

Benchmark test	Database size		Without cacheing	With cacheing	Percentage improvement
	Vertex database	Edge database			
1.1	16	-	360	300	16.7
	64	-	440	360	18.2
	256	-	700	480	31.4
	1024	-	1420	880	38.0
	4096	-	3360	1880	44.0
	16384	-	8720	5660	35.1
1.2.1	16	15	300	300	0.0
	64	63	480	480	0.0
	256	255	640	620	3.0
	1024	1023	1100	1080	1.8
	4096	4095	2460	2440	0.8
	16384	16383	7980	7940	0.5
1.2.2	16	15	400	400	0.0
	64	63	760	700	7.9
	256	255	1780	1620	9.0
	1024	1023	7940	7620	4.0
	4096	4095	45360	44580	1.7
	16384	16383	348200	343240	1.4
1.2.3	16	15	440	420	4.5
	64	63	860	760	11.6
	256	255	2440	2120	13.1
	1024	1023	10180	9580	5.9
	4096	4095	62380	61020	2.2
	16384	16383	459860	451020	1.9
1.3	16	1023	1480	860	41.9
	64	1023	2860	1360	52.4
	256	1023	5720	2220	61.2
	1024	1023	11480	3840	66.6
	4096	1023	23520	7520	68.0
	16384	1023	48920	16180	66.9

```
duration(Len, Dep_hour, Dep_min, Arr_hour, Arr_min) :-
        Arr_hour > Dep_hour, Len is (60*(Arr_hour-Dep_hour)+Arr_min-Dep_min).
duration(Len, Dep_hour, Dep_min, Arr_hour, Arr_min) :-
        Arr_hour =:= Dep_hour, Arr_min >= Dep_min, Len is (Arr_min-Dep_min).
duration(Len, Dep_hour, Dep_min, Arr_hour, Arr_min) :-
        Arr_hour =:= Dep_hour, Arr_min < Dep_min,
        Len is (60*24+Arr_min-Dep_min).
duration(Len, Dep_hour, Dep_min, Arr_hour, Arr_min) :-
        Arr_hour < Dep_hour, Len is (60*(Arr_hour+24-Dep_hour)+Arr_min-Dep_min).
```

A number of flight timetable databases of different size (ranging from 20 to 39800 clauses) were generated.

They were then exercised to ascertain the answers to queries such as

```
within(edinburgh,City2,Len,60).
```

which asks for a list of cities which can be reached within 60 minutes by flight from Edinburgh.

The second application program was designed to look for a connection between two cities via a third city by flight. This requires a join operation. The Prolog program is listed as follows:

```
connecting(P1,P2,P3,Time) :-
        flight(P1,P2,Dep_hour1,Dep_min1,Arr_hour1,Arr_min1,Fnum1),
        flight(P2,P3,Dep_hour2,Dep_min2,Arr_hour2,Arr_min2,Fnum2),
        duration(Len1,Dep_hour1,Dep_min1,Arr_hour1,Arr_min1),
        duration(Len2,Dep_hour2,Dep_min2,Arr_hour2,Arr_min2),
        transfer(Len3,Arr_hour1,Arr_min1,Dep_hour2,Dep_min2),
        (Len1+Len2+Len3) < Time.

duration(Len, Dep_hour, Dep_min, Arr_hour, Arr_min) :-
        Arr_hour > Dep_hour, Len is (60*(Arr_hour-Dep_hour)+Arr_min-Dep_min).
duration(Len, Dep_hour, Dep_min, Arr_hour, Arr_min) :-
        Arr_hour =:= Dep_hour, Arr_min >= Dep_min, Len is (Arr_min-Dep_min).
duration(Len, Dep_hour, Dep_min, Arr_hour, Arr_min) :-
        Arr_hour =:= Dep_hour, Arr_min < Dep_min,
        Len is (60*24+Arr_min-Dep_min).
duration(Len, Dep_hour, Dep_min, Arr_hour, Arr_min) :-
        Arr_hour < Dep_hour, Len is (60*(Arr_hour+24-Dep_hour)+Arr_min-Dep_min).

transfer(Hour1,Min1,Hour2,Min2,Len) :-
        Hour2 > Hour1, (60*(Hour2-Hour1)+Min2-Min1) >= 40,
        Len is (60*(Hour2-Hour1)+Min2-Min1).
transfer(Hour1,Min1,Hour2,Min2,Len) :-
        Hour2 =:= Hour1, Min2 >= Min1, (Min2-Min1) >= 40,
        Len is (Min2-Min1).
transfer(Hour1,Min1,Hour2,Min2,Len) :-
        Hour2 =:= Hour1, Min2 < Min1, Len is (60*24+Min2-Min1).
transfer(Hour1,Min1,Hour2,Min2,Len) :-
        Hour1 > Hour2, (60*(Hour2+24-Hour1)+Min2-Min1) >= 40,
        Len is (60*(Hour2+24-Hour1)+Min2-Min1).
```

A typical query in this benchmark performed on the flight timetable databases is:

```
connecting(london,City3,paris,1440).
```

asking for a list of cities via which London and Paris can be connected by flight provided that the total time required is within 1440 minutes (24 hours).

Table 2 shows the performance results when applying the second set of benchmarks. Benchmark test 2.1 uses the first application program and test 2.2 the second application program.

Table 2. Comparative study of performances of the system with and without cacheing using the second set of benchmarks (all times given in msec.)

Benchmark test	Database size	Without cacheing	With cacheing	Percentage improvement
2.1	20	480	420	12.5
	56	660	560	15.2
	240	760	620	18.4
	1056	1780	1380	22.5
	4032	6320	3260	48.4
	15500	13960	10840	22.3
	39800	31440	23520	25.2
2.2	20	800	760	5.0
	56	1280	1200	6.2
	240	2740	2560	6.5
	1056	9120	8520	6.6
	4032	50120	47720	4.8
	15500	340880	328520	3.6
	39800	1288820	1246380	3.2

It can be seen from Tables 1 and 2 that the percentage improvement in benchmark test 1.2 is smaller than that in test 1.1 and that in benchmark test 2.2 is smaller than that in test 2.1. This is caused by the join between the two subgoals - *vertex(M,N,0)* and *edge(X,M,bound_value)* in benchmark test 1.2, *flight(london,City3,X1,Y1,Z1,M1,N1)* and *flight(City3,paris,X2,Y2,Z2,M2,N2)* in benchmark test 2.2. While the Prolog system is evaluating the second sub-goal, the solutions to the first sub-goal are cached in the solution buffer area. At this point, *M* or *City3* in the second sub-goal is bound which makes the evaluation of the second sub-goal a process of search for a single fact in the external database. The retrieval of the single solution to the second sub-goal in the external database does not exploit the cache mechanism and in fact the cache mechanism caused a small overhead in evaluating the sub-goal. The *next query lookahead* mechanism provided by the RDS system is not very efficient in benchmarks where two sub-goals share a variable. However, it is well suited to deal with two independent sub-goals with no joint variables. This is shown in benchmark test 1.3 in Table 1, where the percentage improvements are considerably larger.

Some aspects of the cache and lookahead mechanism might be taken into account to make further improvements to system performance. Since decompiling the code for the next sub-goal is time-consuming when a *next query lookahead* takes place and it would be better to perform the *next query lookahead* when the current sub-goal and the next sub-goal are independent, the Prolog compiler might be modified to achieve this. A user program can make a declaration to the compiler which indicates the external predicates and dependencies among sub-goals. With this approach, some of the lookahead information can be compiled into the abstract machine instructions. Therefore, improved system performance might be obtained.

5. Conclusion

The coupling of Prolog and database systems has raised a problem of buffering the solutions retrieved from the external databases. This paper describes the cacheing and lookahead strategy adopted in a Prolog database system based on a multi-process architecture. The strategy has provided a method to accommodate a large number of retrieved solutions in a buffer area, to minimize the number of accesses to the external databases, and to provide some extent of parallelism in the Prolog database system. Some performance figures are given to evaluate this strategy. This strategy has produced a considerable improvement in performance in cases where a large number of solutions are retrieved from the external databases.

Acknowledgements

The authors would like to thank the Prolog Database Machine Group in the Department of Computer Science, Heriot-Watt University and in particular Mr D. J. Ferbrache for their support and advice. Thanks also go to the EC who through the Esprit project European Declarative System (EDS) have provided funding for the presentation of this paper at British Computer Society's 14th Research Colloquium on Information Retrieval.

References

1. J. Bocca, "EDUCE - A Marriage of Convenience: Prolog and a Relational DBMS," *Proc. of the 1986 Symposium on Logic Programming*, pp. 36-45, Salt Lake City, Utah, (September 1986).

2. J. Bocca, M. Dahmen, M. Freeston, G. Macartney, and P. J. Pearson, "KB-PROLOG, a PROLOG for Very Large Knowledge Bases," *Proc. of the 7th British National Conference on Databases (BNCOD-7)*, Edinburgh, Scotland, (July 1989).

3. S. Ceri, G. Gottlob, and G. Wiederhold, "Interfacing Relational Databases and Prolog Efficiently," *Proc. of the 1st International Conference on Expert Database Systems*, pp. 141-153, Charleston, SC, (April 1986).

4. S. Ceri, G. Gottlob, and G. Wiederhold, "Efficient Database Access from Prolog," *IEEE Transactions on Software Engineering*, **Vol. 15**, (2), pp. 153-164, (February 1989).

5. C. L. Chang and A. Walker, "PROSQL: A Prolog Programming Interface with SQL/DS," *Expert Database Systems, Proceeding from the First International Workshop*, pp. 233-246, Benjamin/Cummings, Menlo Park, CA, (1986).

6. W. F. Clocksin, "Design and Simulation of a Sequential Prolog Machine," *New Generation Computing*, **Vol. 3**, pp. 101-120, (1985).

7. J. S. Conery, in *Parallel Execution of Logic Programs*, Kluwer Academic Publishers, (1987).

8. Y. E. Ioannidis, J. Chen, M. A. Friedman, and M. M. Tsangaris, "BERMUDA - An Architectural Perspective on Interfacing Prolog to a Database Machine," *Proc. of the 2nd International Conference on Expert Database Systems*, Tysons Corner, Virginia, USA, (April 1988).

9. Y. Kohda and J. Tanaka, "Deriving a Compilation Method for Parallel Logic Languages," in *Logic Programming*, LNCS, pp. 80-94, Springer-Verlag, (1988).

10. Y. -J. Lin, V. Kumar, and C. Leung, "An Intelligent Backtracking Algorithm for Parallel Execution of Logic Programs," *Proc. of the 3rd International Conference on Logic Programming*, pp. 55-68, London, (July 1986).

11. P. A. Massey, K. F. Wong, D. J. Ferbrache, and M. H. Williams, "Adapting Prolog for Large Knowledge Based Applications," *Internal Report*, Heriot-Watt University(Edinburgh), Department of Computer Science, (1989).

12. D. S. Moffat and P. M. D. Gray, "Perlog: A Prolog With Persistence and Modules," *The Computer Journal*, **Vol. 31**, (2), pp. 110-115, (April 1988).

13. K. Ramamohanarao and J. Shepherd, "A Superimposed Codeword Indexing Scheme for Very Large Prolog Databases," *Proceedings of 3rd International Logic Programming Conference*, London,England, (July 1986).

14. K. Ramamohanarao, J. Shepherd, I. Balbin, G. Port, L. Naish, and J. Thom, "The NU-Prolog Deductive Database System," *Prolog and Databases*, Ellis Horwood Limited, (1988).

15. D. H. D. Warren, "Parallel Execution of Logic Programs," in *ECRC Workshop on Parallel Logic Programming Systems*, (1987).

16. M. H. Williams, "The Design of a Prolog Database Machine," *Proceedings of the 1st Workshop for SIGKME*, pp. L1-L5, Reading University,England, (Jan 1987).

17. M. H. Williams, G. Chen, D. Ferbrache, P. Massey, S. Salvini, H. Taylor, and K. F. Wong, "Prolog and Deductive Databases," in *Knowledge-based Systems*, pp. 188-192, Butterworth&Co(Publishers)Ltd, (June 1988).

18. M. H. Williams, P. A. Massey, and J. A. Crammond, "Benchmarking Prolog for Database Applications," *Prolog and Databases: Implementations and directions*, pp. 161-187, Ellis Horwood Limited, (1988).

19. M. J. Wise and D. M. W. Powers, "Indexing PROLOG Clauses via Superimposed Code Words and Field Encoded Words," *Proceedings of the IEEE Conference on Logic Programming*, pp. 203-210, Atlantic City, NJ, (Jan 1984).

20. S. Zhou, M. H. Williams, and D. Ferbrache, "Architecture for a Prolog Database System," *submitted for publication to Data & Knowledge Engineering*, North-Holland.

21. S. Zhou and M. H. Williams, "An Assessment of Two-level Codeword Indexing Applied to a Prolog Database," *Information and Software Technology*, **Vol. 33**, (2), pp. 157-162, (March 1991).

61. M.H.WILLIAMS, N.W.PATON, A.ABDELMOTY, A.A.A.FERNANDES and M.L.BARJA, Applying advanced database techniques to geographic information systems, Proceedings of JFIT Conference 93, to appear in March 1993.

APPLYING ADVANCED DATABASE TECHNIQUES TO GEOGRAPHIC INFORMATION SYSTEMS

JFIT Project No. IEATP 4/1/1391

M.H. Williams, N.W. Paton, A. Abdelmoty, A.A.A. Fernandes and M.L. Barja

Department of Computing and Electrical Engineering
Heriot-Watt University
Riccarton, Edinburgh EH14 4AS.
Tel: 031-451 3430
Fax: 031-451 3431

ABSTRACT

This paper outlines certain of the challenges presented to database systems by Geographic Information Systems (GISs), and indicates how these can be addressed using recent developments in database systems. In particular, it is shown how geographic data can benefit from a combination of object-oriented data modelling mechanisms and deductive reasoning facilities, as provided by a deductive object-oriented database system.

1. Introduction

Geographic Information Systems have attracted interest in recent years both from potential users and database specialists alike. Potential users of a GIS include not only geographers, but also environmentalists, planners, local authorities, etc. Users are interested in having a system which will store vast quantities of data associated with different types of maps, and which can perform the complex manipulations required in order to answer meaningful queries about such data. On the other hand, database specialists are interested in the area because it provides a challenging application area for database systems. Current relational technology is inadequate for describing the rich structure of GIS data, and lacks flexible techniques for performing sophisticated analysis.

The Deductive Object-Oriented Database (DOOD) project is developing a database system which combines object-oriented database concepts with those of deductive databases in a sound and consistent manner. To validate this approach some problems arising from geographic information systems are being investigated.

The object-oriented approach provides an effective means of describing the complex features of geographic information. In particular, different classes of data can be identified (e.g. roads, houses, railways, etc), each of which corresponds to particular sets of objects, with particular relationships between them. There is a natural inheritance of properties and operations between classes representing geographic concepts.

In: Proc. JFIT Technical Conf., Keele, March 1993.

Deductive databases are useful for a range of tasks involving geographic data. For example, an important area is the transformation of digitised maps represented by simple geometric entities, such as points and vectors, to a database of objects representing real-world geographic phenomena. Subsequently further deductive rules can be used to determine where one road ends and another begins using incomplete information on road numbers as a guide [Abdelmoty 93].

This paper indicates how deduction and object-orientation can be seen as necessary and complementary mechanisms for a database system which is to be used as part of a Geographic Information System. The paper is structured as follows: section 2 outlines the information storage requirements of a GIS; section 3 provides an introduction to deductive object-oriented databases; sections 4 and 5 indicate how deduction and object-orientation can be used with geographic data; section 6 presents some conclusions.

2. Background on GIS

Geographic Information Systems need to store a wide range of different kinds of data, some of which is of general interest to GIS users (road networks, river courses, etc), and some of which is highly application specific (gas pipe locations, property ownership information, etc). Such information can be classified as belonging to four distinct but closely related dimensions:

1. **Space**: The absolute and relative positions and sizes of data items.
2. **Theme**: A particular interpretation which is placed upon an area of land. Themes can be regarded as consisting of a number of map layers representing land use, weather patterns, soil types, etc.
3. **Resolution**: The scale at which a piece of information is recorded. This depends upon the surveying technique used to collect the information, and different applications need to describe information at very different levels of detail. For example, if the road is to be dug up to fix a gas leak then accuracy is very important to minimise costs and inconvenience, whereas weather details are necessarily much less exact.
4. **Time**: The way in which the use of an area or feature changes with time. It may be important to record such change of use, for example, in agricultural applications, or for legal purposes.

The dimensions of geographic data are depicted in Figure 1. As well as the dimensions mentioned above, complex relationships exist between geographic objects. Such relationships may be spatial (near to, inside, west of, etc), may involve theme (has higher rainfall than, has more acidic soil than, etc), may involve change over time (which fields have been used for oilseed crops during the last five years which have previously been used for grain, etc), or a mixture of dimensions (which areas of forest within the national park are less than 200m above sea level, etc).

Such storage and analysis requirements present a significant challenge to database systems, and are discussed in more detail in sections 4 and 5. In existing GISs the emphasis is often on the storage of low-level spatial data, but future systems are likely to focus upon adding value to the underlying cartographic data, thereby enabling applications to be constructed easily from tailorable high-level components.

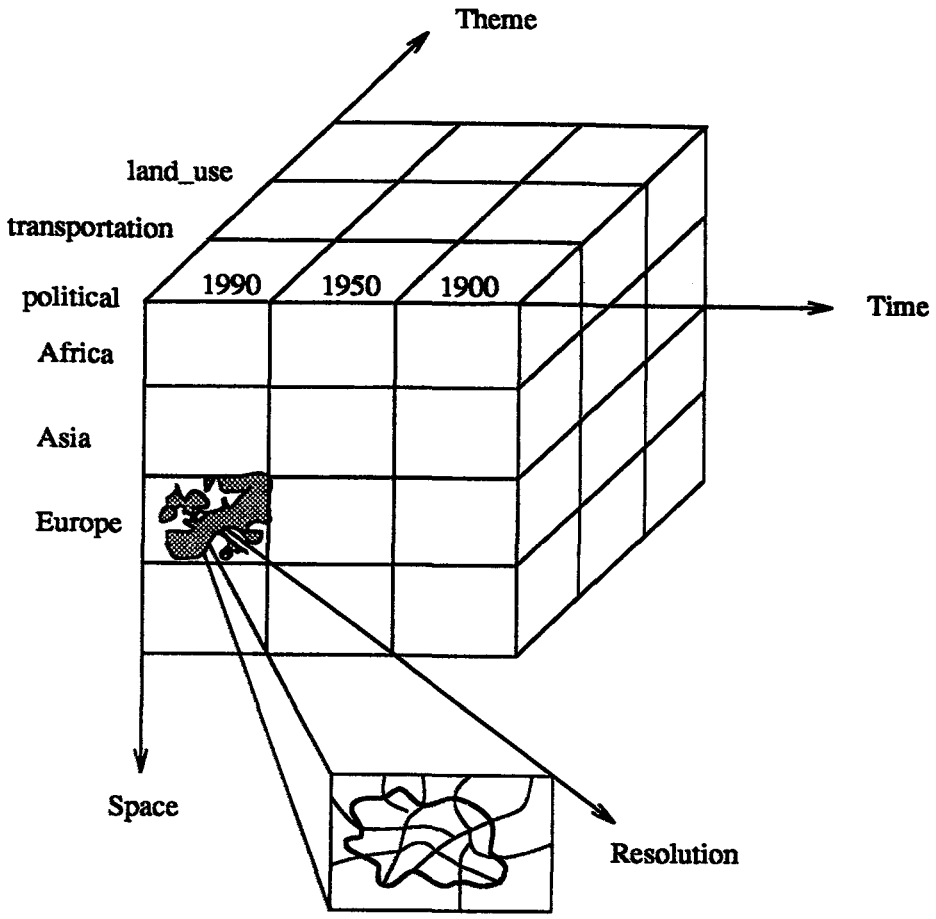


Figure 1: Dimensions of Geographic Data

3. Background on DOOD

The areas of deductive databases and object-oriented databases have independently been the focus of considerable interest to database researchers for almost a decade. Each approach endeavours to overcome recognised weaknesses in the relational model for advanced applications.

Research into *deductive databases* (DDBs) looks to develop powerful rule-based languages for inferring new facts from those which are stored explicitly in a database [Ceri 91]. Deductive databases use languages which are based upon a subset of first order logic, and store data using the same tabular structure of the relational model. Thus DDBs do not provide richer data structuring facilities than the relational model, but they do enable additional information to be derived using a concise, declarative formalism.

By contrast, *object-oriented databases* (OODBs) are based upon new richer data models, and are normally programmed using highly procedural data manipulation languages [Gray 92]. As such, they improve upon the modelling power of the relational model, but have developed in an ad-hoc manner, and often lack both the theoretical underpinning of relational/deductive models and their declarative data retrieval languages.

The aim of research into *deductive object-oriented databases* (DOODs) is to combine the benefits of deductive and object-oriented systems without sacrificing the strengths of either. A number of approaches have been taken to the development of such systems [Fernandes 93],

most of which have less expressive data models and manipulation languages than current commercial OODBs. The approach to the development of a DOOD at Heriot-Watt has started with an expressive data model, from which are derived two complementary languages – a logic language which permits reasoning over data described using the data model, and an imperative programming language which is used to manipulate the database and to express complex algorithms [Fernandes 92(a)]. This architecture has a number of advantages over earlier DOODs:

1. **Rich Data Model**: The system is based upon an independently defined data model, which supports a range of expressive abstraction mechanisms [Fernandes 92(b)].
2. **Declarative Query Language**: The logic language is able to specify complex derived concepts, without being complicated with extra-logical features associated with data manipulation.
3. **Imperative Manipulation Language**: The database is manipulated using an imperative programming language which can be smoothly integrated with the declarative logic language because they both operate on the same underlying data types.

The resulting system is thus able to describe the complex structures associated with geographic data, to define rules which capture important relationships between the stored objects, and to express complex algorithms which are not easily expressed using a declarative formalism.

4. Object-Orientation for GIS

Existing approaches to the representation of geographic data are often characterised by the lack of a uniform framework for describing the features of geographic data – for example, the spatial and aspatial components of geographic phenomena may have to be stored using different formalisms. Future data models for the storage of GIS data should support the following features:

1. Uniform representation of geographic objects (spatial and aspatial).
2. Application independence (extensible object definitions).
3. Conceptual abstraction of geographic objects (input independence).
4. Overlapping objects (objects with same location from different theme layers).
5. Complex structure representation (complex and composite objects).
6. Multiple representations of a geographic object.

An overall aim of the DOOD project is to develop a data model which is application independent, and an approach has been followed in which a general-purpose model has been developed while keeping the requirements of the geographic application in mind [Fernandes 92(b)]. This model has subsequently been applied to the storage of geographic data, as outlined below.

It is common for GISs to model spatial data using low-level concepts such as points, lines and polygons. However, it is possible to abstract over such scale-specific concepts to yield the representation-independent spatial types *site*, *flow-segment* and *enclosure*, as shown in Figure 2. These can in turn be abstracted over to yield application-specific concepts, allowing the user of the system to ignore low-level details.

Application Specific Types	⟷	Representation Independent Spatial Types	⟷	Representation Dependent Spatial Types

This classification of GIS concepts provides a framework within which new geographic concepts can be considered for inclusion within a database, and encourages consistent use of modelling features and terminology across GIS applications. For details of how specific concepts are represented using this approach, see [Abdelmoty 92]. An schema fragment which outlines how roads can be represented using the model is given in Figure 3.

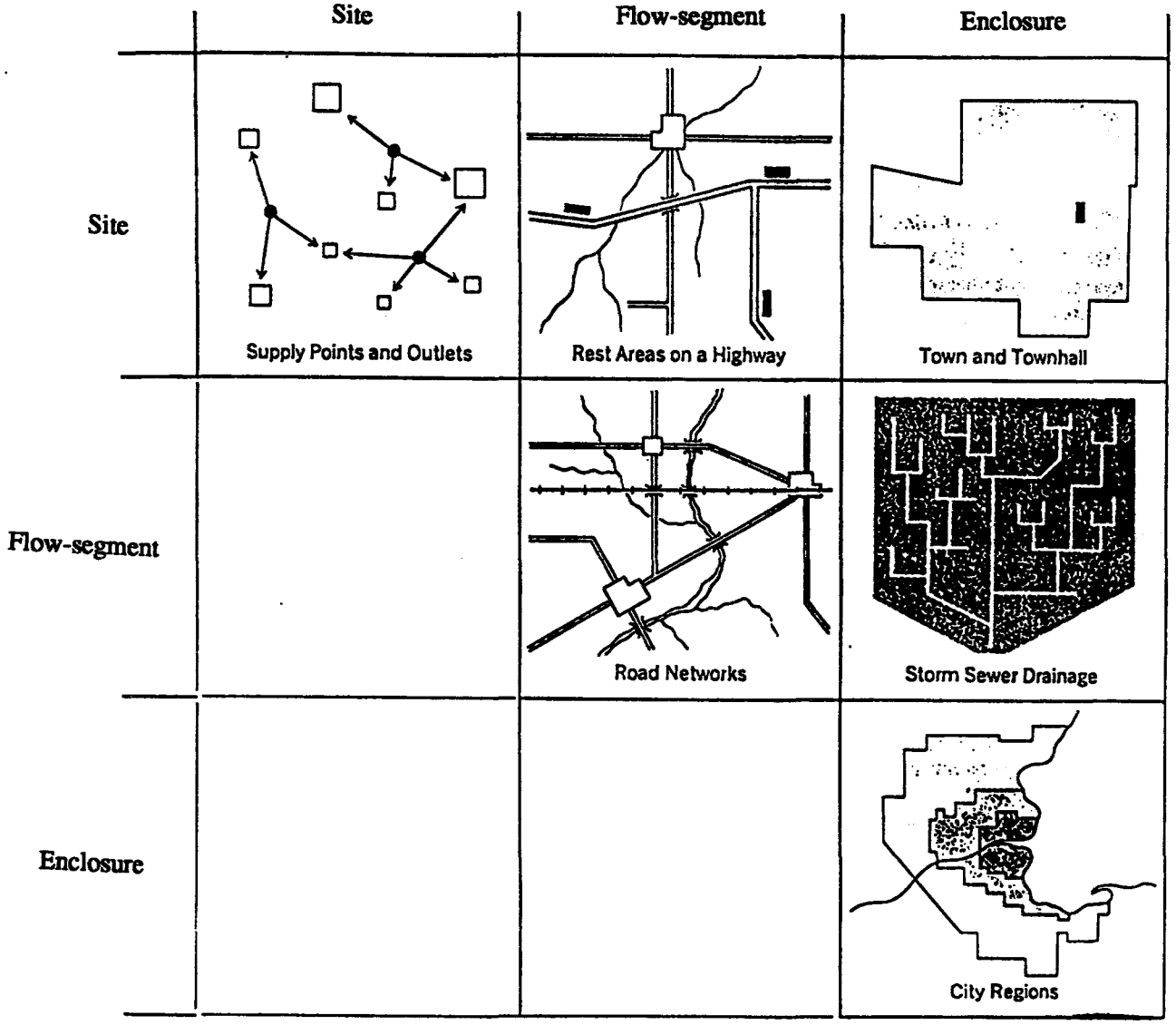


Figure 2: A model for geographic data object relationships.

5. Deduction for GIS

While modelling GIS data cleanly within a uniform formalism is an important first step in the provision of effective database support for geographic applications, the provision of appropriate data analysis facilities is equally important. Conventional database query languages, even when extended with facilities relating to spatial data, are not adequate for handling the variety of retrieval requirements of different geographic applications. The following types of analysis

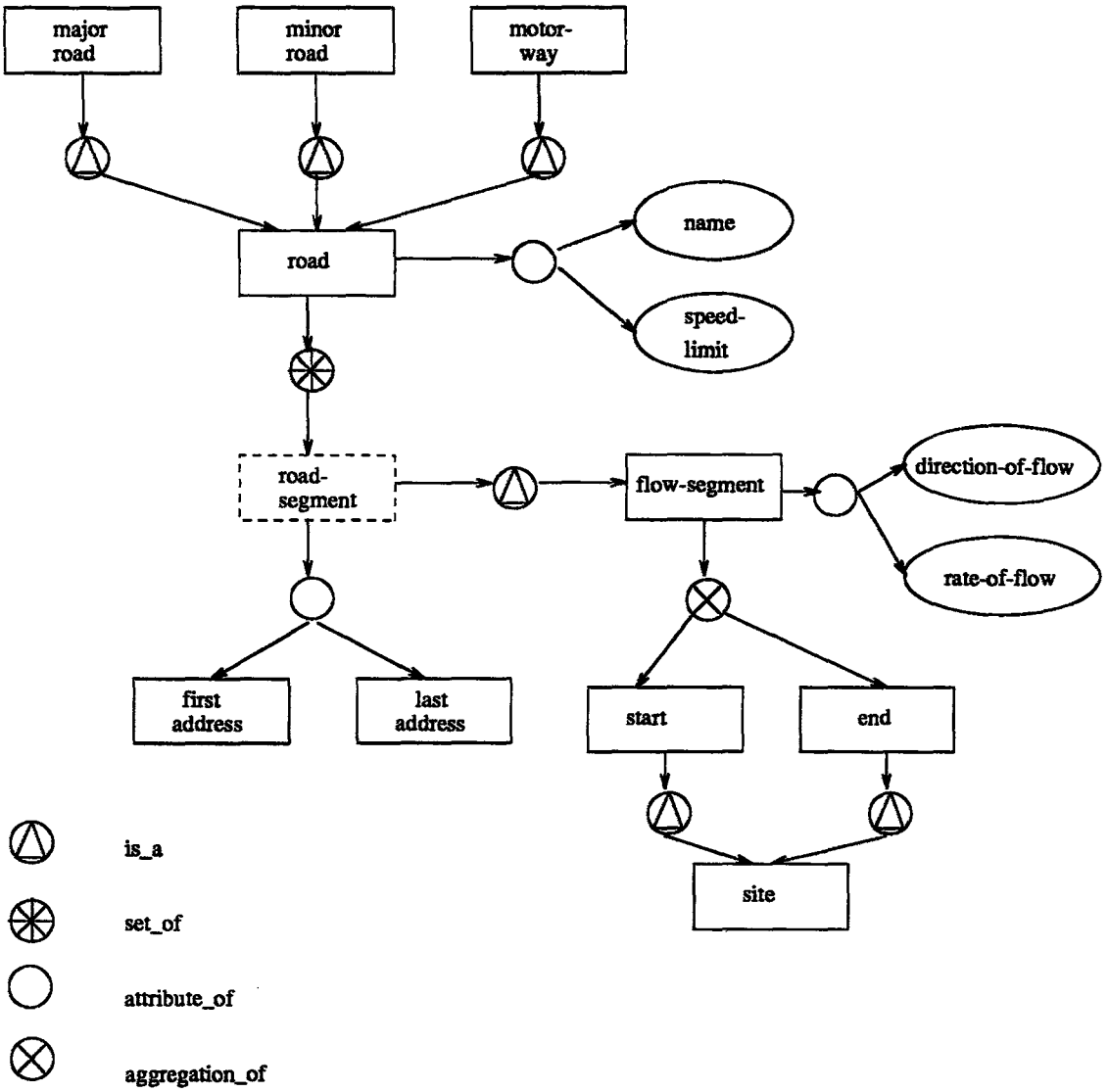


Figure 3: Schema fragment used for representing roads.

are typical of those required by users of GIS data which can be expressed using rule-based techniques (the syntax used here is that of standard deductive rules; for the syntax of a logic language developed by the DOOD project see [Fernandes 92(a)]):

1. The definition of sets of derived objects. There are potentially a vast number of such sets, and their definitions using rules are both concise and easy to understand.

   detached_building(B) ← building(B) & land_parcel(Y) & inside(B,Y).

2. The definition of generic spatial relationships.

   inside(A,C) ← inside(A,B) & inside(B,C).

3. The definition of themes from existing concepts.

```
landslide_risk_area(A) ← land_parcel(A) &
    slope_angle(A,Slope) & Slope > 40 &
    land_cover(A,sparse_vegetation) &
    relative_relief(A,Relief) & Relief > 800.
```

4. Application queries. For example, the following query selects areas of landslide risk within National Parks:

```
← landslide_risk_area(A) & national_park(P) & inside(A,P).
```

5. Feature recognition, which infers the existence of higher-level concepts from low-level map data [Abdelmoty 93].

```
railway(X) ← flow_segment(X) &
    width(X,4.5) &
    maximum_curve(X,Curve) & Curve < 5 &
    maximum_gradient(X,Gradient) & Gradient > -15 & Gradient < 15.
```

The examples given above are extremely simple, and are given only to emphasise different areas to which deduction can be applied. Rules can also be used to perform more complex analysis (such as looking for effective routes through a road network), and can be combined with algorithms for performing complex computation or spatial indexing.

6. Conclusions

This paper has outlined the requirements which geographic applications place upon the data storage component of a GIS. Two important conclusions have emerged:

1. No single paradigm provides all the data storage and analysis requirements of GISs.
2. A fully integrated approach is essential.

The first of these conclusions has been recognised for some time, but has often led to systems which are a poorly integrated collection of independently developed specialised information processors. Such systems are likely to suffer from high complexity and low performance. This paper has outlined an approach based upon a deductive object-oriented database, in which an object-oriented data model is used to define the structural characteristics of the application, and two fully integrated languages – one deductive and the other imperative – are used to perform analysis. Such a system does not in itself guarantee the effective storage of geographic data, but it can be used to provide a framework within which such data can be managed, and provides a context for the sharing of geographic data and operations. Future work at Heriot-Watt will put such a system to work on large-scale geographic databases, to evaluate the current system, and to enable experimentation with different modelling approaches, rule types and applications.

Acknowledgements

The work that resulted in this paper has been funded by the Science and Engineering Research Council through the IEATP programme, and their support is duly acknowledged. We would also like to thank Dr. Keith G. Jeffery of Rutherford Appleton Laboratory for useful discussions on the subject of this paper, and Dr. J.M.P. Quinn representing ICL and Mr Neil Smith of

Ordnance Survey as the industrial partners in the project.

References

Abdelmoty, A., Object-Oriented Representation and Deductive Manipulation: A Geographic Application Example, *DOOD Project Deliverable, Department of Computing and Electrical Engineering, Heriot-Watt University*, 1992.

Abdelmoty, A., Williams, M.H., and Quinn, M.J.P., A rule-based approach to computerised map reading, to be published in *Information and Software Technology*, 1993.

Ceri, S., Gottlob, G., and Tanca, L., Logic Programming and Databases, Springer-Verlag, 1991.

Fernandes, A.A.A., Barja, M.L., Paton, N.W., and Williams, M.H, An Object-Oriented Database for Large-Scale Application Development, submitted for publication, 1992(a).

Fernandes, A.A.A., Paton, N.W., and Williams, M.H., Object-Oriented Databases and First-Order Theories, submitted for publication, 1992(b).

Fernandes, A.A.A., Paton, N.W., Williams, M.H., and Bowles, A., Approaches to Deductive Object-Oriented Databases, to be published in *Information and Software Technology*, 1993.

Gray, P.M.D., Kulkarni, K.G. and Paton, N.W., Object-Oriented Databases: A Semantic Data Model Approach, Prentice-Hall, March, 1992, 230 pages + xv.

62. M.H.WILLIAMS, A.G.SARTORI-ANGUS and G.P.STROMBERG, Alternative approach to a microprocessor teaching laboratory, Microprocessors and Microsystems, vol. 4, no. 7, 219-222 (1980).

Alternative approach to a microprocessor teaching laboratory

Microprocessors are particularly useful for teaching aspects of computer science at an undergraduate level. **M H Williams, A G Sartori-Angus and G P Stromberg** describe a microprocessor network being developed for this purpose

A recent paper discussed the way in which microprocessors are being used for teaching computer science at Strathclyde University. This article discusses an alternative system being implemented at Rhodes University. The chief difference between the two is the attitude towards breakdowns.

The falling cost of microprocessors has lead to their increasing popularity in teaching certain courses in computer science[1,2]. They are particularly useful for teaching courses on systems programming, including the development of operating systems, data communications, distributed processing and networks, etc., where project work by undergraduate, or even postgraduate, students is impractical on the main computer system within a university. Not only is it possible for student projects in these areas to be performed on microcomputers, but also the simplicity of microprocessors makes them ideally suited for this purpose.

Their usefulness is not limited to these areas alone, however, and one can and should use them for other courses as well, such as the development of simple translators, or for projects involving the control of special peripheral devices including colour graphics, voice input/output, analogue-to-digital and digital-to-analogue converters, etc. They can also serve a very useful purpose as machines on which students can play games without arousing the ire of other serious users of the main university computer.

The advantages of using microcomputers for teaching purposes in these areas have been spelled out by various authors and there is general consensus on this. However, there are differences in opinion as to what sort of configuration one should invest in and how this should be set up.

In a recent paper Colin *et al*[1] described a microprocessor teaching laboratory which is being implemented at Strathclyde University. The system is based on a minicomputer to which is attached a number of small microprocessor work stations, as illustrated in Figure 1. Each workstation consists of a microprocessor with a minimal amount of main memory (1k ROM and 1k RAM), a video terminal and four interfaces. The main system software is resident in the minicomputer, which is responsible for storing the source programs for the microcomputers, compiling or assembling them and sometimes even emulating their execution on a microprocessor. Thus the system is completely controlled by the minicomputer, which performs most of the operations required by the workstations.

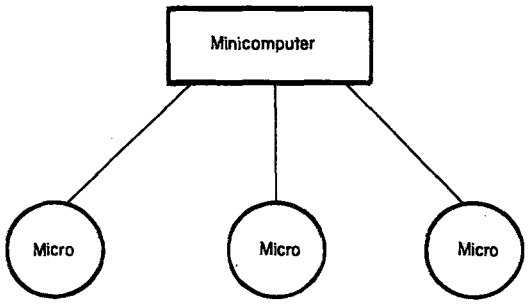


Figure 1. Strathclyde University's configuration

While this structure has its advantages, it also has one serious drawback, namely that the system is completely dependent on one machine – the central minicomputer. When this breaks down, everything comes to a standstill.

A similar type of system has been implemented at Rhodes University. The chief objective of this system is identical to that of the Strathclyde one, namely to give students hands-on experience with computer systems and an insight into the problems of systems programming. However, the second objective differs in that the system should not be dependent on any single machine and that even if breakdowns do occur one should be able to continue to use as many facilities as possible within the reduced system.

THE RHODES SYSTEM

Before the system is described, consider the background behind its development. Prior to the setting up of a microcomputer laboratory, our department, like many others, possessed its own minicomputer (an Interdata 6/16) and students had access both to it and to the university mainframe computer (an ICL 1904S).

When the decision was taken to acquire a number of microcomputer systems, this raised the problem of how

Computer Science Department, Rhodes University, Grahamstown, 6140, South Africa

 0141–9331/80/060219–04 $02.00 © 1980 IPC Business Press

these were to be configured. Two possible strategies are either to purchase a number of very small systems which are not capable of doing very much on their own and are dependent on some other machine (as Strathclyde and others have done) or to purchase larger systems which are capable of operating independently. In our case the latter option was chosen on the understanding that these systems could be used as independent, stand-alone units, or could be connected together into a network in order to make good use of the resources available.

In such a system the microcomputer units could be put to a wider range of uses than would be the case for a number of small microcomputers, and the system as a whole would be more flexible. However, this approach also requires more software to be developed for it in order to get the best advantage from it.

Hardware configuration

The current hardware configuration is arranged as a star network with a single dedicated 6800 microprocessor at the centre and the Interdata 6/16, the ICL 1904S and the other microprocessor workstations connected to this central node, as illustrated in Figure 2. Each workstation consists of a 6809 processor with 32k to 56k RAM, a single 8 inch floppy disc drive, a video terminal and several interfaces. The dedicated controller microprocessor consists of a 6800 processor with 16k RAM.

Command language

Rather than subject students to learning three different command languages, three different editors, and so on, which would inevitably lead to confusion, a single command language was chosen. This is a subset of the command language of the mainframe computer (MAXIMOP[3]). It includes facilities to create and delete files, input source, edit files, copy files, etc. The operating systems of the minicomputer and microcomputer subsystems have been extended to include this command language.

In addition to this the operating systems are being extended to enable the various subsystems to be connected together to form a loosely coupled network (as shown in Figure 2) in which information (usually in the form of files) can be passed from one processor to another in the system. The single dedicated microprocessor at the centre of the star acts both as controller and as an interface between the different components of the system. In particular the protocol of the mainframe computer differs from that used by the other machine in the network (both minicomputer and microcomputers), and rather than tamper with the operating system of the mainframe computer, a dedicated microprocessor offers a safe way of connecting the mainframe computer into the network.

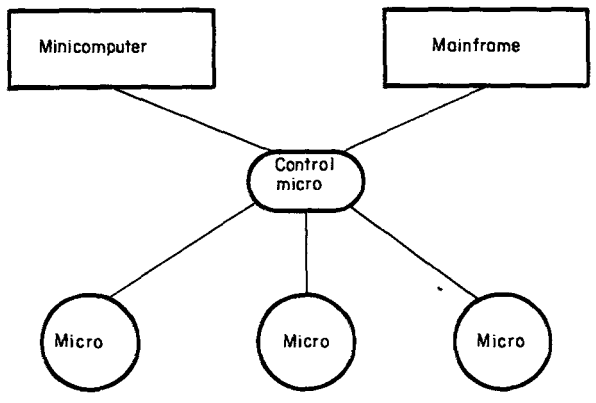


Figure 2. Star network used by Rhodes University

In this form each component of the system (mainframe, minicomputer and microcomputers) is self contained with its own local compilers, assemblers and interpreters but is connected to all the others so that information may be transferred freely between them. Thus if a user wishes to keep a backup copy of files which are resident on the minicomputer or mainframe computer, he can direct these to one of the microcomputers and dump them onto a floppy disc. Likewise if he wants to enter or edit information on the mainframe computer when the machine is not available (during maintenance periods, over weekends, etc.) he may do so using any of the microcomputers provided that he has a copy of the file on floppy disc. He may even work at home on a microcomputer and then copy what he has done into the system at a later stage.

Translators

The programming languages which are currently being used by our undergraduate computer science students are:

- high level languages – PASCAL, FORTRAN, COBOL and BASIC
- low level languages – assembly language of ICL 1900 (PLAN) or Interdata 6/16 (CAL), assembly language of 6809 (6809 assembler)

Of these, COBOL and, for less obvious reasons, FORTRAN are best handled by the mainframe computer. As far as BASIC is concerned there are interpreters for it for the mainframe, the minicomputer and the microcomputers – but, of course, the language accepted by each system has its own peculiarities which make it different from the rest. As it is not a language of major importance to us there are currently no plans to produce a 'standard' BASIC system for the network.

PASCAL is easier to standardize and it may be possible to produce a standard form of PASCAL for all machines in the network so that programs entered on one machine may be run on any machine in the network. In addition a new language BPL[4] (which is a language developed from BASIC and PASCAL) is being implemented and it is hoped that this will be available on the network too.

Besides learning the assembly language of the 6809 microcomputers, our students also learn the assembly language of either the minicomputer or the mainframe computer. Since our minicomputer is not covered by a comprehensive maintenance contract as is the main university computer, it is rather vulnerable and breakdowns can cause serious disruption. For this reason a crossassembler for the language CAL has been implemented on the 1904S as well as an emulator for the Interdata machine code. Thus programs written in CAL can be assembled on either the minicomputer or the mainframe computer. The resulting object program can either by executed directly on the minicomputer or its execution can be emulated on the mainframe computer. The assembly listings produced by the two systems are identical as are the results of execution (except in certain special circumstances, where, for example, machine timings are important).

Crossassemblers and emulators for the 6809 are being developed for the minicomputer and for the mainframe so that a program written in 6809 assembler can be assembled and executed on any machine in the system.

System software

The system software which is required for the network as a whole is as follows:

- control program in the control microprocessor
- modifications to the minicomputer operating system to cater for the common command language and the network link
- modifications to the microcomputer operating system to cater for the command language and the network link
- 6809 crossassembler for the minicomputer and for the mainframe
- 6809 emulator for the minicomputer and for the mainframe
- CAL crossassembler for the mainframe
- Interdata 6/16 emulator for the mainframe

Future expansion

The system is currently designed to cater for up to six microcomputer subsystems. When this number is exceeded a second control microprocessor will need to be introduced, as demonstrated in Figure 3. This will require a slight modification to the control program.

In due course the intention is to develop compatible software for other microprocessors and add these to the system.

Financial considerations

Colin *et al* set the cost of a development system or independent microcomputer subsystem at £6000: these can, however, be obtained at a cost in the region of £2000 – 2500. The cost of the control microprocessor is close to £300. If one includes the minicomputer at, say, £25 000 then the cost of a system with a minicomputer and five independent microcomputer units is less than £38 300 (as opposed to £30 000 for the Strathclyde equivalent system).

This is slightly more expensive than the Strathclyde system, but it is also more powerful since the minicomputer is not dedicated to performing functions for the microcomputer subsystems and the microcomputer subsystems themselves are capable of doing a lot more.

The system could also be configured without the minicomputer in which case the crossover in the cost function between the Strathclyde system and the one described here occurs at about 15 microprocessor subsystems or workstations. For anything less than this the system described here will be cheaper; for more microprocessor units the Strathclyde system will be the less expensive.

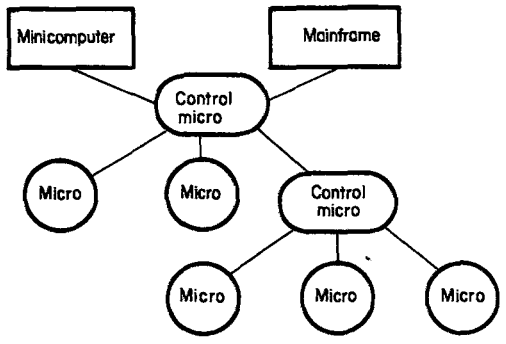


Figure 3. Extending the star network

INITIAL REACTIONS TO THE SYSTEM

Although the system is not yet fully operational, the reactions of students to the first phase have been very favourable. Having learnt to use MAXIMOP on the ICL 1900 they are able to use the other systems with virtually no further instruction. Two additional commands are required for transferring files from one subsystem to another within the network, and these are easy to use. The system should be working fully by the beginning of 1981.

The system should be easily transported to other sites where either an ICL 1900 system running MAXIMOP or an Interdata 6/16 are available. Obviously it is less advantageous and will involve more work to adapt the system to a different minicomputer or mainframe system.

CONCLUSIONS

Any of the computers in the network can be used via a simple common command language. Information can be transferred from any node in the system to any other node. This is particularly useful if parts of the system are not available at certain times, for example during regular maintenance periods, over weekends or (more seriously) when a machine has broken down and is awaiting repair. It is also useful if a user wishes to work at home and is permitted to take one of the microcomputer systems home with him. The common command language makes the system easy to use and avoids the confusion which can arise from having to cope with three different systems.

Some programming languages (e.g. COBOL) are 'machine specific' in that they can only be compiled, assembled or interpreted on a specific machine in the network; others are not machine specific in that they can be translated on at least two different types of machine in the system (for example, the assembly language of the minicomputer). Thus if the assembly language of the minicomputer is taught, there are normally two machines on which programs can be translated and executed. In the event of a breakdown of one of the machines, work can continue on the other without disrupting the course. Another benefit is that emulators can provide more detailed diagnostics than are obtained on executing a program normally on the machine for which it is intended.

Since the microcomputer systems are of a reasonable size (main memory and floppy disc), they can be used for a wide variety of applications, from operating systems to translators, from driving special peripheral devices to a wide range of games, for specific courses or general student projects, for programming in assembly language or in PASCAL. These systems are a lot more flexible than small microprocessor units with little main memory and no disc memory.

The chief disadvantage is the fact that this system requires more software to be developed than does the Strathclyde system, and the addition of different types of microprocessors to the system will require the development of yet more software.

ACKNOWLEDGEMENTS

The authors wish to thank those students who have contributed towards this project, particularly J P Brooks, M J H Will and A M C St Quintin.

REFERENCES

1 **Colin, A J T, Hutchinson, D and Shepherd, D** 'Microprocessor teaching laboratory' *Microprocessors and Microsystems*, Vol 3 No 4 (May 1979) pp 169–172

2 **Danhof, K J and Smith, C L** 'The impact of microcomputers in computer science' *Quaestiones Informaticae*, Vol 1 No 2 (September 1979) pp 38–40

3 *MAXIMOP User Manual*, Queen Mary College, University of London, London, UK

4 **Williams, M H** 'The programming language BPL' to appear in *The Computer Journal*

MICROSYSTEMS '81

11-13 March 1981 **Wembley Conference Centre, London**

CALL FOR PAPERS

Microsystems '81 is the fourth conference and exhibition to meet the rapidly changing information needs of managers and engineers developing and using microprocessor-based systems.

The first two days of the conference are dedicated to design engineers seeking answers to problems encountered in designing and implementing microsystems through discussion and sharing experiences with other experts in the field. The third day is devoted to personal computers and small business systems and their use in industry, commerce and education.

Microsystems '81 is sponsored by the publications *Microprocessors & Microsystems, Computer Weekly* and *Practical Computing.*

SCOPE

- case studies
- communications
- design aids
- distributed processing
- education and training
- industrial control
- interfacing
- multimicroprocessors
- project management
- real-time languages
- signal processing
- software development
- standards
- testing

Synopses of papers for consideration should be sent by 12 September 1980 to: **Robert Parry, Microsystems '81,** IPC Science and Technology Press Ltd., PO Box 63, Westbury House, Bury Street, Guildford, Surrey, GU2 5BH., UK. Telephone: 0483 31261

63. M.H.WILLIAMS, The problem of absolute privacy, Information Processing Letters, vol. 17, 169-171 (1983).

Information Processing Letters 17 (1983) 169–171
North-Holland

5 October 1983

THE PROBLEM OF ABSOLUTE PRIVACY

M.H. WILLIAMS

Computer Science Department, Heriot-Watt University, Edinburgh EH1 2HJ, Scotland (U.K.)

Communicated by W.L. Van der Poel
Received February 1982

Keywords: Operating systems, privacy, security

Security and privacy have been subjects of concern to the designers of operating systems for many years [1]. Ever since users have committed their information to computer systems to be stored on devices beyond their control (such as disk units), and have had to share the facilities of a computer system simultaneously with other users, the protection of information stored in users' files (or indeed anywhere in the computer system) has led to problems of unauthorized access and corruption.

One of the early examples of file protection was that used in the filing system part of the Cambridge multi-access system [2,3]. Each file has a designated *owner* in whose file directory the file occurs. The file owner may nominate one or more *partners* who may have access to the file too. He may also specify a keyword and anyone who can produce this keyword is termed a *keyholder* and may access the file. Finally anyone who does not fall into at least one of the above three categories is simply a *general user*.

When a file is created, any nominated partners and/or keyword must be specified. In addition the type of access (no access, read only, execute only, read and change status, etc.) to be granted to each of the above four categories of user (owner, partner, keyholder and general user) must be given.

Since then many systems have been designed with less flexibility than the Cambridge system, and some with greater flexibility [4]. However, there is one weakness which appears to be a common feature of all of these systems. This weakness arises from the fallibility of human beings.

Consider the following two sets of circumstances:

(a) A user stores important information on a file and then forgets the password which gives him (her) access to the system or which enables him (her) to access the file. Naturally he (she) will expect some means by which he (she) can regain access to the system and recover information from the file.

(b) A user may suddenly leave an organisation (through death, crime, retrenchment or simply to take up an appointment elsewhere) without revealing his (her) password to anyone else in the organisation. As a result work done by the user for the organization may be inaccessible to anyone in the organization. Again it would be expected that a mechanism would be provided whereby the information which belongs to the organization could be recovered from the departed user's files.

For these reasons every system has a loop-hole which enables access to be gained to any user file in the system. Usually this is accomplished by classifying users into two or more categories (cf. the 'super-user' referred to in [5]) and permitting any user from the most privileged category to perform system functions not available to other categories of user. In particular

(i) the system may permit a privileged user to access the password of any user in the system,

0020-0190/83/$3.00 © 1983, Elsevier Science Publishers B.V. (North-Holland)

(ii) it may permit a privileged user to change the password of any user in the system without actually being able to see it, or

(iii) an underprivileged user may not be able to set up or alter his (her) own password at all, and all operations concerning passwords may have to be performed by privileged users.

In all three cases it would be a simple matter for a privileged user to have access to any user's files. The difference between them is that in the first and third cases the privileged user can do so without the victim's knowledge, while in the second case the victim will be aware that something has taken place (since his (her) password will have changed).

It is reliant on the integrity of the individual(s) concerned not to misuse this privilege to inspect or tamper with the contents of another user's files without authorization. Although this arrangement appears to work satisfactorily in practically all cases, there are numerous exceptions where, for whatever reason, abuse has occurred.

If one wanted to create a system with a higher degree of privacy, one would need to close this loop-hole. Indeed suppose that one were to construct a system in which each user may change his (her) own password whenever he (she) wishes but *no-one* may read or alter another user's password. In this case no-one may access another user's files without permission from the owner of the file (unless he (she) has found a weakness in the system). This would be ideal from the point of view of privacy but would leave the problem of human fallibility unaccounted for.

One way of overcoming this latter problem is to introduce the idea of an heir, i.e., another user to whom you are prepared to entrust the contents of your file in the event of something happening to you. Thus whenever a user creates a new file he (she) may be required to specify not only the access attributes for the file but also a list of heirs who would inherit the file in the event of something happening to the user. Just as with passwords the user may alter this list from time to time as the need arises.

Now if a user forgets his (her) password, the only way in which he (she) may regain access to the system is to be deleted from the list of users and reinstated afresh as a new user. When he (she) is deleted from the system, each file which he (she) owned is reassigned so that the first user on the list of heirs becomes the new owner (or if this user has been deleted from the system, then it is assigned to the next user on the list of heirs, and so on). Thus when the user is 'restored to life again' as a new user in the system his (her) first task will be to retrieve his (her) files from those who have inherited them. The onus is on the user to do this since no-one else can do it for him (her). Obviously this does place an additional burden on the user but, on the other hand, this might act as a deterrent to forgetting one's password.

Similarly if a user leaves the company unexpectedly (through death, resignation, etc.), his (her) files will automatically be inherited by other users in the system. However, if several users leave simultaneously it is possible that the list of theirs associated with a file may be exhausted without finding a 'live' user to inherit the file. In this case the file becomes the property of the system until it receives instructions as to what to do with it. The question is then, who should control this process?

In order to answer this question, consider the problem of breach of privacy in such a system. It is no longer possible for some privileged user to access another user's files without seriously affecting a number of other users in the system. Thus a breach of privacy is no longer a simple matter and the temptation to perform such an act must be greatly diminished. However, it is still possible that a privileged user might under certain circumstances be sufficiently motivated to attempt such a breach despite the consequences. If there is any fear of this, then the process by which 'dead' files are released by the system could be made more difficult by requiring instructions from two, or possibly more privileged users before a file is released, otherwise dead files could simply be reassigned by any privileged user. Indeed the user could be allowed to decide for each file whether it required one, two or more than two privileged users to procure its release.

One problem with such a system is the way in which off-line files are handled. Obviously if these files are not on-line when a user is deleted, then by the time that they are brought on-line the user may

have been reinstated and the files may not be reallocated to the appropriate heirs. This would leave a loop-hole which is equivalent to the loop-holes described earlier. To prevent this from happening a unique number may be associated with each user identifier in the system. As each new user is added to the system, the next largest integer is generated by the system and coupled with the user's identifier. This number is used internally by the system and is not accessible to anyone. When a file is allocated, its owner is identified by both the user's identifier and his (her) unique number. This will ensure that if a user is removed from the system, *all* of his (her) files will be reassigned to his (her) heirs.

The approach described here goes some way towards providing a solution to the problem of giving the user absolute control over the privacy accorded to his (her) data within the computer system while still leaving a loop-hole to recover this data under certain circumstances.

References

[1] L.J. Hoffman, Computers and privacy: A survey, ACM Comput. Surveys 1 (1969) 85–103.

[2] D.W. Barron, A.G. Fraser, D.F. Hartley, B. Landy and R.M. Needham, File handling at Cambridge University, AFIPS SJCC Conf. Proc. 30 (1967) 163.

[3] D.W. Barron, Computer Operating Systems (Chapman and Hall, London, 1971).

[4] D.K. Hsaio, A file system for a problem solving facility, Ph.D. Thesis, University of Pennsylvania, Philadelphia, 1968.

[5] R. Morris and K. Thompson, Password security: A case history, Comm. ACM 22 (11) (1979) 594–597.

64. M.H.WILLIAMS, A useful trace facility for microcomputers, Microprocessors and Microsystems, vol. 5, no. 3, 99-102 (1981).

A useful trace facility for microcomputers

A powerful trace facility for 6800 microprocessor systems which does not require any hardware modifications is described by **M H Williams**

Recently a single-stepping debugging aid which could be used on 6800-based microprocessor systems provided that slight modifications are made to the hardware of the machines was described. This paper describes a more elaborate trace facility which does not require any modifications to the hardware but which uses the software interrupt facility in combination with an interpreter for branch instructions as the basic mechanism for single-stepping.

The usefulness of some form of single-stepping or trace facility as an aid to debugging programs has long been acknowledged, and many high-level language systems (compilers or interpreters) provide such a facility[1]. Tracing facilities at a low level are less common but even here trace facilities are available on some machines. In some cases the addition of tracing facilities at this level is dependent on modifications being made to the hardware[2] but this is not always the case.

A simple single-stepping facility (such as that described by Cahill[2]) will trace each instruction executed. In cases where the number of instructions being executed is small, such a facility is very useful; however, if the number of instructions being executed is large, a simple single-stepping facility can be rather cumbersome. In this case what is required is some mechanism whereby the tracing facility can be switched on and off.

The tracing facility described here is a flexible debugging aid which has been written in 6800 assembler and added to the SWTBUG monitor program for 6800 microprocessors. It does not require any modifications to the hardware and it can be switched on and off according to the position in main memory of the currently executed instruction and to any additional predefined conditions specified by the user.

PRINCIPLE OF THE SYSTEM

The tracing mechanism may be set up to operate in any number (from 1 to 254) of contiguous regions of main memory. These regions are referred to as trace regions. Within each trace region one of three modes of operation is possible.

Display mode

This is somewhat similar to the TRACE facility in BASIC. Whenever an instruction is to be executed which lies within a trace region whose mode is display mode, the address of that instruction together with its first byte and the states of the internal registers (in the case of the 6800 these are: condition codes, accumulators A and B, the index register and the stack pointer) are displayed before the instruction is executed.

Store mode

If the mode of a trace region is store mode then whenever an instruction in such a region is executed, its address and first byte together with the states of the internal registers will be placed on a circular list before the instruction is executed. One may use a separate list for each trace region, or two or more trace regions may share a common list. When a list becomes full, further additions to the list will cause the least recent entries to be overwritten by the incoming ones.

Stop mode

Whenever an instruction is to be executed which is situated in a trace region whose mode is stop mode, execution will stop before the instruction is executed. The address of the instruction, its first byte and the states of the internal registers will be displayed and control will then return to the monitor program. The user may display and alter any memory locations, display registers, display the contents of a trace list, etc. Finally execution may be continued by typing the appropriate start command (in the case of the 6800, the SWTBUG command G) which will cause the instruction to be executed and the system to proceed to the next instruction.

In addition the tracing mechanism in each region may be specified to be a conditional trace with some associated condition C, in which case the appropriate trace will apply whenever an instruction in this region is executed and the condition C is true; if no condition is specified, the system will assume the tracing mechanism to be unconditional, and the appropriate trace action will be performed every time an instruction in this region is executed.

USER'S VIEW OF THE SYSTEM

The command language used by the SWTBUG is a very crude one in which each command consists of a single letter followed by zero or more parameters. Since this trace system has been designed to be added onto the SWTBUG system, the commands have a similar format. From the user's point of view there are three additional commands which are required in order to use the system:

- T – sets the trace regions (TRACE)
- A – adds the conditions to the trace regions (ADD)
- S – displays the contents of a trace list set up in store mode (SHOW)

Consider each of these commands in turn.

Computer Science Department, Heriot-Watt University, 79 Grassmarket, Edinburgh, EH1 2HJ, UK

 0141–9331/81/030099–4 $02.00 © 1981 IPC Business Press

T command

In order to use the tracing facility one must first set up the trace regions which are required. This is done using the T command which has the form

T *n m*

with *n* = number of trace regions desired (a hexadecimal number between 01 and FE)
m = start address of an area of length 7**n*+*i* byte which can be used to store the details of the trace regions; *i* is the number of byte required to store the condition information

The system will then attempt to read *n* entries (each on a separate line) of form

x y z

with *x* = start address of a trace region
y = end address of a trace region
z = mode of the trace region. This may be 00 (display mode), FF (stop mode) or a hexadecimal number between 01 and FE (store mode — the number denotes the number of the list on which entries are to be stored)

To alter the details of a trace region, a T command of form

TFF *n*

may be used, where *n* denotes the number of the trace region to be altered. The system will then require to read a single entry of form

x y z

as above. These details will then replace the details currently stored for region *n*. A command of form

T00

will remove the tracing mechanism from all trace regions, and delete the details of these regions.

Lists

If a T command is given which necessitates the use of lists (either T*n m* or TFF *n*), then at the end of the *x y z* sequence the system will ask for the size and start address of each new list required. For example if lists 1 and 2 are called for, a typical conversation might be:

```
GIVE SIZE AND START OF LIST1*10 2000
GIVE SIZE AND START OF LIST2*FF 3000
```

The size *n* represents the number of elements of the list and must be a two-digit hexadecimal number. The actual space occupied by any list is 6+*n**10 (where 10 is a decimal number). The start address must be a four-digit hexadecimal number.

A command

When the trace regions are set up with a T command, they are set up as unconditional trace regions. If one wishes to add a condition to one of the regions, one types the command

A*n*

with *n* = the number of the trace region (a two-digit hexadecimal number in the range 01 to FE)

The system will then expect the condition which is to apply to trace region *n*. The first line of a condition is a relation of the form

address relop value

where address may be a four-digit hexadecimal number or a letter denoting a register (e.g. S = stack pointer, X = index register, H = half-carry bit, N = negate bit, etc.)
value may be a two-digit hexadecimal number followed by a space or a four-digit hexadecimal number
relop may be any of the relational operators <, <=, >, >=, = or <>

Each subsequent line has the form

boolop address relop value

where boolop is one of & (AND) or ! (OR),

terminated finally by a line containing a single question mark

?

For example, the command

```
A02
X=0011
&2A33>00
?
```

will create a condition for trace region 2 which will cause the trace to be performed only if the content of the index register is equal to the hexadecimal value 11 and the content of location 2A33 is greater than zero.

S command

Once execution has stopped (SWI or stop mode trace), one may display the contents of any trace list by typing the command

S*n*

with *n* = the two-digit number of the trace list concerned.

Example

The following set of instructions

```
$T03 0600
*0100 0200 FF
*2000 2100 01
*1000 1600 02
  GIVE SIZE AND START OF LIST1*10 2101
  GIVE SIZE AND START OF LIST2*20 21A7
$A01
*0200>1FFF
*!X<>0
*?
```

will set up regions as shown in Figure 1. Tracing will only be performed in region 1 if the contents of bytes 0200—0201 exceed the hexadecimal value 1FFF or if the index register does not contain the value 0. In the other two regions the tracing is unconditional.

Once the trace regions, lists and conditions have been set up, the program is entered in the usual way with a J command. On stopping one may display and possibly alter the contents of memory locations, display registers or display the contents of a trace list. If the halt is due to a stop mode trace, execution may be resumed by typing G.

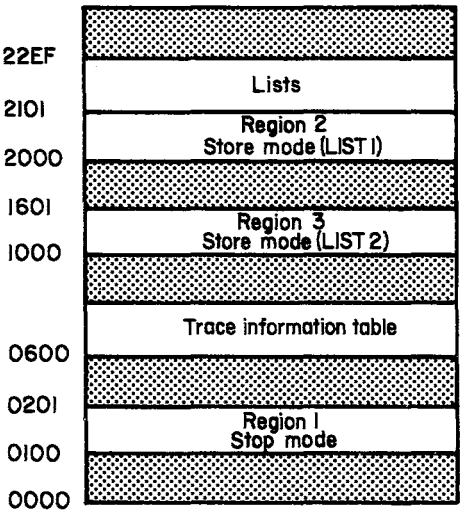


Figure 1. Composition of main memory after setting up trace regions

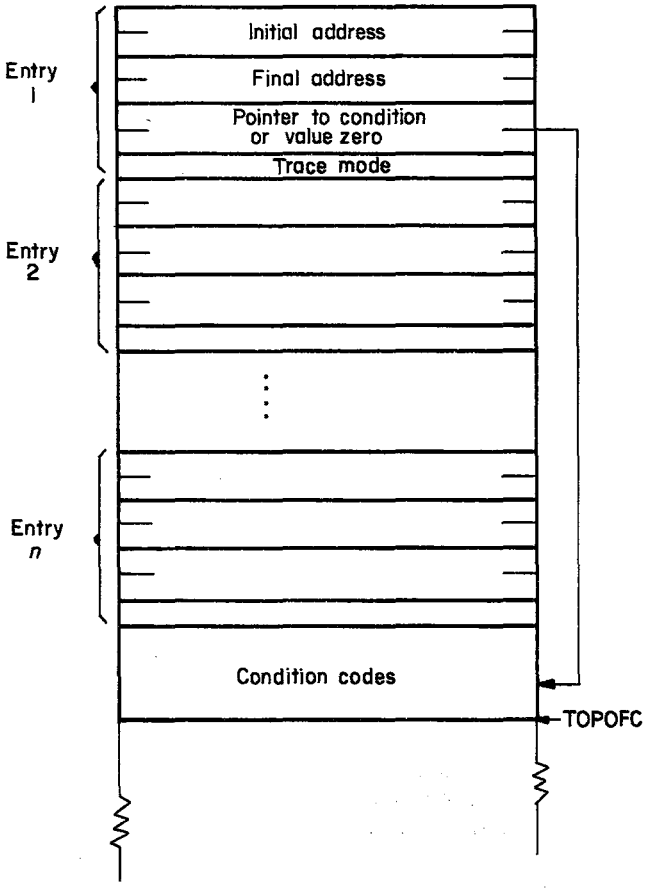


Figure 2. Layout of the trace information table

STRUCTURE OF DATA AND PROGRAM

When a T*n m* command is given, the system expects to read *n* entries of the form *xyz* and set them up in a table starting at address *m*, as shown in Figure 2. If additional lists are called for, the user is prompted to give two numbers: the size (*n*) and the start address (*m*) of each list, and each area is set up as shown in Figure 3 and linked together into a chain.

If the command

A*n*

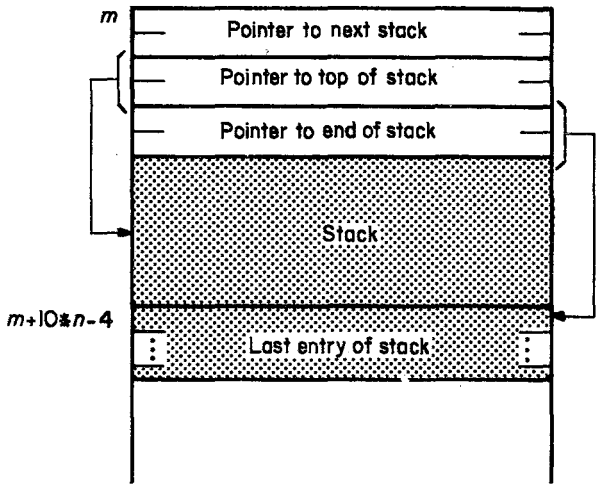


Figure 3. Format of a trace list

is given, the system alters the condition pointer in entry *n* of the trace information table to contain the current value of TOPOFC. As each line of the condition is read in, additions are made to the condition part of the trace information table which have the form:

1-byte CODE 2-byte	ADDRESS 1-byte VALUE for address relop 1-byte value
1-byte CODE 2-byte	ADDRESS 2-byte VALUE for address relop 2-byte value
1-byte CODE 2-byte	VALUE for 2-byte register relop 2-byte value
1-byte CODE 1-byte	VALUE for 1-byte register relop 1-byte value
1-byte CODE	for AND or OR or terminator

until the terminator '?' is reached.

The overall structure of the trace subsystem is described in flowchart and algorithmic form in Figure 4.

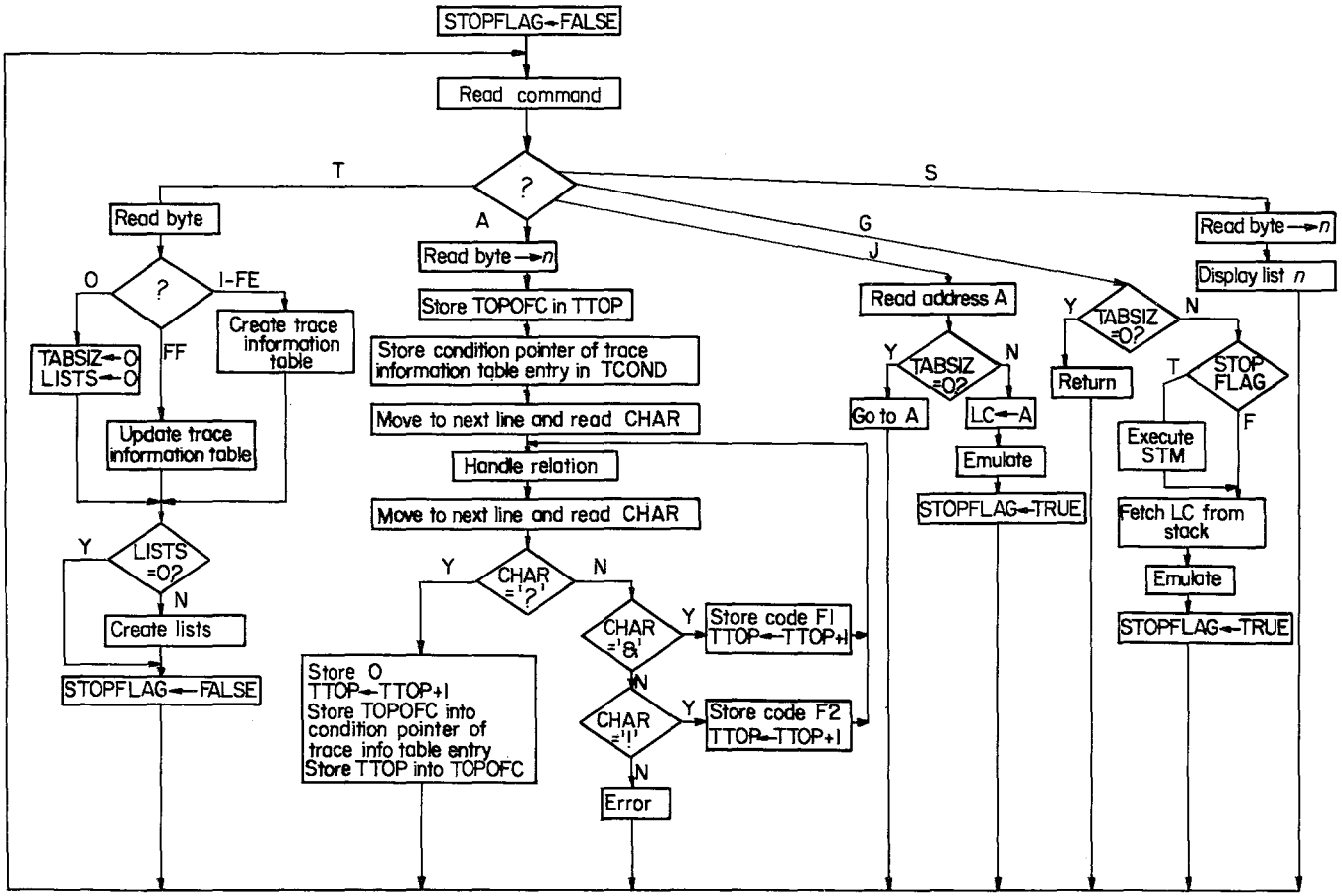
CONCLUSIONS

The tracing facility described here has three important advantages over a simple single-stepping facility:

- it permits the user to define regions of his program which are to be traced
- any one of three different types of trace mechanism may be associated with each region to be traced
- the user may attach conditions to each of the trace regions so that tracing will only take place if these conditions are satisfied

The system has been implemented on a 6800 microcomputer although it should be possible to implement it on other systems too. The system was developed by making a copy of part of the monitor and extending it to include the trace facility. This extended monitor was then run under the control of the existing monitor. Once it is completely finished it can be put onto a PROM in place of the existing monitor.

Consideration was given to the possibility of adding two further relational operators to the A command. These are >> which is used to denote 'changes by greater than' and << which is used to denote 'changes by less than'. These operators would be used to monitor changes in the values contained in registers and store locations. For example




```
EMULATE =
    FLAG:=true;
    while FLAG do begin
        NOTRACE:=true;i:=1;
        while NOTRACE and i≤TABSIZ do
            if LC lies between initial addr and final addr of TRACE INFO TABLE entry i
            then NOTRACE:=false else i:=i+1;
        if NOTRACE then EXECUTESTM
        else if not CONDITIONMET then EXECUTESTM
        else if TRACTYP = DISPLAY then begin PRINTLINE; EXECUTESTM end
        else if TRACTYP = STORE  then begin STORELINE; EXECUTESTM end
        else begin PRINTLINE; FLAG:=false end;
```

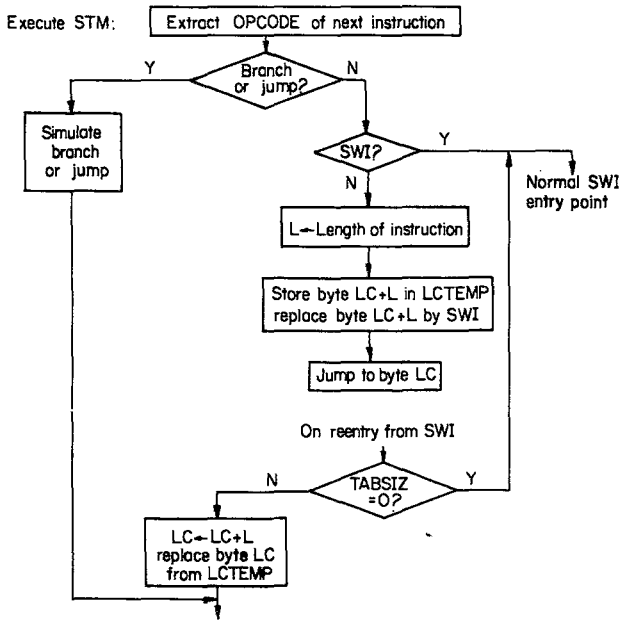


Figure 4. Overall flow diagram for trace subsystem

```
A01
X>>0005
?
```

specifies that the trace mechanism associated with region 1 must only operate when an instruction from this region is executed which causes the value in the index register X to change by more than 5.

However, the implementation of these operators would have made the system considerably more complicated and it was felt that their usefulness was not sufficient to warrant the additional space which would be required for their implementation.

It has been suggested that an additional facility which might be added to the system is the ability to register only branch instructions (including call and return). In this case if the user chooses to operate in this mode, any non-branch instruction in a trace region which satisfies the trace condition for that region will be executed but not traced; any branch instruction in the same circumstances will be traced and executed. This facility has not been incorporated into the system as yet.

ACKNOWLEDGEMENTS

The author wishes to thank Mr C Hannah who assisted with programming and testing the software.

REFERENCES

1 Bull, G M 'Dynamic debugging in BASIC' *Comput J* Vol 15 No 1 (January 1972) pp 21–24

2 Cahill, S J 'Low cost trace facility for development systems' *Microprocessors and microsystems* Vol 3 No 9 (November 1979) pp 400–403

65. M.H.WILLIAMS and H.L.OSSHER, A computer model for instructional purposes, The Computer Journal, vol. 18, no. 4, 333-341 (1975).

A computer model for instructional purposes

M. H. Williams and H. L. Ossher

Department of Computer Science, Rhodes University, P.O. Box 94, Grahamstown, South Africa

This paper describes a computer simulator which is used as an aid to teaching certain aspects of computer science. The properties of the computer to be simulated are determined by parameters which are read in as data by the simulator. Once these are set up, further parameters are used to enter a program coded in the machine code of the simulated computer. The program will be executed with monitor printouts of specified registers or portions of memory.
(Received June 1974)

1. Introduction

A convenient method of communicating the basic principles of operation of a particular computer is by specifying its semantics in some suitable notation. Not only does this provide the receiver with the essential information in a conveniently condensed form but it also causes the person providing the information to think carefully and to crystallise his own ideas on the exact manner of operation of the particular machine.

This technique is thus useful when one is studying the basic manner of operation of a computer in courses such as machine architecture, or as a background to the hardware required for certain aspects of operating systems. However, if one is to use such a system for teaching, it should preferably be both simple and powerful. For example, one could express the semantics of a computer in a system such as VDL (Lee, 1972); but while this is powerful, it lacks simplicity and is certainly not the most convenient system for this type of application.

The present paper describes a simulator which employs a 'language' (semantic notation) that is both simple and powerful. The user provides parameters specifying the characteristics of the particular computer he wants to simulate and the program to be run on this computer. The simulator then simulates the effect of running this program on the specified machine, monitoring the contents of certain registers during the run.

It can thus be used to demonstrate hardware features of a wide variety of existing or hypothetical computers, or to allow students to design computers of their own to meet certain specifications, or to implement their own interpretation of existing computers. The language includes a certain amount of duplication in parts (e.g. input/output) to allow it to be used to teach students at different levels.

The data required by the simulator can be divided into two categories, viz.:

(*a*) COMPUTER DEFINITION PARAMETERS—defining the properties of the computer to be simulated, and

(*b*) PROGRAM DESCRIPTION PARAMETERS—defining the machine code program to be run on the simulated computer and providing 'running instructions' to control its execution.

These two types of parameters are described in the following two sections.

2. Computer definition parameters

The first set of parameters, the computer definition parameters, define the structure of the machine in terms of the registers, the machine code instruction set and the instruction set-up/execute cycle. There are also parameters for defining the peripherals and the character set of the simulated computer. The set comprises seven parameter types in all, namely *B*, *D*, *I*, *E*, *O*, *P* and *C* parameters.

2.1. *The BEGIN parameter*

Before a computer definition, the parameter

B

is used; this causes all pointers to be initialised. It is used since one may want to run several batches of data representing different computers.

2.2. *The definition of registers*

The *D* parameter defines the registers of the computer. Each definition associates a name with a register or set of registers of a particular size as follows:

	D r:*n*	defines a register *r* of length *n* bits,
or	*D r*:*n*,*m*	defines a set *r* of *m* registers each of length *n* bits,
or	*D r*:*n*,*m*,*p*	defines a set *r* of *p* blocks, each containing *m* registers of length *n* bits.

The first form is used to define single registers such as accumulators, next-address register, function register, etc. The second form can be used to define main memory, viz. a set of *m* registers (numbered 0 to $m-1$) of equal length; while the third form is used to define auxiliary memories, such as disc or drum, consisting of *p* blocks or buckets, each containing *m* words of length *n* bits apiece.

More than one definition may be punched on a card, in which case one uses the form

$$D\ d_1; d_2; \ldots; d_k$$

where each d_i is one of *r*:*n*, *r*:*n*,*m* or *r*:*n*,*m*,*p*. If a definition overflows onto a new card, the symbol '&' in the first column of the new card indicates the continuation. An example is shown in **Fig. 1.**

2.3. *Definition of machine code instructions*

The machine code instruction set (or possibly a subset of the full instruction set) is set up by defining the effect of each instruction on the appropriate registers of the computer. This is done by means of parameters of the form:

$$I\,n{:}s_1; s_2; \ldots; s_i$$

where *n* is the instruction code in binary (preceded by the symbol '#') or decimal, and $s_1, s_2, \ldots, s_i$ is a sequence of substeps or statements representing the effect of the instruction code *n*.

Statements may consist of any of the following

(*a*) Arithmetic:	register = expression
(*b*) Data transfer:	MOVE
(*c*) Input:	READ
(*d*) Condition:	IF
(*e*) Unconditional transfer:	GOTO
(*f*) Shift:	SLL, SRL, SLC or SRC
(*g*) Halt:	STOP or ILLEGAL

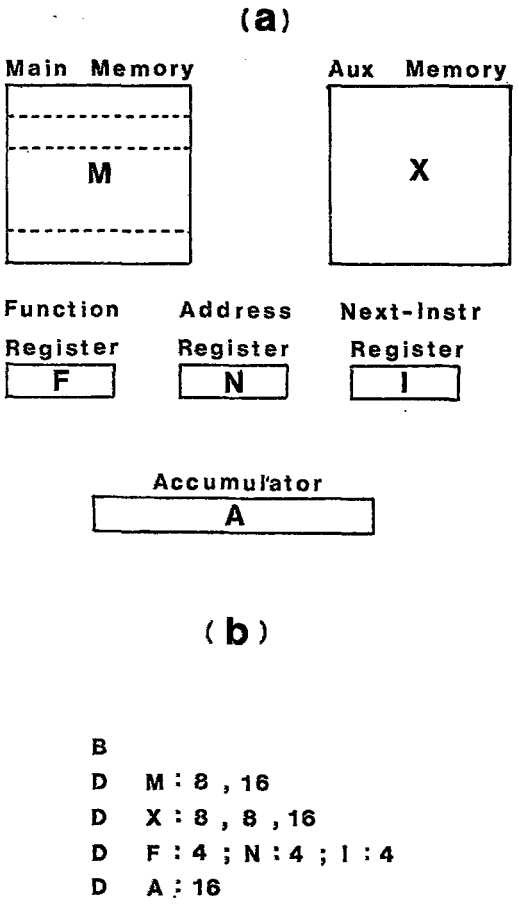


Fig. 1 **An illustration of how the registers of a simple hypothetical computer can be defined using *D*-parameters.**
(*a*) The registers of a simple hypothetical computer: main memory (consisting of 16 8-bit words), auxiliary memory (comprising 16 blocks or buckets, each containing eight 8-bit words), function register, address register and next-instruction register (each of four bits) and an accumulator (16-bits).
(*b*) The corresponding computer definition parameters defining these registers.

(*h*) Execute: EXECUTE
(*i*) Subroutine call: see section 2.5
(*j*) Initiate IO: see section 2.6

Arithmetic
An arithmetic statement is similar to the assignment statement of many high-level languages. The contents of registers (simple or subscripted) and integer constants may be combined by means of +, −, *, / and parentheses in the usual way. An example of an arithmetic statement is

$$A = -(B + C(I))* 2$$

which is interpreted as follows: add the contents of register *B* and the contents of the register from the set *C* pointed to by the register *I*. Multiply the result by two, negate and store in register *A*.

To access only a portion of a register, the bit positions of the beginning and end of the required field must be specified in square brackets after the register name, e.g.

$$A[1, 5] = E[2, 4] + C(I + 1)[3, 7] .$$

The convention adopted here is that the least significant bit is numbered zero.

If one wishes to propagate the sign bit of the number contained in a register before using it in an expression or storing it in another register, the suffix

@(expression)

Table 1 **Specification of a simple set of machine code instructions for a computer with registers as defined in Fig. 1.**

(*a*) Description of instructions

Machine code function	*Interpretation*
#0000	Load main memory register into acc., propagating sign
#0001	Store least significant 8 bits of acc. into main memory
#0010	Add main memory register into accumulator
#0011	Subtract main memory register from accumulator
#0100	Unconditional jump
#0101	Jump if accumulator negative
#0110	Jump if accumulator is non-zero
#0111	if $N \neq 0$, CALL if $N = 0$, EXIT
#1000	Fetch 8 words from auxiliary into main memory from $M(8)$, then call address 8
#1001	Dump 8 words from main memory to auxiliary memory
#1010	Fetch 8 words from auxiliary and exit to address given in accumulator
#1011	Add into store
#1100	Shift left logical
#1101	Shift right logical
#1110	Read *n* words into main memory
#1111	Stop

(*b*) The corresponding instruction definition parameters

I	#0000:	$A = M(N)@(16)$
I	#0001:	$M(N) = A[0, 7]$
I	#0010:	$A = A + M(N)@(16)$
I	#0011:	$A = A - M(N)@(16)$
I	#0100:	$I = N$
I	#0101:	IF(A[15, 15].EQ.1), 1; $I = N$
I	#0110:	IF(A.NE.0), 1; $I = N$
I	#0111:	IF(N.EQ.0), 2; $I = A[0, 3]$;
&		GOTO END; $A = I$; $I = N$
I	#1000:	MOVE $X(0, N)$, $M(8)$, 8; $A = I$;
&		$I = 8$
I	#1001:	MOVE $M(8)$, $X(0, N)$, 8
I	#1010:	MOVE $X(0, N)$, $M(8)$, 8; $I = A[0, 3]$
I	#1011:	$M(N) = M(N) + A[0, 7]$
I	#1100:	SLL A, N
I	#1101:	SRL A, N
I	#1110:	READ $M(A)$, N
I	#1111:	STOP

is appended to the register name. The value of the expression specifies the number of bits to be occupied by the result. For example, if one wishes to add the contents of an 8-bit register Z to the contents of the least significant 16 bits of a 24-bit register Y and store the result in a 16-bit register X, propagating the sign bit of Z before addition, one may write:

$$X = Y[0, 15] + Z@(16) .$$

Data transfer
To transfer a number of registers at a time, say from main memory to auxiliary memory or main memory to main memory, the statement

MOVE r_1, r_2, e

is used, where r_1 is a register from a set of registers,
r_2 is a register from the same or a different set of registers,
and e is an expression specifying the number of

registers to be transferred.

For example,

MOVE $C(8)$, $X(0, N)$, 8

transfers the contents of 8 registers starting at $C(8)$ to the 8 registers starting at $X(0, N)$.

Input

The statement

READ r, n

causes n data items to be read from the data cards and stored in registers starting from register r. This is used to obtain the effect of peripheral transfers at a simple level (without using the P parameter).

Condition

To obtain conditional execution of substeps, the statement

IF (condition), n

is used. A condition has the form

expression relation expression

where relation is one of the following: .GT., .GE., .EQ., .LE., .LT. or .NE., and n is a constant. For example,

IF(A[4, 7].NE.B[0, 1]@(4) + C), 3

If the condition is true, then the next n substeps are to be executed; otherwise they are to be skipped.

Unconditional transfer

The statement

GOTO n

where n is a constant, causes control to be transferred to the nth substep of that instruction definition. Similarly

GOTO END

causes all further substeps of the instruction definition to be skipped.

Logical or circular shift

Statements of form

SLL r, e
SRL r, e
SLC r, e
or SRC r, e

are used for left or right logical or circular shifts. In each case r is the register to be shifted and e an expression specifying the number of places it is to be shifted.

Halt

The statement

STOP

halts the simulated computer. The statement

ILLEGAL n

halts the simulated computer and prints a suitable message on the line printer.

Execute

The statement

EXECUTE e

causes e to be evaluated and the result to be interpreted as an instruction code. This instruction code is then executed. Finally control returns to the substep following the EXECUTE statement.

A simple set of machine code instructions is illustrated in **Table 1.**

2.4. *Definition of instruction set-up/execute cycle*

The E parameter is used to define the process involved in fetching an instruction from main memory, resolving the separate fields into separate registers, determining the modified address if indexing or indirect addressing is allowed, updating the next-instruction register and finally executing the instruction. It has the form

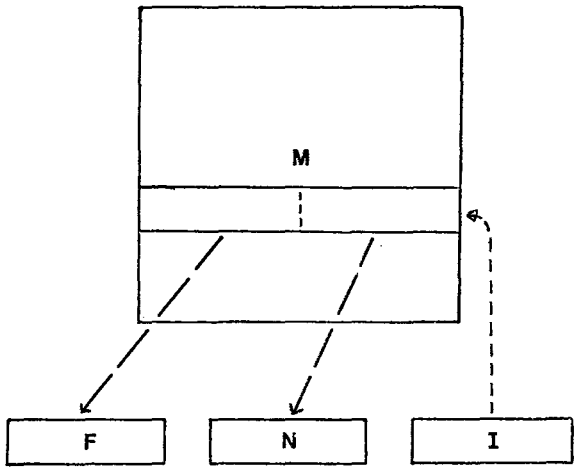
$$E\, s_1; s_2; \ldots; s_j$$

where $s_1, s_2, \ldots s_j$ is a sequence of substeps of the same form as used in instruction definitions. For a simple example, see **Fig. 2.**

2.5. *Definition of subroutines and overflow conditions*

Where sequences of substeps occur repeatedly, subroutines

(a)



(b)

```
E   F=M(I)[4,7]; N=M(I)[0,3]; I=I+1;
&   EXECUTE F
```

Fig. 2 (*a*) **The instruction set-up/execute cycle for the machine of Fig. 1: fetch contents of main memory word pointed to by I and separate function and address components into registers F and N; increment register I and then execute the instruction in F.**
(*b*) **The corresponding instruction set-up/execute parameter for the simulation program.**

```
S   RESET;
S   M(0):#1110010001; M(1):N; M(3):1; M(4):#1110010001; M(5):#1001000000; M(6):2
S   M(7):#1010000000; 1:3
M   F,F,N,I+A
R   5
E   14,#0001111111,#0001111110,#0101111110,#1001000000,#0111000001,55;R
N   R,1
R   1
E   1;15;#1110010000,#1001000010,0,#0010000011,#0001000011,#1000000000
M   R,1
R   12
E   0,#0011111111,#0001011110,#0111011101,#1111000001,#1000100000,#0100001111,1
M   R,1
R   7s
E   #0001010000,15,#0001000010,14,#0001000011,#1000010000,8,0
E   1,#1100001111,#0111011000,#0101011111,0,#0010000010,#0001000010,#1000001001
E   1,#1110100001,#0111011101,4,#1010000010,#0001000001,#1000011000,0
E   0,#1100000001,#0001000003,#0011111111,#0001000011,#1000011111,1
E   3,#0111011000,4,#1010000010,#1000010000,0,0,0
E   #0001010002,2,#0001000001,5,#0001000004,#1010000010,0
R   R,F,I,A
```

Fig. 3 Program Description Parameters to set up and execute a bootstrap loader program which loads a binary loader which in turn loads a large program in the hypothetical computer of Figs. 1 and 2 and Table 1.

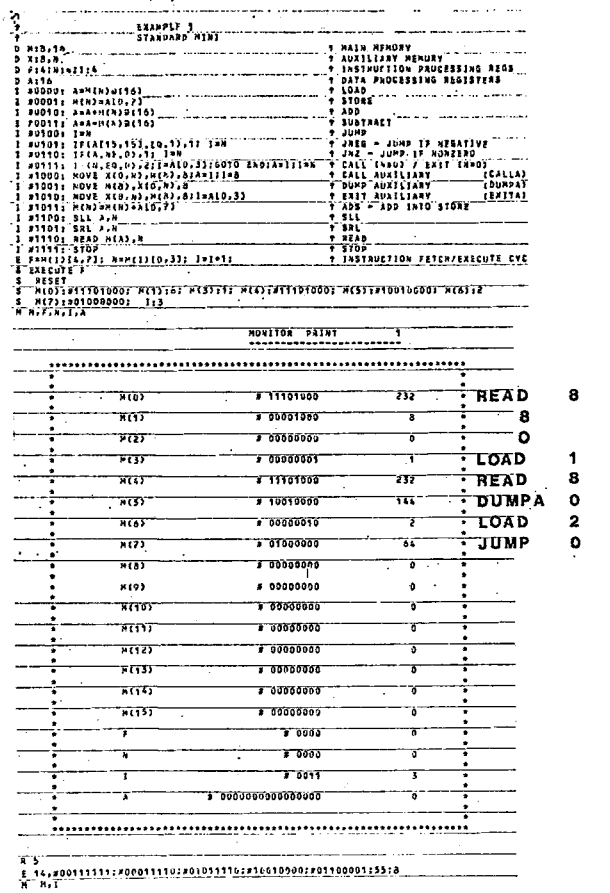


Fig. 4 (a)

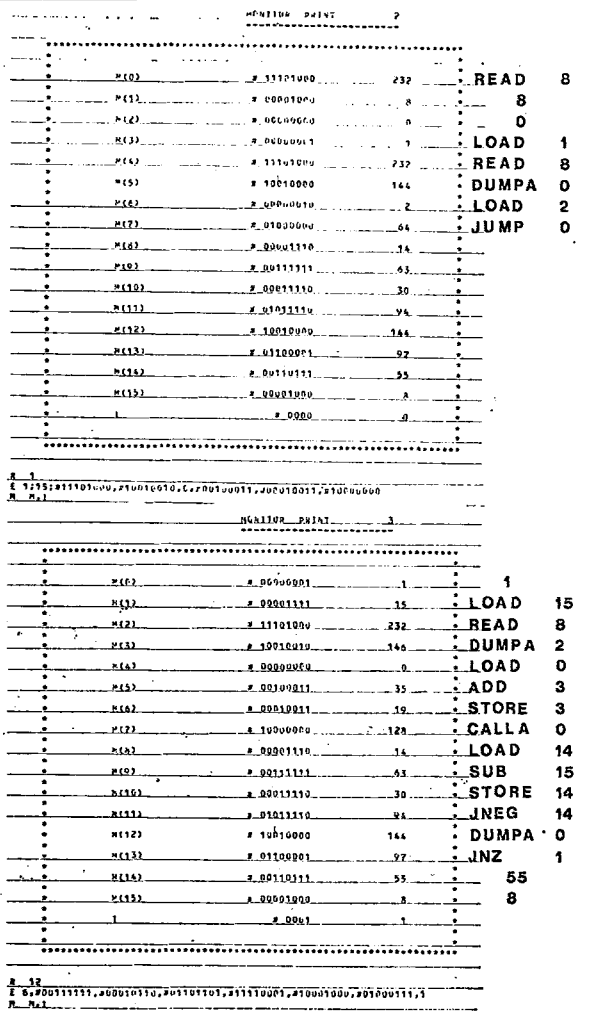


Fig. 4 (b)

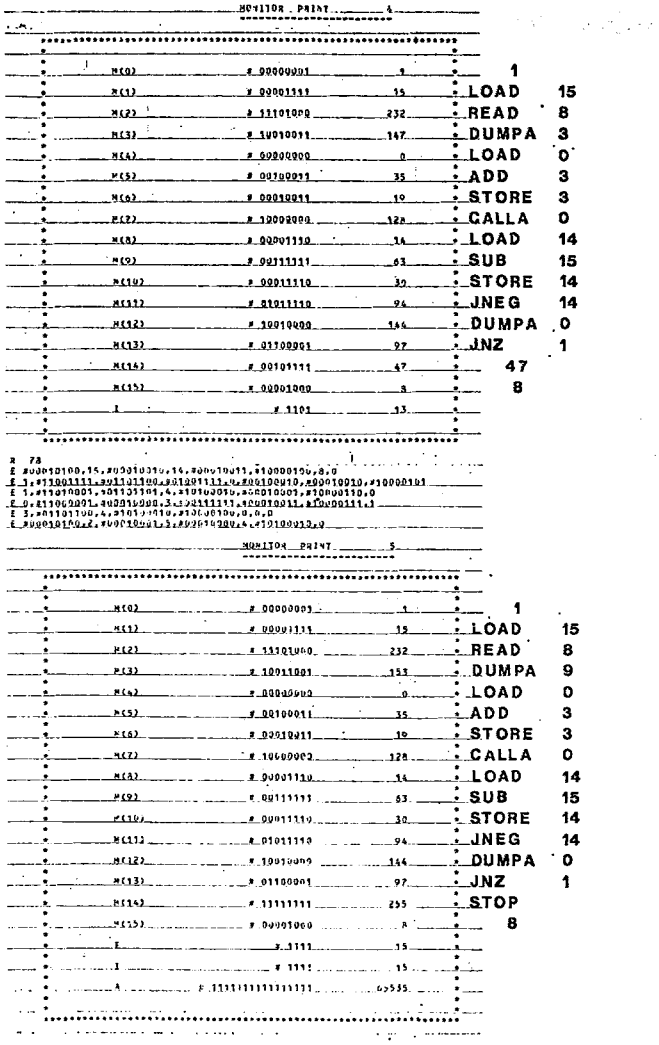


Fig. 4 A simulation run using the parameters shown in Figs. 1 to 3 and Table 1. The contents of registers are printed in binary and decimal form.

may be used to reduce repetition. A subroutine without dummy variables is defined as follows:

$$0\ \%sn : s_1; s_2; \ldots; s_k$$

where sn is the subroutine name and $s_1, s_2, \ldots s_k$ is a sequence of substeps which constitutes the subroutine body. A subroutine with dummy variables is defined by a parameter of form

$$0\ \%sn(p_1, p_2, \ldots p_i) : s_1; s_2; \ldots; s_k$$

where $p_1, p_2, \ldots p_i$ is a list of dummy variables, each of form

? letter

Subroutines are called by statements of form

$\%sn$ if the subroutine has no dummy variables

or $\%sn$ (actual parameters) if the subroutine has dummy variables.

Overflow conditions are treated as a special form of subroutine call. An assignment statement which tests for overflow is written in the form

register ! sn,r = expression

where sn is the name of the overflow subroutine concerned, and r is the register to be set on overflow.

This is interpreted as: evaluate the expression on the right hand side, test for overflow (by calling subroutine sn) setting register r as required and store the result in the register on the left hand side.

An overflow subroutine is defined with three dummy vari-

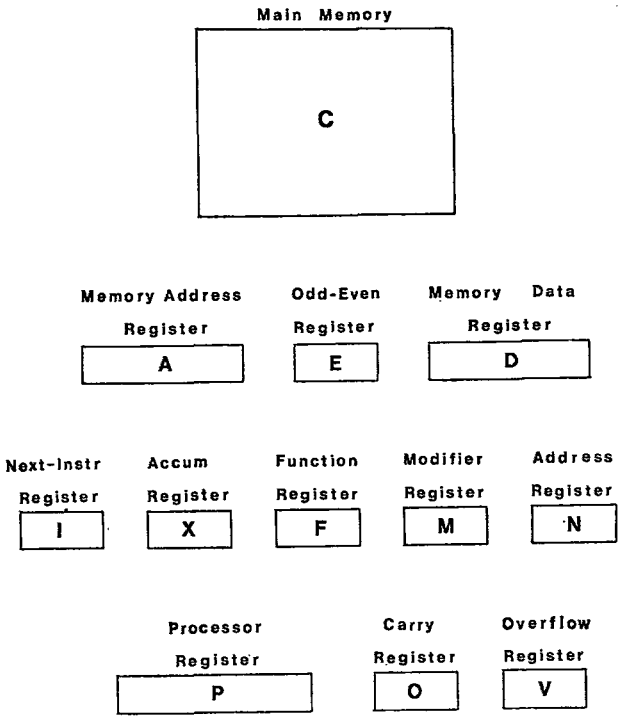


Fig. 5 A simple model of some of the registers of the ICL 1901A.

ables. When it is called, the register on the l.h.s., the overflow register (r) and the value of the expression on the r.h.s. will be substituted as actual parameters in their place.

Examples of subroutines and overflow routines can be seen in Figs. 6 and 8.

2.6. *Definition of peripherals*

To demonstrate independent I/O and processing, details of each peripheral must be specified by means of *P* parameters. These have the form

$$P\ a, c, r, h, n: s_1; s_2; \ldots; s_j$$

where a is the absolute peripheral number (used in the IO statement),

c is the channel number,

r is the name of a single register into which each character being 'read' by the device is stored or from which each character is 'written',

h is a measure of the delay time between requests for hesitations (cycle stealing), which is specified as the number of substeps of the simulated program to be executed between requests,

n is the number of characters in the record to be read or written,

and $s_1, s_2, \ldots s_j$ is a sequence of substeps defining the action to be taken when the peripheral is activated by an

$$\text{IO } a$$

statement in an *I* parameter or a subroutine.

The sequence of substeps $s_1, \ldots s_j$ in a *P* parameter should contain either the statement

INITIATEREAD

or INITIATEWRITE

An INITIATEREAD statement causes control to be returned to the main program sequence, to the substep after the IO statement. After this all substeps executed are counted. When h substeps have been executed, the character read will be stored in the register r and control passed back to the substep after the INITIATEREAD in the peripheral definition. The substeps after the INITIATEREAD could check for parity errors and set the hesitation register. If more characters are to be read, one should branch back to the INITIATEREAD, otherwise exit normally. Similarly for INITIATEWRITE.

An example of this is shown in Fig. 8.

2.7. *Definition of character set*

The parameter *C* is used to define the internal character codes of the computer to be simulated. It has the form

$$C\ \text{“character”}: i$$

where i is the code for the character specified in quotation marks. As with the *D* parameter, any number of these character definitions may be punched on a single card using the form:

$$C\ \text{“}c_1\text{”}: i_1; \text{“}c_2\text{”}: i_2; \ldots; \text{“}c_n\text{”}: i_n\ .$$

3. Program description parameters

This set consists of five parameter types, namely *S*, *R*, *M*, £ and *T*. These control the setup of a program (*S*), the execution (running) of a given number of instructions (*R*), the printing of monitor printouts (*M*), the data required by the program (£) and finally the termination of the simulation (*T*).

3.1. *Setting up a program*

A program is set up by specifying the contents of each register of the simulated computer. The parameter

$$S\ \text{RESET}$$

sets all registers to zero, while the parameter

$$S\ r : n$$

sets up the value n (a decimal or binary number) into the register r. Again, more than one register-value pair may be punched on a card using the form:

$$S\ r_1 : n_1; r_2 : n_2; \ldots; r_i : n_i\ .$$

3.2. *Executing a program*

To control the simulated execution of the program, one has the parameters

R n execute n machine code instructions (or, more strictly, n setup/execute cycles)

and *M* list print the contents of the registers specified in the list (List items may consist of single register names, names of sets of registers or components of form r_1 : TO : r_2, where r_1 and r_2 are registers of a set of registers),

interspersed with the data parameters which have the form £ $n_1; n_2; \ldots; n_i$ where $n_1, n_2, \ldots n_i$ is a sequence of numbers

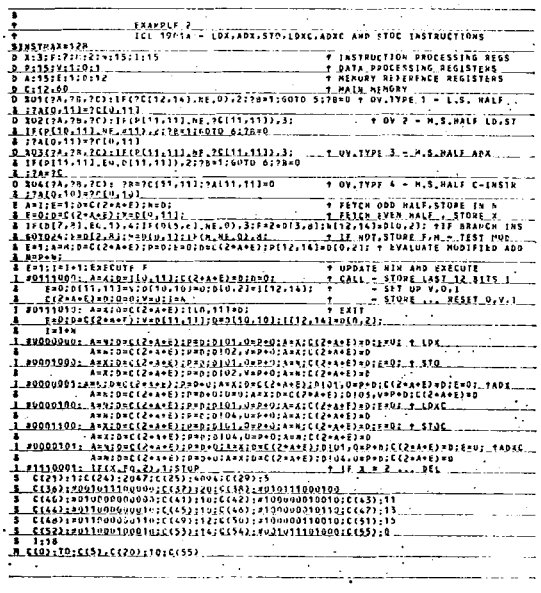
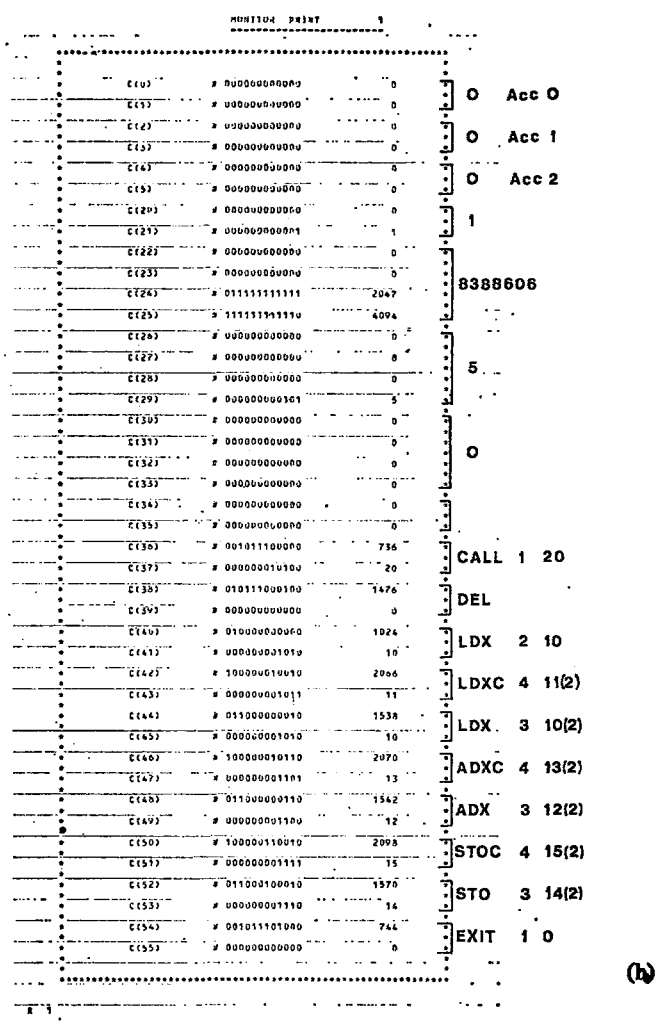
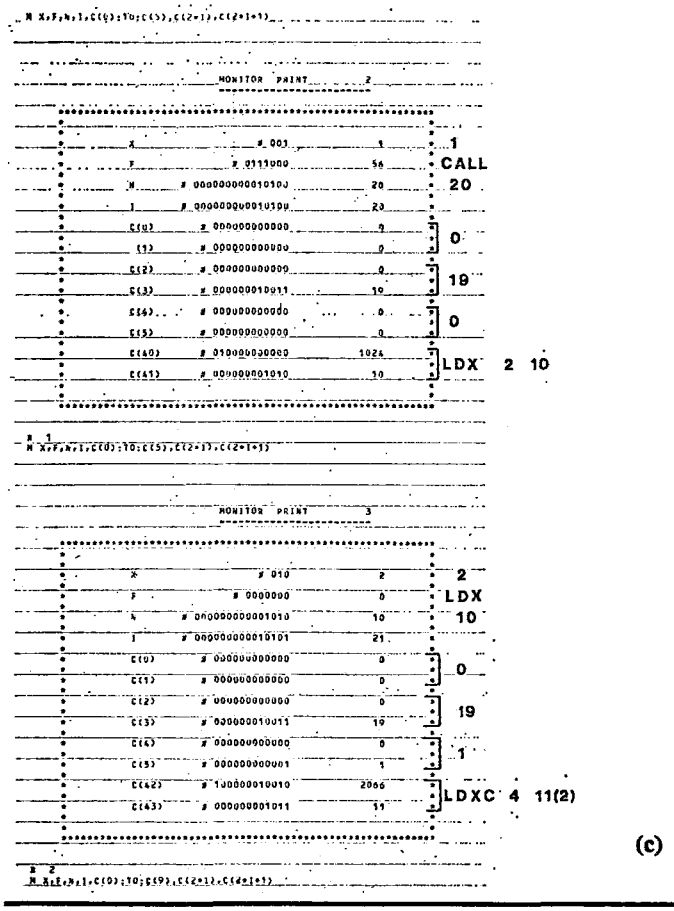


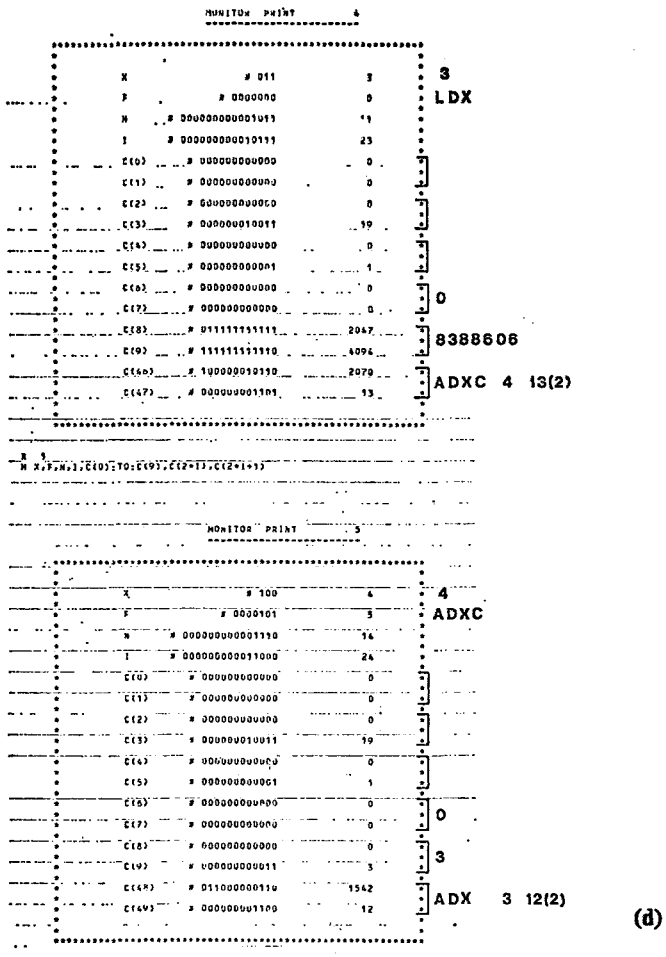
Fig. 6 (a)



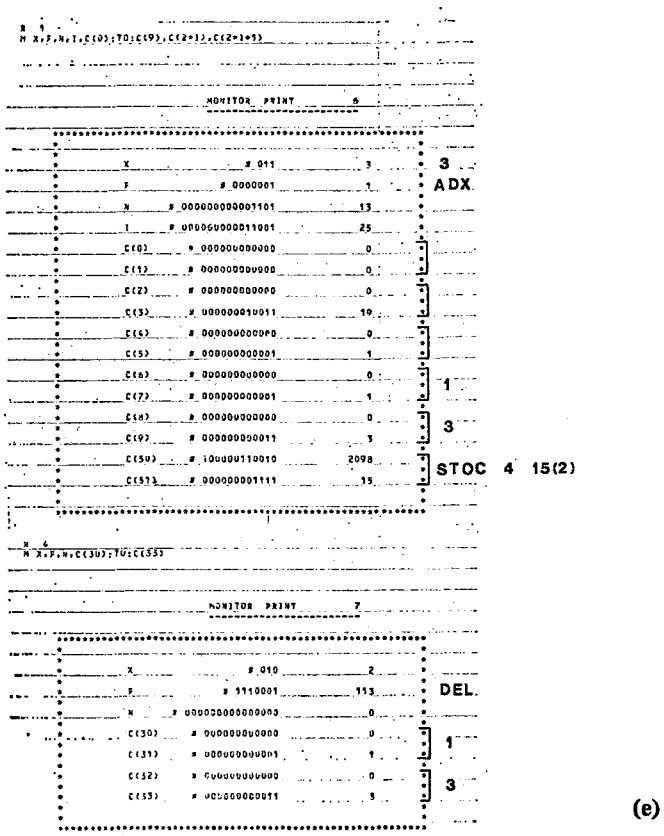
(b)



(c)



(d)



(e)

Fig. 6 A simulation run demonstrating certain aspects of the operation of the ICL 1901A. These include the half-word concept, various types of overflow, different instruction formats for different instructions and relative addressing. The contents of registers are printed in binary and decimal form.

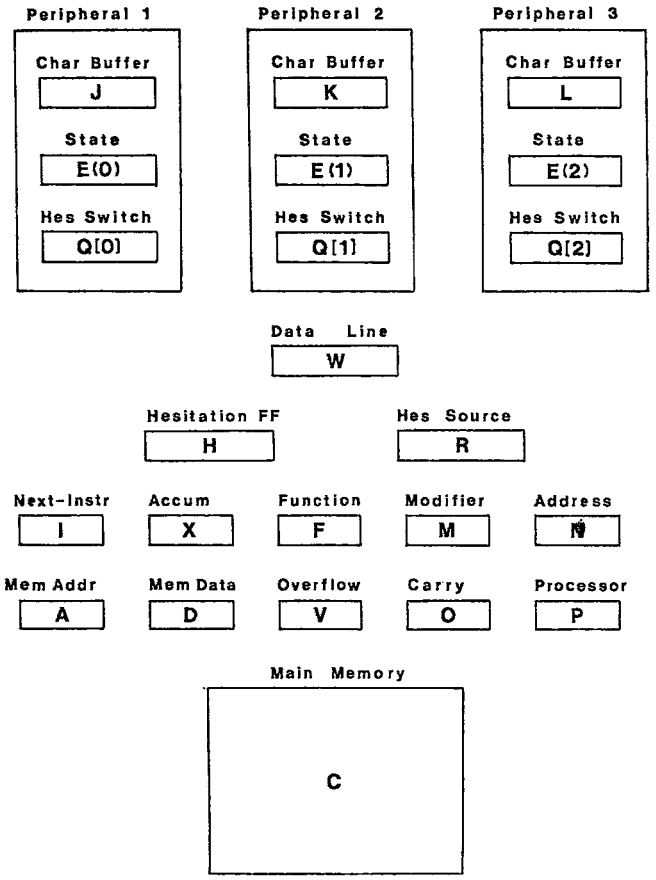


Fig. 7 A simple model of the registers of the ICL 1901A, including three peripherals.

as read from the input device by a READ statement,

and data cards read by an INITIATEREAD statement.

3.3. *Terminating a program*

A program is terminated when a new *B* parameter is read. This announces the start of a new computer definition. Alternatively, the parameter

T

may be used to terminate the simulation run completely.

A typical set of program description parameters is shown in **Fig. 3.**

4. Other facilities available on the simulator

Comments may be included at any point on a card. They are simply preceded by the symbol '↑'. Upon reaching such a symbol, the simulator will ignore the rest of the card.

There are various other facilities which are available on the simulator but which cannot be described here for lack of space. These include tracing facilities controlled by parameters

$TRACE

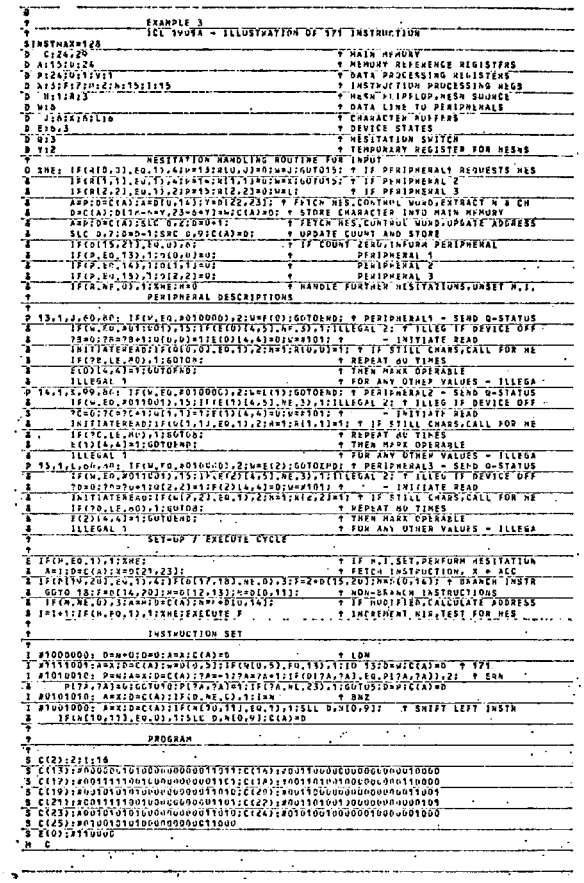
and $NOTRACE

and a set of auxiliary variables which are similar in format to dummy variables.

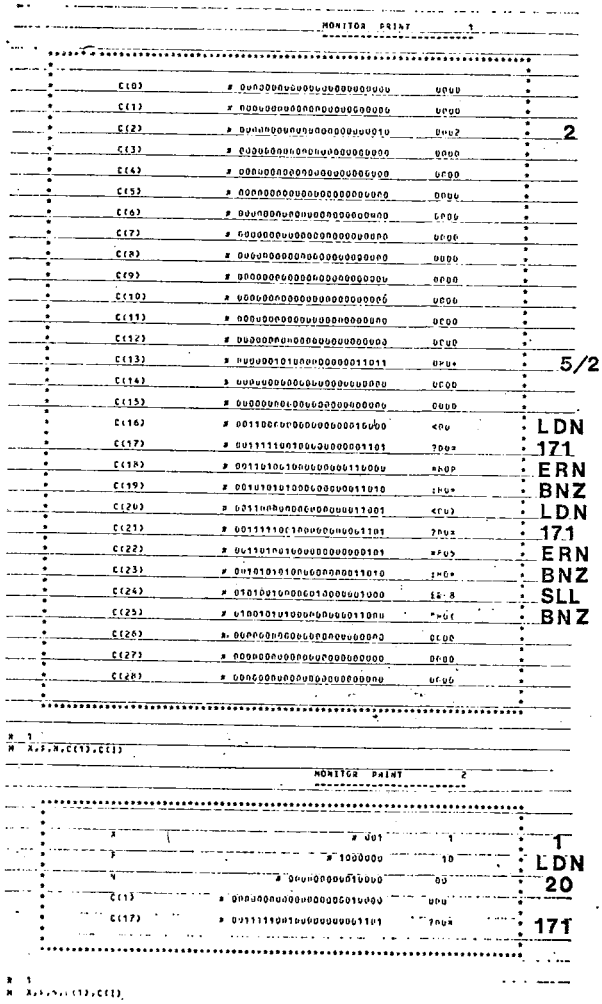
5. Examples

The simulation run for the computer program defined in Figs. 1-3 and Table 1 is shown in **Fig. 4.** This example illustrates the concept of a simple bootstrap loader program loading a more complex loader program which in turn loads the program of interest. This is done within the framework of a very simple machine.

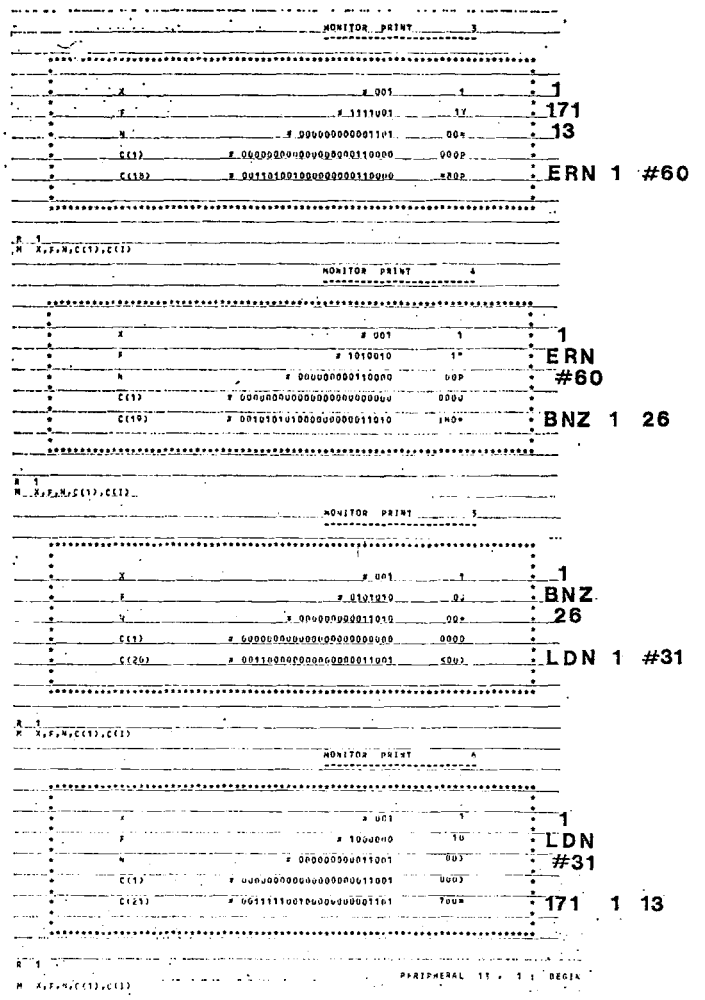
The next example demonstrates some simple aspects of the



(a)



(b)



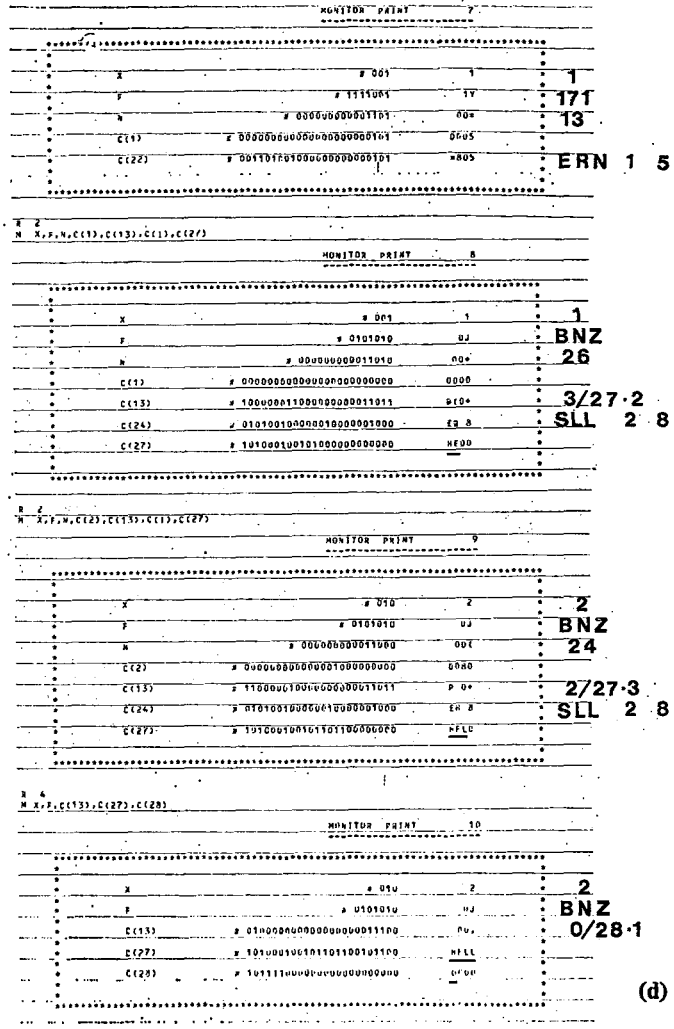


Fig. 8 A simulation run demonstrating how the 171 instruction (an Executive mode instruction) is used first to test the state of a peripheral and then to initiate a peripheral transfer on an ICL 1901A. The handling of hesitations is also shown. The contents of registers are printed in binary and character form.

ICL 1901A. This is a 24-bit word machine, which actually works on a half-word principle. Furthermore there are two different add instructions for adding from store into an accumulator:

ADXC (used for adding together the least significant halves of two double-length numbers) always leaves the most significant bit zeroised;

ADX (used for adding together the most significant halves of two double-length numbers, or for adding together two single-length numbers) retains the most significant bit, whether 0 or 1.

This results in three different types of overflow condition:

(*a*) on adding the least significant 12 bits of any two 24-bit words (type 1)

(*b*) on adding the most significant 12 bits of the least significant halves of two double-length numbers (type 4), and

(*c*) on adding the most significant 12 bits of the most significant halves of two double-length numbers (type 3).

Besides the two add instructions, the 1901A also has two load instructions (LDX and LDXC), two store instructions (STO and STOC), etc. A simple model of this aspect of the 1901A and a program to illustrate it, are shown in **Figs. 5** and **6**. This model also demonstrates the idea of relative addressing, the implementation of branch type instructions (CALL and EXIT) and of different instruction formats for different types of instruction.

The final example illustrates how the simulator can be used to demonstrate certain hardware features of a machine as might be useful in a course on operating systems. The example (**Figs. 7** and **8**) demonstrates the operation of the 171 instruction within Executive for an ICL 1901A computer. This instruction can be used either to test the status of a peripheral or to initiate a transfer on the peripheral. In this example, the program first calls for the status of the peripheral and checks that it is equal to 60_8, and then initiates a transfer. The reply is checked to see whether the command has been accepted by the peripheral, and the program (Executive) then continues, with hesitations (cycle-stealing) being handled at fixed points in the set-up/execute cycle. The incoming characters are stored in the correct locations and the count is updated on each hesitation (as seen in Monitor Print 8 to 10).

The example can be modified quite simply to illustrate the effect of two or more peripherals operating simultaneously. The streams of characters to and from different peripherals are handled by the simulator while the effects of crisis times can clearly be seen if the hesitation rates (h) in the P parameters are too small.

Other examples have been run on the simulator illustrating a host of other machine concepts; however, most of these examples are too large to print in a paper such as this.

After submitting this paper our attention was drawn to the similarity of our notation to that of ISP (Bell and Newell, 1971).

6. Conclusion

This simulation program, written in FORTRAN and PLAN, allows one to simulate the functioning of a variety of actual or theoretical computers. It is useful in teaching basic concepts of machine architecture and in illustrating hardware design con-

cepts necessary to understand operating systems—for example, hesitations, interrupts, various addressing techniques (such as relative, indirect and two-component addressing—with page and word registers), storage protection techniques such as base-limit or protect key systems, privileged and unprivileged modes, program status word, interrupt priorities, handling of multiple interrupts, stack machines, etc. In general, we have found it an extremely useful aid to teaching these aspects of computer science.

References

BELL, C. G., and NEWELL, A. (1971). *Computer Structures Readings and Examples*, McGraw-Hill.

LEE, J. A. N. (1972). *Computer Semantics*, New York: Van Nostrand Reinhold Co.

Book reviews

Computer Aided Control System Design, 1973; 244 pages. (IEE Conference Publication No. 96, £8·30)

This publication consists of a set of papers presented at an IEE Conference on Computer Control System Design, 2-4 April 1973, and represents the state of work in this field in the UK at that time. The papers can be split into roughly three sections: (i) identification and modelling, (ii) design, and (iii) simulation, which correspond to the three main activities of the design engineer. The number of papers which fall into each of these categories are, respectively, eight, nineteen and three.

The papers on identification and modelling include descriptions of three comprehensive packages of interactive programs for identification (Clarke, Shellswell and Young, Goodwin *et al*), the latter incorporating an optimal test signal design method. The remaining papers in this area are concerned with various techniques for the reduction of high order system models, a survey of the field being given by Towill. An interesting discussion of some practical modelling problems in relation to steel rolling mills is given by McClure.

Several papers describe comprehensive interactive design program packages and CAD techniques for: (i) linear single input, single output (siso) systems (Allen and Atkinson, Shearer *et al*, Webb, Woodward and Daly), (ii) nonlinear siso systems (Gray and Savvides), (iii) linear multivariable systems (Belletrutti, Fallside *et al*, MacFarlane, Mayne and Chuang, Munro and Ibrahim, Seraji, Young *et al*), and (iv) optimal control (Brown, Burt, Elkin and Daly, Healey and Jones, Mayne, Mobley and Paddison, Weislander). Clearly these areas could be further subdivided to display their specialisations. Two noticeable unifying features are the extensive use of interactive programming techniques, and the importance of graphical presentation of results, features which will undoubtedly form the basis of all control systems CAD programs in the future.

Simulation methods in design are described by Harris and Miles, and Revett. A survey of continuous system simulation languages, with particular emphasis on their industrial usage, is given by Gulland.

As a whole this set of papers describes, or makes reference to, most of the work done in CAD in this field in the 1970-73 period. There is a nice balance between papers which describe successfully implemented packages, and those concerned with future projects. Implicit in their descriptions are the essential features of CAD programs in this area: interactive, command driven, graphical. The only noticeable shortcoming is the absence of any papers which describe real problems which have been approached and solved making use of CAD programs. This can be taken as an indication of the state of acceptance of these design tools by industry, a situation which one would hope will be improved in the future. Another important feature which is absent from many papers is a deep appreciation of the numerical problems implicit in many of the proposed algorithms, especially in relation to implementation on machines with short word length.

M. J. DENHAM (London)

Human Congenital Malformations, the Design of a Computer-aided Study by E. Gal and I. Gal. 1975; 194 pages. (*The Butterworth Group*, £7)

This is an excellent book in almost all respects except its unfortunate title, in which the emphasis is quite misleading. The object is to explain how to design and carry out computer-aided studies and surveys. Since the authors took part in one such study on human congenital malformations, it is used partly as an example to illustrate the points: but other examples are used too.

The central theme is how to make sure that data obtained for future analysis, is useful, relevant and above all completely reliable. Thus the forms or questionnaires on which the information is recorded should be both as acceptable as possible to the specialist in the field and easy for the card-punch operator to use. It is essential to have advice from experts to know what information is useful and relevant. It is equally important to have the questions vetted by laymen and others to make sure that they are easy to understand, unambiguous, and likely to result in truthful answers. Methods of checking the accuracy of the information in various stages (when it is obtained and recorded, when it is punched, and when it is processed) are outlined. The emphasis is on seemingly simple and commonsense precautions which can be easily and disastrously overlooked. For technical details, such as programming or statistical methods, the advice is to consult professional programmers and statisticians.

Anyone but the most seasoned expert considering a survey type of investigation in any field (medical, psychological, social) would be well-advised to read this book first.

A. B. SMITH (London)

66. M.H.WILLIAMS, G.R.POTE and J.P.BROOKS, A simulator for teaching the internal workings of an assembler, Computers in Education, vol. 7, 55-63 (1983).

Comput. Educ. Vol. 7, pp. 55 to 63, 1983
Printed in Great Britain. All rights reserved

0360-1315/83/010055-09$03.00/0
Copyright © 1983 Pergamon Press Ltd

A SIMULATOR FOR TEACHING THE INTERNAL WORKINGS OF AN ASSEMBLER

M. H. WILLIAMS*, G. R. POTE and J. P. BROOKS
Rhodes University, Grahamstown, South Africa

(*Received* 4 *November* 1981; *revision received* 15 *December* 1981)

Abstract—In many courses on systems programming students are taught about assemblers. However, due to constraints on the time available they seldom get the opportunity to construct one.

This paper describes an assembler simulator which can be used as a tool to assist in teaching about assemblers. Using it students can design and simulate the running of their own assemblers in a very short time. This approach not only stimulates the student's interest but also gives him a deeper insight into assemblers.

INTRODUCTION

Many courses on systems programming or compiler construction devote some attention to the subject of assemblers and assembly languages [1–3]. However, since assemblers are generally fairly large programs and since the amount of time available is usually limited, there is seldom sufficient time for the student to obtain any practical experience on the construction of an assembler. For this reason courses sometimes concentrate on what an assembler should do rather than providing an understanding of how it does it.

We have attempted to solve this problem by creating an assembler simulator and using it in such a course. An assembler simulator is a program which simulates the actions of an assembler. In order to use it, the user must decide on the details of the particular assembly language which he wants to simulate (the assembly language instruction and pseudo-op formats, the machine code produced, the actions to be performed, etc.). These details are then encoded in a suitable notation and are referred to as the *assembler specification.*

The notation used for the specification is a special-purpose high level language. Using this notation one can describe the operation of a wide range of different assemblers with a variety of different properties.

Once an assembler specification for a particular assembler has been drawn up, it is read in by the simulator, whereupon the latter will adopt the characteristics of this assembler and go through the motions of the assembler, accepting as input source lines of a program written in the particular assembly language, extracting the information from these, assembling machine code and printing the listings expected as assembler output. At any point in the assembly language source code one may include directives to print out the contents of the assembler's tables, variables or the machine code produced, or to switch the tracing mechanism on or off. The whole process is illustrated diagrammatically in Fig. 1.

Thus the simulator serves a dual purpose in teaching students. In the first place it provides a model which enables one to "see" the inner workings of an assembler; in the second place, in formulating the

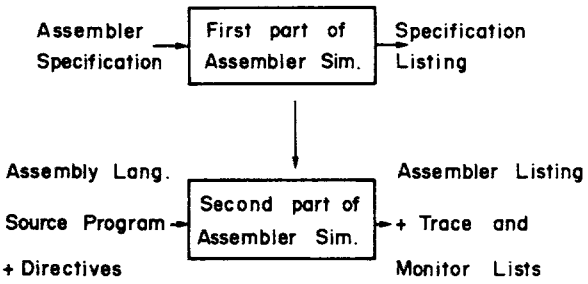


Fig. 1. Diagrammatic representation of assembler simulator.

* Present address: Department of Computer Science, Heriot-Watt University, Edinburgh, EH1 2HJ, Scotland.

assembler specification one obtains an insight into the way in which different facilities may be implemented.

Besides its use as an educational device the simulator can also be used as a design tool in designing assemblers, especially for microprocessors. The mechanism behind the desired assembler can be specified and its operation debugged and tested on the simulator in a relatively short space of time, before proceeding with an actual implementation.

In the following sections the notation used for the assembler specification is described briefly and then applied to an example.

THE ASSEMBLER SPECIFICATION

The assembler specification describes the operation of the assembler to be simulated. It is divided into two parts:

(a) The Declaration Section: This consists of a sequence of declarations describing the formats of tables used by the assembler, the variables used by the assembler, etc.
(b) The Assembler Description: This is an algorithm in a high level notation which specifies the operation of the assembler.

Data types and identifiers

There are two basic types in this system: numbers and strings. In the declaration section one may declare variables of either type—a numeric variable is denoted by a numeric identifier (which consists of a letter followed by zero or more letters and digits), a string variable is denoted by a string identifier (which is similar except that it is preceded by a "$" symbol).

In other declarations the format of the input record is described in terms of named fields, the format of a machine code instruction is described in terms of the fields which comprise the machine code word, the entries of a table are described in terms of the fields of which they are composed. These named fields may be either string or numeric and the same naming convention applies.

The Declaration Section

The declaration section is divided into eight subsections, each preceded by an asterisk followed by a two-letter qualifier code, viz. *CO, *IF, *OF, *MF, *TF, *EF, *EM, *VA.

Constant section

The constant section is used to set up certain constants for the simulator. The most important of these is the word length of the machine for which code is being generated. This is set up by means of a simple assignment statement. For example

```
*CO
WORDLENGTH = 16
```

will set up a word length of 16 for the machine code produced.

Input formats

The input format specifications describe the different forms which input records read by the assembler, can have. Each specification indicates how a line of source code may be broken into fields. There can be any number of different input format specifications, each consisting of a sequence of field specifications separated by commas and terminated by a semi-colon. Each field specification consists of one of the following:

A fixed length field. This is specified by writing the name of the field followed by its length in parentheses, e.g. $LABEL(9).

A variable length field. This is written as a string identifier followed by a delimiter character in quotation marks, e.g. $LAB",". Under certain circumstances the delimiter character may be omitted from the assembly language source which is to be assembled.

A constant field. This is indicated by a numeric identifier followed by a string constant, e.g. ASTERISK "*". If the string is recognised, the numeric variable is set to 1, otherwise it is set to zero.

A dummy field, written as (0). This indicates the presence of zero or more spaces. For example

```
*IF
$LABEL(9), $OP(6), $OPERAND(60);
$LAB" ", (0), $POP" ", (0), $ARGUMENT" ";
ASTERISK"*", $ADDRESS" ";
```

defines three input formats: the first consists of three fixed length fields $LABEL, $OP, and $OPERAND of length 9, 6 and 60 characters respectively; the second has three variable length fields $LAB, $POP and $ARGUMENT, each terminated by one or more spaces; the third indicates that if the line begins with an asterisk the remainder of the line is to be taken as a variable length field $ADDRESS, terminated by a space.

Output formats

The output format specifications indicate how a line of output is to be assembled. Again there can be any number of different output format specifications, each consisting of a sequence of field specifications separated by commas and terminated by a semicolon. Field specifications are the same as fixed length input field specifications. For example

```
*OF
$SPACE(8), $CARD(80);
```

Machine formats

The machine format specifications are similar to the output format specifications except that they are used to describe the formats of machine code instructions. There can be any number of machine format specifications, each consisting of a sequence of fixed length numeric fields. For example

```
*MF
F(4), N(12);
NUM(16);
```

Table formats

In this section the user defines the tables (Symbol Table, Machine Opcode Table, etc.) which his assembler will use. Each table format specification has the form

Table name (max entries) = $\text{field}_1, \text{field}_2, \ldots, \text{field}_n$;

followed by zero or more entries of form

@$\text{dat}_1, \text{dat}_2, \ldots, \text{dat}_n$;

where max entries is an integer specifying the maximum number of entries in the table for which space is to be reserved;

$\text{field}_1, \ldots, \text{field}_n$

are string or numeric field names referring to the different fields within each entry of the table; and

@$\text{dat}_1, \text{dat}_2, \ldots, \text{dat}_n$;

initializes an entry in the table.

In the case of a string field name, the name must be followed by an integer in square parentheses, representing the maximum length which a string in this field may have. This is necessary in order for the system to print the contents of tables neatly. Each dat_i is a string constant (enclosed in quotation marks), a binary integer (preceded by #) or a decimal integer. For example

```
*TF
ST(20) = $SYMBOL [8], $TYPE [1], ADDRESS, VAL;
MOT(3) = $OPCODE [6], MCOP, BRN;
@"LOAD", #0000, 0;
#"ADD", #0001, 0;
@"JMP", #0010, 1;
```

defines a table ST with a maximum of 20 entries, each of which consists of four fields: a string field $SYMBOL whose maximum size is 8 characters, a string field $TYPE whose maximum size is 1 character and two numeric fields ADDRESS and VAL; and a table MOT consisting of 3 entries (each with three fields—a string field $OPCODE and two numeric fields MCOP and BRN). The entries in MOT are initialized with the data given.

Expression formats

Since an operand field may contain an expression which may have one of a number of different possible forms, this set of declarations defines the set of possible expression forms which might occur. Each expression format is written in a form similar to a BNF production, with a string variable followed by an equal sign followed by one or more alternatives separated by exclamation marks and

terminated by a semi-colon. Each alternative consists of a sequence of string constants, string variables or pseudo-terminals (e.g. $LETTER, $DIGIT, $LAMBDA) separated by commas. If the left hand variable is separated from the equal sign by a slash symbol and an integer *i*, this means that the token *i* will be returned whenever this symbol is recognized. For example

```
*EF
$ADDR    = $EXP;
$EXP     = $TERM! $TERM, $PLUS, $EXP;
$TERM    = $ID! $INT;
$ID/1    = $LETTER, $RESTID;
$RESTID  = $LETTER, $RESTID! $LAMBDA;
$INT/2   = $DIGIT, $RESTINT;
$RESTINT = $DIGIT, $RESTINT! $LAMBDA;
$PLUS/3  = "+";
```

Thus an expression of form $ADDR consists of one or more terms separated by the symbol "+", where each term is either an identifier or an integer. If an expression is recognized, the tokens associated with the nonterminals will be used by the assembler in analyzing the expression and generating code.

Error messages

If the user wants his assembler to print error messages at certain points, he specifies these by number in the Assembler Description. This section relates the numbers to actual error messages. Each entry has the form

Error number, error message text;

For example,

```
*EM
1, "INVALID OPCODE";
2, "INVALID OPERAND";
```

Variables

The variables required in the Assembler Description are declared by writing them out as a list separated by semicolons. For example

```
*VA
LC; FLAG; ENDFLAG;
```

Comments

Any line which has asterisks in the first two character positions is treated as a comment.

The Assembler Description

This section (preceded by the characters *AD) contains a description of the action of the assembler written as a program in a high level notation. It consists of a single compound statement (i.e. a sequence of labelled or unlabelled statements enclosed between BEGIN and END) followed by a full stop. A label has the form

#⟨label identifier⟩:

while a statement may be any of the following:

(a) An assignment statement, assigning the value of a numeric or string expression to an output field, a machine code field, a table field or a simple variable.
(b) An unconditional transfer:

GOTO⟨label identifier⟩

(c) A simple conditional statement

IF⟨condition⟩THEN⟨statement sequence⟩ELSE
⟨statement sequence⟩FI

or IF⟨condition⟩THEN⟨statement sequence⟩FI

(d) A multi-way branch:

SWITCHON⟨var⟩INTO⟨case sequence⟩ENDCASE

where ⟨case sequence⟩ is two or more cases having the format

CASE⟨const⟩:⟨statement sequence⟩ESAC

(e) A repeat loop:

REPEAT⟨statement sequence⟩UNTIL⟨condition⟩

(f) A while loop:

WHILE⟨condition⟩DO⟨statement⟩

(g) A foreach loop:

FOREACH⟨table name⟩DO⟨statement⟩

This loop steps through each entry of the given table and performs the statement specified.
(h) A compound statement:

BEGIN⟨statement sequence⟩END

(i) A call statement:

CALL⟨label identifier⟩

(j) A return statement:

RETURN

(k) A stop statement:

STOP

(l) An error statement:

ERROR(n)

causes the error message text corresponding to the error number n under *EM to be printed.
(m)A print statement:

PRINT

causes whatever is in the output buffer of the simulated assembler to be printed.
(n) A newcard statement:

NEWCARD

causes the assembler to read in a new line of input and provided it is not a simulator directive, it scans each input format specification in turn and matches it against the input line. As it matches each field, the contents of the input line are stored in the field. If it encounters a mismatch, all remaining fields of that particular input format specification are set to null.
(o) A table insert statement:

INSERTIN⟨table name⟩ENTRY(⟨list⟩)

This creates a new entry in the table specified and assigns the data items in the list to the fields of the table entry. Each item in the list may be a string or numeric constant, a variable or an input field. The Current Entry Pointer for the table is set to point to this new entry.
(p) A table unstack statement:

UNSTACK⟨table name⟩

removes the last entry of the table specified and sets the Current Entry Pointer to the entry preceding the last one.
(q) A table position statement:

POSITION⟨table name⟩TO n

where n is a numeric variable or constant, moves the Current Entry Pointer to the nth entry in the table.

(r) A next entry statement:

NEXTENTRY⟨table name⟩

moves the Current Entry Pointer to the next entry in the table.
(s) A monitor statement:

MONITOR(⟨list⟩)

where list is a list of names of tables, fields or variables, causes the contents of the specified tables, fields and variables to be printed.

In addition to these statements there are a number of special functions available for use in expressions. These include:

(1) FIND (⟨sv or if⟩, ⟨table name⟩, n)—this function searches the table concerned for an entry whose nth component matches the string in the string variable or input field (⟨sv or if⟩). If such an entry is found, the Current Entry Pointer is set to point to it and the function returns the value **true**; if the search fails, the Current Entry Pointer is left unchanged and the function returns the value **false**.
(2) FORMAT(⟨sv or if⟩, ⟨expression var⟩)—this function checks the string contained in the string variable or input field to see whether it matches the format defined by the expression variable (one of the variables defined under *EF). If the match succeeds, the function returns the value **true** otherwise the function returns **false**.

On a successful match two lists are set up: a token list (which contains a sequence of tokens corresponding to the nonterminals recognized) and a pointer list (which contains a sequence of pointers pointing to the places in the string where the nonterminals begin). Once a successful match has been obtained, the following three functions may be used:

(a) NEXTT —fetches the next token from the token list and returns the value of this token;
(b) $NEXTS —yields the string corresponding to the current token;
(c) NEXTV(b)—treats the current token as an integer with base b, converts it to a binary number and returns the value of this number.

For example, given the expression formats

```
*EF
$ADDR       = $SYMBOL!$AST, $REL;
$SYMBOL/1   = $LETTER, $RESTSYM;
$RESTSYM    = $LETTER, $RESTSYM! $DIGIT, $RESTSYM! $LAMBDA;
$AST/5      = "*";
$REL        = $PLUS, $VAL! $MINUS, $VAL;
$PLUS/2     = "+";
$MINUS/3    = "−";
$VAL        = $INTVAL! $SYMBOL;
$INTVAL/4   = $DIGIT, $RESTINT;
$RESTINT    = $DIGIT, $RESTINT! $LAMBDA;
```

if one had the string "*+10" in the field $OPERAND and one encounters the function call

FORMAT($OPERAND, $ADDR)

the function will return the value **true** and will set up the token list as shown in Fig. 2. Thus the first call to NEXTT will yield the value 5; a reference to $NEXTS at this stage will yield the symbol "*" while a reference to NEXTV is meaningless. The next call to NEXTT will yield 2 ($NEXTS will yield "+", a call to NEXTV is meaningless). The next call to NEXTT will yield the value 4 ($NEXTS will yield the string "10", NEXTV(10) will yield the value 10).

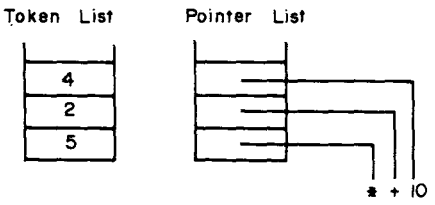


Fig. 2. The token list and pointer list set up for "*+10".

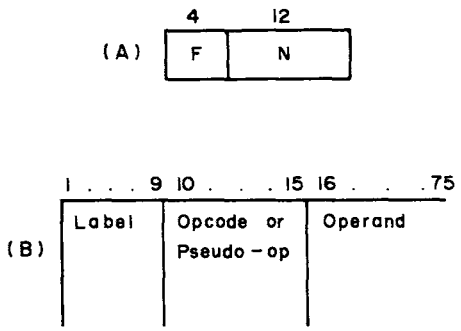


Fig. 3. (a) Machine Code format; (b) Assembly Language format.

Other numeric functions include:

VALUE(⟨string exp⟩)—treats a string as an ASCII representation of a decimal number and converts it to the equivalent binary representation of that number;
INDEX(⟨table name⟩)—yields the index of the current entry in the table.

EXECUTION OF THE SIMULATED ASSEMBLER

As the assembler specification is read in and verified, tables are set up for the second part of the simulator. The second part acts as interpreter, using the tables to simulate the effect of the assembler.

The interpreter steps through the tables and, whenever it encounters the statement NEWCARD, a new line of source is read. However, if this line contains a control character in the first column, the line is taken to be a directive to the interpreter and is read and acted upon. The possible directives are:

#MONITOR(⟨list⟩)—where list is a list of tables, variables, and/or "MC"; this prints the contents of the specified tables/variables/machine code area
#TRACE —causes tracing to be switched on
#UNTRACE —causes tracing to be switched off
#DUMP —gives a trace of the last 100 steps executed.
#FINISH —terminates the simulator run.

Thus the user is not only able to simulate the given assembler but is also able to obtain a picture of what is happening inside the assembler at any particular instant.

A SIMPLE EXAMPLE

Consider a simple machine with a word length of 16 bits and a single accumulator which has a machine code format as shown in Fig. 3(a). The assembly language for this machine consists of instructions and pseudo-ops each of which are written according to the format given in Fig. 3(b).

The OP/POP field contains a machine op mnemonic or a pseudo-op. The OPERAND/ARGUMENT field contains a symbol, a decimal integer, an octal integer or nothing. The machine opcodes included in the specification are: LOAD, ADD, STORE, JUMP, HALT and LDI. The pseudo-ops include:

START —announces the start of the program
END —marks the end of the program
DC —define constant (OPERAND = value of constant)
DS —define storage (OPERAND = length of storage block)
COM —comment.

An assembler specification for a one-pass assembler for this language is given in Fig. 4, together with a typical program in the assembly language to be used as data for the simulated assembler.

CONCLUSIONS

The notation described here enables one to specify the operation of a simple assembler in a reasonably concise manner. The notation is flexible and enables one to illustrate the important

```
**
** ONE-PASS ASSEMBLER
**
*CO
WORDLENGTH = 16;
*IF
$LABEL(9),$OP(6),$OPERAND(60);
$LAB(9),$POP(6),$ARGUMENT(60);
$CARD(75);
*OF
$SPACE(10);$OCARD(75);
*MF
F(4),N(12);
NUM(16);
*TF
ST(10)=$SYMBOL[9],$STYPE[2],ADDR;
BA(10) = P1,LINK;
OT(11)=$MNEM[5],OPTYPE,BIN;
@"START",1,0;
@"END",1,0;
@"DC",1,0;
@"DS",1,0;
@"COM",1,0;
@"LOAD",2,#0001;
@"ADD",2,#0010;
@"STORE",2,#0011;
@"JUMP",2,#0100;
@"HALT",2,#0101;
@"LDI",2,#0110;
*EF
**
**ADDRESS FIELD CONSISTS OF ONE OF:
**SINGLE SYMBOL-CONSISTING ONLY OF LETTERS,
**DECIMAL INTEGER, OR
**OCTAL INTEGER-PRECEDED BY &
**
$ADDNT = $SYMNT!$DECNT!$OCTNT;
$SYMNT/1 = $LETTER,$RESTSYM;
$RESTSYM = $LETTER,$RESTSYM!$LAMBDA;
$DECNT/2 = $DEC;
$OCTNT/3 = "&",$ODIG,$RESTOCT;
$RESTOCT = $ODIG,$RESTOCT!$LAMBDA;
$ODIG = "0"!"1"!"2"!"3"!"4"!"5"!"6"!"7";
*EM
1,"OPCODE NOT RECOGNISED";
2,"LABEL DEFINED MORE THAN ONCE";
3,"VARIABLE DEFINED MORE THAN ONCE";
4,"UNDEFINED SYMBOL";
5,"INVALID OPERAND";
*VA
LC;FLAG;ENDFLAG;P;Q;$I;$TEMPSYM;TEMPTOKN;
*AD
BEGIN
LC := 0;  ENDFLAG := 0;
REPEAT FLAG := 0;
  REPEAT NEWCARD; $OCARD:=$CARD; PRINT;
    IF FIND($OP,OT,1) THEN FLAG := 1 ELSE ERROR(1) FI
  UNTIL FLAG = 1;
  IF OPTYPE = 2 THEN

**
**MACHINE OP
**
    F(LC) := BIN;
    IF $LABEL <> "" THEN
      IF FIND($LABEL,ST,1) THEN
        IF $STYPE = "U" THEN $STYPE := "L";CALL PROCESSBACHAIN ELSE ERROR(2) FI
      ELSE INSERTIN ST ENTRY ($LABEL,"L",LC) FI
    FI;
  IF $OPERAND <> "" THEN
    IF FORMAT($OPERAND,$ADDNT) THEN
       TEMPTOKN:=NEXTT;
      IF TEMPTOKN=1 THEN
         $TEMPSYM:=$NEXTS;
      IF NOT FIND($TEMPSYM,ST,1) THEN INSERTIN BA ENTRY (LC,0);
        INSERTIN ST ENTRY ($TEMPSYM,"U",INDEX(BA));N(LC) := 0
      ELSE IF $STYPE = "U" THEN INSERTIN BA ENTRY (LC,ADDR);
        ADDR := INDEX(BA);N(LC) := 0
      ELSE N(LC) := ADDR FI FI
      ELSE IF TEMPTOKN=2 THEN N(LC):=NEXTV(10) ELSE
        N(LC):=NEXTV(8) FI FI
  ELSE ERROR(5)
    FI ELSE N(LC):=0 FI; LC := LC + 1;MONITOR("MC")
```

```
  ELSE
**
** PSUEDO-OP
**
    SWITCHON $POP INTO
        CASE"DC":CALL PROCESSVAR; NUM(LC) := VALUE($ARGUMENT);
                                  LC := LC + 1 ESAC
        CASE"DS":CALL PROCESSVAR; LC := LC + VALUE($ARGUMENT) ESAC
        CASE"END":ENDFLAG := 1 ESAC
    ENDCASE
  FI;MONITOR("MC")
UNTIL ENDFLAG = 1; $I := "U"; IF FIND($I,ST,2) THEN ERROR(4) FI;STOP;
**
** SUBROUTINES
**
#PROCESSVAR: IF $LAB = "" THEN RETURN FI;
IF FIND ($LAB,ST,1) THEN
  IF $STYPE = "U" THEN $STYPE := "V";CALL PROCESSBACHAIN ELSE ERROR(3) FI
ELSE INSERTIN ST ENTRY ($LAB,"V",LC) FI;
RETURN;
#PROCESSBACHAIN: Q:= ADDR; ADDR := LC;
REPEAT P := Q;POSITION BA TO P;Q := LINK;N(P1) := LC UNTIL Q = 0;
RETURN
END.
BEGIN    START A
A        DS    1
B        DS    1
D        LOAD  C
         ADD   B
         STORE C
# MONITOR(ST,BA,LC)
# TRACE
         JUMP  A
# UNTRACE
         LDI   101
         LDI   &52
         LDI   B
         HALT
C        DC    20
         END
#MONITOR(ST,BA,LC)
#DUMP
#FINISH
```

Fig. 4. An assembler specification for a one-pass assembler.

features of a simple assembler (e.g. one-pass or two-pass assembly, simple macro facilities, a macro assembler with multilevel definition and expansion, assemblers producing binary object code or relocatable binary, assemblers producing a single block of code and data or producing separate blocks of code and data, free format or fixed format input, complex operand fields, assembly time constants, etc.).

A simulator which accepts a specification of an assembler in this notation and simulates its behaviour, serves a dual purpose in teaching about assemblers. In the first place the design of the specification gives the student an insight into how the assembler operates, in the second place the monitor and trace facilities enable him to form a picture of the data structures within the assembler and the actions performed. In a relatively short period of time a student can obtain a reasonable depth of understanding of the properties of an assembler and how these might be implemented.

Such a simulator has been implemented by us on an ICL 1903T and has been used as a teaching aid to supplement part of the course on systems programming in the Computer Science undergraduate programme at Rhodes University. The most important result from our point of view was the extent to which it stimulated students to think about the implementation of different facets of assembly languages and the novel ideas with which students came forward.

REFERENCES

1. Seegmüller G., Systems programming as an emerging discipline. In *Information Processing 74* (Edited by Rosenfeld J. L.). North Holland, Amsterdam (1974).
2. *Curriculum 68, Commun. Ass. Comput. Mach.* **11**, 151–197 (1968).
3. Austing R. H., Barnes B. H., Bonette D. T., Engel G. L. and Stokes G., *Commun Ass. Comput. Mach.* **22**, 147–166 (1979).

67. M.H.WILLIAMS and A.J.STEWART, Machine cycle simulator for a microprocessor, Microprocessors and Microsystems, vol. 6, no. 3, 131-134 (1982).

Machine cycle simulator for a microprocessor

A simulator has been constructed for the MC68000 microprocessor and the technique used is described by **M H Williams and A J Stewart**

The technique used in the construction of a simulator for a processor affects the range of applications for which the simulator can be used. A machine cycle simulator has been constructed for the MC68000 microprocessor, and some of the advantages of this technique are discussed.

microprocessors simulators MC68000

The rapid growth in the microprocessor population and the increasing use of microprocessors in education, both at secondary and tertiary levels, has resulted in a proliferation of cross-assemblers and, to a lesser extent, simulators and emulators for these microprocessors on larger machines.

Machine emulation or simulation is the process whereby a program written in the machine code of one type of computer (say machine X) can be read into a machine of a different type (say machine Y) which will then produce the effect of executing that code on machine X. The distinction between emulation and simulation is in the level at which this process is implemented[1].

In the case of a simulator, implementation is usually in a high-level language. In addition it is customary to include a variety of facilities which would not be available as features of the hardware of the machine being simulated. These facilities include extended tracing capabilities, break options, more complete error reporting and so on.

The advantages of a simulator are:

- one is able to simulate the execution of programs on machine configurations which one does not possess. For instance, one may not possess a processor of this type at all, or one may possess a processor with insufficient main memory, etc
- one is able to execute a program in a controlled manner and monitor its execution in a way which would not normally be possible with the hardware alone

However, the usefulness of a simulator, measured in terms of the range of applications for which it can be used, is affected by the method of simulation used. This paper considers one approach and the advantages which follow from using it.

BASIC METHODS OF OPERATION

When a simulator is constructed, the simplest approach and that which is usually adopted, is to simulate the machine at the instruction level. The basic structure of such a simulator is:

```
while not end do
    begin
    trace;
    fetch-next-instruction;
    execute-instruction
    end
```

where execute-instruction is implemented as a large case statement with a case option for each opcode (or some suitable variation of this). This may be termed an instruction simulator.

An alternative approach is to follow the operation of the machine concerned cycle by cycle through the execution of a program. This may be termed a machine cycle simulator. In this case the basic structure of the simulator might be regarded as:

```
while not end do
    begin
    trace;
    execute-next-cycle
    end
```

where execute-next-cycle is again implemented as a large case statement but this time each case option corresponds to the action taken in a single machine cycle, and at the end of each case option, the next case option is determined.

The distinction between the fetch and execute phases is no longer necessary since in each machine cycle the type of the subsequent machine cycle is determined; indeed it is an advantage to ignore this devision since different instructions require a different number of machine cycles for instruction fetch as well as for instruction execute.

A SIMULATOR FOR THE MC68000 MICRO-PROCESSOR

The machine cycle approach has been used to construct a simulator for the MC68000 microprocessor on a PDP11/34. Some aspects of this simulator are described here briefly.

The first step in any simulation is to fetch an instruction. In the case of the 68000 each instruction may occupy between one and five words. The first word is called the operation word and this specifies the operation to be performed and (indirectly) the number of words which make up the instruction. The operation word may be followed by an immediate operand consisting of one or two words or an extension to the source effective address, also one or two words in length. After this there may be a one- or two-word extension to the destination effective address. The format is illustrated in Figure 1.

Computer Science Department, Heriot-Watt University, 79 Grassmarket, Edinburgh, EH1 2HJ, UK.

 0141–9331/82/030131–04 $03.00 © 1982 Butterworth & Co (Publishers) Ltd

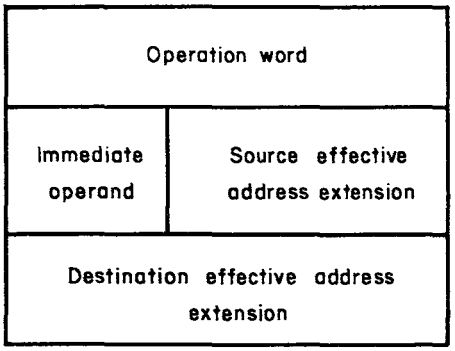


Figure 1 Format of an instruction for MC68000 microprocessor

The time taken for any operation on the 6800 is largely determined by the accesses it makes to main memory. Most other activities which take place, proceed in parallel with the memory accesses and are usually completed by the time that the memory accesses are complete.

The time taken for a memory access (data read or data write) is two internal cycles, or eight half-cycles of the processor clock. However, in the case of a data read, the data fetched is actually available for use before the protocol of the memory access is complete. Thus each memory access is divided into two phases. In the case of the data write this makes little difference, but in the case of the data read the data which has been fetched in the first phase can be used during the second phase while the memory access is being completed. Although these two phases are not of equal length, they will always occur in pairs and each such pair corresponds to two internal cycles. Thus for convenience we could take each phase as being one internal cycle in length.

Thus in the first cycle of a simulation the operation word is fetched. If there are any additional words to be fetched, the second cycle is spent waiting for the memory access to complete; thereafter the appropriate words will be fetched in the correct order and stored in internal registers. Once the whole instruction has been fetched, execution of the instruction can proceed. A rough outline of the first part of the simulator which handles this is given in Figure 2.

The majority of instructions on the 68000 involve fetching one or two operands, performing some operation and storing the result. However, one of the main features of the machine is its wide range of addressing modes and thus the process of fetching or storing an operand is quite complex. In particular one needs to define actions to fetch or store a one byte, one word or two word operand for all the various addressing modes. A typical set of actions for fetching a one word operand is given in Table 1. The actions performed in each cycle are enclosed in square brackets and the corresponding machine timings (excluding the time taken to fetch words which form part of the instruction) are given in the final column. Note that for simplicity the action

$E \leftarrow A(reg) + SA$

in mode 101 and the action

$E \leftarrow PC + SA$

in mode 111, register 010 have been included as part of these actions. In practice they would actually occur in the last phase of the operation to fetch the source effective address extension.

The operations themselves can be classified into various

```
case nextcycle of
fetopwd: begin op:= fetchwd(I); I:=I + 1;
      determine instruction type and store in instr;
      if immediate operand
        reqd then begin imop:=0; imopflag:= true;
                    place size in imopsize
                    end
             else imopflag:= false;
      if source effective address
          reqd then begin sa:= 0; saflag:= true;
                      place size in sasize
                      end
               else saflag:= false;
      if destination effective address
          reqd then begin da:= 0; daflag:= true;
                      place size in dasize
                      end
               else daflag:= false;
      if imopflag or saflag or deflag then nextcycle:= waitinst
                                      else nextcycle:= instr
      end;
waitinst: if imopflag then nextcycle:= fetimop
          else if saflag then nextcycle:= fetsa
          else if daflag then nextcycle:= fetda;
fetimop: begin x:= fetchwd(I); I:= I + 1;
          store x in appropriate position in imop;
          imopsize:= imopsize-1;
          if imopsize = 0 then imopflag:= false;
          if imopflag or saflag or daflag
            then nextcycle:= waitinst
              else nextcycle:= instr
         end;
fetsa:
```

Figure 2 Rough outline of first part of machine cycle simulator

categories, namely:

- arithmetic (ADD, SUB, MULS, MULU, DIVS, DIVU, CMP, TST, NEG, ABCD, SBCD and NBCD)
- logical (AND, EOR, OR and NOT)
- shift (ASL, ASR, LSL, LSR, ROL, ROR, ROXL and ROXR)
- bit test operations (BCHG, BCLR, BSET and BTST)
- stack operations (LINK, PEA and UNLK)
- move operations (MOVE, LEA, EXG, SWAP, CLR, MOVEM and MOVEP)
- jump operations (B_{CC}, BRA, BSR, DB_{CC}, JMP, RTE, RTR and RTS)
- trap operations (CHK, TRAP and TRAPV)
- miscellaneous (EXT, NOP, RESET, S_{CC}, STOP and TAS)

Almost all the operations are straightforward and similar to equivalent operations on other machines.

Finally, in addition to fetching and executing instruc-

Table 1. Actions for fetching a one word operand under the various addressing modes

Addressing mode	Mode	Reg.	Action for 1 word fetch	Number of internal cycles
Data register direct	000	reg	–	0
Address register direct	001	reg	–	0
Address register indirect	010	reg	[FETCHWORD (A(reg))], [ ]	2
Address register indirect with postincrement	011	reg	[FETCHWORD (A(reg))], [A(reg) ← A(reg) +2]	2
Address register indirect with predecrement	100	reg	[A(reg) ← A(reg) -2], [FETCHWORD (A(reg))], []	3
Address register indirect with displacement	101	reg	[E ← A(reg) + SA; FETCHWORD (E)], [ ]	2
Address register indirect with index	110	reg	[E ← A(reg) + R(index) + SA], [FETCHWORD (E)],[ ]	3
Absolute short	111	000	[FETCHWORD (SA)], [ ]	2
Absolute long	111	001	[FETCHWORD (SA)], [ ]	2
Program counter with displacement	111	010	[E ← PC + SA; FETCHWORD (E)], [ ]	2
Program counter with index	111	011	[E ← PC + R(index) + SA], [FETCHWORD (E)], [ ]	3
Immediate or status register	111	100	–	0

tions the simulator must also test for interrupt conditions (caused by trap operations within the program or by circumstances external to the program) and act upon them when they occur. Processor interrupts or 'exceptions' as they are referred to in the 68000 manual[2], are classified into three groups. These are handled as follows:

- the exception with the highest priority is the external RESET. When the RESET line is asserted, the processor stops whatever it is doing, the Interrupt mask is set to 7, trace mode is turned off, the mode is set to supervisor mode and the contents of the first two words of the supervisor's code space are placed into the Stack Pointer and the second two words into the PC. Execution resumes at the location pointed to by the contents of the PC
- other exceptions are handled in four steps:
  - copy the Status Register internally, set supervisor mode and clear trace mode. If the exception is an interrupt, the Interrupt Mask is set
  - determine the Vector Number of the exception (for interrupts this is obtained from the interrupting device – lines D0 to D7; for exceptions it is generated by logic within the MC68000)
  - save the current processor status, that is the current contents of PC and the contents of the Status Register
  - fetch the new value of the PC from the Supervisor's data space and resume execution at that address

Interrupts can be stacked, and interrupts of higher priority can preempt lower priority ones. RESET has the highest priority and will preempt any other type of interrupt.

ADVANTAGES OF THIS APPROACH

The machine cycle approach to simulation involves a more detailed analysis of the order in which operations are performed in the machine concerned, and probably results in a larger program than in the case of an instruction simulator. What, therefore, is to be gained from this approach?

The first benefit to be obtained is that of realistic timings for the processes involved. This is important in the study of the performance of an operating system since the timings enable one to make accurate assessments of the proportion of time spent in the operating system as opposed to that spent executing user programs, the amount of time taken to switch programs, the proportion of time spent doing nothing, and so on. This is of particular benefit as a tool for teaching computer science.

The second advantage of this approach is the ease with which it can be adapted to simulate a network of processors of the same type. In this case the basic structure is altered to:

```
while not end do
  for i: = 1 to noofprocessors do
    begin
    trace(i);
    execute-next-cycle(i)
    end
```

Since the timings of this simulated multiple processor system accurately reflect the operation of the physical system, the simulator can be used to study the interactions between processors in such a system. This facility is useful not only from the point of view of teaching but also for the purpose of debugging network programs. Such programs are considerably more difficult to debug than programs which involve only a single processor, and a simulator can be of great assistance in tracing what is happening. However, it is only a machine cycle simulator which can be

used in this way as an instruction simulator cannot keep track of the synchronization between processors.

Thus before embarking on the construction of a simulator for a processor, careful consideration should be given to the technique to be used since this will affect the range of applications for which the simulator can be used.

REFERENCES

1 **Cooper, R E M** 'Emulation on the DG Eclipse S/130' *Euromicro J.* Vol 6 No 3 (1980) pp 179–183

2 'MC68000 16-bit Microprocessor User's Manual' Motorola (1979)



Design notes

Engineers — have you a design hint or trick of the trade you'd like to share with your colleagues, but cannot spare the time to author a full-scale refereed paper? Or do you have experience with a particular part or piece of equipment you feel may be of interest to others? Why not write a Design Note?

It can be as short as you like — consistent with clarity — and we can reproduce your software listings straight from printer output. Send us your sketches and our studio will turn them into works of art (although we would prefer unlettered tracings).

At present we can offer rapid typeset publication — which can be turned quickly and simply into glossy reprints for use by your company.

Send your ideas (or ring for more advice) to Simon Middelboe **Microprocessors and Microsystems** Butterworth Scientific Limited — Journals Division PO Box 63 Westbury House Bury Street Guildford Surrey GU2 5BH UK
Telephone: 0483 31261 Telex: 859556 SCITEC G

68.M.H.WILLIAMS, A question answering system for automatic program synthesis, SIGPLAN Notices, vol. 11, 63-68 (1976).

A QUESTION-ANSWERING SYSTEM FOR AUTOMATIC PROGRAM SYNTHESIS

M.H. Williams
Department of Computer Science
Rhodes University, Grahamstown
South Africa

1. INTRODUCTION

With the increasing attention which is being focussed on the problems of writing correct programs and the difficulties involved in maintaining and adapting existing programs to the changing requirements of the user, the importance of subjects such as automatic program verification and automatic program synthesis is growing rapidly. The term "automatic programming", which was once used to refer to the process of writing a program in some high level language (1), is now generally taken to mean a "higher level" programming system which gives the programmer considerably more assistance in producing correct programs that do present compiling systems. However, there are different interpretations of this term and different automatic programming projects have widely differing aims. Balzer et al (2) have as their goal a generalized domain-independent system which transforms a specification of a problem together with an ill-defined algorithm for solving the problem into a correct program; whereas Cheatham and Wegbreit (3) aim at a system which will accept a non-trivial program and convert it into a significantly improved one.

Nevertheless the basic problem in automatic programming is that of synthesizing a program according to some specification (whether this be of the problem or of its solution). But consider what type of program automatic programming systems are aiming at producing. For this purpose one might group the majority of programs into three very broad categories:

(i) the small relatively straightforward program written by physicists, engineers, psychologists, etc.
(ii) the larger but relatively straightforward program (or set of programs) written by commercial programmers, and
(iii) the large but by no means straightforward program written by systems programmers (such as an operating system).

General automatic programming systems aim at solving problems in all three categories by providing the user with some suitable language for expressing either the problem itself or an algorithm for solving the problem. Furthermore the intention is that the solution obtained by the automatic programming system should be logically correct and free of programming errors.*

---

* Note that the term "programming error" is used to cover errors made by the programmer in converting an algorithm for solving a problem into a program in a high-level language for achieving the same effect, as opposed to "problem errors" which are errors in the programmer's basic understanding or formulation of the problem.

However, such ambitious aims are not likely to be realized for some time yet, and some people have predicted that any higher level languages developed in the near future would be very specialized and only of benefit in solving very limited classes of problems. Another fear is that such systems will not completely remove the problem of programming errors but simply transfer them to a different point in the process. At present the process of expressing a solution to a problem in a high-level language is very prone to errors, but it is possible that in automatic programming systems the process of expressing the problem itself in some problem domain language might be somewhat error-prone too.

As a result this paper examines a different type of automatic programming system in which the user simply answers questions put to him by the system and from the user's replies the system constructs a program to fit the user's problem. In practice such a system on its own may prove too inflexible and thus a combination of this question-answering system with a generalized problem domain language may be a better approach. However, the question-answering system does have the advantage that it removes the problem of the user having to be familiar with the peculiarities of a computer language in order to use it properly. In addition it is simpler to implement and can be extended without the major changes required in the case of a problem domain language.

2. STRUCTURE OF THE AUTOMATIC PROGRAMMING SYSTEM

In the project described here, the process of automatic program synthesis is broken down into two stages. The first stage interacts with the user to produce a flow diagram while the second stage automatically converts this flow diagram into a program in some suitable programming language. However, since certain programs (e.g. commercial applications) depend heavily on very rigorous input/output specifications, these are treated separately from the flow diagram and a separate module has been set aside to deal with them.

The system consists basically of four components: the Problem Definition Module, the Flowchart Synthesizer, the Format Editor and the Program Generator as illustrated in Fig. 1.

(1) THE PROBLEM DEFINITION MODULE

This module attempts to define the user's problem by asking the user a series of simple questions and noting his replies. It is implemented by setting up the questions themselves as a tree-like structure in a data file. The Problem Definition Module then follows a path through the tree which is determined by the user's replies. The information obtained from the user's replies is processed and handed over to the Flowchart Synthesizer.

(2) THE FLOWCHART SYNTHESIZER

This takes as input the specification of the problem as produced by the Problem Definition Module and produces as output a flow diagram for the program to solve the user's problem.

(3) THE FORMAT EDITOR

This module is used to set up information on input/output formats for the intended program on a disc file called the Format File.

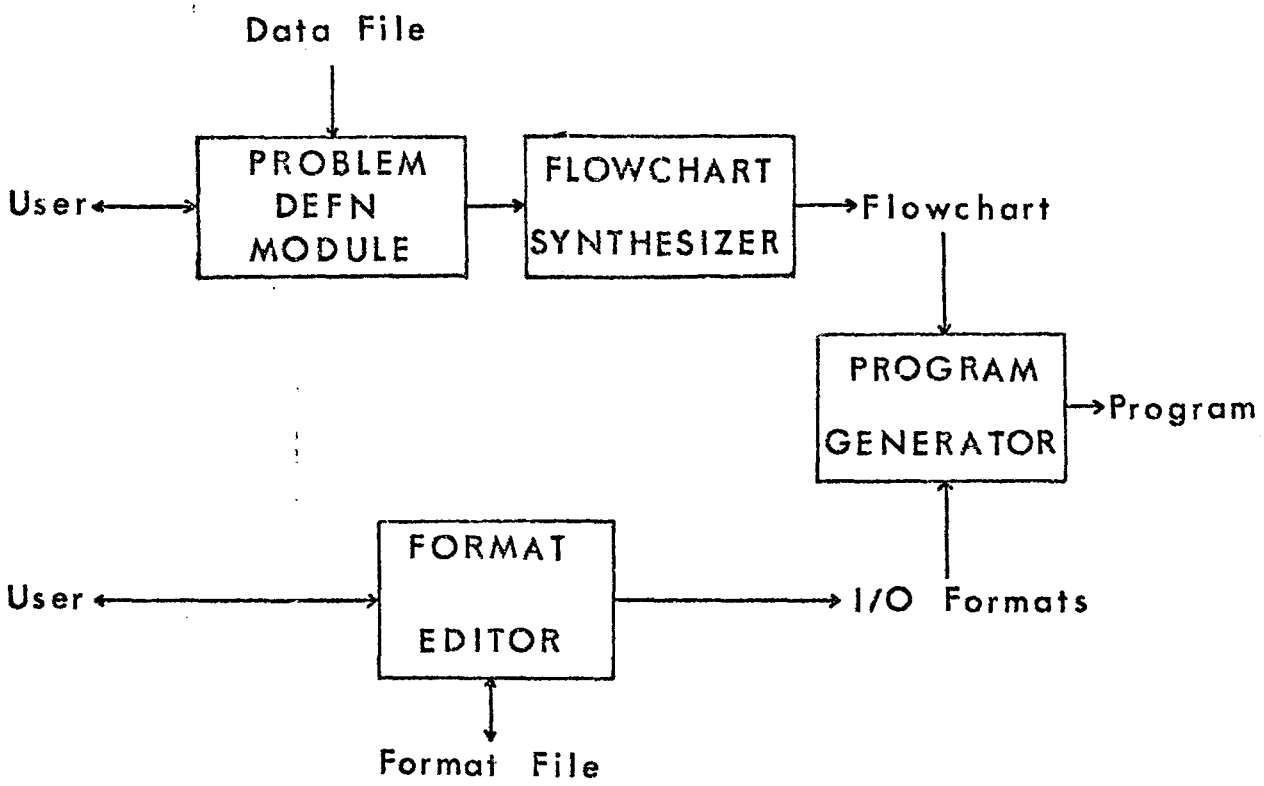


Figure 1. Structure of automatic programming system.

(4) THE PROGRAM GENERATOR

The flow diagram produced by the Flowchart Synthesizer and the formats stored in the Format File are combined by the Program Generator to produce a program in the desired programming language.

The problem-dependent units of this system are the Problem Definition Module and the Flowchart Synthesizer. However, the problem-dependent portion of the Problem Definition Module is largely contained in the tree structure in the data file. Thus it is possible to write both of these Modules so that the code is relatively independent of the problem. This means that if one designs two sets of data in separate data files for solving two different types of problem, these can be accessed separately, or they can be combined into a single system without much difficulty - something which is not possible in general language-based automatic programming systems.

There are no plans for including an automatic program verifier in the system at this stage, although this can certainly be interposed between the Flowchart Synthesizer and the Program Generator (see King (4)) at a later stage if necessary. If this is done, the information provided by the Problem Definition Module can be used to assist the program verifier in its task.

An example of the dialogue between the Problem Definition Module and the user is illustrated in Fig. 2. This is a simplified picture of the type of questions which might be put to a user wishing to write a simple program to solve a differential equation.

```
   1. WHAT SORT OF PROBLEM DO YOU WISH TO SOLVE:
      A. LARGE SET OF DATA TO BE ANALYZED?
      B. MODEL TO BE SOLVED?

B
   2. WHAT SORT OF MODEL DO YOU WISH TO SOLVE:
      A. DISCRETE SIMULATION?
      B. DIFFERENTIAL EQUATION(S)?
      C. LINEAR EQUATIONS?

B
   3. DO YOU HAVE:
      A. A SINGLE DIFFERENTIAL EQUATION?
      B. A SET OF DIFFERENTIAL EQUATIONS?

A
   4. IS THIS:
      A. AN ORDINARY DIFFERENTIAL EQUATION?
      B. A PARTIAL DIFFERENTIAL EQUATION?

A
   5. IS THIS:
      A. A FIRST ORDER DIFFERENTIAL EQUATION?
      B. A SECOND ORDER DIFFERENTIAL EQUATION?
      C. OF HIGHER ORDER THAN TWO?

A
   6. WHAT IS THE NAME OF THE DEPENDENT VARIABLE?

X
   7. WHAT IS THE NAME OF THE INDEPENDENT VARIABLE?

Y
   8. IS THE LOWER BOUND OF THE RANGE OF X OVER WHICH THIS EQN IS TO BE
      SOLVED:
      A. CONSTANT?
      B. TO BE EVALUATED FROM SOME FORMULA?
      C. TO BE READ IN?

A
   9. WHAT IS THE VALUE OF THE LOWER BOUND OF THE RANGE OF X?

0
   10. IS THE UPPER BOUND OF THE RANGE OF X OVER WHICH THIS EQN IS TO
       BE SOLVED:
       A. CONSTANT?
       B. TO BE EVALUATED FROM SOME FORMULA?
       C. TO BE READ IN?

C
   11. DO YOU WISH THE STEP-LENGTH TO BE:
       A. CONSTANT?
       B. TO BE EVALUATED FROM SOME FORMULA?
       C. TO BE READ IN?

B
   12. WHAT FORMULA IS TO BE USED FOR STEPLENGTH?

UPPERBOUND/20
```

Figure 2. An oversimplified picture showing how a user might generate a program to solve a simple differential equation.

3. CONCLUSION

The generally accepted approach to automatic programming is based on a general language for expressing problems, and aims at creating "machines that write programs better than we can, given only the problem specification" (5). The system outlined here (and which is presently under development) is based on questioning of the user by the system. The main aim of this system is to assemble a collection of well-understood techniques for solving straightforward standard problems, and combine these to produce simple programs. If this approach proves successful, it can be used to relieve the burden of designing straightforward programs for solving standard problems such as commercial applications, engineering applications, etc.

The difference between this system and the general language-based system is somewhat analogous to the difference between an applications package and a compiler. Although this system is not as powerful as the general system, it has advantages of simplicity of use, simpler implementation and ease of extension.

References

1. J.E. Sammet, Programming Languages: History and Fundamentals, Prentice-Hall, Englewood Cliffs, N.J., 1969.

2. R. Balzer, N. Greenfeld, M. Kay, W. Mann, W. Ryder, D. Wilczynski and A. Zobrist, "Domain - Independent Automatic Programming", in Proceedings of IFIP Congress 74, North-Holland, Amsterdam, 1974, pp326-330.

3. T.E. Cheatham and B. Wegbreit, "A Laboratory for the Study of Automatic Programming", AFIPS Conf. Proc., 40, 11-21 (1972).

4. J.C. King, "A Program Verifier", in Proceedings of IFIP Congress 71, North-Holland, Amsterdam, 1972, pp.234-239.

5. D.E. Knuth, "Computer Programming as an Art", Comm ACM, 17, No. 12, pp.667-673 (1974).

69. M.H.WILLIAMS and E.V.C.FIELDING, Automatic generation of payroll programs, The Computer Journal, vol. 21, 378-380 (1978).

Discussion and correspondence
Automatic generation of payroll programs

M. H. Williams and E. V. C. Fielding

Department of Computer Science, Rhodes University, Grahamstown, South Africa

A system for automatically generating programs for a particular class of problem—in this case the class of payroll programs—is discussed. The technique used in the generation process is based on the stepwise refinement approach to problem solving. The method described is applicable to other similar problems and ultimately could be used in a general automatic programming system.
(Received February 1978)

1. Introduction

The term 'automatic programming' refers to a system which will automatically create a program from some specification. Initially (Sammet, 1969) this term was used in the narrow sense to refer to compilers (where the specification is simply a program written in a high level language). More recently (Knuth, 1974) the term has been used to refer to much more ambitious systems which in some way accept the definition of a problem from a user and from this construct a program to solve the problem.

Various workers are already engaged in the task of creating systems to solve general forms of problems (eg. Balzer *et al.*, 1974) while others are approaching the problem from the point of view of accepting an existing program and converting it to a significantly improved form (Cheatham and Wegbreit, 1972).

An alternative approach to automatic programming is to consider specific applications for which the techniques of solution are known but for which many variations in user requirements exist. If one can create a system which will adequately determine the user's needs and generate a unique program to solve each individual user's particular problem, this will provide a small step in the direction of the ultimate goal of automatic programming. Furthermore if the same technique can be applied to other types of problems and eventually the different systems can be merged into a single system, one may indeed obtain a limited form of the all purpose automatic programming system—one which may not be capable of handling 'any' problem but which will certainly be capable of handling the more common problems from a particular area.

One of the most common programs in existence today and certainly one of the most important in most institutions, is the payroll program. The payroll program is also a program for which user requirements differ widely.

The present paper describes a system, the aim of which is the automatic generation of payroll programs. This involves both the determination of user requirements and the generation of a unique program to meet each user's particular needs.

2. Structure of the system

The current approach to designing a system, which will provide for a variety of different user requirements, is to construct a package. In order to use a package, the user must decide which facilities he requires, set up the appropriate parameters and feed these into the package.

From experience the task of deciding on and setting up the parameters can be a lengthy one and one which requires someone with considerable computer expertise to perform. In one instance the setting up of a set of parameters for one computer payroll package which is currently being marketed, took one institution (with under a thousand employees) many man-months to complete. In general parameters may lead to problems unless they are well designed and documented.

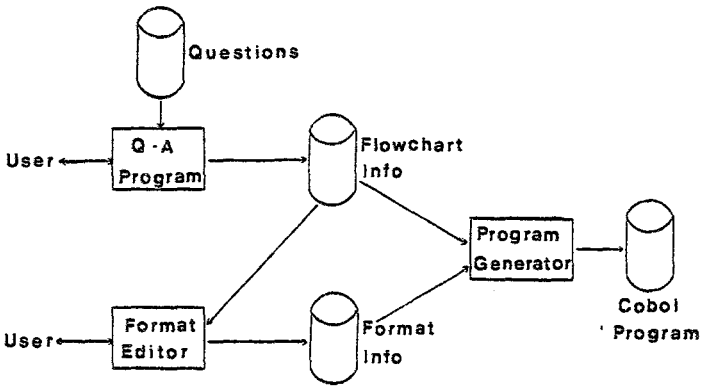


Fig. 1 An overall view of the system

On the other hand a question answering approach is a much more efficient and effective means of determining the user's needs. The question answering system can be designed to ask the user only those questions which are relevant, omitting those which from the user's previous replies are obviously irrelevant. Thus this approach was adopted here.

The question answering system used here is a table driven program for which the data is set up initially on a disc file. The data for each question consists of three components:

(*a*) the text of the question

(*b*) the expected answer

(*c*) the action.

The expected answer field contains a little procedure which must be executed once the text of the question has been displayed. It is responsible for displaying any additional information and reading in and checking the user's reply. It also performs certain actions on the basis of the user's reply.

The action field also contains a small procedure which is executed after the user's reply has been read and the expected answer procedure is completed. The action procedure is concerned essentially with the generation of 'flowchart boxes' on the basis of the user's reply and with deciding which question should be asked next.

The output from the question answering system comprises a set of flowchart boxes which contain most of the information necessary to construct the program. The only details which are missing at this stage are details about input/output formats. A separate program, the Format Editor, is responsible for setting these up. Thus when the user finishes with the question answering system, he is directed to the Format Editor which obtains the necessary information and stores this away on a disc file.

The final step in the process is the Program Generator. This

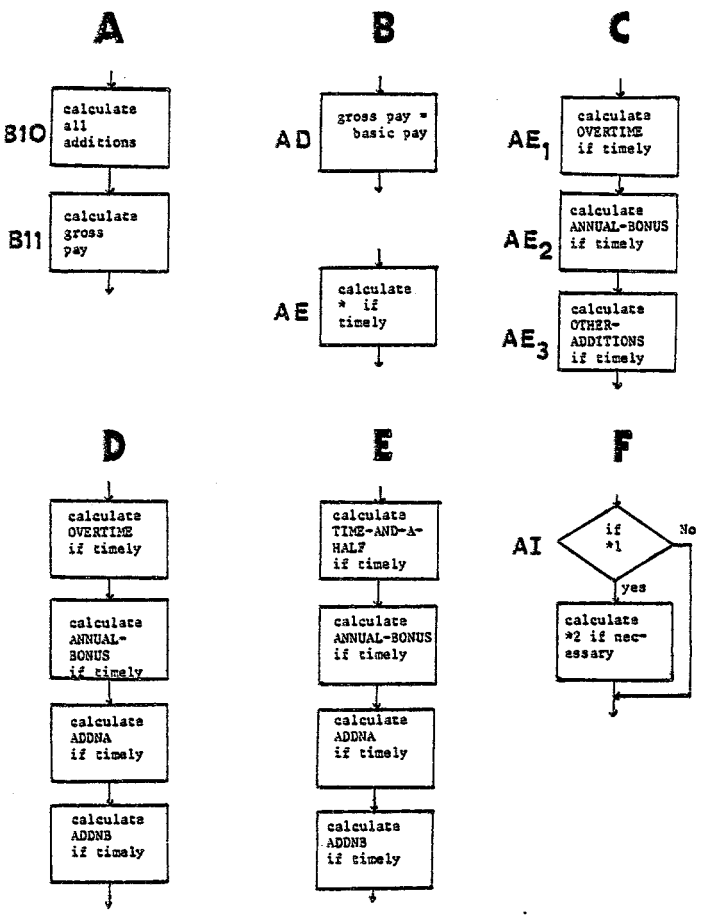


Fig. 2 (*a*) A portion of the initial flowchart;
(*b*) possible replacement boxes;
(*c*) part of the flowchart after expansion by question 9;
(*d*) the same part after expansion by question 10;
(*e*) same as (*d*) after expansion for overtime;
(*f*) the pair of boxes which replace each box of type AE

accepts as input the information set up by the question answering system and that by the Format Editor and uses these to generate a COBOL program which is tailored to the user's requirements (see Fig. 1).

3. The generation process

The strategy followed in the process of generating flowchart boxes is based on the principle of stepwise refinement. The first few questions determine the nature of the program to be generated (e.g. weekly payroll, single shot program, setup program, etc.); from these the initial flow diagram is determined. Each subsequent question, which is asked, is concerned with the refinement of boxes in this flowchart and, depending on the user's reply, one or more boxes in the flowchart may be replaced by other boxes with more specific information.

To demonstrate this, consider the portion of flowchart in Fig. 2(*a*). This represents part of the whole flowchart at an early stage in the question answering process. One will then encounter the question

Q.9. WHAT ADDITIONS?
WHAT ADDITIONS ARE THERE TO BASIC PAY/ STANDARD SALARY?
A) OVERTIME
B) ANNUAL BONUS
C) HOLIDAY BONUS
:
N) NONE
O) OTHER?

The expected answer procedure will read the user's reply—a string of characters—checking that each character lies in the range A to O. The characters are stored on a stack. The action procedure does the following:

1. If the answer is 'N', remove box B10 completely and replace box B11 by box AD (see Fig. 2(*b*)); goto question 13.
2. Otherwise replace box B10 by a series of boxes of type AE (see Fig. 2(*b*)) replacing the parameter * by the appropriate string. If 'A' occurs in the stack, set the overtime pointers; if 'O' occurs in the stack, set the 'other' pointers and go to' Q. 10. If the overtime pointers are set but not the 'other pointers, go to Q. 11 otherwise goto Q. 13.

Assuming that the user has replied giving the letters A, B and O, then box B10 would be replaced by the flowchart in Fig. 2(*c*). Now since the user has specified other additions (O), the next question to be asked will be:

Q.10 OTHER ADDITIONS
WHAT OTHER ADDITIONS ARE THERE?

The expected answer procedure reads one or more strings and stores them on a stack. The action procedure replaces the OTHER ADDITION box (in this case box AE_3) by one or more boxes of type AE with the appropriate addition name substituted in place of the parameter. Thus if there are two other additions, say ADDNA and ADDNB, the updated flowchart will look like that in Fig. 2(*d*).

The next question is responsible for handling different types of overtime and expanding box AE_1 accordingly, e.g. as in Fig. 2(*e*). One might then be asked about the frequency of each addition as follows:

Q. 15 FREQUENCY OF ADDITIONS
AN ADDITION MAY BE CALCULATED IF
(A) IT IS A WEEKEND

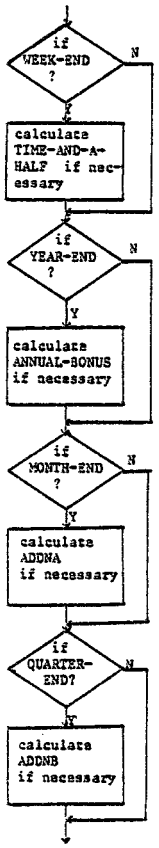


Fig. 3 The portion of the flowchart after question 15

(B) IT IS A FORTNIGHTEND
(C) IT IS A MONTHEND
⋮
(I) A SPORADIC INDICATOR HAS BEEN SET.
THE ADDITIONS ARE LISTED BELOW. USE THE ABOVE LIST TO INDICATE NEXT TO EACH WHICH FREQUENCIES ARE APPLICABLE.

In this case the action field replaces each box of type AE by the pair AI with the appropriate substitution of parameters. The result is shown in Fig. 3.

4. Conclusion

The question answering interface with the user is an effective way of establishing the user's requirements. Even someone with little or no computer experience can generate a payroll program within a matter of hours and a complex system can be set up in a short space of time.

The question answering system produces as output a flowchart structure. This has several advantages, the most important of which is that the use of a flowchart structure makes this part of the system to a certain extent language independent. It is possible to write a Program Generator for other languages, e.g. FORTRAN, in which case one could produce one's final generated program in a variety of languages.

The reason for producing the final generated program in a high level language instead of producing a machine code program directly is that the user may then inspect the result to check that it does suit his requirements. Thus if he should want a facility not catered for by the system it is possible to add the appropriate COBOL statements to the generated program. Since COBOL is a reasonably efficient language for this type of problem the loss of efficiency in generating COBOL statements is not particularly serious.

The use of the stepwise refinement approach meant that the questions had to be organised in such a way that information is obtained in order of increasing detail. This is the most logical way of presenting the questions and is the most useful from the point of view of manipulating the flowchart structure produced.

The system has been implemented using PASCAL on an ICL 1902T.

Acknowledgements

One of the authors, Miss E. V. C. Fielding, was in receipt of a CSIR bursary when this work was carried out.

References

BALZER, R., GREENFELD, N., KAY, M., MANN, W., RYDER, W., WILCZYNSKI, D. and ZOBRIST, A. (1974). Domain-independent Automatic Programming, in *Proceedings of IFIP Congress 71*, North-Holland, Amsterdam, pp. 326-330.

CHEATHAM, T. E. and WEGBREIT, B. (1972). A Laboratory for the Study of Automatic Programming, *AFIPS Conf. Proc.*, 40, pp. 11-21.

KNUTH, D. E. (1974). Computer Programming as an Art, *CACM*, Vol. 17 No. 12, pp. 667-673.

Predicting student success in an introductory programming course

L. J. Mazlack

Computing and Information Science, University of Guelph, Guelph, Ontario, Canada

Over several years, more than a thousand students have taken the same introductory programming course. The students were drawn from all academic disciplines and academic experience levels. No correlations were found between success in the course or its components when posed against academic programme, gender or semester in school. Additionally, when the IBM Programmer's Aptitude Test was administered to a smaller group of mathematically orientated students taking the same course, the predicative value of the test was found to be low.

There are several questions that immediately come to mind when constructing an introductory course. Some of them are:

(*a*) who should it serve?

(*b*) should the disciplines be separated?

(*c*) are the students from varying disciplines equally competitive?

(*d*) what is the potential performance difference due to the number of semesters of academic experience?

(*e*) does one gender perform better than the other?

(*f*) is it possible to predict performance by an aptitude test?

The importance of these questions varies from person to person. Answers which may appear to be self-evident to one person may be considered to be an emotional response to another. An empirical study was developed to shed some light on these questions. The course material was held constant over a period of several semesters. The student's performance was then correlated against a standard aptitude test and against various personal descriptions (semester in school, sex, etc.).

The three PAT (IBM Programmer's Aptitude Test) test components and PAT total score were correlated against the various course components. The highest correlations existed between student performance in the PAT scores and the midterm examination results. The midterm examination contained the least programming of any component in the course. The correlation with the pure programming and problem analysis component of the course (assignments) was very low. It logically follows that this type of test should not be administered to college level people as the results are meaningless and might have adverse effects on the test takers.

No significant correlation was found between academic performance and academic discipline. This is directly opposed to the often made assumption that a student's academic discipline is a good predictor of potential competency in programming. This implies that if a course is properly constructed and presented, there is no need to segregate students from differing academic disciplines due to concerns based on learning ability or interdiscipline competitiveness.

No significant difference was found in academic performance between the genders. The only observable difference is that women appear to be more consistent than men.

The correlations found between semester in school and academic performance were very low. This clearly indicates that in a properly constructed course, there is no reason to worry more about those beginning their university experience

70. M.H.WILLIAMS, A system to assist survey analysts, Software Practice and Experience, vol. 12, 487-491 (1982).

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 12, 487–491 (1982)

A System to Assist Survey Analysts

M. H. WILLIAMS
Computer Science Department, Rhodes University, Grahamstown 6140, South Africa

SUMMARY

The researcher who knows little about computers but wants to conduct a survey and analyse the results by computer can land himself in some difficulty if he does not appreciate some of the problems of computerization. This paper describes a system which is designed to aid such a person by providing assistance with the design of the questionnaire, the capturing of the data and the final analyses.

KEY WORDS Survey analysis Statistical analysis

INTRODUCTION

Sociologists, psychologists, political scientists, educationalists, and various other academics frequently make use of the computer to help them to analyse surveys. The more experienced among them are familiar with the process and have no problems with planning a questionnaire, conducting the survey, encoding the data and using a statistical analysis package[1, 2] to obtain their results. However, one often encounters less-experienced researchers who have no computer background and who do not fare as well.

The latter type of individual may design his questionnaire without giving any thought to the problems of encoding the data at a later stage. He conducts the survey and at the end of it, arrives at the Computer Centre with a heap of completed questionnaires which he wants analysed.

Some Computer Centres will assist such a person by encoding the data for him; in other institutions he may have to employ someone to do it for him; or even do it himself. But if the questionnaire has been poorly laid out, the task of encoding the data can be rather difficult and prone to error, and this process could end up taking considerably longer than is actually necessary.

Once the data have been encoded, the research worker can begin with the analyses. For this he will usually make use of a statistical analysis package. All that needs to be done in this case is for the parameters for the package to be set up correctly. However, for the novice this can be very confusing, and it may take some while before he is able to handle this process on his own.

Thus the novice will usually depend on a reasonable amount of advice and assistance from other research workers and Computer Centre staff, and if this is not sought at the appropriate time it can lead to a lot of wasted time and energy. For this reason a system has been developed to assist users who are about to conduct a survey

* Current address: Department of Computer Science, Heriot–Watt University, 79 Grassmarket, Edinburgh EH1 2HJ, Scotland.

0038–0644/82/050487–05$01.00 *Received 5 September 1980*
© 1982 by John Wiley & Sons, Ltd.

but who do not have a detailed knowledge of computers and computing. The system is intended to assist the user from the initial stages of the project (questionnaire design) through to the final stages of analysis of the results. It is divided into three sections aimed at assisting the user with three separate stages of the survey.

PREPARATION OF THE QUESTIONNAIRE

The first section assists the user in laying out a questionnaire. It is assumed that he has designed the set of questions which he wants to ask and has thought about the type of answer which he is expecting in response to each question.

The program concerned here is the Questionnaire Program. It is responsible for obtaining from the user the details of the questionnaire and for printing out the questionnaire in a suitable format (see Figure 1).

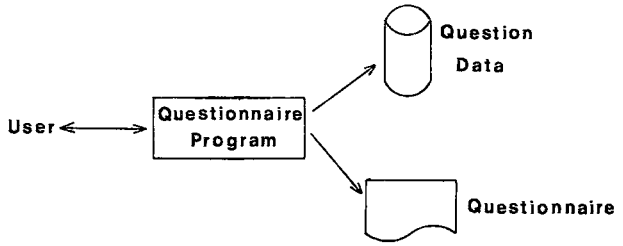


Figure 1. The Questionnaire Program

This program is an interactive program which obtains the information it requires using a question-answering approach. For each question in the questionnaire the user is asked to provide the text of the question and is also asked about the type of answer expected, the maximum and minimum values of each answer, and so on. These details are stored on a disc file. The program has a break facility which enables the user to stop at a suitable point and return later to continue from where he left off (or from a previous question if he wishes).

```
PAGE 1          COMPUTER MANAGER SURVEY
*******         ********************************
(1) AGE
    ****
HOW OLD ARE YOU?                                                  1-2|  |  |

(2) SEX
    ****
WHAT SEX ARE YOU (M OR F)?                                          3|  |

(3) INCOME
    ********
HOW MUCH DO YOU EARN?
CHOOSE YOUR ANSWER'S UNITS FROM THE FOLLOWING:
(A) POUNDS PER WEEK
(B) POUNDS PER MONTH                                    4-9*   *|  |  |  |  |  |
(C) POUNDS PER YEAR
** NB—WRITE CODE REFERRING TO CHOICE OF UNITS IN BOXES MARKED
  WITH STARS

(4) STAFF
    *******
HOW MANY OF EACH OF THE FOLLOWING CATEGORIES OF COMPUTER
PERSONNEL DO YOU EMPLOY?
    (A)
    ***
PROGRAMMERS                                                      10-11|  |  |
```

(B)
***
ANALYST/PROGRAMMERS 12–13 | | |

(C)
***
SYSTEMS ANALYSTS 14–15 | | |

(D)
***
PROJECT MANAGERS 16–17 | | |

PAGE 2 COMPUTER MANAGER SURVEY
****

(5) MACHINE
****
GIVE DETAILS OF THE TYPES OF COMPUTER CONFIGURATIONS WHICH YOUR COMPANY OPERATES. PLEASE SPECIFY MODEL, SIZE OF MAIN MEMORY, ON-LINE DISC STORE CAPACITY, NO. OF TAPE DRIVES, VDUS, ETC.

.....

.....

.....

.....

.....

.....

.....

(6) SALARY
****
WHAT IS THE AVERAGE MONTHLY SALARY (IN POUNDS) WHICH YOU PAY TO EMPLOYEES IN EACH OF THE FOLLOWING CATEGORIES?

(A)
***
PROGRAMMERS 18–21 | | | | |

(B)
***
ANALYST/PROGRAMMERS 22–25 | | | | |

(C)
***
SYSTEMS ANALYSTS 26–29 | | | | |

(D)
***
PROJECT MANAGERS 30–33 | | | | |

PAGE 3 COMPUTER MANAGER SURVEY
******* **********

(7) EDUCATN
****
THE FOLLOWING QUESTIONS RELATE TO THE STAFF MENTIONED IN QUESTIONS 4 AND 6.

(A) LEVEL
****
HOW MANY OF YOUR EMPLOYEES POSSESS A COMPUTER-RELATED DEGREE? (E.G. B.SC. IN COMPUTER SCIENCE OR INFORMATION PROCESSING, B.COM. IN BUSINESS DATA PROCESSING, ETC.) 34–36 | | |

(B) EXPECTN
****
WHAT IS YOUR IMPRESSION OF THESE GRADUATES AS A WHOLE?
(A) VERY DISAPPOINTING—WILL NEVER HIRE AGAIN
(B) A LITTLE DISAPPOINTING
(C) NO BETTER THAN AVERAGE
(D) ABOVE AVERAGE
(E) VERY IMPRESSIVE
37 | |

(C) APPTMENT
****
IF TWO ROUGHLY EQUAL CANDIDATES WERE TO APPLY TO YOU FOR EMPLOYMENT, ONE WITH A COMPUTER-RELATED DEGREE, THE OTHER WITHOUT, WOULD YOU APPOINT THE PERSON WITH THE COMPUTER-RELATED DEGREE IN PREFERENCE TO THE OTHER ONE? 38 | |

Figure 2. A sample questionnaire produced by the Questionnaire Program

When all the details have been entered the user can request the program to print the questionnaire. The program reads back the details from the disc file, decides on a suitable format for the answers to each question and prints the questionnaire in a format designed to ease the task of encoding the data at a later stage, with the question text suitably set out on the left hand side and boxes numbered according to card column numbers on the right hand side, etc. Figure 2 shows a typical questionnaire generated by the program.

ENTERING THE DATA

Once the user has prepared a questionnaire and conducted his survey, he will return with a pile of completed questionnaires. The next task is to transfer the data on these questionnaires to a file on disc within the computer system.

At this stage the user makes use of the Verifier Program. This program reads back the details of the types of questions from disc and, using these, it reads the lines of data entered by the user and checks that the answer in each field has the correct type, lies within range of the expected maximum and minimum values, etc. (see Figure 3).

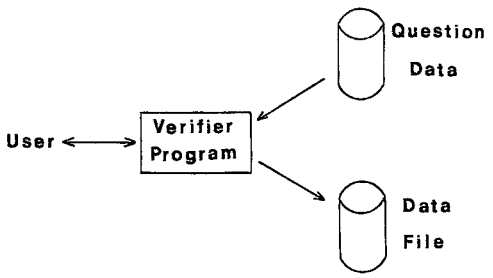


Figure 3. The Verifier Program

In some cases an answer may be given in various units. For example, the area of a farm may be given in acres, hectare, morgen, square miles or square kilometres, wages/salary may be given in pounds per week, pounds per month or pounds per year. This type of answer is encoded using two fields—one indicating the type of units used and the other containing the value. This type of answer is checked and converted to a common unit by the Verifier Program.

Once again the system provides a facility to stop and continue at a later stage, to list the data entered thus far and to correct data which is in error.

ANALYSING THE DATA

When all the data have been entered, the user will be ready to perform statistical analyses on them. For this purpose there are various statistical analysis packages available to the user. He must simply choose the package which he wishes to use, set up the parameters for the analyses which he wishes to perform and run the package.

In our case SPSS is the most commonly used statistical package on our machine. In order to use it one must draw up a set of parameters consisting of three types. The first few parameters describe the format of the data and the variable names; this is followed by the data, and finally the parameters describing the analyses to be performed on the data.

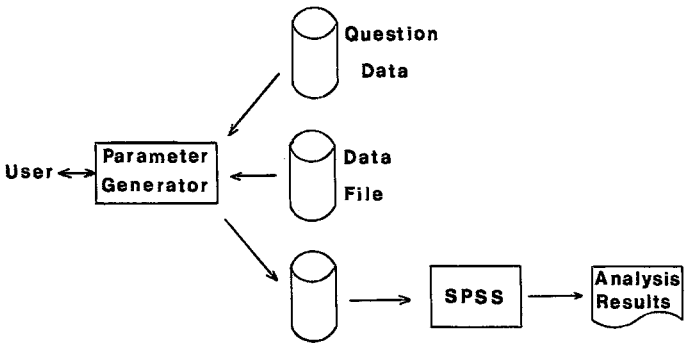


Figure 4. The Parameter Generator Program

Once again the system is designed to assist the user, in this case by setting up the parameters for him. The program for doing this is called the Parameter Generator Program. This program reads the details of the question names and types of answer expected from the disc file and uses this information to set up the first set of parameters. It then asks the user about the analyses which he wishes to perform and from his replies it sets up the analysis parameters for SPSS. Finally the two sets of parameters are combined into a single file with the questionnaire data, and SPSS is executed with this file as input. The whole process is illustrated in Figure 4.

CONCLUSIONS

The design of this system was prompted by the experience of a number of novices who have conducted surveys and analysed them on the computer, and in the process have made unnecessary mistakes and considerable demands on the time of their colleagues. The aim of this system is to reduce both of these as much as possible.

The system has been implemented in Fortran on an ICL 1904S, and is designed for use with MAXIMOP.

ACKNOWLEDGEMENT

The author expresses his gratitude to the Council for Scientific and Industrial Research for a grant for the project and to the various students who have assisted in various ways.

REFERENCES

1. N. H. Nie, C. H. Hull, J. G. Jenkins, K. Steinbrenner and D. H. Bent, *SPSS Statistical Package for the Social Sciences*, McGraw-Hill, New York, 1975.
2. W. J. Dixon, *BMD Biomedical Computer Programs*, University of California Press, Berkeley, 1973.

71. K.T.MILNE, M.H.WILLIAMS, B.J.CLARK and A.F.FELL, An expert system for the analysis of fluorescent substances, Proceedings of the Sixth International Workshop on Expert Systems and their Applications, Agence de L'Informatique, Avignon, vol. 2, 1203-1217 (1986).

AN EXPERT SYSTEM FOR THE ANALYSIS OF FLUORESCENT SUBSTANCES

K.T. Milne, M.H. Williams, B.J. Clark*, and A.F. Fell*
Department of Computer Science
Heriot-Watt University
79 Grass Market
Edinburgh EH1 2HJ, Scotland
United Kingdom
***School of Pharmaceutical Chemistry**
University of Bradford
Bradford BD7 1DP
United Kingdom

Abstract

The process of identifying chemical substances through spectral analysis is one which could benefit from computerisation using a knowledge-based approach. An expert system has been developed to apply spectral analysis to the problem of identifying fluorescent substances, known as polynuclear aromatic hydrocarbons, which may be present either as single substances or as components of a mixture. The system controls a luminescence spectrometer, directing the capture of fluorescence data and performing the subsequent analysis to produce the result.

Keywords : Expert Systems, Fluorescence, Spectral Analysis.

1. Introduction

The identification of chemical substances by luminescence analysis techniques involves the control of an instrument to capture the data as well as the analysis of this data. The process requires a high degree of skill on the part of the operator and can be very time-consuming; it is especially difficult if a mixture of substances is involved [1]. Consequently there is considerable benefit to be gained from the application of expert systems techniques to this problem.

One area of luminescence analysis which is of particular interest is that of fluorescence. A major problem with the identification of an unknown substance using fluorescence analysis is that identification relies on pattern recognition which is a time-consuming process, particularly in view of the sizes of the data sets involved. Several different identification techniques were implemented and used to identify unknown substances correctly, although they still proved time-consuming. Consequently a process was developed which obtained a rapid initial estimate consisting of a candidate list with a limited set of possibilities, which may be used before embarking on a more thorough examination. The process of identifying the components of a chemical mixture is more complex and requires the combination and expansion of these techniques.

This paper will describe the process of fluorescence and the problems encountered in completing the analysis of a compound. The methods used to identify a single substance and analysé a mixture will also be described.

2. Fluorescence Spectroscopy

Fluorescence spectroscopy enables chemical substances to be examined by using light to observe absorption and emission characteristics [2]. It has two main advantages. As it is extremely sensitive it can detect materials in concentrations as low as 10^{-9} to 10^{-10} grams per millilitre. It is also very specific, having two distinguishing characteristic wavelengths - the wavelength of maximum excitation (or absorption) and the wavelength of maximum emission. To observe fluorescence analytically an instrument such as a luminescence spectrometer is required. The essential components of this instrument, also known as a fluorescence spectrometer or fluorimeter, are illustrated in figure 1.

The fluorimeter's source lamp emits an intense beam of ultra-violet radiation, the excitation beam, and directs it at the sample which is dissolved in a suitable solvent and held in a quartz cell. In order that a number of different wavelengths of light can be selected individually, the excitation beam passes through a diffraction grating or prism known as the excitation monochromator. The monochromator can be moved electro-mechanically to select any particular excitation wavelength λ_{ex}. When the beam hits the sample it will be absorbed and then re-emitted as fluorescent light. This emitted light will then pass through the emission monochromator which selects a particular wavelength λ_f or λ_{em} from the spectral range present. The intensity of fluorescence at this wavelength is then detected by a photomultiplier tube, amplified, and then recorded on one of a variety of devices.

If one monochromator is held at a fixed wavelength while the other is varied over a predetermined range, the intensity can be measured to produce a spectrum representing the intensity of fluorescence as a function of the varied wavelength. Two spectra can be produced in this way - the excitation spectrum and the emission spectrum. The emission spectrum for phenanthrene where λ_{ex} = 245 nanometers (nm) and λ_{em} = 220-600 nm is shown in figure 2, and the excitation spectrum for the same substance where λ_{em} = 365 nm and λ_{ex} = 200-400 nm is shown in figure 3. A number of different emission spectra can be produced at a variety of excitation wavelengths to build an Emission-Excitation Matrix (EEM). This 3 dimensional picture can be represented by a contour plot, in which points of equal intensity are joined together by a line. A contour plot of phenanthrene is shown in figure 4.

3. Process of Analysis

The manual analysis of a compound is normally carried out by a chemist who interacts with an instrument such as a luminescence spectrometer to obtain certain sets of data for subsequent interpretation. First he will set various parameters, such as the excitation and emission wavelength limits, to enable a set of intensity data to be collected. Then, from a representation of this data, the expert will try to establish the probable identity of the chemical substance(s) present in the sample. In order to automate this identification process and incorporate it into an expert system, it was decided to interface a microcomputer with an instrument to control the data collection. Once this had been achieved, expertise could be applied to the data sets to obtain

the necessary results. There are a number of problems, however, which have to be solved.

Sets of fluorescence spectra are often represented by as many as 31000 integer values and therefore, a considerable amount of time is required to obtain a set of data from the fluorimeter. The large amount of data also makes the comparison of different substances difficult as certain substances possess very similar spectra. Moreover, the intensity data obtained will vary from one analysis to the next due to instrument noise and solvent effects. This exacerbates the problem of identification, particularly when an unknown substance is present in an extremely low concentration near to the limits of detection. Finally, spectral overlap in mixtures of compounds introduces a number of additional problems. The fluorescence obtained for different substances present in a mixture are often affected by each other, causing swamping and shifting of certain characteristic features. The manner in which this will occur is not predictable and the problem increases as the number of overlapping components increases thus making the identification of all the individual components far more difficult.

A knowledge-base was built using an example set of polynuclear aromatic hydrocarbons (PAHs). They were selected as examples of fluorescent substances and are of environmental interest as they are known to be powerfully carcinogenic. They are produced by the burning of fossil fuels such as coal and oil.

4. Identification of Single Substances

Three identification techniques exist to represent the intensity data of a single substance in a form that enables an unknown to be easily compared with all substances in the knowledge-base. The three methods are as follows.

1. Topographical features such as peaks, saddles and valleys can be very distinctive and are usually visible in a contour plot. They can also be detected automatically by computer software and stored in the knowledge-base with importance values attached by an expert.

2. The EEM can be reduced so that it can be more easily compared without significant loss of detail. This is achieved by a set of matrices of various different sizes.

3. Excitation and emission spectra through the point of maximum intensity can be obtained by slicing through the EEM. As these can be highly characteristic they are known as fingerprints.

The features, matrices and fingerprints can each individually identify a single unknown substance even if present in a very low concentration near to the fluorimeter's limits of detection. They also represent the intensity data in such a way that instrument noise and inconsistencies can be handled and differentiations can be made between similar substances.

Approximate fingerprints can be measured directly from the fluorimeter, unlike the features and matrices which require a full set of data to be collected. They can thus be obtained quickly and

are accurate enough to enable an initial estimation as to the identity of a single substance, consisting of a reduced set of candidates, to be produced. The estimate is produced by applying rules on chemical structures which govern where in the spectrum certain groups of substances will fluoresce. It not only reduces the expert system search space but also indicates areas where no fluorescence will exist and therefore no data need be collected there, thus reducing the amount of time required to collect a 'full' set of data by approximately 50%.

5. Identification of Mixtures

The problem of identifying the components of a mixture of substances is more complex than recognising a single substance as each of the identification techniques described earlier will perform less well when applied to a multi-component compound. The characteristic features of any substance will usually be reduced in number due to swamping by other substances and shifting of certain features from their expected location. The matrices too will contain less useful information as the number of components increases, and greater spectral overlap will reduce the distinctiveness of the fingerprints. An expert attempting to identify the components of a mixture will also find the task more difficult to solve because representations of the intensity data, such as a contour plot, will contain less information.

A new technique, variable-angle synchronous scanning (VASS) [3] is introduced to assist in the identification process. Variable-angle synchronous scans are similar to excitation and emission scans except that they are more general and can be

obtained by slicing through the intensity data in the EEM. An excitation scan is a slice taken at 90° and an emission scan one taken at 0°, whereas a variable-angle synchronous scan is produced by slicing through the EEM at any angle. If the angle is approximately 45° then the amount of spectral overlap will be reduced and this may enable a feature, characteristic of a particular substance that is not visible in either an excitation or emission scan, to be identified.

In view of this, a composite method comprising the three identification techniques mentioned in the previous section and VASS has been developed for the analysis of mixtures. First the topographical features of the mixture are extracted and compared with all the known substances in the knowledge-base to produce a list of probable and possible substances. This initial result is then improved by a matrix manipulation program. Next, the technique of VASS is used to look for additional evidence to confirm or deny the presence of those substances on the probable and possible lists, and also to provide information about any substance not already recognised. Finally, all this information is used to synthetically build profiles (equivalent to the excitation and emission fingerprints) that can be compared with a scan taken at the same point through the mixture EEM. This produces the final list of probable and possible components of the mixture which is then reported to the user.

Some preliminary results, shown in Table 1, were obtained using six mixtures with between 2 and 6 components. The results obtained by both the manual analysis and the expert system are given. The first two columns contain the number of components

which were correctly identified as either probable or possible substances. The third column shows the total of these correct substances and the actual number present. The final column indicates the number of substances which are not actually present in the mixture but were, however, identified as additional possible components. These illustrate that there is still some way to go to achieve precise answers. Nevertheless the results obtained so far are better than those produced by manual analysis. They also contain less mistakes and identify more components as probable substances rather than merely possible ones.

--- TABLE 1 ---

Manual Analysis Results

	Probable Substances	Possible Substances	Total Correct/ Actual No. Present	Errors
mixture-1	2	0	2/2	1
mixture-2	2	0	2/2	2
mixture-3	1	2	3/4	2
mixture-4	2	1	3/5	2
mixture-5	2	1	3/6	2
mixture-6	2	3	5/6	2
TOTAL	11	7	18/25	11

Expert System Results

	Probable Substances	Possible Substances	Total Correct/ Actual No. Present	Errors
mixture-1	2	0	2/2	0
mixture-2	2	0	2/2	0
mixture-3	3	0	3/4	1
mixture-4	3	1	4/5	0
mixture-5	4	0	4/6	1
mixture-6	5	1	6/6	0
TOTAL	19	2	21/25	2

6. Conclusion

The expert system can correctly identify an unknown PAH, using any one of the identification methods, assuming it is only a single substance, and that its spectral data are present in the knowledge base. This recognition is successful even if the unknown is present at an extremely low concentration. An initial estimate of an unknown's identity can also be given almost immediately, and PAH mixtures can be analysed using a combination of the methods and the technique of VASS.

In addition to identifying PAHs, however, the expert system could be applied to the analysis of any fluorescent substances. For this purpose a variety of different knowledge-bases could be built to identify substances such as drugs, their principal impurities and metabolites. It has been established that this approach is suitable for the detection of fluorogenic solutes in high-performance liquid chromatography [4].

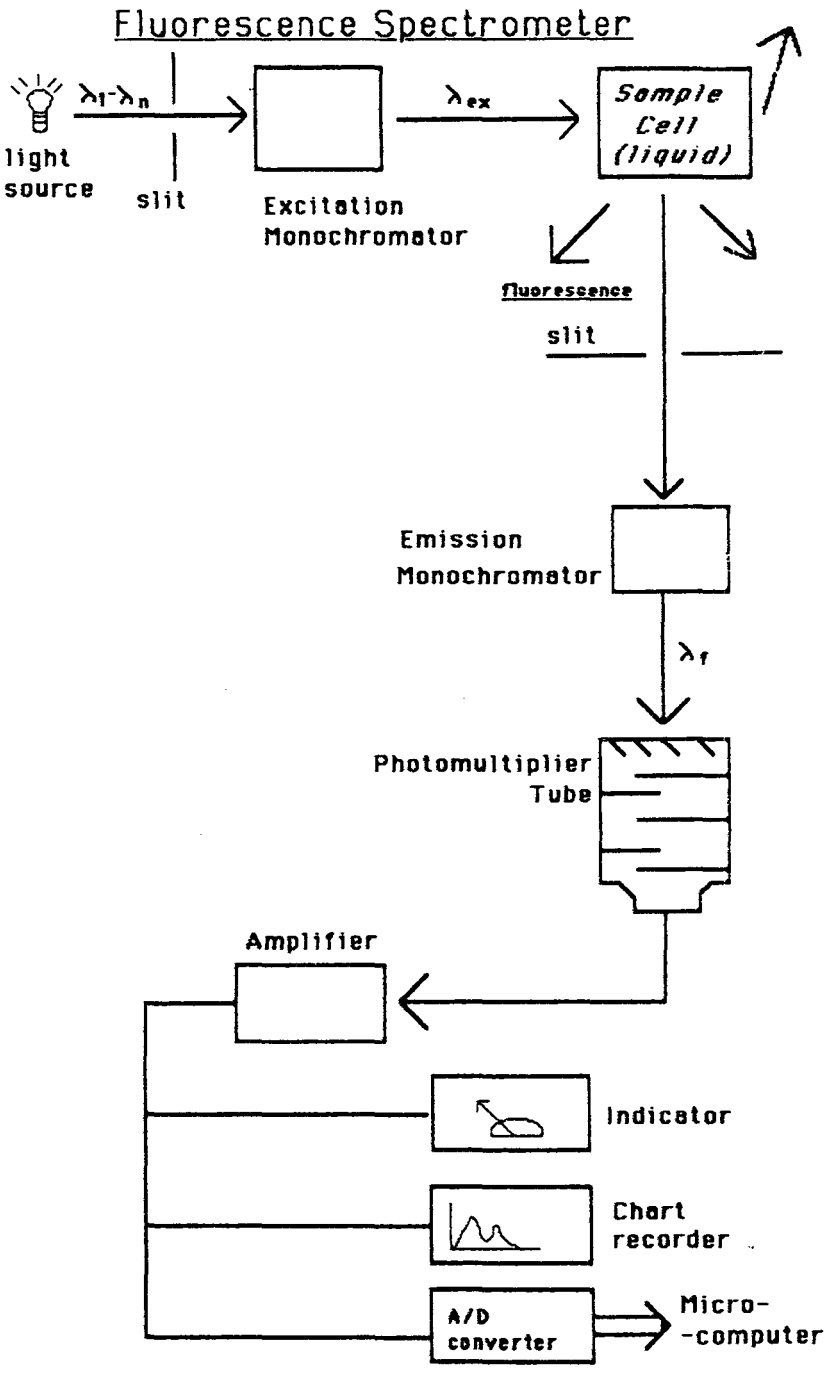
References

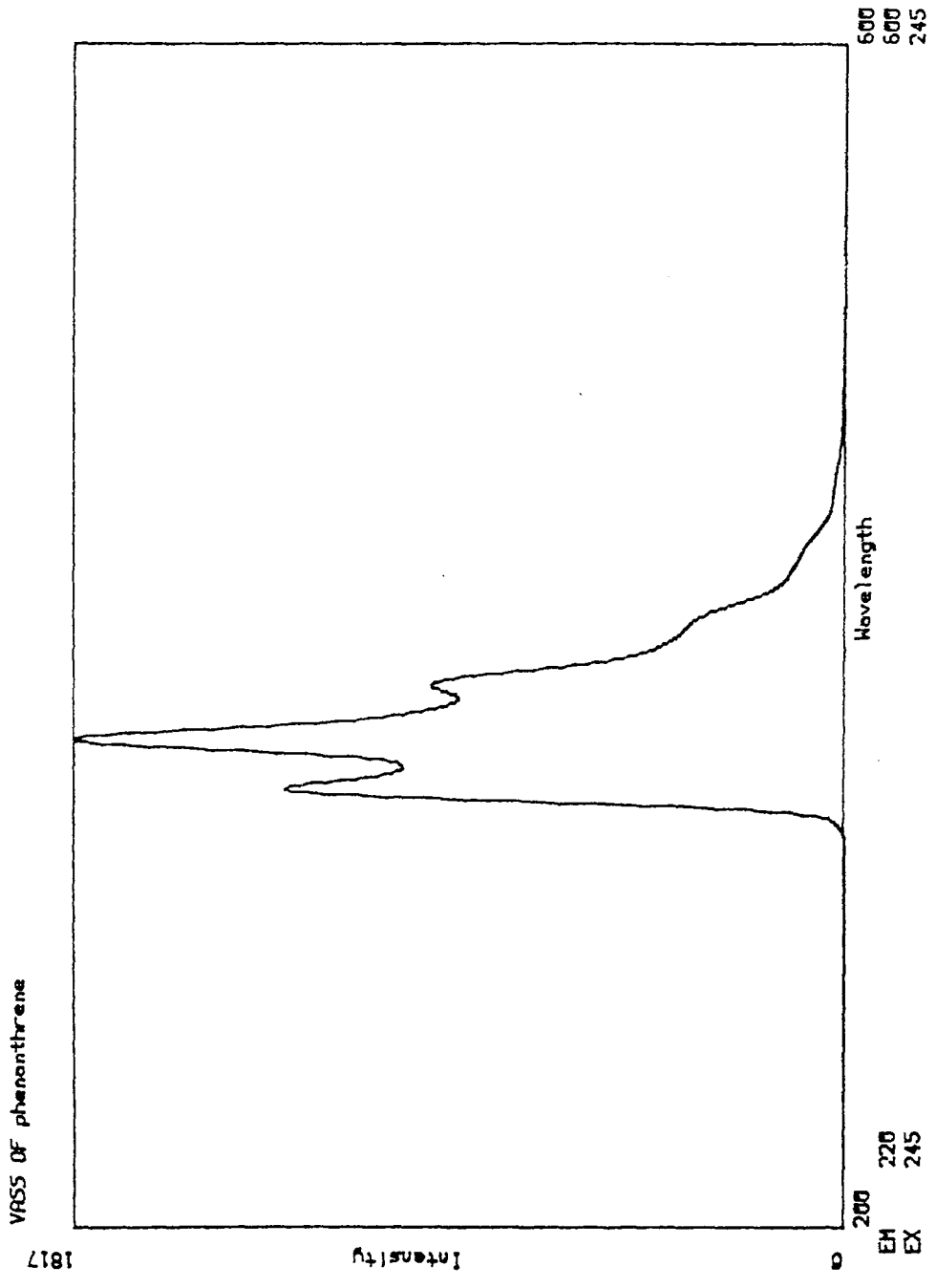
1. K.T. Milne, M.H. Williams, B.J. Clark and A.F. Fell "Expert Systems in Luminescence Analysis", Analytical Proceedings of the Royal Society of Chemistry (1985).

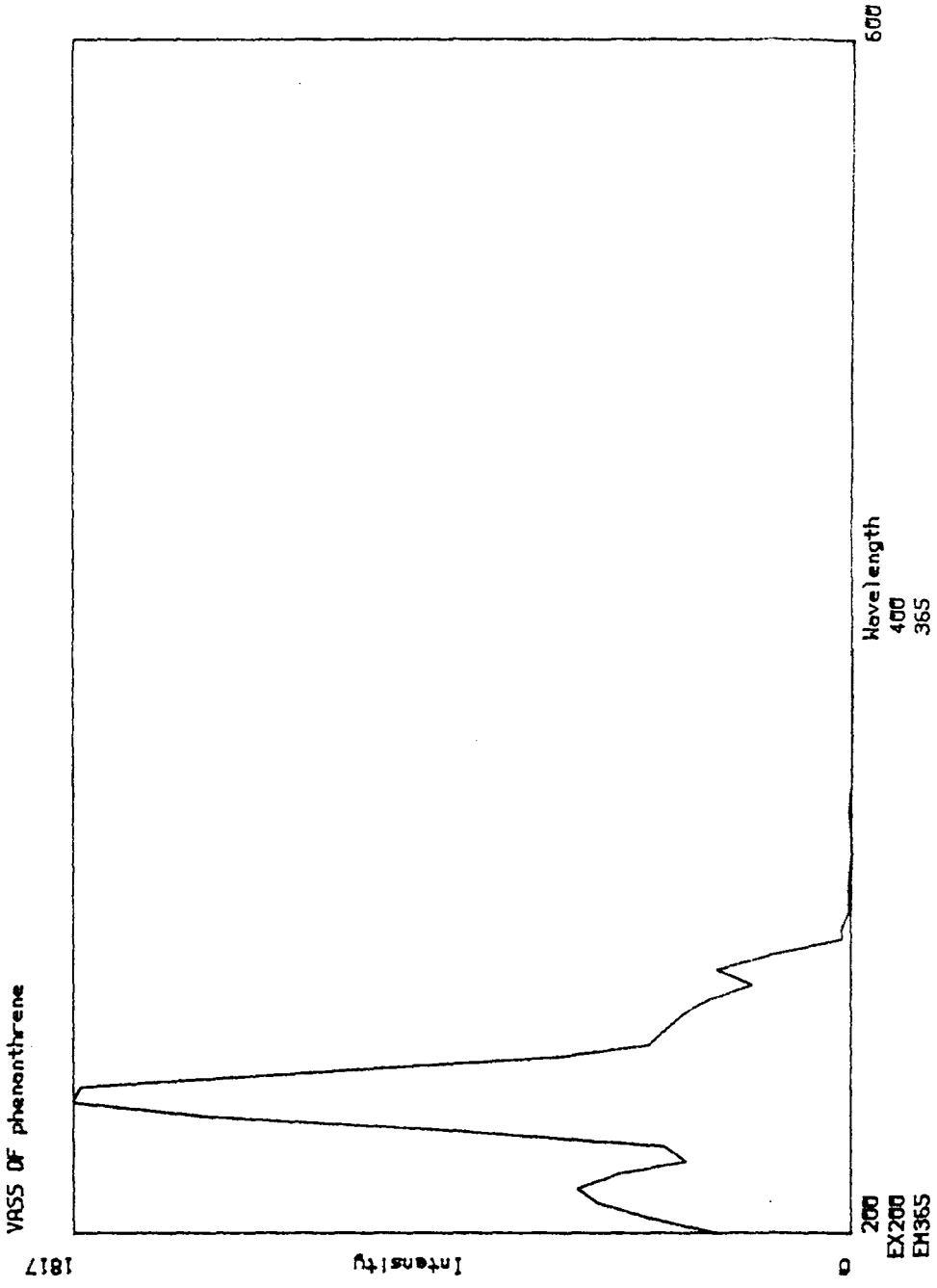
2. E.L. Wehry (Ed) "Modern Fluorescence Spectroscopy Vol. 4" Plenam Press, New York (1981).

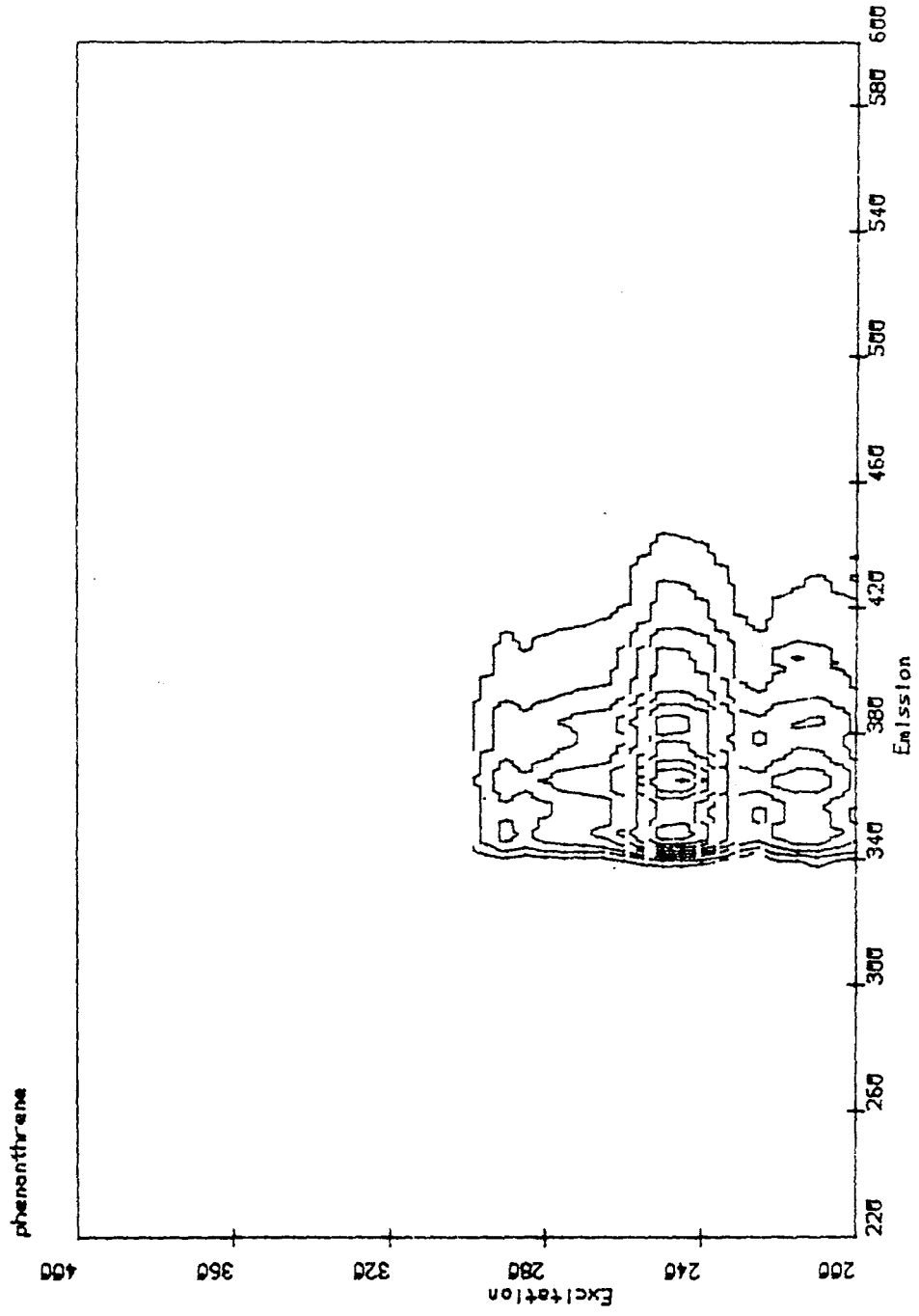
3. B.J. Clark, A.F. Fell, K.T. Milne, D.M.G. Pattie and M.H. Williams "Pharmaceutical Applications of Variable-Angle Synchronous Scanning Fluorescence Spectroscopy" Anal Chim Acta 170 35-44 (1985).

4. B.J. Clark, A.F. Fell, K.T. Milne and M.H. Williams "The Use of an Expert System based on Total Luminescence Spectra for the Identification of Drugs separated by HPLC" J. Pharm. Pharmacol. 37 (suppl.) 129p (1985).


Fluorescence Spectrometer
λ_1-λ_n
light source
slit
Excitation Monochromator
λ_{ex}
Sample Cell (liquid)
fluorescence slit
Emission Monochromator
λ_f
Photomultiplier Tube
Amplifier
Indicator
Chart recorder
A/D converter
Micro--computer


600
600
245
Wavelength
200
220
245
EM
EX
0
Intensity
1817
MASS OF phenanthrene


VRSS OF phenanthrene
1817
Intensity
0
200
EX200
EM365
Wavelength
400
365
600


phenanthrene
Excitation
200
240
280
320
360
400
Emission
220
260
300
340
380
420
460
500
540
580
600

72. K.T.MILNE, M.H.WILLIAMS, B.J.CLARK and A.F.FELL, Expert systems in luminescence analysis, Analytical Proceedings, vol. 23, 157-160 (1986).

each hour of the day, and for the day as a whole, and calculated the "filter efficiency" for each gas. Finally, a graph was produced showing the variation throughout the day. This allowed an instant evaluation of the working of the filter units (Fig. 1) and identified problems such as over-range or instrument malfunction at a glance. These printed data sheets were stored as the primary record of the monitoring system.

Approximately once per week the floppy disk in the second drive unit, which contained the coded raw data, was replaced. The floppy disk was then returned to the laboratory for copying on to a mainframe database for future detailed analysis.

This monitoring system has proved very effective and has now operated continuously for over 3 years with very little "down time." In addition to the advantages of "real time" graphical output, menu-driven programs have been written (also in PASCAL) to allow for calibration of the gas analysers. Data points are displayed on the screen as they are obtained and a "best straight-line" calibration fitted using the method of least squares. Again, the graphical output is most useful in providing a permanent record of each calibration. Several other programs have also been produced to allow for data manipulation and the drawing of graphs for each data channel over the period of a week using the raw data stored on disk.

This method of data logging provides many of the facilities (such as graphical output) found only on the more expensive off-the-shelf data logging systems. It also allows complete freedom, subject only to machine memory size, in manipulating the data, at a price equivalent to that of a simple data logger. All of the plug-in cards came with sufficient software to allow the construction of a monitoring program without too many interfacing problems. The structured nature of the PASCAL programming language also helped during program development.

We acknowledge the contribution to this work made by colleagues at ITE Bush and ITE Banchory. The crop-growth experiment is directed by Mr. I. A. Nicholson and was funded in part by the Ministry of Agriculture, Fisheries and Food and the Department of the Environment.

Reference

1. Elphinstone, G. B., Fowler, D., Owen, G. H., and Nicholson, I. A., "An Open-top Chamber for Studying the Effects of Air Pollution on Plants," *Ins. Terrestr. Ecol. Annu. Rep.*, 1983, 54.

Expert Systems in Luminescence Analysis

K. T. Milne, M. H. Williams, B. J. Clark* and A. F. Fell*
Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ

Introduction

With the current availability of artificial intelligence languages and techniques there are now a number of possibilities for extending the capabilities of conventional analytical spectroscopy.

As the first stage of a collaborative project, an expert system has been developed for the identification of fluorescent polynuclear aromatic hydrocarbons (PNAH), which are of environmental interest. A luminescence spectrophotometer has been interfaced with a 16-bit microcomputer and the expert system developed in a micro-PROLOG environment with access to PASCAL routines for instrument control, data transfer and analysis.

A number of analytical techniques have been incorporated into this system and are invoked as required when the system is applied to the qualitative analysis of synthetic mixtures.

Expert Systems

With the recent developments in artificial intelligence techniques, it is now possible to translate human expertise in a particular domain into a form that makes it possible for a computer to manipulate it, and thus convincingly perform tasks previously only undertaken by experts. Such "expert" or "knowledge based" systems are now widely used in many disciplines, including medicine, chemistry and mining.[1]

An expert system consists of three components: an inference engine or interpreter, a knowledge base and an explanation facility. The knowledge base is the repository for knowledge in the system and consists of a set of rules that have been composed from the ideas of one or more human experts, either by interviews, or (automatically) by computer software. One particular advantage of storing knowledge in this form is that it can be updated incrementally, thereby enabling new rules to be added and corrections to be made with ease.

The inference engine controls the sequence of operations performed by the expert system. At each stage this involves the selection of an appropriate rule from the knowledge base, followed by the interpretation of that rule. In the process of interpretation the user may be asked to provide information about the subject (*e.g.*, details of a patient's condition in the case of medical expert systems).

An essential component of an expert system is the facility to explain its line of reasoning by informing a user how it arrived at a particular answer, or why some other answer was not identified. This justification is vital in de-bugging the expert system as the control path is determined by the system rather than by the programmer.

Process of Luminescence Analysis

The process of determining the composition of some unknown sample by luminescence analysis involves an interaction between an expert and an instrument; the expert first uses his knowledge to set the parameters of the instrument and collect an appropriate data set, and then applies his expertise to interpret the results.

In order to automate this process it is necessary to interface a computer with the instrument so that it can control the data collection process. Once this has been carried out the data that is captured in this way can be analysed and manipulated so that the unknown sample can be identified.

In certain areas, for example high-performance liquid chromatography (HPLC), it may be necessary to capture several different sets of data, at various parameter settings, in order to complete an analysis and identify positively the components. However, in the instance of a luminescence spectrophotometer, it is possible to set up the parameters so that a complete set of data is obtained in a single run. In view of this difference it was decided to concentrate on the simpler instrument, the luminescence spectrophotometer, in the first instance.

Even with the simpler instrument there are a number of problems which have to be solved.

1. Each set of spectra is composed of a large number of integer values, thus raising the question as to how to compare such

* Present address: School of Pharmaceutical Chemistry, University of Bradford, Bradford BD7 1DP.

spectra. This problem is exacerbated by the fact that the spectra for certain substances are very similar.
2. Instrument noise and solvent effects give rise to a lack of consistency, which reduces the amount of useful information and increases the difficulty of the matching process. This is particularly noticeable when an unknown substance is present at an extremely low concentration.
3. In the instance of mixtures of substances, spectral overlap frequently leads to swamping and shifting of characteristic features as well as the creation of new features. Also, as the number of overlapping components in a mixture increases, the total luminescence data become less useful, thereby making the task of identifying individual components far more difficult.

Development of the Expert System

The hardware configuration used for the system is shown in Fig. 1. The microcomputer is an Advance 86b, which is an IBM-PC compatible machine running under MS-DOS. It is based on an 8086 microprocessor and thus, unlike the IBM-PC, it is a true 16-bit machine.

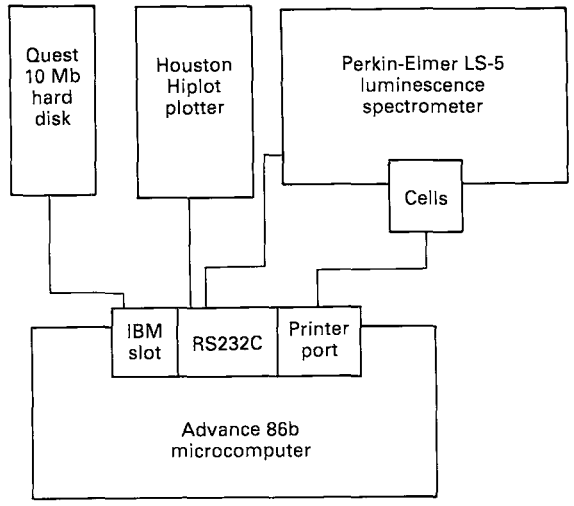


Fig. 1. Hardware configuration

In the configuration, a 10 Mb hard disk unit acts as a temporary store for the large amounts of luminescence data that are generated. The RS232-C interface can be connected to either a Perkin-Elmer LS-5 Luminescence Spectrometer for collecting data, or to a Houston Digital Plotter for producing contour or single-scan plots. The LS-5 has four sample cells, which can be changed by a stepper motor controlled by the computer through a modified Centronics printer port.

As the aim of this project was to develop an expert system that involves a large amount of numeric processing, two programming languages were used in its implementation. As logic programming languages are generally regarded as the most suitable for implementing expert systems, micro-PROLOG[2] was selected for control and decision making. Because this programming language is unsuitable for large-scale numeric work, the imperative language PASCAL[3] handles all the data transfer and numeric processing.

There are, however, no mechanisms available to link these two languages directly so that control can be passed from one to the other. For this reason a "load and execute" primitive has been developed to allow a micro-PROLOG program to load any number of compiled programs from a disk into main memory and execute them, thereby giving the expert system access to fast numeric processing utilities.

The first stage in developing the over-all system was to link the various hardware components and create a sub-system to control the operation of the spectrophotometer. This sub-system controls the positioning of the sample cells in the LS-5, the issuing of commands to the spectrophotometer to set parameters and control its operation, the capture of data and storage on a disk file, and preliminary numeric processing.

In order to analyse this data a number of identification techniques were employed. This required the establishment of a knowledge base with appropriate information on the characteristics of different spectra. Although the process of setting up this knowledge base was largely automatic, it did require the involvement of an expert to confirm the conclusions and to add additional information where necessary.

The system was tested by building up a knowledge base for a typical set of fluorescent substances from the set of polynuclear aromatic hydrocarbons (PNAH).

Once the knowledge base was complete a simple expert system was developed which obtained information about an unknown and applied the knowledge in the knowledge base to the problem of identifying a single substance. The system was able to give a qualitative answer as to the identity of an unknown, and also explain to the user how it arrived at the particular answer and why it did not identify some other substance instead. Currently, this expert system is being extended to handle the more complex problem of identifying the components of mixtures.

Identification Techniques

Three methods have been used to reduce the large data matrix to a more manageable size, giving a description of the data that can be compared with an unknown in a reasonable amount of time. Each method can individually identify a single substance, and they can be adapted to identify the constituent components of a mixture.

Distinctive Features

One way of determining the identity of an unknown substance is to identify its characteristic features. Three different types of features that may be present are the standard topographical features, namely peaks, saddles and valleys. By looking at the points at which both emission and excitation scans have turning points[4,5] these features can be identified. Fig. 2 gives a summary of the definitions used.

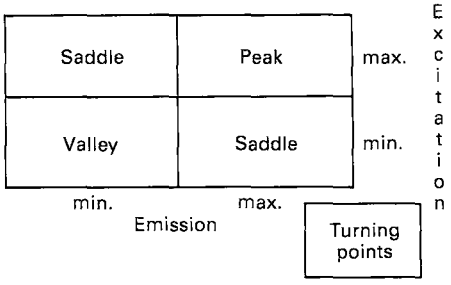


Fig. 2. Definition of distinctive features of traces

The features that are detected by computer software are usually visible in a contour plot of the emission - excitation matrix (EEM), see Fig. 3. By using plots of this kind, an expert can attach importance values to each feature.

Matrix Method

To be able to compare one substance with another easily it is obviously necessary to reduce the EEM to a more manageable size without significant loss of detail. In this system the EEM is a 761 by 41 array, which represents excitation wavelengths of between 200 and 400 nm and emission wavelengths of between 220 and 600 nm. There is, however, no one size that incorporates both easy comparison and no great loss of detail. For this reason a set of matrices of different sizes has been selected. The sizes used here are 80 by 40, 20 by 10, 10 by 5 and 5 by 5.

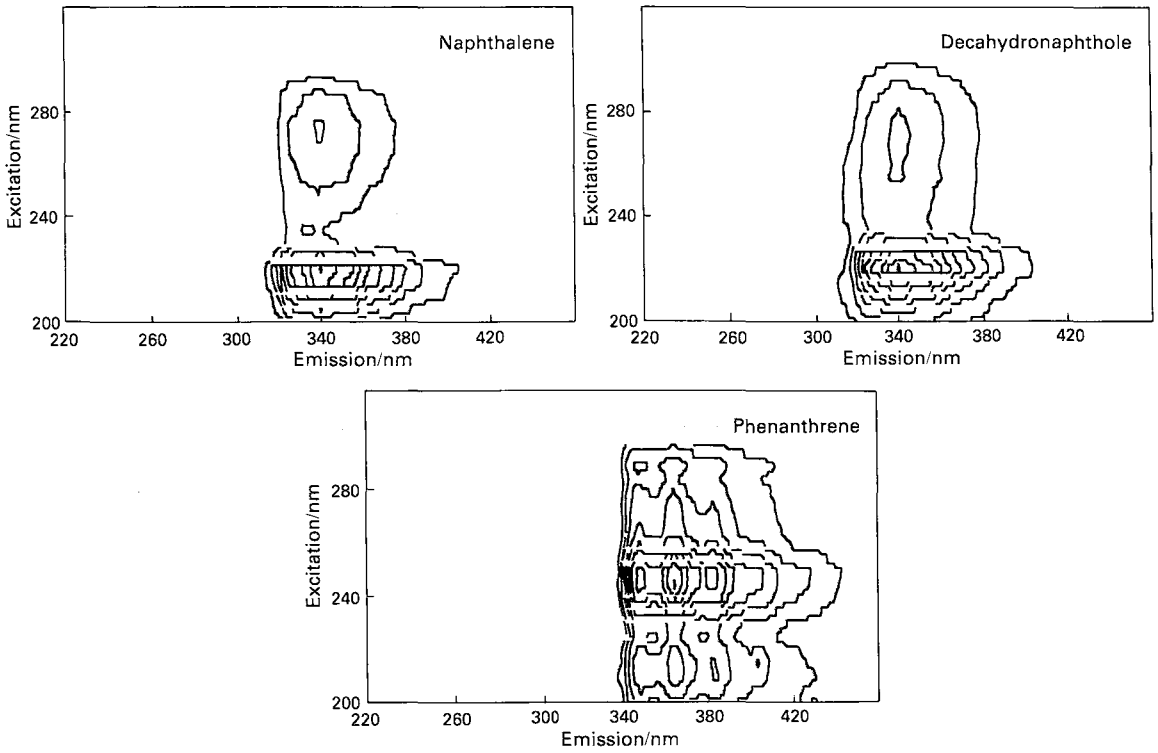


Fig. 3. Contour plots of the emission - excitation matrix

For each of the four sizes there are two different types of matrix: the density matrix and the activity matrix. The density matrix is simply a reduction of the larger data matrix, whereas the activity matrix gives an indication of the degree of change in intensity.

These matrices, once normalised, can be compared with an unknown substance using the smallest one first and moving progressively towards the largest matrix if and only if more than one substance matches with the unknown.

Fingerprint Method

Single excitation and emission scans through the peak of maximum intensity can provide a great deal of information as other features tend to lie in line with this point. These excitation and emission "fingerprints" can be normalised and then compared with an unknown using the method of least squares.

Identification of Mixtures

The three methods described above work well for a single unknown substance, identifying it correctly in each instance, even when present in a very low concentration. However, the problem of a mixture of unknown substances is more complex and introduces further problems. Much detail can be lost because of swamping and shifting of features, and it is no longer a straightforward matter to deduce the components of a mixture simply by looking at a contour plot (see Fig. 4 for a mixture of six components consisting of anthracene, carbazole, diphenylmethane, fluorene, perylene and phenanthrene). As a result, the three methods which have been used to identify accurately a single unknown substance have to be adapted to identify the components of a mixture.

Conclusions

It has been shown that an expert sytem that controls a luminescence spectrophotometer can easily identify a single unknown polynuclear aromatic hydrocarbon using a variety of methods, provided that its spectral data are present in the knowledge base. Moreover, it is also possible to use the identification methods and the technique of variable-angle synchronous scanning[6] to identify the constituent components of a mixture. Although the results produced by the expert system for mixtures are not always perfectly correct, work is continuing on improving the identification process within the limits of the instrument.

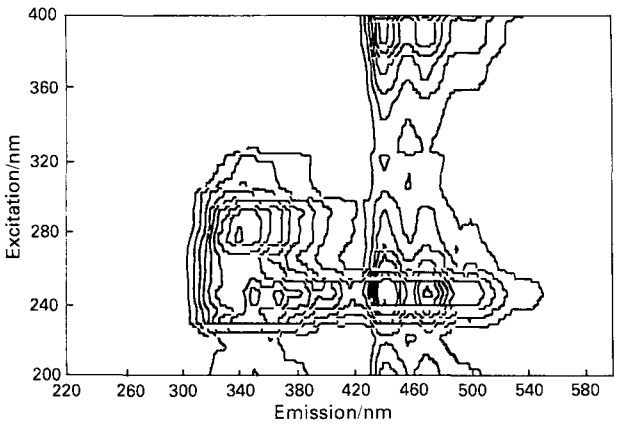


Fig. 4. Contour plot for a mixture of anthracene, diphenylmethane, perylene, carbazole, fluorene and phenanthrene

Although PNAHs have been used to demonstrate the system, it is clearly not restricted to these, and any other fluorescent substances can be handled in a similar way. Preliminary work has also established that this approach is suitable for the detection of fluorogenic solutes in HPLC.[7]

References

1. Dessy, R. E., *Anal. Chem.*, 1984, **56**, 1200.
2. Clark, K. L., and McCabe, F. G., "micro-PROLOG—Programming in Logic," Prentice Hall, London, 1984.
3. Jensen, K., and Wirth, N., "PASCAL User Manual and Report," Springer-Verlag, Berlin, 1975.
4. Savitzky, A., and Golay, M. J. E., "Smoothing and Differentiation of Data by Simplified Least Squares Procedures," *Anal. Chem.*, 1964, **36**, 1627.

5. Steiner, J., Termonia, Y., and Deltour, J., "Smoothing and Differentiation of Data by a Simplified Least Squares Procedure," *Anal. Chem.*, 1972, **44**, 1906.
6. Clark, B. J., Fell, A. F., Milne, K. T., Pattie, D. M. G., and Williams, M. H., "Pharmaceutical Applications of Variable-Angle Synchronous Scanning Fluorescence Spectroscopy," *Anal. Chim. Acta*, 1985, **170**, 35.
7. Clark, B. J., Fell, A. F., Milne, K. T., and Williams, M. H., "The Use of an Expert System Based on Total Luminescence Spectra for the Identification of Drugs Separated by HPLC," *J. Pharm. Pharmacol.*, 1985, **37** (suppl.), 129.

4 Analytical Journals

from The Royal Society of Chemistry

The Analyst

An international journal of high repute containing original research papers on the theory and practice of all aspects of analytical chemistry drawn from a wide range of sources. It also publishes regular critical reviews of important techniques and their applications, short papers and urgent communications (which are published in 5–8 weeks) on important new work, and book reviews.
12 issues per annum; annual author and subject index.
See Subscription Rates 1986 (below) for price details.

Analytical Proceedings

Analytical Proceedings is the news and information journal of the Analytical Division. It contains special articles, reports of meetings, extended summaries of original papers, safety articles, details of recent legislation, surveys of equipment, and many other items of general interest to analytical chemists both in Britain and overseas.
12 issues per annum (plus index).
See Subscription Rates 1986 (below) for price details.

Journal of Analytical Atomic Spectrometry

An International Journal on the Development and Application of Atomic Spectrometric Techniques
Journal of Analytical Atomic Spectrometry (JAAS) is a new international journal, first published in February 1986, which contains original research papers, short papers, communications and letters concerned with the development and analytical application of atomic spectrometric techniques. *JAAS* is published bimonthly and includes comprehensive reviews on specific topics, general information and news of interest to analytical atomic spectroscopists, including information on forthcoming conferences and book reviews. Special issues of *JAAS* will be published, devoted to subjects highlighted by particular symposia. Also included in *JAAS* are the literature reviews previously covered in *Annual Reports on Analytical Atomic Spectroscopy*. *JAAS* has a style and format similar to that of the well-established journal *The Analyst*. *JAAS* provides an improved publication service to support the growing research efforts in, and applications of, atomic spectrometric techniques.
Six issues per annum
First Published February 1986
£165.00 ($319.00) Rest of World £182.00
RSC Members £33.00

Subscription Rates 1986

The Analyst
Analytical Abstracts
Analytical Proceedings

The Analyst
£147.00 ($285.00) Rest of World £162.00
RSC Members £33.00

Analytical Abstracts
£219.50 ($425.00) Rest of World £241.00
RSC Members £57.50

Analytical Proceedings
£69.00 ($134.00) Rest of World £76.00
RSC Members £11.00

The Analyst, Analytical Abstracts and Proceedings
£375.00 ($726.00) Rest of World £412.00

The Analyst and Analytical Abstracts
£329.00 ($636.00) Rest of World £361.00

The Analyst and Proceedings
£184.00 ($356.00) Rest of World £202.00

N.B. *The version of Analytical Abstracts printed on one side of the page only is no longer available.*

Analytical Abstracts

Analytical Abstracts endeavours to cover the whole field of analytical chemistry, providing more than 12,000 abstracts per annum of papers and books considered to be of importance and interest to analytical chemists. A subject index is provided in each issue and a volume index is produced annually.
See Subscription Rates 1986 (below) for price details.

Ordering:
RSC Members should send their orders to: The Royal Society of Chemistry, Assistant Membership Officer, 30 Russell Square, London WC1B 5DT.
Membership Manager, 30 Russell Square, London WC1B 5DT.
Distribution Centre, Blackhorse Road, Letchworth, Herts SG6 1HN, UK.

ROYAL SOCIETY OF CHEMISTRY
Information Services

73. B.J.CLARK, A.F.FELL, K.T.MILNE, D.M.G.PATTIE and M.H.WILLIAMS, Pharmaceutical applications of variable-angle synchronous scanning fluorescence spectroscopy, Analytica Chimica Acta, vol. 170, 35-44 (1985).

Analytica Chimica Acta, 170 (1985) 35—44
Elsevier Science Publishers B.V., Amsterdam — Printed in The Netherlands

PHARMACEUTICAL APPLICATIONS OF VARIABLE-ANGLE SYNCHRONOUS SCANNING FLUORESCENCE SPECTROSCOPY

BRIAN J. CLARK[a,*], ANTHONY F. FELL[a], KENNETH T. MILNE[b], DONALD M. G. PATTIE[b] and M. HOWARD WILLIAMS[b]

Analytical Research Group, [a]Department of Pharmacy and [b]Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ (Great Britain)

(Received 7th September 1984)

SUMMARY

The development of variable-angle synchronous scanning (v.a.s.s.) in fluorescence spectrometry is reported, based on a computer-aided spectrofluorimeter. The technique permits a linear path to be scanned at any preselected angle through the emission-excitation matrix defined by (I_{em}, λ_{em}, λ_{ex}), by effectively scanning the emission and the excitation monochromators at different speeds under computer control. When applied to pharmaceutical dosage forms, v.a.s.s. gave good selectivity for chlorpromazine in the presence of its principal degradation product, chlorpromazine sulphoxide, and for oxytetracycline in the presence of the additives vitamin C, thiamine, nicotinamide and riboflavin. Good calibration linearity, precision and recovery were observed for both principal drug components. The angle of the scan trajectory can also be varied continuously through the emission-excitation matrix, to describe any desired path under computer control. This novel technique of non-linear v.a.s.s. can provide an improved method for generating diagnostic profiles of drugs, degradation products and metabolites.

In luminescence studies, high sensitivity and selectivity are generally expected. However, problems of selectivity and ultimately of sensitivity can occur in multicomponent analysis because of the overlap of the broad-band spectra observed. Specificity is a particular problem in the determination of luminescent drugs in their dosage forms, which are often multicomponent preparations. In consequence, a number of procedures has been proposed to increase selectivity in quantitative luminescence spectrometry.

One approach is to exploit a multichannel detector, based either on the vidicon tube [1, 2], which effectively captures the 2-dimensional emission-excitation matrix (EEM), or on the linear photodiode array [3, 4], which operates only in the emission wavelength domain. These detectors are capable of generating a large amount of luminescence data in just a few milliseconds. Moreover, the utility of these detectors can be extended to allow quantitation by incorporating deconvolution algorithms [5]. However, the vidicon is at present relatively expensive, while the linear photodiode array spectrofluorimeter is not yet widely accessible. An alternative to multichannel detection involves the improvement of spectral resolution of

0003-2670/85/$03.30 © 1985 Elsevier Science Publishers B.V.

overlapping peaks, captured by conventional means, by utilising computer-based mathematical techniques such as least-squares deconvolution [6], derivative transformation [7, 8] and matrix-isolation spectroscopy [9].

A further technique in this category of spectral resolution improvement is synchronous luminescence spectroscopy, also described as synchronous excitation spectroscopy. This method was originally proposed by Lloyd [10] and has been recently reviewed by Vo-Dinh [11]. It involves simultaneous scanning of the excitation and emission monochromators, separated by a constant wavelength interval, $\Delta\lambda$. The resulting spectra represent the intensity profile of a 45° section cut through the EEM, and are generally characterised by decreased bandwidths, which can lead to improvements in selectivity. The main areas of application have been in air pollution studies [11, 12] and as an aid to 'fingerprinting' in the oil industry [13, 14]. Relatively few applications have been reported in the field of pharmaceutical and biomedical analysis [15—20].

One drawback to conventional synchronous scanning spectroscopy is that it is limited to producing 45° sections, whose locus is defined by the constant wavelength interval $\Delta\lambda$ employed. This is also true of the constant-energy modification in the frequency domain ($\Delta\nu$), proposed recently by Inman and Winefordner [21]. A novel approach which offers considerable flexibility is variable-angle synchronous scanning (v.a.s.s.), where the wavelength separation between the monochromators is continuously varied [22]. This can be achieved either mechanically, by varying the relative scan speeds of each monochromator [23], or digitally, where the luminescence data are stored for subsequent processing by microcomputer [24]. In the present work, the increased selectivity afforded by the v.a.s.s. technique has been examined in studies on the determination of psychotropic and antibiotic drugs in their respective dosage forms.

EXPERIMENTAL

Instrumentation and software

All fluorescence spectra were recorded on a Perkin-Elmer LS-5 spectrofluorimeter which includes a quantum-corrected reference system for fully-corrected excitation spectra. Slit-widths for both monochromators were set at 5 nm. The spectrofluorimeter was operated in the computer-controlled mode via the RS232C serial interface by either of the following systems. In one, a 48 kbyte Apple II microcomputer with dual disk-drive was used with a Houston DMP4 Hiplot digital plotter (Bausch and Lomb, TX); instrument control, data control and manipulation were achieved by a suite of programs written by the authors in UCSD Pascal. In the other, a Perkin-Elmer Model 3600 Data Station microcomputer was used; instrument control and data collection were achieved by using the commercially available Perkin-Elmer Computerised Luminescence Software (PECLS). 'OBEY' programs using this software were written to perform certain tasks.

Programs were developed here to permit variable-angle synchronous scanning using either of the two systems. Two approaches were used. In the first, the initial wavelengths for the emission and excitation monochromators were sequentially stepped in specified increments over the required range. A variable-angle scan at the desired angle can be obtained by selecting the appropriate value and sign for each of the two increments. In the second approach, which was implemented only for the Apple system, the EEM is first collected and stored on disc for processing at a later stage. The angle is selected by choosing the appropriate increments at the processing stage. With this method, it is possible to obtain a number of sequential segments, each at a different angle. Each pair of adjacent segments can be linked together, thereby defining any desired trajectory through the EEM.

A contour-plotting routine was written in BASIC for the Perkin-Elmer system. Contours in the (λ_{em}, λ_{ex}) plane were produced by joining points of equal fluorescence intensity. The routine was based on a recent algorithm developed to present the 3-dimensional data obtained from photodiode-array u.v. detection in high-performance liquid chromatography [25].

Reagents, solutions and materials

Oxytetracycline, thiamine, riboflavin, nicotinamide and ascorbic acid were obtained from the Sigma Chemical Co. Chlorpromazine hydrochloride and its sulphoxide were kindly supplied by May and Baker (Dagenham, U.K.). Anhydrous aluminium chloride was a pure grade (Koch-Light) and potassium dihydrogenphosphate was AnalaR grade (BDH).

Standard solutions. Standard solutions of oxytetracycline and of the vitamins were prepared each day in 0.05 M potassium dihydrogenphosphate (pH 5.0). Aluminium chloride (0.80 M) was added to these solutions and to the dosage form solutions to give a final concentration of 0.20 M. The solutions were allowed to stand for 15 min before measurement. These solutions were found to be stable for at least 1 h. Chlorpromazine hydrochloride (1—10 μg ml^{-1}) and its sulphoxide (0.1—1.0 μg ml^{-1}) were prepared in 0.05 M potassium dihydrogenphosphate (pH 3.0).

Capsule and tablet dosage forms. The capsules analysed for oxytetracycline (Terramycin-SF; Pfizer, Sandwich, U.K.) contained oxytetracycline (250 mg), ascorbic acid (75 mg), thiamine (2.5 mg), riboflavin (2.5 mg) and nicotinamide (25 mg). The contents of each capsule were extracted into phosphate buffer, pH 5.0, followed by dilution to the working range of 1—10 μg ml^{-1} oxytetracycline. For chlorpromazine tablets (Largactil, 25 mg; May and Baker) 20 tablets were weighed and ground and the equivalent of one tablet was extracted with phosphate buffer, pH 3.0, as previously described [20].

RESULTS AND DISCUSSION

The determination of tetracycline and phenothiazines in their dosage forms is possible by various methods. Among these, fluorescence spectro-

metry would be expected to be suitable, although the method in its conventional implementation suffers from the fact that the major peak of interest overlaps with peaks attributable to degradation products or other formulation constituents.

Tetracyclines may be administered in combination with a vitamin complex, either as an intravenous infusion (extemporaneously prepared) or as one of several commercially available dosage forms. In one of these preparations, oxytetracycline is formulated with vitamin C (ascorbic acid) and with the B-complex vitamins thiamine, nicotinamide and riboflavin. Overlap of the emission spectra is not observed for oxytetracycline in the presence of these vitamins, with the exception of riboflavin for which the overlapping emission spectrum is shown in Fig. 1. The enhanced fluorescence intensity for oxytetracycline observed in this spectrum was achieved by complexation with aluminium chloride; tetracyclines in general are only weakly fluorescent. The complexation of oxytetracycline with aluminium has been previously reported to increase fluorescence intensity tenfold [26].

The extent of this overlap can usefully be examined by interfacing a microcomputer to the spectrofluorimeter to obtain the total fluorimetric information available in the EEM. With suitable computer programs,

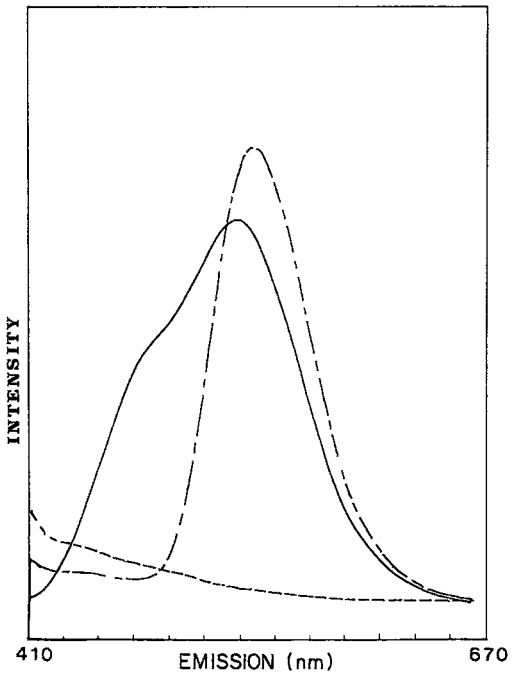


Fig. 1. Emission spectra (λ_{ex} = 378 nm): (—) oxytetracycline (5.01 μg ml^{-1}) and riboflavin (0.189 μg ml^{-1}) in phosphate buffer (pH 5.0)/aluminium chloride (0.2 M); (—·—) riboflavin alone in the buffer; (--) pH 5.0 buffer alone.

3-dimensional spectra can be obtained and presented as the isometric projection, where the emission spectra at stepped increments of excitation wavelength are recorded and plotted (Fig. 2A). A reversed projection of the data can sometimes indicate emission peaks hidden by the foreground (Fig. 2B). Alternatively, the 3-dimensional spectra can be effectively transformed to a plot in the two dimensions of excitation and emission wavelength by linking points of equal intensity to form contours, as shown for riboflavin (Fig. 3A) and for a standard mixture of oxytetracycline and riboflavin (Fig. 3B). This contour presentation has generally been found to be more useful than the isometric projection for indicating the presence of hidden emission peaks. In this case, however, the position of the riboflavin peak is not readily discriminated from the large oxytetracycline emission in the contour plot (Fig. 3B), whereas in the forward projection (Fig. 2A) and more particularly in the reversed projection (Fig. 2B), the riboflavin peak can be more easily seen.

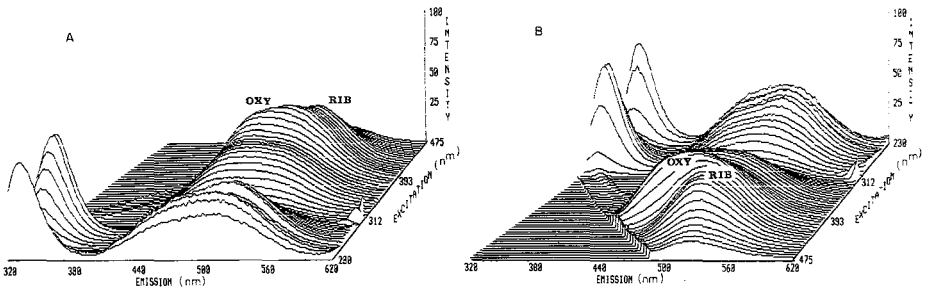


Fig. 2. Isometric plots of the emission-excitation matrix of oxytetracycline (OXY, 10.75 μg ml^{-1}) and riboflavin (RIB, 0.94 μg ml^{-1}) in phosphate buffer (pH 5.0)/ aluminium chloride (0.2 M). (A) Forward projection; (B) reversed projection. Data were obtained using the Perkin-Elmer Model 3600 microcomputer; excitation and emission bandwidths, 5 nm; increments in excitation wavelength, 5 nm for each emission scan; scan speed 240 nm min^{-1}.

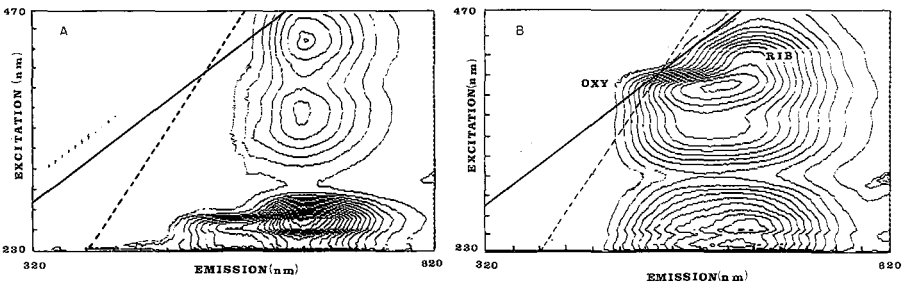


Fig. 3. Contour plots: (A) riboflavin fluorescence; (B) oxytetracycline and riboflavin fluorescence. (—) The conventional synchronous luminescence path slicing the data matrix at 45° ($\Delta\lambda$ = 40 nm); (---) the v.a.s.s. path at 63.4° from (λ_{ex}, λ_{em}) = (230, 340 nm). Experimental conditions as for Fig. 2.

The contour plots are probably of greatest use in indicating the most suitable trajectory to follow in the EEM, in order to obtain conventional synchronous luminescence spectra for the complete resolution of overlapping component peaks. The synchronous scan describes a 45° trajectory through the EEM and, for oxytetracycline and riboflavin, the scan ($\Delta\lambda$ = 40 nm) shown on the contour plot (Fig. 3B) yielded the best attainable resolution of the two drug components, within the constraints of the method. The resolution is shown in Fig. 4. It did not, however, afford sufficient selectivity to resolve completely the overlapping emission peaks of oxytetracycline and riboflavin.

A similar degree of overlap was observed in the spectra of a number of phenothiazines and their spectrally similar sulphoxide oxidation products, as exemplified by the antipsychotic drug chlorpromazine, which is formulated in a number of dosage forms. The spectra are shown in Fig. 5B. The synchronous scan illustrated in Fig. 5A at an optimum separation of $\Delta\lambda$ = 20 nm gave a single sharp peak (λ_{ex} = 294 nm) for the sulphoxide as previously described [20]. At $\Delta\lambda$ = 130 nm (Fig. 5C), which is the most favourable value for the determination of the parent phenothiazine, the major peak shows a small degree of fine structural overlap with the oxidation product (λ_{ex} = 281 nm), which does not, however, interfere in the assay. By employing variable-angle synchronous scanning, complete resolution of the overlapping emission spectra can be achieved, as discussed below.

For the overlapping peaks of oxytetracycline and riboflavin, it was possible

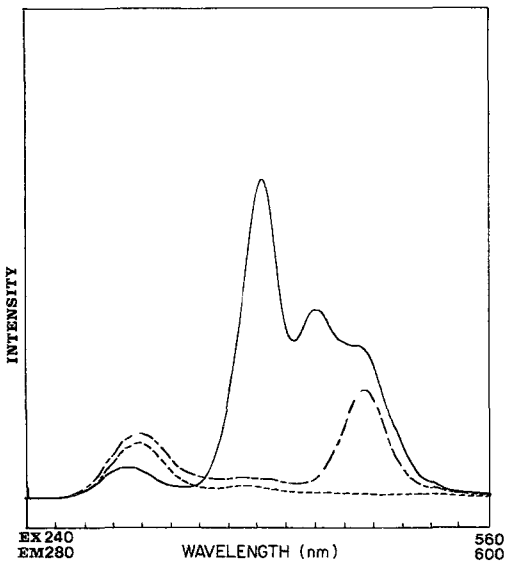


Fig. 4. Synchronous luminescence spectra ($\Delta\lambda$ = 40 nm): (—) oxytetracycline and riboflavin in phosphate buffer (pH 5.0)/aluminium chloride (0.2 M); (—·—) riboflavin alone; (---) pH 5.0 buffer alone. Concentrations and instrumental conditions as for Fig. 2.

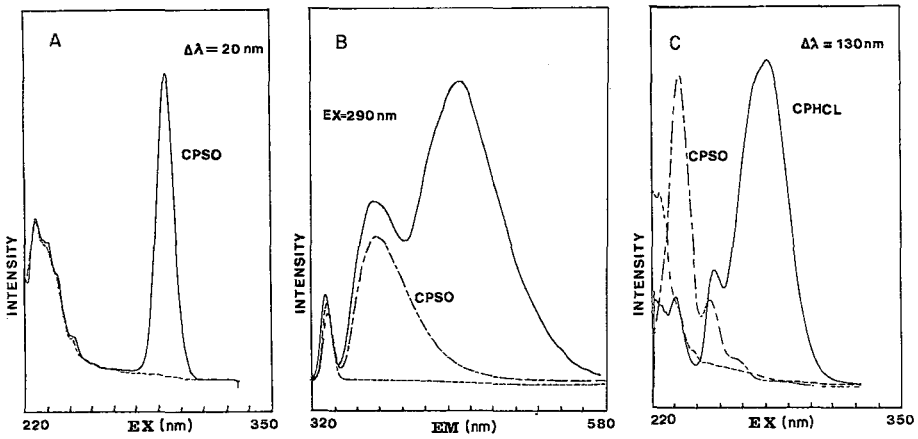


Fig. 5. (A) Synchronous luminescence spectra: (—) chlorpromazine sulphoxide ($\Delta\lambda$ = 20 nm) in pH 3.0 buffer; (---) buffer alone. (B) Emission spectra of chlorpromazine and its sulphoxide impurity, illustrating the extent of spectral overlap (λ_{ex} = 290 nm): (—) chlorpromazine hydrochloride (10.40 μg ml^{-1}) with chlorpromazine sulphoxide (0.258 μg ml^{-1}, CPSO) in phosphate buffer (pH 3.0); (— · —) CPSO alone (0.258 μg ml^{-1}) in phosphate buffer (pH 3.0); (---) buffer alone. (C) Synchronous luminescence spectra in pH 3.0 buffer: (—) chlorpromazine hydrochloride (CPHCl) ($\Delta\lambda$ = 130 nm); (— · —) CPSO; (---) buffer alone.

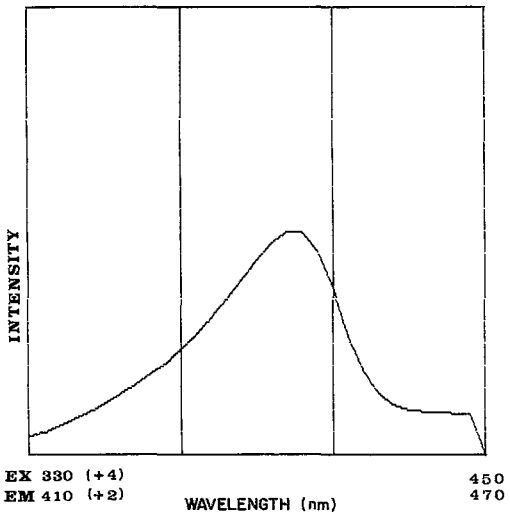


Fig. 6. Variable-angle synchronous scanning spectrum of oxytetracycline at 63.4° from (λ_{ex}, λ_{em}) = (330, 410 nm) through the emission-excitation matrix. Excitation monochromator increment, +4 nm; emission monochromator increment, +2 nm; concentration and instrumental conditions as for Fig. 2.

to obtain spectral resolution by slicing the matrix at 63.4° (Fig. 6). The v.a.s.s. technique allowed the quantitation of oxytetracycline (Table 1), independently of the riboflavin concentration up to 15% (w/w). Attempts to quantify riboflavin in the presence of oxytetracycline were not successful. However, riboflavin could readily be assayed by measurement in the absence of aluminium, when the tetracycline contribution is very slight. The calibration graph was linear for 0—6.1 mg ml^{-1} oxytetracycline in the presence of a constant concentration of riboflavin. The regression line was $y = 12.30\ x + 16.1$ ($n = 5$, $r = 0.991$). The relative standard deviation (r.s.d.) at 4.88 $\mu g\ ml^{-1}$ was 1.96% ($n = 10$).

The chlorpromazine hydrochloride (CPHCl) emission peak was similarly resolved from the interfering sulphoxide by slicing the EEM at a 'reversed' angle of 135° (i.e., λ_{ex} values decreased with increasing λ_{em}). In Fig. 7, both the contour plot and the optimum v.a.s.s. plot for CPHCl are shown. A linear calibration graph was obtained for the parent phenothiazine over the range 0—7.35 $\mu g\ ml^{-1}$, independently of the amount of sulphoxide up to 10% degradation. The regression line was $y = 26.2\ x + 10.24$ ($n = 5$, $r = 0.9894$). The r.s.d. at 5.88 $\mu g\ ml^{-1}$ was 2.34% ($n = 10$). The results shown in Table 1 indicate that the amount of the parent drug in each of the two dosage forms was in reasonable agreement with declared label strength.

The ability to describe any path through the data matrix by non-linear v.a.s.s., without the restriction of scanning in a straight line, raises a number of possibilities. In a complex system, the maximum and minimum emission intensities can be explored by traversing the peaks and valleys. Or, as illustrated in Fig. 8, a curved trajectory can be described through the EEM which permits the light-scattering peaks to be avoided. Improved resolution of overlapping systems which cannot be separated by linear scanning may be possible by this technique. The use of non-linear v.a.s.s. to generate diagnostic profiles of drugs, their degradation products and metabolites offers a new technique for qualitative characterisation in pharmaceutical analysis.

Conclusions

The flexibility conferred by v.a.s.s. has been shown to be successful in resolving peak overlap experienced in two differing problem areas, that of

TABLE 1

Results for the assay of parent drug in the respective dosage forms by v.a.s.s.[a]

Drug	Found (mg)	Recovery (%)	R.s.d. (%)
Oxytetracycline capsules (250 mg)[b]	236.6	94.6	1.96
Chlorpromazine hydrochloride tablets (25 mg)[b]	24.1	96.4	2.34

[a]Conditions and experimental parameters as in Figs. 6 and 7. [b]Nominal content.

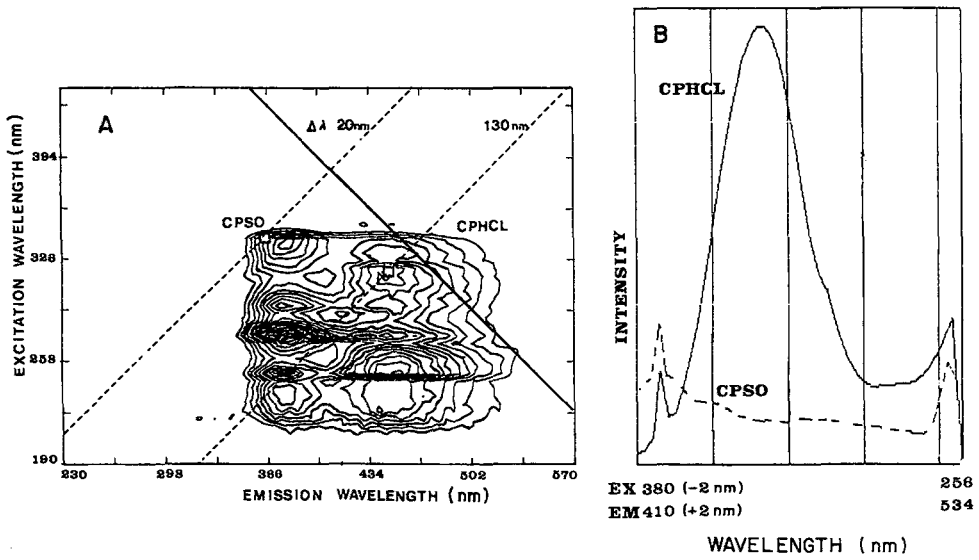


Fig. 7. (A) Contour plot of the fluorescence of chlorpromazine hydrochloride (CPHCl) and its sulphoxide impurity (CPSO): (---) conventional synchronous luminescence paths at 45° through the data matrix for $\Delta\lambda$ = 20 nm and 130 nm; (—) v.a.s.s. path at a 'reversed' angle of 135°. (B) V.a.s.s. spectra at a 'reversed' angle of 135° from (380, 410 nm) (excitation monochromator increment, −2 nm, emission monochromator increment, +2 nm).

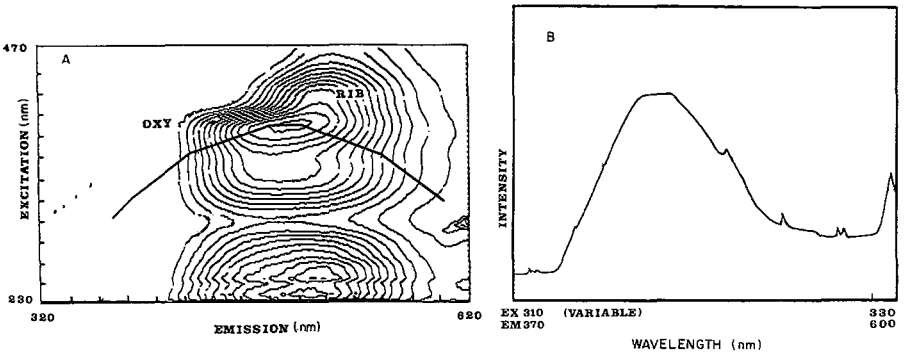


Fig. 8. (A) Contour plot of oxytetracycline and riboflavin demonstrating a curved trajectory cut through the data matrix (cf. Fig. 3B). (B) V.a.s.s. spectrum obtained from the curved trajectory scan. The contour plot was generated by linking 6 consecutive segments in sequence. The excitation and emission monochromator increments varied progressively for each segment, starting at (λ_{ex}, λ_{em}) = (310, 370 nm). Experimental conditions as in Fig. 2, except that the Apple-II microcomputer was used.

a drug and its interfering degradation product and of a drug and its co-formulated components. The technique shows promise as a new approach in the analysis of materials other than pharmaceuticals and could also be applied to further areas of luminescence spectroscopy, such as phosphorescence.

The possibility of describing paths cut at any angle and direction through the data matrix, reported here for the first time, merits further investigation as a novel technique for qualitative and quantitative analysis.

We thank Mr. Barry A. Rowe and Dr. Steven G. Winfield of Perkin-Elmer, Beaconsfield, U.K., for valuable and stimulating discussions. Thanks are also extended to Dr. Hugh P. Scott and Mr. Sandy Yates (Heriot-Watt University) for their valuable assistance in the design of some of the programs employed. The helpful encouragement of Professor James N. Miller (Loughborough University of Technology) in the early stages of this work is gratefully acknowledged.

REFERENCES

1 R. E. Santini, M. J. Milano and H. L. Pardue, Anal. Chem., 45 (1973) 915A.
2 I. M. Warner, J. B. Callis, E. R. Davidson, R. Gouterman and G. D. Christian, Anal. Lett., 8 (1975) 665.
3 M. A. Ryan, R. J. Miller and J. D. Ingle, Anal. Chem., 50 (1978) 1772.
4 Y. Talmi, Appl. Spectrosc., 36 (1982) 1.
5 C. N. Ho, G. D. Christian and E. R. Davidson, Anal. Chem., 53 (1981) 92.
6 W. Lindberg, J. A. Persson and S. Wold, Anal. Chem., 55 (1983) 643.
7 J. N. Miller, Proc. Anal. Div. Chem. Soc., 16 (1979) 203.
8 G. L. Green and T. C. O'Haver, Anal. Chem., 46 (1974) 2191.
9 T. C. O'Haver and W. M. Parks, Anal. Chem., 46 (1974) 1886.
10 J. B. F. Lloyd, Nature (London), Phys. Sci., 231 (1971) 64.
11 T. Vo-Dinh, Modern Fluorescence Spectroscopy, Vol. 4, Plenum Press, New York, 1981, p. 167.
12 T. Vo-Dinh, R. B. Gammage and P. R. Martínez, Anal. Chem., 53 (1981) 253.
13 P. John and I. Soutar, Anal. Chem., 48 (1976) 520.
14 I. Soutar, Anal. Proc., 20 (1983) 19.
15 J. C. André, Ph. Baudot and M. Niclause, Clin. Chim. Acta, 76 (1977) 55.
16 J. N. Miller and A. F. Fell, J. Pharm. Pharmacol., 32 (1980) 70P.
17 J. N. Miller, T. A. Ahmad and A. F. Fell, Anal. Proc., 19 (1982) 37.
18 Ph. Baudot and J. C. André, Anal. Lett., 15 (1982) 471.
19 L. J. C. Love and R. Weinberger, Spectrochim. Acta, Part B, 38 (1983) 1421.
20 B. J. Clark and A. F. Fell, J. Pharm. Pharmacol., 35 (1983) 22P.
21 E. L. Inman, Jr. and J. D. Winefordner, Anal. Chem., 54 (1982) 2018.
22 D. L. Eastwood, in E. L. Wehry (Ed.), Modern Fluorescence Spectroscopy, Vol. 4, Plenum Press, New York, 1981, p. 251.
23 J. N. Miller, Analyst (London), 109 (1984) 191.
24 B. J. Clark, A. F. Fell, I. E. Aitchison, D. M. G. Pattie, M. H. Williams and J. N. Miller, Spectrochim. Acta, Part B, 38 (1983) 61.
25 B. J. Clark, A. F. Fell, H. P. Scott and D. Westerlund, J. Chromatogr., 286 (1984) 261.
26 D. Hall, J. Pharm. Pharmacol., 28 (1976) 420.

74. T.P.BRIDGE, M.H.WILLIAMS and A.F.FELL, Analytical chemistry and expert systems, Chemistry in Britain, vol. 23, no. 11, 1085-1088 (1987).

Analytical chemistry and expert systems

T. P. Bridge, M. H. Williams and A. F. Fell

Early computer systems were developed simply for dealing with arithmetical problems. However, with the development of storage devices such as the Winchester disc-drive, the range of applications has increased so that information storage and retrieval are now as important as numerical computation. With the advent of knowledge based systems computing is on the threshold of another step forward in the types of applications which can be handled.

The most common form of knowledge based system is the expert system,[1] which contains encoded expert knowledge and is capable of exploiting this knowledge for problem solving purposes. One of the earliest implementations of an expert system designed to solve a chemical problem was the Dendral system, developed at Stanford University.[2] This system uses data from mass spectrometry to predict the structure of target molecules by using rules about possible structures and fragmentation processes in the spectrometer. These rules have been obtained from two main sources. They have either been supplied by an expert, or been derived by an associated program, *meta*-Dendral,[3] which analyses the mass spectra produced from a particular class of compounds to suggest probable fragmentation rules. The main Dendral program continues to be developed to include information from additional sources, including facilities to allow data from both proton[4] and ^{13}C magnetic resonance spectroscopy[5] to be incorporated into the structure elucidation process.

An expert system is a knowledge based system and may consist of one or more programs. It is characterised by three essential features—(*i*) a knowledge base containing modules of organised knowledge derived from an expert's skill; (*ii*) an inference engine that uses the knowledge from the knowledge base to supply intelligent advice or to assist with decision making on a specialised topic; and (*iii*) an explanation facility that can be invoked to explain the line of reasoning which has been followed to arrive at the advice given. A typical structure for an expert system is represented in *Fig. 1*.

The knowledge base

The knowledge, which is stored in the knowledge base and used by the expert system, may be derived from various sources. Two classes of knowledge are particularly important. The first is the public knowledge which is generally available from books and journals and which provides a theoretical grounding in a particular problem and a reference base for the second class of knowledge, *ie* the private knowledge of one or more experts. This includes various empirically derived rules and other factors accumulated over several years, and represents the experience, as opposed to the theoretical knowledge, of the expert. It is this experiential knowledge that is difficult to capture and which is essential for the success of a system.

The knowledge is usually stored in the computer as a set of production rules or as statements in a first-order logic structure.[6]

● *Rule-based production systems*[7] represent knowledge as a collection of production rules with the general form:

IF condition THEN action

The condition, or antecedent, comprises one or more statements connected by the logical connectives *and, or* and *not*. If the condition is satisfied when the rule is tested by the inference mechanism, then the action, or consequent, is performed. The consequent will usually cause changes in the state of the database, which may, in turn, allow other rules to become active.

The facts stored in the database may be definitive assertions, or they may contain information on the probability of the particular fact being true. The order in which rules are selected is governed by the inference engine, but the selection and testing process is repeated until either a solution to the original query has been found, or until no further rules exist for which the antecedents can be satisfied, in which case the final state is reported.

● *A system based on first-order logic* has the advantage of well defined semantics. The most successful approach is that in which the knowledge is stored as a collection of Horn clauses,[8] each of which has the general form:

$$R \text{ if } A_1 \,\&\, A_2 \,\&\, \ldots \,\&\, A_k$$

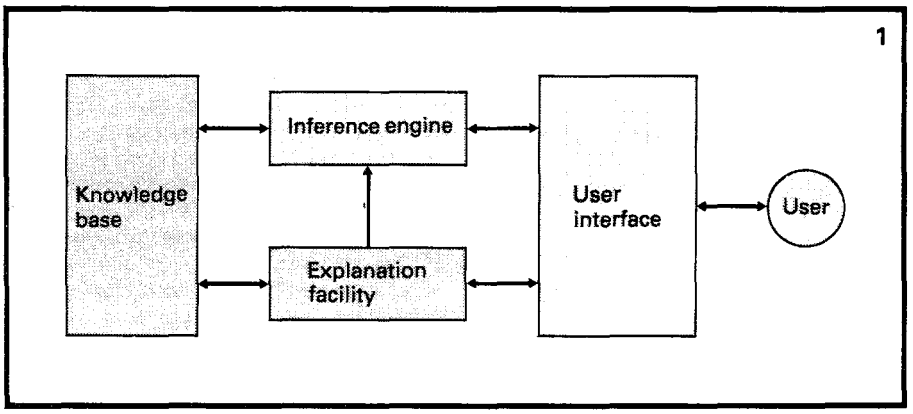
where R is the consequent and $A_1, \ldots A_k$ are the antecedents. This clause may be interpreted as either:

(*i*) if all the statements A_i are true then so is R, or:

(*ii*) to prove R, one must first prove all the statements A_i.

Although this representation may appear to be similar to that used in production rule systems, the representation of knowledge as Horn clauses is less general and implies restrictions on the manner in which the rules may be invoked. In particular the use of Horn clauses excludes the use of probabilities to direct the inference process.

Fig. 1. An outline of the structure of a typical expert system.



The inference engine interprets the knowledge stored in the knowledge base. It is responsible for selecting an appropriate rule from the knowledge base and then interpreting that rule. The way in which rules are selected varies from one inference engine to the next, but there are two fundamental approaches which underlie all selection strategies. These are referred to as forward chaining and backward chaining strategies.

In a forward chaining system the knowledge base is searched for a rule or clause, the antecedent of which is satisfied by the current state of the database. Once a rule has been identified the consequent can be established and the appropriate modifications made to the database. This process is repeated until the final goal is established. Forward chaining is data driven in that the system reasons from the available data towards a final hypothesis. In contrast, backward chaining is goal driven in that a rule is selected if the consequent matches the goal that one wishes to prove, and the system then attempts to prove the antecedent. Thus one starts with the overall goal and searches for a rule, the consequent of which matches the goal. If one is found, the statements in its antecedent, if any, become the sub-goals and the system searches for rules whose consequents match each of these sub-goals. This process is repeated until all of the sub-goals have been proved.

The explanation facility

Because the way in which rules are selected and executed can be very difficult to follow, particularly in a system containing a large set of rules or clauses, an explanation facility forms an essential component of any expert system. This facility is used both as a means of debugging a knowledge base during development and as a key part of the diagnostic process itself. Such a facility will generally provide an explanation of the reasoning process which was followed to arrive at a particular answer; it can explain why a possible answer was rejected by the system; or it can give a reason for asking the user for a particular piece of information. It may therefore be used as a means of teaching users about the system or of gaining their confidence in the system.

The inference engine and explanation facility together form the program, which operates on the knowledge base as data to solve a particular problem. Just as with programs in conventional applications, it is possible to run the same program with different sets of data. In this case it is possible to load the knowledge base with different sets of rules and to run the inference engine and explanation facility with these different knowledge bases. Such a program, an expert system with an empty knowledge base, is referred to as an expert system shell. An example is the shell Emycin,[9] which was derived from the medical consultant system Mycin[10], and subsequently used in various other applications such as the system for diagnosing pulmonary function disorders, known as Puff[11], and in Sacon,[12] which provides guidance in the use of a comprehensive structural analysis program.

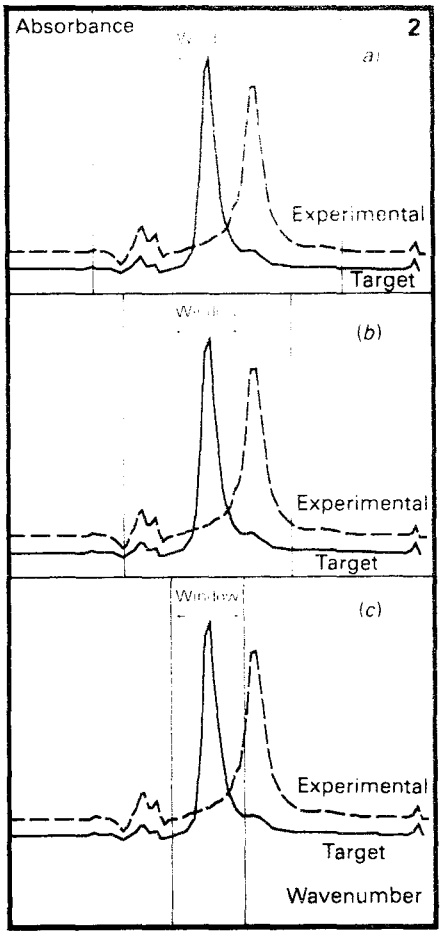


Fig. 2. The incremental windowing peak selection process used in Pairs (broken line: experimental; solid line: target). (*a*) A wide window is used to check for a peak in the experimental spectrum. Because a peak is present in this band a low score is assigned to the match. (*b*) A narrower window is now employed. This match increases the probability score to a higher level. (*c*) A further reduction of the width of the search window excludes the peak in the experimental spectrum with no change to the overall probability score. If this match had proved successful then the probability score would have been increased.

Expert systems have been applied successfully to diagnostic problems in a number of different areas. The increase in success rate over human experts may be attributed to the fact that the expert system may contain knowledge derived from many sources, and may therefore know more about the immediate problem domain than any one person; or to the fact that its knowledge is well organised and is applied in a systematic and exhaustive fashion. An additional benefit is that the formalisation of the knowledge, required for the construction of the knowledge base, may lead to a deeper insight into the problem area, with a consequent benefit to the overall understanding of the subject concerned.

Applications in analytical chemistry

Expert system applications within the field of analytical chemistry fall into two broad categories.

- Systems developed to interpret experimental results.
- Systems that provide guidance as to the course of action the analyst should take.

The expert system Pawmi is used to analyse materials recovered from hazardous waste disposal sites.[13] This system attempts to match features from the Fourier transform infrared (FTIR) spectrum of the analyte with features from a library of reference spectra, and thus to assign a numerical value to the probability of any of the target compounds being present in the mixture. The analysis requires three stages of processing by the expert system. First the interferogram is reduced to a table of peaks, identified by wavenumber. This table is then examined by a second expert system, Pairs,[14] which operates as a subroutine of Pawmi, and matches the features found in the mixture against each of the reference spectra. To allow for the experimental variation in the wavenumber at which the peaks are located, the system uses an incrementally reducing window around the expected location (see *Fig. 2*). The smaller the difference between the wavenumber of the peak in the library spectrum and that in the experimental spectrum, the higher is the degree of probability assigned to the match. Once a match between the experimental spectrum and the reference has been confirmed, the spectrum of the identified component is subtracted from the total experimental spectrum, leaving a reduced number of peaks. The matching and subtraction are continued until no identifiable peaks remain in the spectrum.

In a comparison exercise,[15] the results of the FTIR analysis by Pawmi were compared with results obtained by gas chromatography–mass spectrometry (GC–MS). In its original form the expert system correctly identified 44 per cent of the components identified by GC–MS and correctly eliminated 98.9 per cent of the candidate compounds, that is, compounds known to the expert system. After modifying the Pairs routine the identification success rate improved to 74 per cent, but a slight deterioration (to 98.3 per cent) in elimination was also produced. Given the relative costs of FTIR and GC–MS analysis, coupled with the shorter analysis time for FTIR, the expert system approach provides a suitable analytical method for major component identification in the analysis of organic wastes.

Another expert system interprets spectral information from luminescence spectra.[16] The system examines the emission–excitation data produced by a fluorescence spectrophotometer in three ways to establish the identity of the components in a mixture.

- Identification of the characteristic features that are present in the

emission-excitation matrix (EEM). These features may be peaks, valleys or saddles, and may be recognised from a contour plot of the EEM, as illustrated in *Fig. 3*.

- Comparison of the normalised EEM of sample and target. Because this represents a very large amount of data the matching is first attempted by using a reduced data matrix. The EEM produced by the system is an array of 761 by 41 data points. For comparison this is reduced to a set of arrays of 80 by 40, 20 by 10, 10 by 10 and 5 by 5 data points. The EEM arrays of sample and standard are first compared at the low-resolution level, comparison of the higher resolution versions only being attempted once a match has been found at low resolution.
- Examination of the emission and excitation spectra that lie across the maximum intensity peak. These scans can contain a great deal of qualitative information, because the major features associated with a compound tend to lie on these lines.

To characterise a mixture, however, is more complex because spectral features associated with one compound may be swamped or shifted by signals from other components of the mixture. Despite this complication the performance of the system has been found to compare favourably with that of an expert for the identification of polynuclear aromatic hydrocarbons.[17] An extension to this system for the detection and identification of fluorogenic solutes in HPLC has also been proposed.[18]

The interpretation of X-ray spectral data has provided the stimulus for a number of expert systems. One such project[19] attempts to infer three dimensional structures for proteins from X-ray crystallography experiments, by using a number of independent knowledge based systems in a co-operative framework incrementally to propose, augment and verify or discard a working hypothesis of the structure under investigation.

The system developed by Amoco to analyse mineral samples by X-ray powder diffraction[20] was constructed as a series of production rules to extract spectral features, compare the extracted features with a library and, if the identity of the component was confirmed, subtract the relevant signal from the spectrum. The system can recognise 20 minerals typical of those found in sedimentary rocks, by using a strategy similar to that of Pawmi, *ie* an iterative recognition and subtraction of spectral profiles. This system was subsequently implemented in various forms as a test of commercial expert system shells.[21]

All of the systems described are involved in interpreting the results of experiments done under the control of an operator, who selects the appropriate conditions for each analysis. Systems are being developed to reproduce the skills required to select instrument conditions or even the form of analysis most appropriate for a given sample, *eg* there is an expert system available which offers guidance in the selection of methods for the analysis of pharmaceutical samples.[22] This system considers the factors affecting the solubility, spectral properties and electrochemical behaviour of the analyte to propose an appropriate analytical method. The capabilities of the system are currently limited, but have provided the basis of systems which can suggest conditions for spectroscopic analysis,[23] or select the detector for use in liquid chromatographic analysis.[24]

Similar guidance for the liquid chromatographer is offered by a number of systems currently under development. Some of these systems concentrate on providing information for the analysis of one particular class of compounds such as the steroids,[25] whereas others attempt to guide the chromatographer through the separation of any mixture of analytes

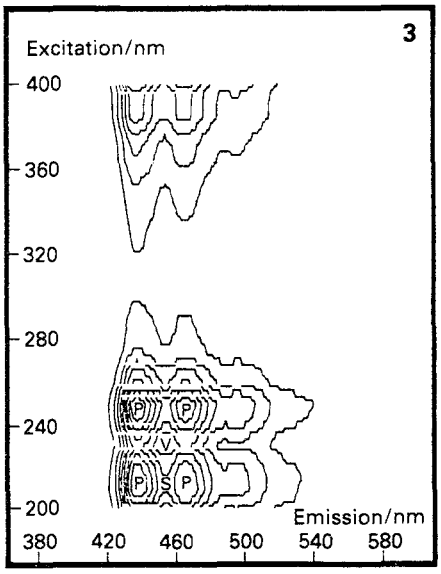


Fig. 3. Contour map of the excitation-emission matrix (EEM) of perylene.[17] The features marked are: peak P, valley V and saddle S.

for which chemical information is available. One such system[26] first selects between gas and liquid chromatography, and then searches a library of separation conditions to identify suitable conditions for the analysis. An alternative system[27] directs the optimisation of the mobile phase in reversed phase liquid chromatography (RPLC). This system requires no initial information about the sample to be separated, but does assume that an isocratic RPLC separation is feasible. After selecting the eluotropic strength of the mobile phase by a gradient elution experiment, the system then does three isoeluotropic chromatograms by using mobile phases of the same eluotropic strength but of differing selectivity. By using spectral information from a photodiode array detector to locate each solute in each of the chromatograms, a model is produced of the retention behaviour of all the solutes for all mobile phase compositions. This is then used to select the optimum mobile phase for the separation. Because of the limitations of the model the optimum is unlikely to be located at the conditions suggested by the initial model, but an iterative refinement allows the true optimum to be located. Additional reasoning facilities are provided to solve the problem when the spectral data are ambiguous owing to peak overlap or other external causes.

In addition to providing assistance in finding an optimised separation, expert systems are being developed to diagnose faults in chromatographic hardware and to recommend corrective action.[28] This would be similar to having an experienced technician on call to monitor continuously the performance of the pumps and fittings.

Another expert system—Exmat[29]—provides guidance on the types of analysis appropriate for a given problem and interprets the results of six sets of instrumental analysis. This system consists of a network of seven linked expert systems, each of which controls the decision making in one particular aspect of the problem, under the overall control of a commercial expert system shell which combines both numerical and symbolic processing elements. The seven expert systems within Exmat have the following specific roles.

1. Problem definition and analytical strategy. This relates the nature of the analytical problem to the amount and physical form of the sample and determines the techniques for the assay. An account is also taken of the current availability of the various instruments in the laboratory.

2. Instrument configuration and conditions. Once the instrument has been selected, this module determines the conditions under which it should be used. Where appropriate a suitable detector is suggested, *eg* flame ionisation or electron capture detectors for GC. Sample pretreatment may also be specified by this module.

3. Data generation. This controls the construction of result databases and a sample tracking mechanism, based on the requirements of the preceding modules. For example, data from FTIR would be added to an FTIR database, but the mode in which the instrument was used must be identified. For instance, was the FTIR used in direct spectral capture mode or as a detector for the GC?

4. Chemometrics and search algorithms. This includes a range of techniques for chromatographic and spectral analysis such as normalisation, deconvolution and pattern recognition, and in addition a number of search routines for infrared and MS libraries.

5. Correlation of results. The results from the analysis are combined and documented.

6. Interpretation. This is guided by the requirements and constraints of the analyst and may range from visual pattern matching to comprehensive multilibrary comparisons. A facility to include decisions has been incorporated to assess the reliability of various analyses.

7. Determination of analytical goals: this

module uses the interpreted results to guide the analyst toward the goal, as defined by module *1*.

Conclusions

The various expert systems described are a representative selection of the systems currently available or under development. The computer power required to run them ranges from a small desk-top computer to large mainframe machines. The techniques of artificial intelligence are still at a relatively early stage of development. It seems probable that as progress is made in this field, coupled with the continuing increase in available computer power, the tasks to which expert systems are directed will become increasingly complex, thereby adding a powerful tool to the armoury of analytical expertise.

References

1. F. Hayes-Roth, D. A. Waterman and D. B. Lenat, *Building expert systems*. Reading, MA: Addison-Wesley, 1983.
2. R. K. Lindsay *et al, Applications of artificial intelligence for organic chemistry*. New York: McGraw-Hill, 1980.
3. B. G. Buchanan *et al, J. Am. Chem. Soc.*, 1976, **98**, 6168.
4. H. Egli, D. H. Smith and C. Djerassi, *Helv. Chim. Acta*, 1982, **65**, 1898.
5. N. A. B. Gray *et al, Org. Magn. Reson.*, 1981, **15**, 375.
6. J. R. Quinlan, *Aust. Comput. J.*, 1980, **12**, 56.
7. R. Davis and J. King, *A survey of production systems* in *Machine intelligence 8*, E. Elcock and D. Michie (eds), p300. Chichester: Ellis Horwood, 1981.
8. M. H. van Emden, *Programming with logic* in *Machine intelligence 8*, E. Elcock and D. Michie (eds), p266. Chichester: Ellis Horwood, 1981.
9. W. van Melle, Proceedings of the 6th international joint conference on *Artificial intelligence*, 1979, p923.
10. E. Shortliffe, *Computer-based medical consultations: Mycin*. New York: Elsevier, 1976.
11. E. A. Feigenbaum, Proceedings of the 5th international joint conference on *Artificial intelligence*, 1977, 1014.
12. J. S. Bennet and R. S. Engelmore, Proceedings of the 6th international joint conference on *Artificial intelligence*, 1979, p47.
13. M. A. Puskar, S. P. Levine and S. R. Lowry, *Anal. Chem.*, 1986, **58**, 1156.
14. H. B. Woodruff and G. M. Smith, *Anal. Chem.*, 1980, **52**, 2321.
15. M. A. Puskar, S. P. Levine and S. R. Lowry, *Environ. Sci. Technol.*, 1987, **21**, 90.
16. K. T. Milne *et al, Anal. Proc.*, 1986, **23** 157.
17. K. T. Milne, *An investigation into IKBS techniques for analytical spectrofluorimetry*. PhD thesis, Heriot-Watt University: Edinburgh, 1986.
18. B. J. Clark *et al, J. Pharm. Pharmacol.*, 1985, **37** (suppl).
19. R. S. Engelmore and H. P. Nii, US NTIS, AD Rep., AD-A0388866, 1977, 39.
20. S. P. Ennis, *Anal. Chem.*, 1984, **56**, 1326A.
21. S. P. Ennis, Proceedings of the national conference on *Artificial Intelligence*, 1982, 319.
22. M. R. Detaevernier *et al, J. Pharm. Biomed. Anal.*, 1986, **4**, 297.
23. L. Buydens *et al, Chemometrics Intelligent Lab. Sys.*, 1986, **1**, 99.
24. G. Musch, M. De Smet and D. L. Massart, *J. Chromatogr.*, 1985, **348**, 97.
25. H. Gunsingham, B. Srinivasan and A. L. Anada, *Anal. Chim. Acta.*, 1986, **182**, 193.
26. R. Bach, J. Karnicky and S. Abbott, ACS symposium series 306, *Artificial intelligence applications in chemistry*, T. H. Pierce and B. A. Hohne (eds), p278. Washington DC: ACS, 1986.
27. T. P. Bridge *et al, Chromatographia*, in press.
28. *Focus, Anal. Chem.*, 1986, **58**, 1192A.
29. S. A. Liebman *et al*, ACS symposium series 306, *Artificial intelligence applications in chemistry*, T. H. Pierce and B. A. Hohne (eds), p365. Washington DC: ACS, 1986.

The authors

Dr John C. Berridge is section head and Paul Last is a senior research scientist in the analytical department of Pfizer Central Research, Sandwich, Kent CT13 9N. Professor M. H. Williams is head of the computer science department at Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ; Dr T. P. Bridge is a lecturer in the same department. Professor A. F. Fell is chairman of the school of studies in pharmaceutical chemistry, University of Bradford, West Yorkshire BD7 1DP; Dr Brian J. Clark is a lecturer in the same department. Dr Bob D. McDowall is a senior analyst in the department of drug analysis at Smith, Kline and French Research, The Frythe, Welwyn, Hertfordshire AL6 9AR.

75. T.P.BRIDGE, M.H.WILLIAMS, G.G.R.SEATON and A.F.FELL, An expert system for eluent optimization in liquid chromatography with photodiode array detection, Chromatographia, vol. 24, 691-695 (1987).

CHROMATOGRAPHIA

Vol. 24 (1987)

11th International Symposium on Column Liquid Chromatography

Amsterdam, 28 June-3 July 1987

An International Journal for Rapid Communication in Chromatography and Associated Techniques

REPRINT

Editors

H. Engelhardt
Saarbrücken
Federal Republic of Germany

L. S. Ettre
Norwalk, CT, USA

G. Schomburg
Mülheim
Federal Republic of Germany

R. Stock
Muscat
Sultanate of Oman

Acting Regional Editor

E. R. Adlard
Burton
United Kingdom

Publisher

F. Lube
Wiesbaden
Federal Republic of Germany

Friedr. Vieweg & Sohn
Pergamon Press



CHROMATOGRAPHIA

CHROMATOGRAPHIA

An International Journal
for Rapid Communication in Chromatography and Associated Techniques

Distributed internationally, Chromatographia provides a medium for the rapid and widespread communication of current information in the field of chromatography. This includes associated techniques and combined methods, as well as the application of computers and modern data systems.

Chromatographia publishes primarily reports of original research, as well as short communications, reviews and technical information from instrument manufacturers. In addition, news of meetings and reports on the activities of various organizations are announced. A special section of the journal, the so-called "Green Pages" introduces the reader to specific techniques, instrumentation and applications by offering reports and discussions in simplified form.

The editorial board of Chromatographia consists of internationally renowned scientists from Austria, P. R. China, Czechoslovakia, France, the German Federal Republic, Hungary, Israel, Italy, Japan, the Netherlands, Sweden, Switzerland, the United Kingdom, the United States of America and the U.S.S.R.

Chromatographia is published monthly by Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, P.O. Box 5829, D-6200 Wiesbaden 1, F.R.G. and is distributed in the United States by German Language Publications, Inc., 560 Sylvan Avenue, Englewood Cliffs, NJ 07632.

Editorial Advisory Board
E. Bayer, Tübingen, FRG
F. Bruner, Urbino, Italy
C. A. Cramers, Eindhoven, NL
D. R. Deans, Wilton, UK
P. Devaux, Romainville, France
F. Erni, Basle, Switzerland
R. W. Frei, Amsterdam, NL
F. Geiss, Ispra, Italy
G. Guiochon, Knoxville, TN, USA
E. Grushka, Jerusalem, Israel
J. F. K. Huber, Vienna, Austria
D. B. Jack, Birmingham, UK
K. Jinno, Toyohashi, Japan
B. Karger, Boston, MA, USA
M. L. Lee, Provo, UT, USA
P. C. Lu, Dalian, P. R. China
M. Martin, Palaiseau, France
M. V. Novotny, Bloomington, IN, USA
C. S. G. Phillips, Oxford, UK
L. Rohrschneider, Marl, FRG
K. I. Sakodynskii, Moscow, USSR
P. A. Sewell, Liverpool, UK
R. E. Sievers, Boulder, CO, USA
W. Simon, Zurich, Switzerland
C. F. Simpson, London, UK
L. Szepesy, Budapest, Hungary
D. Westerlund, Uppsala, Sweden
W. L. Yu, Lanchow, Kansu, P. R. China
A. Zlatkis, Houston, TX, USA

Honorary Editorial Board
C. Cremer, Innsbruck, Austria
G. Hesse, Erlangen, FRG
G. Machata, Vienna, Austria
R. L. Munier, Paris, France
H. M. McNair, Blacksburg, VA, USA
G. Schay, Budapest, Hungary
E. Smolková-Keulemansová, Prague, CSSR
K. Tesarik, Brno, CSSR
G. Widmark, Stockholm, Sweden
A. A. Zhukhovitskii, Moscow, USSR

Friedr. Vieweg & Sohn **Pergamon Press**

CHROMATOGRAPHIA

An Expert System for Eluent Optimisation in Liquid Chromatography with Photodiode Array Detection

T. P. Bridge*[1]/M. H. Williams[1]/G. G. R. Seaton[2]/A. F. Fell[2]

[1]Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2 HJ, UK

[2]Department of Pharmaceutical Chemistry, University of Bradford, Bradford BD7 1DP, UK

Key Words

Expert system
Reversed-phase liquid chromatography
Mobile phase optimisation
Peak homogeneity validation
Photodiode array detection

Summary

The problem of eluent optimisation in reversed-phase liquid chromatography is a complex diagnostic situation, amenable to an expert system approach. Such a system has been developed in microProlog, which uses a gradient elution experiment to determine the appropriate initial solvent strength for a given separation, followed by response-surface modelling using an iterative lattice method to determine the mobile phase composition for optimum resolution. Spectral information from a diode array detector is used to track the retention position of each component as the mobile phase composition is varied. Peak homogeneity is assessed by a number of independent modules, the output from which is utilised by the expert system to validate the model constructed by the optimisation procedure.

Introduction

The determination of the mobile phase composition that gives the best possible resolution of the components of a given sample within an acceptable analysis time, is a complex task requiring the adjustment of a number of interacting variables. The identity and concentration of the organic modifier(s), the pH of the mobile phase, the identity of the buffer used, if any, and the identity and concentration of any added components in the mobile phase are amongst the parameters available to the analyst. The adjustment of these parameters to achieve the desired separation is a time consuming operation, requiring a combination of methodological experimentation and knowledge of the probable behaviour of the analyte.

Automated optimisation strategies have been suggested before [1], but most implementations rely upon a search of the parameter space to locate the optimum conditions. The system described in this paper has the capacity to use information derived during the initial stages of the optimisation procedure to predict an optimum composition of the mobile phase for resolution of each separately identifiable solute in an unknown mixture. This prediction is iteratively refined until a true optimum mobile phase composition is established.

Optimisation Strategy

The strategy adopted for use in the expert system is to establish the initial solvent strength required for the separation by means of a water-methanol gradient elution experiment, and then to construct a model of the response surface over the solvent triangle bounded by isoelutropic water-modifier mixtures of equivalent eluting power to that of the initial mixture [2]. A flow-chart of this strategy is reproduced as Fig. 1. The modifiers employed for this stage are methanol, acetonitrile and tetrahydrofuran.

The model construction process is based on an iterative lattice approach [3], whereby a logarithmic relationship is assumed between mobile phase composition and the phase capacity ratio of each component. This allows a prediction to be made of the separation between all pairs of solutes at all points within the solvent triangle, and hence permit selection of the conditions predicted to give the optimum separation. The data provided by the separation performed using this predicted mobile phase composition allow the model to be refined and a new optimum mobile phase predicted. The cycle of prediction, experimentation and refinement continues iteratively until no further improvement in resolution is achieved.

0009-5893/87 0691-05 $03.00/0 © 1987 Friedr. Vieweg & Sohn Verlagsgesellschaft mbH

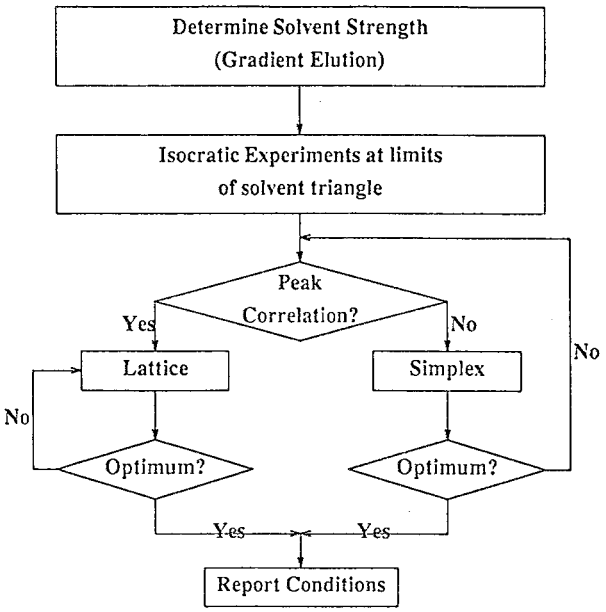


Fig. 1
The optimisation strategy employed by the expert system.

The iterative lattice approach to mobile phase optimisation relies upon the correct and unambiguous identification of each solute in each of the chromatograms. Despite the array of tests for peak homogeneity available to the system, as described below, the situation may still arise in which the unambiguous assignment of a retention time for each component may not be possible. For example the difference between the spectra for a pair of peaks may be so small as to be below the discrimination limit of the system. Ambiguity may also arise in cases of isoelution that are not detected by the peak homogeneity tests currently available to the system.

Under these conditions the system may call on a modified Simplex procedure [4] to enable the optimisation to continue. To utilise the Simplex method the overall quality of each chromatogram is described by a numerical value, calculated according to an appropriate response function [5]. The optimisation proceeds by discarding the mobile phase conditions giving the least favourable value for the response function. A new composition for the mobile phase is predicted by reflecting this discarded point about the line connecting the two remaining data points.

After each new experiment carried out under the control of the Simplex optimisation strategy, the component spectra are examined in an attempt to acquire sufficient spectral information for unambiguous assignation to be made. This then enables the iterative lattice method to continue the overall optimisation process.

Peak Homogeneity Assessment

The construction of the response surface model requires that the retention time of each solute be established for each of the mobile phases used. To assist in this, spectral information is acquired for each peak by means of a photodiode array detector [6]. This allows each eluting peak to be characterised by a spectral profile, which should enable it to be associated with a particular component in the mixture. The assignment of spectral profiles is staightforward for peaks containing only one component unless two or more components exhibit similar spectra. However the process is more complicated in cases of co-elution, when estimates of the true spectral profiles of the contributing components must be produced by a suitable deconvolution of the matrix of (A, λ, t) data. As such a deconvolution process may require the expenditure of a significant amount of computing time, the expert system is provided with a number of tools to assess the homogeneity of each detected peak. The use of deconvolution can therefore be restricted to cases where the chromatographic peak is found to be heterogeneous.

The tests for assessing peak homogeneity currently available to the system are:

(a) Comparison of the normalised spectral profiles recorded on the upslope and the downslope of the peak. The difference between the two curves is indicative of the extent of heterogeneity in the peak.

(b) Evaluation of the derivative total-absorbance chromatogram. For an homogeneous peak the first derivative with respect to time of the total-absorbance chromatogram would be expected to exhibit one zero-crossing, with two zero-crossings present in the second derivative chromatogram. An increase in the number of zero-crossings indicates the presence of two or more components in the peak.

(c) Determination of the apparent retention time as a function of detector wavelength. For an homogeneous peak the apparent retention time determined at any wavelength, will vary only due to noise in the signal. If two or more components are present, with different spectral profiles, then the apparent retention time of the peak will depend upon the detection wavelength [7].

(d) Examination of the absorbance-ratio chromatogram. The ratio of the absorbance detected at two wavelengths will be essentially invariant with time across a pure peak. The presence of a second co-eluting component with a different spectral profile to the first, will result in a variation in the value of the absorbance ratio across the chromatographic peak.

Since the accuracy of the above tests is affected to different degrees by a number of factors (e. g. changes in the peak profile, the signal-to-noise ratio of the detector and the degree of similarity between the component spectra) the interpretation of the information provided by the tests requires consideration of these factors, when they can be determined. To improve the reliability of the system for detecting co-eluting solutes, rules to direct the interpretation of these test results are being progressively developed for inclusion in the system. The modular nature of the expert system will facilitate the incorporation of additional homogeneity tests at a later stage.

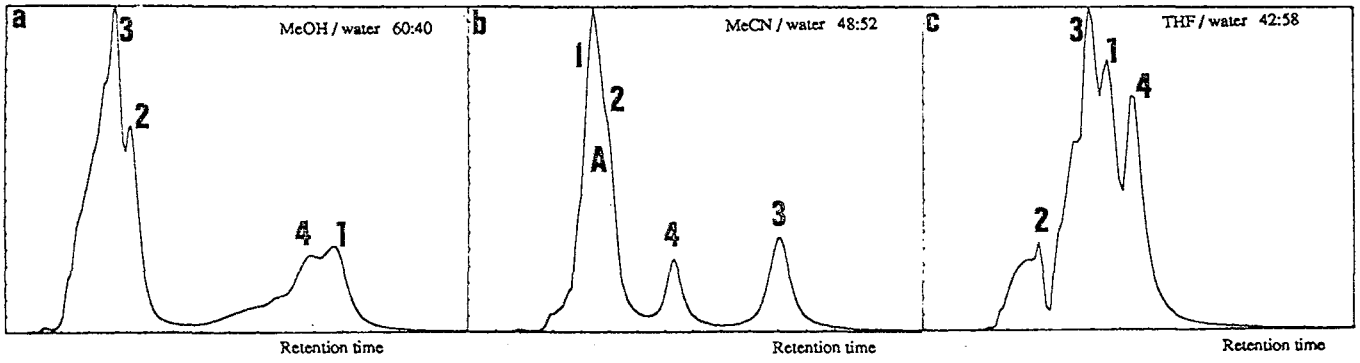


Fig. 2

Initial isoelutropic chromatograms.

(a) Mobile phase methanol-water 60:40 v/v
(b) Mobile phase acetonitrile-water 48:52 v/v
(c) Mobile phase tetrahydrofuran-water 42:58 v/v

Solutes: 1: 3-hydroxybenzaldehyde; 2: phenol; 3: diethyl phthalate; 4: n-propyl-p-hydroxybenzoate. Peak A: Peak for which homogeneity tests are illustrated in Fig. 5.

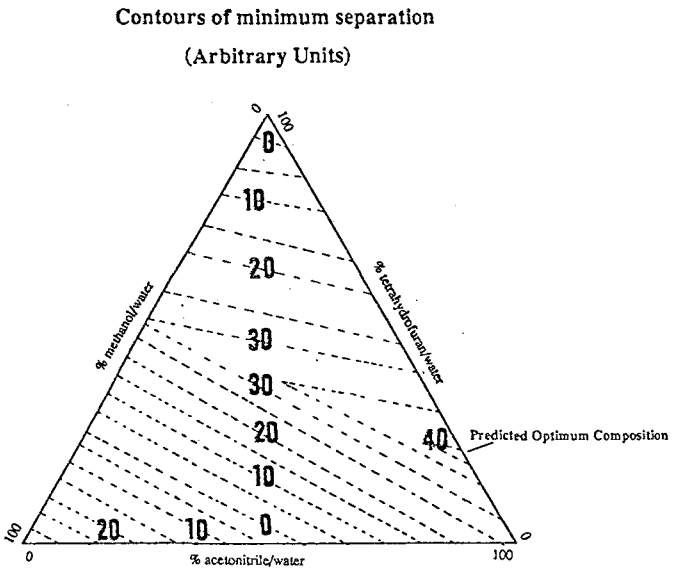


Fig. 3

Response surface predicted by iterative lattice modelling from the chromatograms in Fig. 2.

Implementation

The expert system consists of a number of modular routines, the internal operation of each module being independent of the rest of the system. The decision-making modules are written in microProlog [8] to which an extension has been provided [9] enabling it to call numerical processing modules written in other languages such as Pascal and Basic. The system is hosted on an IBM PC-XT microcomputer, with data from the photodiode array detector passed to the computer via an RS232C serial data link. The present system requires an operator to prepare the mobile phases, operate the pumps and perform the injections. It is intended to bring these tasks under direct control at a later stage in the development of the overall system.

Materials

HPLC-grade methanol, acetonitrile and tetrahydrofuran (Rathburn Chemicals, Walkerburn, UK) were used as received. All eluents were filtered through Millipore® 0.45 μm filters in all-glass apparatus and degassed under reduced pressure in an ultrasonic bath for 10 minutes. 3-hydroxybenzaldehyde, n-propyl-p-hydroxybenzoate, diethyl phthalate and phenol (BDH, Poole, UK) were used without further purification.

Equipment

Eluent delivery was achieved by the use of two PU4010 pumps under the control of a PU 4850 chromatography data station (Pye Unicam Ltd., Cambridge, UK). The 100 x 5 mm column was packed with 5-μm Hypersil ODS; solute detection was achieved using a Pye Unicam PU4021 diode array detector.

Optimisation of the Separation of a Four Component Mixture

The operation of the expert system has been demonstrated for the optimisation of a synthetic mixture. The test mixture contained 3-hydroxybenzaldehyde, n-propyl-p-hydroxybenzoate, diethyl phthalate and phenol dissolved in methanol at a concentration of 20 mg/ml.

The initial gradient elution experiment indicated that the required elutropic strength would be provided by a binary mixture of water and methanol (40:60 v/v). Equivalent combinations of water-acetonitrile (48:52 v/v) and water-tetrahydrofuran (58:42 v/v) were also prepared. The sample was injected onto the chromatograph using these three isoelutropic mobile phases; the chromatograms are presented as Fig. 2. After determining the homogeneity of the detected peaks, spectra were extracted for the four components from each of the chromatograms and a model of the response surface constructed. Here the resolution

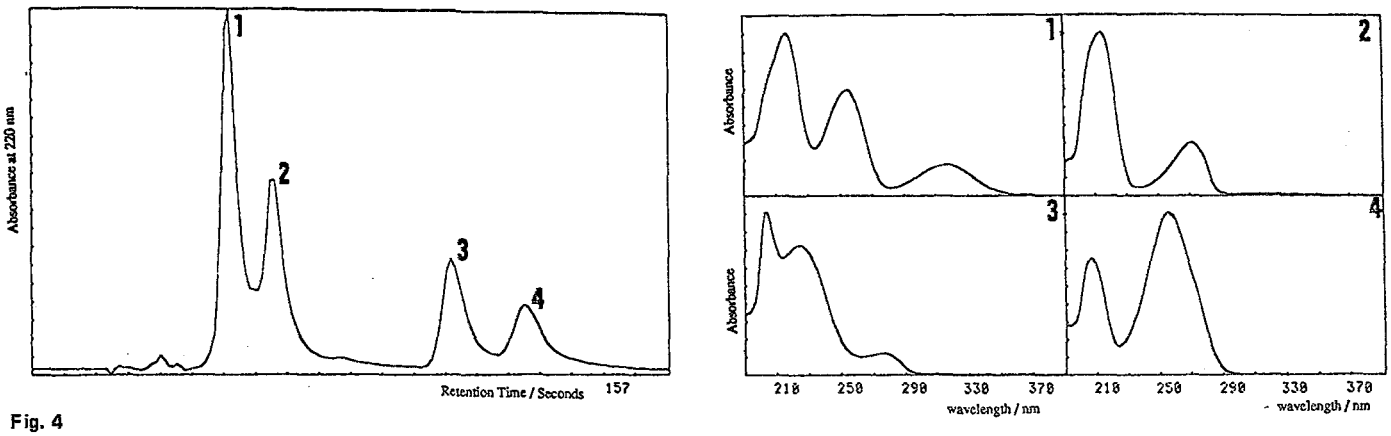


Fig. 4

Chromatogram produced under the optimum conditions predicted by the expert system. Mobile phase methanol-acetonitrile-tetrahydrofuran-water 24:5:21:50 v/v. Solute identities as for Fig. 2.

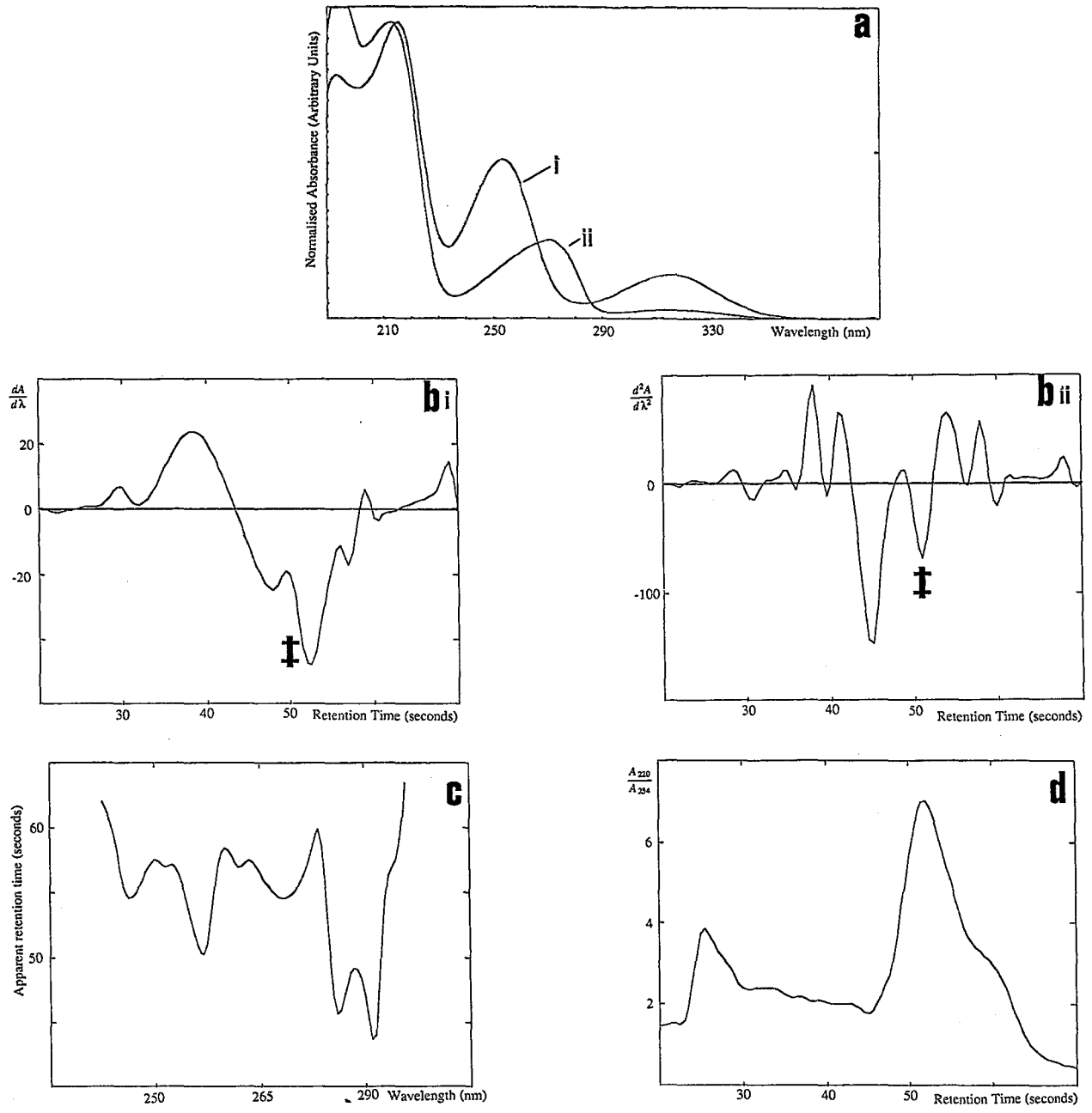


Fig. 5

Homogeneity tests applied to peak A in Fig. 2b.

(a) Comparison of spectral profiles. (i) Spectrum from upslope. (ii) Spectrum from downslope.

(b) First (i) and second (ii) derivative total absorbance chromatograms. Features marked ‡ are indicative of heterogeneity.

(c) Variation of apparent retention time with detector wavelength.

(d) Absorbance ratio chromatogram.

was calculated for each pair of solutes at 2% increments in each of the three solvents, within the limits bounded by the solvent triangle. Fig. 3 represents the contour lines of equal separation for the worst resolved pair of solutes at each eluent combination. The figure depicts the first stage of the search for the true optimum composition. The response surface produced at this stage may be visualised as a set of adjacent planes, tilted relative to one another. In addition to the predicted global optimum composition, with a separation of over 40 arbitrary units, the system predicts the existence of two local optima: one located at the apex representing 100% methanol-water, with a separation of 32 arbitrary units; the other is located at the apex for 100% THF-water and has a separation of under 5 arbitrary units. A chromatogram was then carried out under the conditions predicted by this model and the data produced subsequently included in the model to allow an improved optimum to be established. At this stage the solvent triangle was subdivided by a line (or lines) connecting the latest experimental point with each of the apexes of the triangle surrounding that point, thereby producing up to three new, smaller, triangles. A resolution model was produced for each of these subsets of the solvent triangle and a new optimum composition established. The process of continual refinement was continued until no further improvement in the resolution was indicated. The final optimum chromatogram is shown in Fig. 4.

Assessment of Peak Homogeneity

The operation of the homogeneity assessment tool described above is illustrated in Fig. 5, when applied to the composite peak A in Fig. 2b.

Fig. 5a demonstrates the difference in spectral profiles obtained from the leading and trailing edges of the peak. For an homogeneous peak these normalised spectral profiles should only differ by an amount related to the noise of the detector.

Fig. 5b shows the profiles of the first and second derivative chromatograms, with the presence of a second component indicated by the features marked ‡ and the number of zero crossing points in the second derivative profile.

Fig. 5c is a plot of the apparent retention time as a function of detector wavelength. The variation indicated in this plot again suggests that more than one component is present.

Fig. 5d ist the absorbance ratio chromatogram produced from the signals detected at 220 and 254 nm. The ratio for an homogeneous peak would be expected to be invariant with time.

Conclusions

The system presented here is still in the early stages of development, but it illustrates some of the potential of a knowledge based approach to the problem of eluent optimisation in reversed-phase liquid chromatography. A number of enhancements are proposed, including the development of a series of rules to allow more intelligent use of the homogeneity data, an independent module to assist in the selection of the column and initial solvent selection, and direct control of the chromatographic hardware.

Acknowledgements

The Authors acknowledge receipt of a Post-Doctoral Assistantship from the SERC for one of us (TPB), a studentship from the Royal Society of Chemistry Analytical Trust (GGRS) and the generous loan of equipment by Pye Unicam Ltd., Cambridge.

References

[1] *J. C. Berridge,* "Techniques for the Automated Optimisation of HPLC Separations", Wiley, Chichester, (1985).
[2] *P. J. Schoenmakers, H. A. H. Billiet, L. de Galan,* J. Chromatogr., **205**, 13 (1981).
[3] *A. C. J. H. Drouen,* "Computerised Optimisation and Solute Recognition in Liquid Chromatographic Separations", PhD Thesis, Technical University Delft, (1985).
[4] *J. C. Berridge,* Analyst, **109**, 291 (1984).
[5] *J. C. Berridge,* J. Chromatogr., **244**, 1 (1982).
[6] *A. F. Fell, H. P. Scott, R. Gill, A. C. Moffat,* Chromatographia, **16**, 69 (1982).
[7] *A. G. Wright, A. F. Fell, J. C. Berridge*, Chromatographia **24**, 533 (1987).
[8] *K. L. Clark, F. G. McCabe,* "micro-PROLOG: Programming in Logic", Prentice-Hall International, London, 1984.
[9] *K. T. Milne, M. H. Williams, B. J. Clark, A. F. Fell,* Anal. Proc., **23**, 157 (1986).

Received: July 31, 1987
Accepted: Aug 14, 1987
F

CHROMATOGRAPHIA

An International Journal
for Rapid Communication in Chromatography and Associated Techniques

Some important papers published during 1987

The Allyl Bonded Stationary Phase. Conversions to New Phases for Gas Chromatography
J. J. Pesek / P. Mahabadi / Shih-Jen Chan

Separation of Gonadal Steroid Epimers Using High-Performance Liquid Chromatography
T. G. Watson / D. J. Windmill / A. R. Bourne

Elution Behavior of Styrene Oligomer Fractions in Supercritical Fluid Chromatography
D. Leyendecker / F. P. Schmitz / E. Klesper

Derivatives for Separation of Amino Acid Enantiomers
S. Abdalla / E. Bayer / H. Frank

A Gas-Solid Chromatographic Study of the Adsorption of Hydrocarbons and Alcohols on Graphite
F. J. López-Garzón / I. Fernandez-Morales / M. Domingo-García

Chromatographic Amino Acid Analysis of Protein Hydrolysed in Polyacrylamide after Removal of Ammonia
E. E. Farmer / T. F. Uchytil / J. P. Helgeson / R. D. Durbin

Measurement of Thermodynamic Equilibria by Chromatography
I. Ignatiadis / M. F. Gonnord / C. Vidal-Madjar

Estimation of the Band Broadening Contribution of HPLC Equipment to Column Elution Profiles
H. A. Claessens / C. A. Cramers / M. A. J. Kuyken

Trace Analysis of Volatile Polar Organics by Direct Aqueous Gas Chromatography
B. S. Middleditch / Nak-June Sung / A. Zlatkis / G. Settembre

Optimization of Fat Soluble Vitamin Separation in Reversed-Phase Microcolumn Liquid Chromatography
W. T. Wahyuni / K. Jinno

Characterization of some Commercial Poly (Styrene-Divinylbenzene) Copolymers for Reserved-Phase HPLC
H. W. Stuurman / J. Köhler / S.-O. Jansson / A. Litzén

HPLC in Photokinetics: Determination of Reaction Mechanism and Photochemical Quantum Yields of (E)-1-Phenylpropene
G. Gauglitz / W. Schmid

A Strategy of Two-Dimensional, High-Performance Thin-Layer Chromatography, Applied to Local Anesthetics
B. De Spiegeleer / W. Van den Bossche / P. De Moerloose / D. Massart

New Polymer Coated Anion exchange HPLC-Phases: Immobilization of Poly(2-Hydroxy, 3N-Ethylene-diamino)-Butadiene on Silica and Alumina
G. Heinemann / J. Köhler / G. Schomburg

Polymer-Coated Cation-Exchange Stationary Phases on the Basis of Silica
P. Kolla / J. Köhler / G. Schomburg

Analysis of the Metal-Catalyzed Decomposition of γ- and δ-Haloesters by GC/FT-IR
D. E. Henry / S. L. Pentoney, Jr. / R. W. Kondrat / P. R. Griffiths

Use of Backflushing and Peak-Cutting Technique in Gas Chromatographic Trace Analysis
Yafeng Guan / Lairong Zhang / Xueyan Tang / Peichang Lu

Retention Prediction of o-Phthalaldehyde Amino Acid Derivatives in Reversed-Phase Liquid Chromatography
K. Jinno / E. Tanigawa

Trace Analysis of Sugars by HPLC and Post-Column Derivatization
H. Engelhardt / P. Ohs

Structure-Retention Correlations of Hydrocarbons in Gas-Liquid and Gas-Solid Chromatography. Cyclo-alkenes and Cycloalkadienes
W. Engewald / U. Billing / T. Welsch / G. Haufe

On-Column Catalysis of Hydratization Equilibria During Liquid Chromatographic Analysis of Acetaldehyde and Formaldehyde
P. O. Edlund

Chitosan-Coated Silica Gel as a New Support in High-Performance Liquid Chromatography
V. Carunchio / A. M. Girelli / A. Messina / M. Sinibaldi

A Comparative Study of Large Volume Injection Techniques for Microbore Columns in HPLC
H. A. Claessens / M. A. J. Kuyken

Friedr. Vieweg & Sohn **Pergamon Press**

CHROMATOGRAPHIA

76. T.P.BRIDGE, M.H.WILLIAMS and A.F.FELL, Expert systems design in high performance liquid chromatography, Analytical Proceedings, vol. 25, 43-44 (1988).

Research Topics in Chromatography

The following are summaries of three of the papers presented at a Meeting of the Chromatography and Electrophoresis Group, held on June 10th, 1987, at the University of Technology, Loughborough.

Expert Systems Design in High-performance Liquid Chromatography

T. P. Bridge and M. H. Williams
Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ

and A. F. Fell
Pharmaceutical Analysis Research Group, School of Pharmaceutical Chemistry, University of Bradford, Bradford BD7 1DP

The traditional role of the digital computer has been to perform numerical calculations, usually quickly and repetitively, and consequently its application to problem solving has been largely limited to those areas in which the problem could be expressed in numerical terms. The introduction of intelligent knowledge-based systems has brought computing to the threshold of a significant change in the type of problem which can be successfully addressed, as knowledge-based systems have the capacity explicitly to manipulate facts and information without the need for this information to be expressed in numerical form. The use of such knowledge-based systems to assist in problem solving in analytical chemistry is still restricted to a few special instances,[1] but an increase in the range of applications is likely to occur in the near future.

The most common form of knowledge-based system is the expert system, which contains encoded expert knowledge and is capable of exploiting this knowledge for problem solving purposes. An expert system contains a number of essential features: (a) a knowledge base containing modules of organised knowledge relevant to the problem domain; (b) an inference mechanism which uses the information contained in the knowledge base to supply advice on the problem domain; (c) an explanation facility which allows the user to investigate the chain of inference used by the expert system; and (d) an interface to the user and in some applications to one or more external data sources. The relationships between these modules are illustrated in Fig. 1.

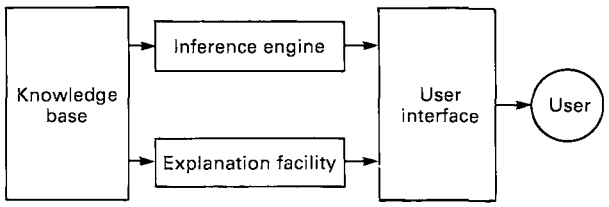


Fig. 1. Outline structure of expert system

The information stored in the knowledge base may be derived from a large number of sources, but these sources generally fall into two classes, public theoretical knowledge and private experiential knowledge. The public knowledge of a problem domain is that information which is available from books or journal articles. The private knowledge is that accumulated by a human expert and is a combination of the basic theoretical knowledge and a number of empirically derived facts. This experiential knowledge is often difficult to translate into a written form, but is essential if a full understanding of the problem is to be developed.

In a conventional program the calculating process is usually independent of the numerical data on which it acts. In an expert system the inference mechanism, explanation facility and user interface together form the program, which operates on the knowledge base as data to solve a particular problem. If the knowledge base is removed from an expert system the remaining units may be referred to as an expert system shell. Provided that any new information is encoded in an appropriate format, a shell derived from an expert system developed for one purpose may be used to tackle problems in an unrelated problem domain. This is so with the shell EMYCIN,[2] which is the medical expert system MYCIN[3] with the medical knowledge base removed. Among the uses to which this shell has been put is the system known as SACON,[4] which provides guidance for civil engineers in the use of a complicated structural analysis program.

One drawback of using an existing shell is that a restriction is placed on the manner in which the data may be manipulated. Whilst this may not be a problem in some instances, and may be reduced in others by selection of a suitable shell for the application, it is sometimes the case that no suitable shell is available. This is the situation faced by the chromatographer who wishes to develop a system to handle data from a chromatograph, as this usually involves combining a numerical processing facility with the logical inference system. In such instances a bespoke expert system can be created, using a suitable computer language or combination of languages.

Development of an Expert System for Optimisation of High-performance Liquid Chromatography

The system currently under development is intended to have the capability of establishing the composition of the mobile phase required to achieve optimum selectivity in reversed-phase liquid chromatography, with minimum intervention by the operator, and with as few intermediate chromatograms as possible. To achieve this goal will require a system with a high level of knowledge about the chromatographic process, the ability to interpret chromatographic data and the facility to exercise control over the chromatograph.

To allow full control over the inference process to remain with the programmer and to allow the numerical processing and communications required to control the instrumentation, the expert system has been written in MICROPROLOG[5] to which an extension has been added, allowing execution of compiled

BASIC or PASCAL programs from within the PROLOG environment.[6] The system runs on an IBM PC-XT or on an Advance 86B, an IBM-compatible computer, connected to a photodiode array detector. The optimisation procedure adopted for use by the expert system is illustrated in Fig. 2.

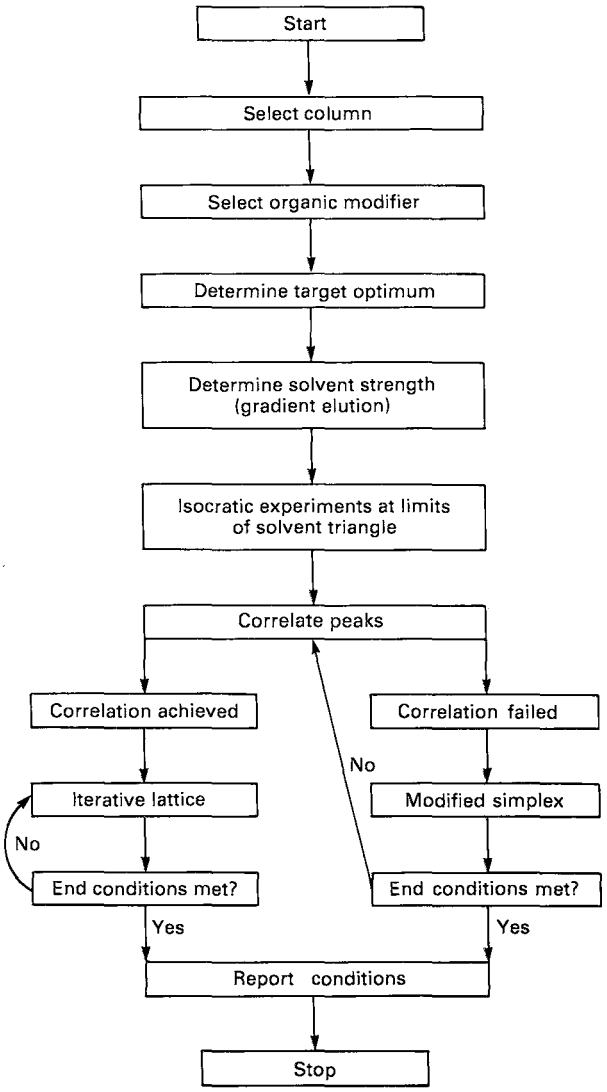


Fig. 2. Optimisation strategy

At this stage, only the interpretive and predictive elements have been implemented and consequently an operator is required to control the operation of the pumps and injector. Automatic control remains an objective for later implementation. Column choice is also a manual task at this time, the preferred stationary phase being ODS. It is intended to develop a rule-based module to assist in column selection at a later date. The expert system determines the solvent strength required for a given separation by performing a water - methanol gradient elution experiment. The user is asked for the desired over-all analysis time and the solvent strength required to give this retention time is calculated accordingly by the system.

The adjustment of selectivity is achieved by adjustment of the relative concentrations of the organic modifiers under the control of an iterative lattice optimisation.[7]

The iterative lattice works by determining the retention time of each component in the mixture under three different mobile phases, hence constructing a model of the variation of resolution as a function of mobile phase composition. The model is updated as the optimisation proceeds until a suitable approximation to reality is produced and the optimum mobile phase composition determined.

The identity of each peak in the chromatograms is established by comparing the spectral data produced by successive experiments. This requires that instances of co-elution be detected so that the spectra of the constituent components may be extracted from the matrix of (A, λ, t) data associated with each peak. Each detected peak is assessed for homogeneity by the following four methods: (a) the normalised profiles of spectra extracted from the upslope and downslope of the peak are compared, and a significant variation indicates a heterogeneous peak; (b) the profile of the time derivative total absorbance chromatogram is examined; (c) the profile of the absorbance ratio chromatogram is characterised and deviation from a square-wave form indicates heterogeneity; (d) the apparent retention time is determined as a function of wavelength. For a homogeneous peak this is essentially invariant, so variation again indicates heterogeneity.

As the sensitivity of these tests depends on the separation of the components in a compound peak, the signal level, peak width and other factors, interpreting the information supplied by these tests is a complex task. Rules are currently being developed to assist in the interpretation of the test results.

Despite the above tests a situation can arise in which co-elution can occur undetected. In this instance construction of the resolution model may be impossible owing to a lack of data. Should this occur then the expert system uses a modified simplex optimisation[7] to continue the optimisation process.

Future Developments

The system described above is the basis from which a more comprehensive system for method development in high-performance liquid chromatography will be developed. It is intended to give the system access to a range of statistical techniques to enhance the reliability of the resolution models further. An intelligent front-end module to assist in the selection of the column and organic modifiers is also planned. In the longer term this will be enhanced by the addition of a library of previous separations, which may be used as a knowledge base for the column selection module.

References

1. Bridge, T. P., Williams, M. H., and Fell, A. F., *Chem. Br.*, in preparation.
2. van Melle, W., "Proceedings of the 6th International Joint Conference on Artificial Intelligence," International Joint Conference on Artificial Intelligence Inc., 1979, p. 923.
3. Shortliffe, E., "MYCIN: Computer-based Medical Consultations," Elsevier, New York, 1976.
4. Bennett, J. S., and Englemore, R. S., "Proceedings of the 6th International Joint Conference on Artificial Intelligence," International Joint Conference on Artificial Intelligence Inc., 1979, p. 47.
5. Clark, K. L., and McCabe, F. G., "micro-PROLOG: Programming in Logic," Prentice-Hall International, London, 1984.
6. Milne, K. L., Williams, M. H., Clark, B. J., and Fell, A. F., *Anal. Proc.*, 1986, **23**, 157.
7. Berridge, J. C., "Techniques for the Automated Optimisation of HPLC Separations," Wiley, Chichester, 1986.

77. A.F.FELL, T.P.BRIDGE and M.H.WILLIAMS, Design and application of an expert system for eluent optimization in reversed-phase high performance liquid chromatography, J. Pharm. Biomed. Anal., vol. 6, 555-564 (1988).

Journal of Pharmaceutical & Biomedical Analysis
Vol. 6, Nos 6–8, pp. 555–564, 1988
Printed in Great Britain

0731–7085/88 $3.00 + 0.00
© 1988 Pergamon Press plc

Design and application of an expert system for mobile phase optimisation in reversed-phase liquid chromatography*

ANTHONY F. FELL,†§ TIM P. BRIDGE‡ and M. HOWARD WILLIAMS‡

† *Department of Pharmaceutical Chemistry, University of Bradford, Bradford, BD7 1DP, W. Yorkshire, UK*
‡ *Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ, Scotland, UK*

Abstract: The selection of the optimum composition for the mobile phase in reversed-phase high-performance liquid chromatography (HPLC) is a complex task; conventional approaches require the expenditure of significant amounts of time by the analyst, particularly for complex mixtures of solutes of biological origin. Some of the existing strategies for the automated optimisation of mobile phase composition (e.g. Simplex), may fail if the elution order of the components changes; or they may require that standards be chromatographed in order to establish the retention behaviour of each component in a mixture (e.g. resolution mapping). These problems may be overcome if the retention behaviour of each individual solute can be established from the chromatogram of the mixture. In this regard, components can be tracked by exploiting the spectral information generated by a rapid scanning photodiode array detector. Unfortunately this information is often insufficiently detailed to allow an unambiguous model of retention behaviour to be constructed. The system developed by the Authors uses these spectral data as a basis for constructing one or more hypothetical retention models, each of which is refined or rejected as further information is obtained during the progress of the experiment. To improve the reliability of the retention models proposed by the system, the spectral data are utilised in a number of tests designed to assess the purity of each chromatographic peak. The information so generated may be used in conjunction with any previously acquired spectral data both to select an appropriate method for extracting spectra for each component from the matrix of (A, λ, t) data and to establish reliability parameters for the resultant spectra. The development and philosophy of the expert system developed for eluent optimisation in reversed-phase HPLC is discussed.

Keywords: *Liquid chromatography; photodiode array detection; expert system; eluent optimisation; peak homogeneity validation.*

* Presented at the "International Symposium on Pharmaceutical and Biomedical Analysis", September 1987, Barcelona, Spain.

§ To whom correspondence should be addressed.

Introduction

The problem of devising a suitable mobile phase for a separation by reversed-phase liquid chromatography, so as to produce the required separation of the solutes in an acceptable analysis time, is one that can require considerable expenditure of time and resources before a solution is identified. The conventional approach to such an optimisation requires the attention of a skilled chromatographer who has the expertise to select the constituents of the mobile phase from a combination of his knowledge of the analytes, the results of some initial experimentation and his experience of other similar optimisation problems.

To reduce the amount of time spent on the problem by the chromatographer a number of automated optimisation strategies have been developed [1, 2]. Unfortunately all of these techniques suffer from limitations in their applicability. They may require a large number of experiments to define the chromatographic response surface, as for instance the search strategy used by the Perkin–Elmer Solvent Optimisation System, (PESOS) or they may require that pure standards be available for chromatographing under a lesser number of conditions to produce a map of the response surface, as for instance in the technique of overlapping resolution mapping [3].

A system which avoids these drawbacks is the technique of Simplex optimisation [4] which assesses the chromatograms by means of a suitable response function [1], and predicts the optimum conditions by a gradient sensing search of the parameter space. However, Simplex optimisation may fail in cases in which the elution order of the components changes during the optimisation process.

What is required is a method of reproducing the skills of the expert analyst in an automated system. Such an approach can be provided by the use of an intelligent knowledge based or expert system [5]. Such a system is a computer program, or suite of programs, which encapsulates expertise derived from a human skill in such a form as to allow the information to be used to solve problems in a specific problem domain. A number of such systems have been created to address problems in the field of analytical chemistry [6].

An expert system for optimisation

The use of an "intelligent" expert system is the basis of the optimisation system currently under development by the Authors [7, 8]. This system attempts to determine the optimum composition of the mobile phase by means of an iterative regression modeling technique [9], with a Simplex optimisation routine [4] available as an alternative method for use in cases in which the iterative regression technique is not applicable. Some of the limitations of the iterative regression method will be dicussed later in this paper. The iterative regression and simplex methods form part of the overall framework of the optimisation system, as illustrated in Fig. 1. This system is organised on modular lines to facilitate the inclusion of novel or improved units at any stage.

The starting point of the optimisation process is the selection of the column and choice of the organic modifiers to be investigated. In the current implementation these choices are made by the operator. However, rules can be continuously formulated and incorporated to allow the expert system to assist in these crucial stages.

The first experimental data required by the system are the results of a linear water–methanol gradient elution chromatogram. This is used to predict the eluting

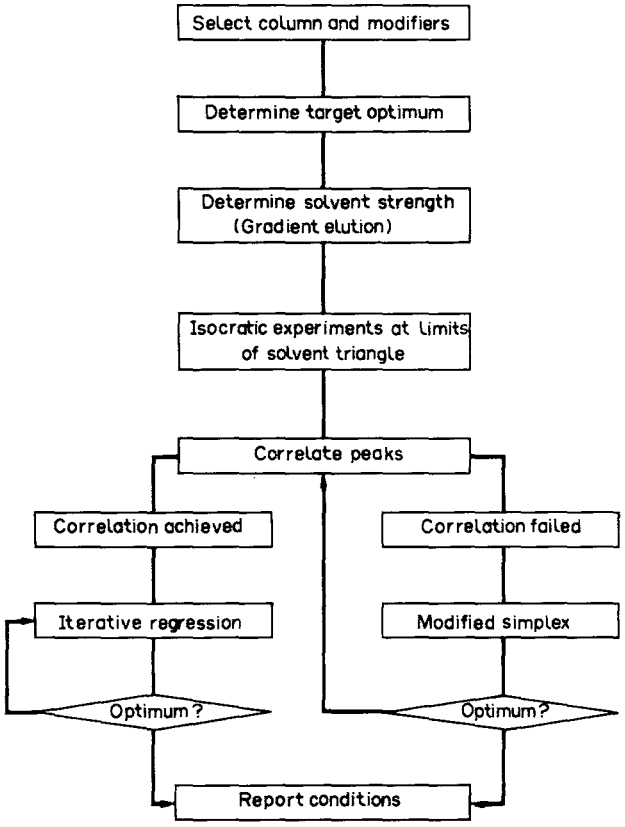


Figure 1
Outline structure of the expert system for HPLC optimisation.

power required to complete the analysis within the desired overall time [10], as indicated by the analyst. Once the eluting power has been determined, three isoeluotropic mobile phases are prepared using the modifiers selected during the initial stage. In the current system, these modifiers are normally restricted to methanol, acetonitrile and tetrahydrofuran. The compositions required to give approximately equal elution times for each mobile phase are determined by means of the transfer rules proposed by Schoenmakers [11].

The results of the three isocratic chromatograms are used as the starting point of the selectivity optimisation process, using either the iterative regression or the Simplex method, depending on the information which the system manages to extract from the chromatograms. As is explained below, the iterative regression technique requires that the retention time of each component be determined under each of the three sets of experimental conditions. If this information is not available due to overlap of the peaks or some other external factor, then the Simplex method is used. In all other cases the control of the optimisation is passed to the iterative regression.

Iterative regression optimisation

The iterative regression optimisation technique works by assuming that there is a linear relationship between the composition of the mobile phase and the logarithm of the

capacity factor of any given solute. If there is a variation in the selectivity between the mobile phases, then the relationship will be dependent upon the identity of the solute, that is, if the capacity factor k and the mobile phase composition Φ are related by the equation: $\ln(k) = m\,\Phi + c$, then the values of the constants m and c are dependent on the solute and the chromatographic conditions. This stage is illustrated in Fig. 2, a plot of log capacity factor versus mobile phase composition for four solutes.

The next stage in the construction of a model of the retention behaviour is to determine the separation of each pair of solutes with all possible mobile phases. This is

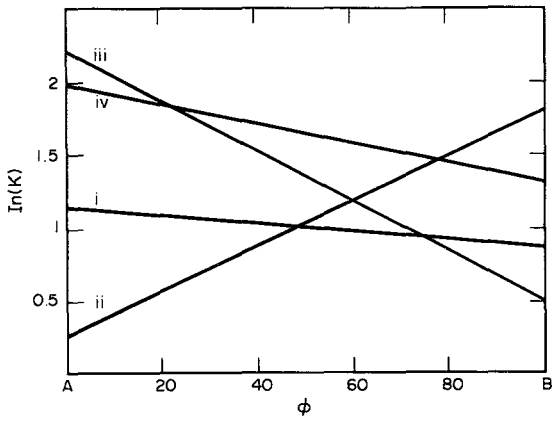


Figure 2
A plot of log capacity factor versus composition for a four component mixture. The solutes represented are: (i) phenol; (ii) diethylphthalate; (iii) *p*-hydroxybenzaldehyde; (iv) *n*-propyl-*p*-hydroxybenzoate. The mobile phase components are: (A) MeOH–water (60:40 v/v); (B) MeCN–water (48:52 v/v).

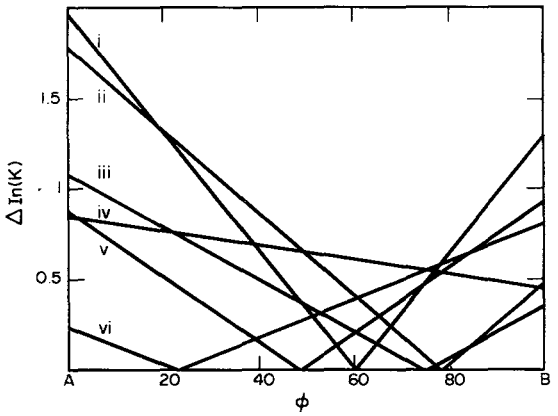


Figure 3
A plot of delta ln k v mobile phase composition derived for each pair of solutes from the data in Fig. 2. The indicated optimum lies at 100% B. The region of achievable resolution at each value of ϕ is bounded by the axes and the lowest line of the plot. A value of 0 for $\Delta\ln(k)$ indicates that the two solutes coelute in that mobile phase. The solute pairs associated with each line are: (i) diethylphthalate and *p*-hydroxybenzaldehyde; (ii) diethylphthalate and *n*-propyl-*p*-hydroxybenzoate; (iii) phenol and *p*-hydroxybenzaldehyde; (iv) phenol and *n*-propyl-*p*-hydroxybenzoate; (v) phenol and diethylphthalate; (vi) *p*-hydroxybenzaldehyde and *n*-propyl-*p*-hydroxybenzoate. The mobile phases are as in Fig. 2.

achieved by considering the separation between the curves of the $\ln(k)$ plot. Figure 3 is a plot of $\Delta\ln(k)$ versus Φ, derived from Fig. 2.

The indicated optimum composition is that at which the separation of the *worst resolved pair* of solutes is maximised. In Fig. 3 this composition is indicated to be 100% component B. An experiment is next carried out using the mobile phase indicated by this plot and the resultant chromatogram examined. If, as is likely during the early stages of the optimisation, the resolution achieved does not match the prediction, then the actual retention behaviour is used to produce a refined model of the response surface and a new mobile phase composition selected. The cycle of experimentation and refinement is repeated until the predicted and actual results agree, at which time the best separation available using the column and modifiers selected will have been obtained.

The example illustrated in Fig. 2 is for two organic modifiers only. The current system considers the use of three modifiers in a direct extension of the modeling process which produces planes of response rather than the lines of Fig. 3. A contour plot of such a response surface is given in Fig. 4. In principle the technique may be extended to consider other factors such as pH or the concentration of ion-pairing reagent present.

Selection of optimisation strategy

As discussed above, the expert system is provided with a choice of optimisation strategies, which may be deployed during the search for the optimum mobile phase composition. The strategy to be deployed at any stage is selected by the expert system on the basis of the spectral information available.

The preferred optimisation strategy is the iterative regression technique, since this requires fewer experimental separations than is the case for a Simplex search. It does, however, rely on the correct assignment of spectral profiles to individual components in the mixture. This assignment of spectra to components takes place in the module labelled "Correlate Peaks" in the diagrammatic representation of the expert system, Fig. 1. This module first examines each peak for homogeneity and then extracts the appropriate spectral profile(s). For a pure peak this is achieved by simply recording the spectra from the peak maximum. For a composite peak, however, the spectral information must be

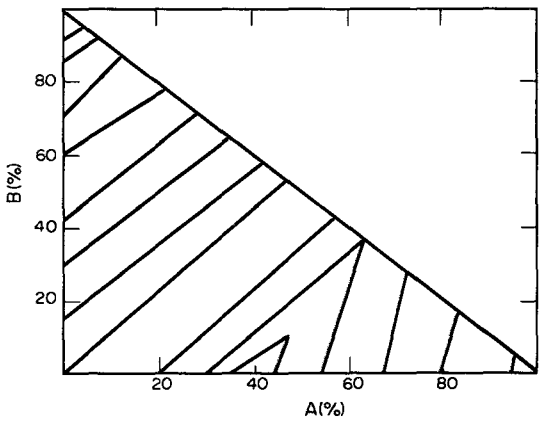


Figure 4
Contour plot of minimum resolution for a four component mixture, chromatographed in three different mobile phases. Resolution contours are based on a scale of arbitrary units. Mobile phase components are: (A) MeOH–water (60:40) v/v); (B) MeCN–water (48:52 v/v); and (C) THF–water (42:58 v/v).

deconvoluted from the total data set. To this end the system employs iterative target transformation factor analysis (ITTFA) [12] to determine the probable number of components contributing to a peak, and to extract their spectral profiles.

The spectra extracted from one chromatogram are next matched with those from any previous chromatograms, to enable the changes in retention time for each component to be assessed. If this correlation procedure succeeds, then the optimisation is directed by the iterative regression routine. If, on the other hand, the correlation procedure fails, then the expert system will select a modified Simplex strategy to continue the optimisation. The system will repeat the correlation step after each new experiment under the Simplex regime, and if sufficient additional data have been acquired to allow a successful correlation to be achieved, the optimisation is redirected to the control of the iterative regression routine.

Determination of retention behaviour

Whilst the technique of iterative regression optimisation is capable of locating the optimum conditions for a given separation after as few as five or six experiments, its success depends entirely on the correct determination of the retention time for each component in each experiment. If pure standards of each solute are available, then this information may be obtained by chromatographing each standard under each condition. This has two significant disadvantages. Firstly, such an approach requires a large number of individual chromatograms and secondly, the constituents of the mixture must be known in advance in order to allow selection of the relevant standards, which may not be available, particularly in the case of samples of biomedical origin.

To avoid this problem the system uses the ultraviolet absorption spectra of the eluting peaks as a label for tracking components between chromatograms. These spectra are obtained on-line by means of a photodiode array detector [13]. Whilst the tracking of components by their UV spectra is straightforward for compounds with significantly different spectral profiles which elute in isolation, the problem assumes significance if two or more components have similar spectra or where co-elution occurs. At this stage there is no easy answer for the first problem case, that is of indistinguishable spectra, but mathematical tools are available to assist in the extraction of spectra from compound peaks [14]. The development of rules to select an appropriate method for deconvoluting the spectral information contained in the spectrochromatogram of an heterogeneous peak will allow the system to operate with a greater degree of independence than is currently the case.

The matrix-based deconvolution methods, whilst offering a significant advance in the extent of usable information which can be extracted from a chromatogram, exact a penalty in terms of the computer power required to run them. To reduce the time spent on analysing the data from each chromatogram the system is provided with a number of tools for the rapid assessment of peak homogeneity. The tests currently available to the system are: spectral difference; first derivative chromatogram; second derivative chromatogram; absorbance ratio (time domain); variation of apparent retention time with detector wavelength; absorbance ratio (wavelength domain); ratio of derivative chromatograms; multiple absorbance ratio.

The intention underlying all these approaches is to give a quick assessment of the homogeneity of the target peak as a filter before any heavy-weight multivariate statistical tools are employed, so as to reduce the total amount of computer time required in examining each chromatogram.

In all cases the tests are applied to a peak or peak group which has been extracted from the total spectrochromatogram. The operation of each of these tests is discussed below.

Spectral difference. This compares the profiles of spectra taken from the leading and trailing edges of the peak. The routine extracts spectra from the data matrix at times 1/3 and 2/3 of the apparent peak width and normalises them at the wavelength of maximum absorption in the first spectrum. The square root of the sum of squares (RSS) difference between the two profiles is calculated and this value is reported. For a pure peak this difference will be due only to noise in the detector, but for an heterogeneous peak containing two or more components with dissimilar spectral profiles a larger difference will be determined.

First derivative chromatogram. The derivative of a chromatographic profile may be expected to be more sensitive than the original curve to the existence of a minor component [15]. To make use of this property, the total absorbance chromatogram is differentiated with respect to time by a simple difference mechanism:

$$\frac{dA}{dt}(i) = (A_{(i+1)} - A_{(i-1)})/2.$$

This method of differentiation was selected to reduce the loss of data inherent in any such averaging differentiation, whilst retaining the advantage of a high speed of operation. The number of zero crossings in the derivative profile is determined. A facility is included to allow crossings produced by noise at low signal amplitudes to be discounted. For a pure peak, one zero crossing is expected, whereas an impure peak would be expected to give rise to a profile exhibiting an increased number of zero crossings.

Second derivative chromatogram. The first derivative chromatogram is further differentiated by the same method, and the number of zero crossings again determined. Two such crossings are expected for an homogeneous peak; an increase in the number of zero crossings is an indicator of heterogeneity. Both derivative tests check to ensure that the available data are larger than the differentiation band width. If differentiation is not possible, due to a lack of available data, i.e. the peak width is less than twice the width of the differentiation window, then a value of -1 zero crossings is returned as a warning "flag" to the system.

Absorbance ratio (time domain). The ratio of absorbances measured at two wavelengths should be invariant with time for an homogeneous peak [16]. This test locates the time at which maximum absorbance occurs (t_{max}) and identifies the wavelength of maximum absorbance (λ_{max}) and the wavelength which gives half this absorbance value at this time. The ratio of the signals at these two wavelengths is then calculated as a function of time across the peak. The RSS deviation from the value of the ratio at t_{max} is determined and reported. Ideally this should be zero, or for a real system, not exceeding the variation in signal due to detector noise.

Variation of apparent retention time with detector wavelength. For a pure peak the measured retention time will be independent of the wavelength used to monitor the

elution. For a composite peak, however, the apparent retention time may vary according to the detector wavelength employed [17]. The extent of this variation is governed by the difference in elution times of the contributing components, the difference, or similarity, of the individual spectral profiles and the relative amounts of the components. The value of t_{max} is determined for each detector channel and the difference between the largest and smallest values reported. A minimum value of absorbance must also be selected, to avoid reporting noise signals for wavelengths at which the solute does not absorb. For a pure peak any variation would be due solely to noise in the detector.

Absorbance ratio (wavelength domain). Just as the ratio of two chromatograms recorded at different wavelengths should be invariant with time, so the ratio of two spectra recorded at different times should be invariant with wavelenth. If the spectra are normalised before the ratio is determined, then the ratio should, of course, be unity. As in the other tests, a variation from this value is indicative of heterogeneity.

Ratio of first derivative chromatograms. The idea of absorbance ratio evaluation may be extended into the derivative domain. This test normalises the chromatograms on each channel with respect to λ_{max} at t_{max} and then calculates the time derivative for each chromatogram. The ratio of these chromatograms to that at λ_{max} (which should be unity) is then calculated and the RSS deviation computed and reported.

Multiple absorbance ratio. This routine is identical to the derivative ratio routine, except that the ratio is calculated on the original, undifferentiated, data. Once more a deviation from unity is a measure of the degree of heterogeneity in the peak under investigation.

The sensitivity of each of the above tests will depend on a number of factors, such as the signal-to-noise ratio, the relative amplitudes of the contributing solutes and the degree of similarity between the component spectra. Since these factors cannot be readily determined for an experimental chromatogram, rules are currently being developed to allow the interpretation of the results of the tests, which may be expected to disagree with each other, in the light of the known signal parameters (peak amplitude, peak width, known detector noise level etc.).

Experimental

The solutes employed, 3-hydroxybenzaldehyde, *n*-propyl-*p*-hydroxybenzoate, diethyl phthalate and phenol (BDH, Poole, UK) were used without further purification. HPLC-grade methanol, acetonitrile and tetrahydrofuran (Rathburn Chemicals, Walkerburn, UK) were used as received. All eluents were filtered through Millipore® 0.45 μm filters in all-glass apparatus and degassed under reduced pressure in an ultrasonic bath for 10 min.

Eluent delivery was achieved by the use of two PU4010 pumps under the control of a PU4850 chromatography data station (Pye Unicam Ltd, Cambridge, UK). The 100 × 5 mm column was packed with 5-μm Hypersil ODS. Solute detection was achieved using a Pye-Unicam PU4021 diode array detector.

The data used to illustrate the process of iterative regression optimisation were taken from an optimisation of a four component mixture, containing the solutes listed above. The final chromatogram produced by the system is illustrated in Fig. 5. The lack of

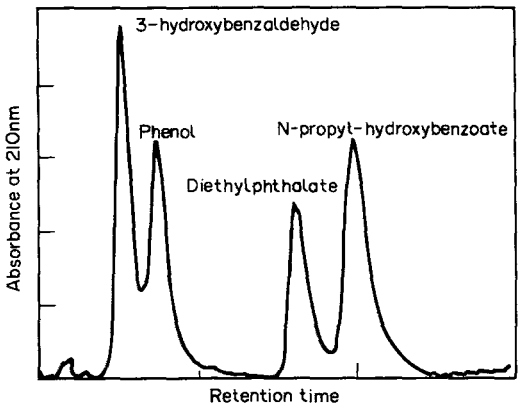


Figure 5
Chromatogram of a four-component mixture using the mobile phase composition predicted by the expert system.

complete baseline-resolution is probably a consequence of the operator-specified analysis time. A longer analysis time would allow the expert system a greater degree of freedom, and may result in improved resolution.

Implementation

The expert system has been implemented on an IBM PS/2 Model 50 microcomputer, which provides a usable memory of 640 Kbytes, with a 20Mbyte hard disk. The logic processing is performed by a program written in microProlog [18]. As this language is not suited to the large amount of numerical processing required in the interpretation of the spectrochromatograms, an extension to the language is used [19] to allow the prolog to execute programs in other languages such as Pascal [20]. A modular design has been adopted, so that changes and extensions to the system may be implemented without forcing modifications to previously implemented sections.

Conclusions

The system including the tests and logical modules described above has been implemented and demonstrated for the optimisation of the separation of a four-component mixture. This system is still under development and will receive additional modules to increase its "intelligence" and thereby reduce the need for operator intervention. In particular rules to guide the selection and use of an appropriate deconvolution technique will be established and will form a key part of the enhanced system. The system, as implemented, combines elements of logic processing with numerical processing routines, and serves to demonstrate the potential advantages to be obtained from such a synergistic combination.

Acknowledgements — The Authors acknowledge receipt of a Post-Doctoral Assistantship from the Science and Engineering Research Council for one of us (TPB) and the generous loan of equipment by Pye Unicam Ltd. Cambridge.

References

[1] J. C. Berridge, *Techniques for the Automated Optimisation of HPLC Separations*. Wiley, Chichester (1986).
[2] P. J. Schoenmakers, *Optimisation of Chromatographic Selectivity*. Elsevier, Amsterdam and New York (1986).
[3] J. L. Glajch, J. J. Kirkland and J. J. Squire, *J. Chromatogr.* **199**, 57–79 (1980).
[4] J. C. Berridge, *Analyst* **109**, 291–293 (1984).
[5] F. Hayes-Roth, D. A. Waterman and D. B. Lenat, *Building Expert Systems*. Addison-Wesley, Reading, MA (1983).
[6] T. P. Bridge, M. H. Williams and A. F. Fell, *Chem. Brit.* **23**, 1085–1088 (1987).
[7] T. P. Bridge, M. H. Williams and A. F. Fell, *Anal. Proc.* **25**, 43–44 (1987).
[8] T. P. Bridge, M. H. Williams and A. F. Fell, *Chromatographia* **24**, 691–695 (1987).
[9] A. C. J. H. Drouen, *Computerized Optimization and Solute Recognition in Liquid Chromatographic Separations*. PhD Thesis, Technical University Delft (1985).
[10] L. R. Snyder, J. W. Dolan and J. R. Grant. *J. Chromatogr.* **165**, 3–30 (1979).
[11] P. J. Schoenmakers, H. A. H. Billiet and L. de Galan, *J. Chromatogr.* **205**, 13–30 (1981).
[12] G. G. R. Seaton and A. F. Fell, *Chromatographia* **24**, 208–216 (1987).
[13] A. F. Fell, H. P. Scott, R. Gill and A. C. Moffat, *Chromatographia* **16**, 69–78 (1982).
[14] E. R. Malinowsky and D. G. Howery, *Factor Analysis in Chemistry*. Wiley, New York (1981).
[15] T. P. Bridge, A. F. Fell and R. H. Wardman, *J. Soc. Dyers and Colourists* **103**, 17–27 (1987).
[16] A. F. Fell, H. P. Scott, R. Gill and A. C. Moffatt, *J. Chromatogr.* **282**, 123–140 (1983).
[17] A. G. Wright, A. F. Fell and J. C. Berridge, *Chromatographia* **24**, 533–540 (1987).
[18] K. L. Clark and F. G. McCabe, *micro-PROLOG: Programming in Logic*. Prentice-Hall International, Englewood Cliffs, NJ (1984).
[19] K. T. Milne, M. H. Williams, B. J. Clark and A. F. Fell, *Anal. Proc.* **23**, 157–160 (1986).
[20] K. Jensen and N. Wirth, *PASCAL User Manual and Report*. Springer, Berlin (1975).

[*Received for review 10 December 1987*]

78. T.P.BRIDGE, M.H.WILLIAMS and A.F.FELL, An expert system approach to eluent optimization in RPLC with multichannel detection, Journal of Liquid Chromatography, vol. 12, no. 1, 23-34 (1989).

JOURNAL OF LIQUID CHROMATOGRAPHY, 12(1&2), 23-34 (1989)

AN EXPERT SYSTEM APPROACH TO ELUENT OPTIMISATION IN REVERSE-PHASE LIQUID CHROMATOGRAPHY WITH MULTICHANNEL DETECTION

T.P. BRIDGE*[1], M.H. WILLIAMS[1]
AND A.F. FELL[2]
[1]*Department of Computer Science*
Heriot-Watt University
79 Grassmarket
Edinburgh EH1 2HJ, UK
[2]*Department of Pharmaceutical Chemistry*
University of Bradford
Bradford, West Yorkshire BD7 1DP, UK

ABSTRACT

The optimisation of eluent composition in reversed-phase liquid chromatography is a complex process, which often requires the expenditure of a significant amount of effort on the part of the chromatographer. To assist in this task an expert system has been developed which uses a gradient elution experiment to determine the appropriate initial solvent strength, followed by response-surface modelling using an iterative regression method to determine the mobile phase composition for optimum resolution. Spectral information from a multichannel diode array detector is used to provide retention information for this model. Peak homogeneity is assessed by a number of independent modules, the output from which is interpreted by the expert system and used to validate the response surface model constructed by the optimisation procedure.

INTRODUCTION

The use of expert systems, which can be described as computer programs with the ability to process knowledge as well as numerical data, is becoming increasingly important within the field

of analytical chemistry (1). Such systems have been developed to assist in the interpretation of experimental results derived, for instance, from a fluorimeter (2) or infra-red spectrophotometer (3), and to guide the analyst in the selection of the appropriate technique for a given analysis (4) or in the establishment of the appropriate instrument conditions once a method has been selected (5). The system described in this paper falls into the last of these categories, in that it has the capability to select the composition of the mobile phase for use in reversed-phase liquid chromatography so as to meet predefined analytical requirements.

To establish the mobile phase composition for a separation by HPLC, that gives the required resolution of the components of a given sample, within a predefined analysis time, requires the adjustment of a number of interacting variables. The parameters available to the analyst include the mode of separation; type of column to be used; and in reversed-phase chromatography, the identity and concentration of the organic modifier(s), the pH of the mobile phase, the identity of the buffer used, if any, and the identity and concentration of any additional components in the mobile phase. The adjustment of these parameters to achieve the desired separation can be a time consuming operation, requiring both an appropriate experimental methodology and information about the likely behaviour of the analyte.

Although several strategies are available for the automated determination of optimum conditions (6), most implementations rely upon a search of the parameter space to locate the optimum conditions, with the inherent need to perform a large number of trial separations. To reduce the number of experiments required during the optimisation process, the Authors have developed an expert system approach to optimisation (7,8). This expert system has the capacity to use the multivariate information extracted from the chromatograms produced during the initial stages of the optimisation procedure, to predict a suitable composition of the mobile phase to resolve each separately identifiable solute in an unknown mixture. This initial prediction is iteratively refined until the final optimum mobile phase composition is established. The available information content from each chromatogram is enhanced by the use of a photodiode array detector (9) to provide spectral information for all eluting peaks.

The photodiode array detector or PDA provides an extra dimension of information from a single chromatogram, as a result of its ability to collect ultra-violet or visible spectral information "on the fly" during the elution process. Thus in place of the conventional chromatogram, a three-dimensional spectrochromatogram is produced, as illustrated in Figure 1, in which the axes are time, wavelength and absorbance. The multi-variate nature of the data produced by a PDA allows the use of sophisticated chemometric tools, which may increase the useable information content of a spectrochromatogram by means of statistical manipulation (10).

OPTIMISATION STRATEGY

The strategy adopted by the expert system at the start of the optimisation process is to establish the initial solvent strength required for the separation by means of a water-methanol

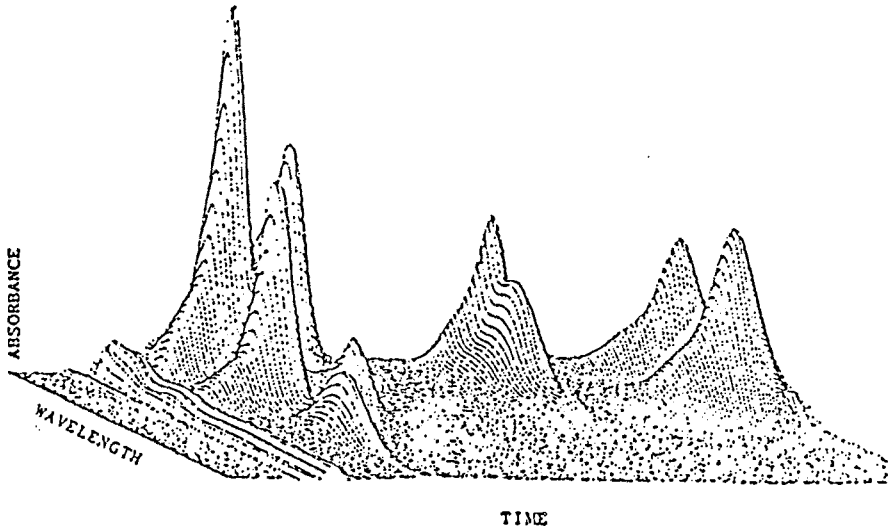


Figure 1: Spectrochromatogram of the test mixture.

gradient elution experiment. Then a model is constructed of the response surface over the solvent triangle bounded by isoeluotropic water-modifier mixtures of equivalent eluting power to that of the initial mixture (11). The modifiers added to the aqueous component at this stage are methanol, acetonitrile and tetrahydrofuran for reversed-phase separations. Other solvents could be introduced as may prove neccessary.

The construction of the response surface model is achieved by the iterative regression method (12). In this approach a logarithmic relationship is initially assumed to hold between the concentration of each component of the mobile phase and the capacity ratio (k') of each detected component. From this relationship a prediction is made of the resolution (R_s) between all pairs of identifiable solutes at all points within the solvent triangle. Thus the conditions can be predicted to give the best separation of the *worst resolved* pair of solutes. A separation is attempted using this predicted mobile phase composition, and the data produced are then incorporated into the model to give a more accurate description of the relationship between concentration and k'. This allows an improved prediction of the optimum mobile phase composition to be made. The iterative cycle of prediction, experimentation and refinement of the model response surface continues either until no further improvement in resolution is produced, or until a pre-determined resolution is achieved.

The iterative regression approach to mobile phase optimisation relies upon the correct and unambiguous identification of each solute in each of the chromatograms. Should this not be achievable, due perhaps to co-elution of two or more components or the close similarity of the uv-absorption spectra, the system then employs a modified Simplex procedure (13) to enable the optimisation to continue. This is possible since the Simplex procedure makes no assumptions based upon spectral data. To utilise the Simplex method the overall quality of each chromatogram must be described by a suitable function. This requires a numerical value, calculated according to an appropriate response function (14). The optimisation proceeds by comparing the value of the response function produced by three different mobile phase compositions. The mobile phase conditions giving the smallest value for the response function are discarded. A new composition for the mobile phase is then predicted by reflecting this discarded point about the line connecting the two remaining data points.

After each separation carried out under the control of the Simplex optimisation strategy, the spectro-chromatographic data are examined in an attempt to acquire sufficient spectral information for unambiguous assignation of identity to be made. As soon as this information becomes available, due to the improvement in resolution achieved, the control of the optimisation process is passed to the iterative regression algorithm for further refinement.

ASSESSMENT OF PEAK HOMOGENEITY

The iterative regression optimisation strategy requires that the retention time of each solute be established for each of the distinct sets of separation conditions employed. To assist in this, each eluting peak is characterised by a spectral profile, produced by a multi-channel diode array detector, which should enable it to be associated with a particular component in the mixture. The assignment of spectral profiles is straightforward for peaks containing only one component unless two or more components exhibit similar spectra, when other factors such as peak area may be employed to reduce any ambiguity. However the process is more complicated in cases of co-elution, when estimates of the true spectral profiles of the contributing components can only be produced after a suitable deconvolution of the matrix of (A, λ, t) data by means of a chemometric method. A number of these powerful statistical tools are available, including, for example, iterative target testing factor analysis (15). Deconvolution may require the expenditure of a significant amount of computing time, particularly when using a bench-top microcomputer. For this reason, the expert system is provided with a number of simpler tools to assess the homogeneity of each detected peak so that the use of deconvolution can be restricted to cases where the chromatographic peak is shown to be non-homogeneous.

The tests for assessing peak homogeneity available to the system have been described previously (6) and include: comparison of the normalised spectral profiles recorded on the upslope and the downslope of the peak; evaluation of the derivative total-absorbance chromatogram;

examination of the absorbance-ratio chromatogram; determination of the apparent retention time as a function of detector wavelength; and examination of the ratio of time-domain derivative chromatograms. These tests are described below.

Comparison of Normalised Spectral Profiles

Spectra are extracted from the data matrix at times before and after the peak where the amplitude of the total absorbance signal is approximately one third of that recorded at the maximum. These spectra are then normalised at the wavelength of maximum absorption in the first spectrum. The square root of the sum of squares (RSS) difference between the two profiles is calculated. This value reflects the variation in the spectral profile across the peak and is indicative of the degree of heterogeneity of the peak.

Evaluation of the Derivative Total Absorbance Chromatogram

The profile of a differentiated curve is often a more sensitive indicator of the existence of an additional, minor component than is the profile of the original zero-order curve (16). To make use of this effect an approximation to the derivative of the total absorbance chromatogram with respect to time is obtained by a simple difference mechanism: $\frac{dA}{dt}(i) = \frac{(A_{(i+1)} - A_{(i-1)})}{2}$. The number of zero crossings in the derivative profile is determined. For a pure peak one zero crossing is expected. To reduce the effects of noise in the derivative signal in determining the number of zero-crossings, a crossing is only accepted as valid if the value of the total absorbance chromatogram at that time is above a threshold of 0.1. The first derivative chromatogram is further differentiated and the number of zero crossings in the second derivative chromatogram determined. Two such crossings are expected for a pure peak. The derivative chromatograms produced by a pure peak are illustrated in Figure 2a. The test is designed to detect variations from this ideal produced by the presence of one or more additional components. Such variations are illustrated in Figure 2b.

Examination of the Absorbance Ratio Chromatogram

The ratio of absorbances measured at two wavelengths should be invariant with time for an homogeneous peak (17). This test locates the time at which maximum absorbance occurs (t_{max}) and identifies the wavelength of maximum absorbance (λ_{max}) and the wavelength which gives half this absorbance value at this time ($\lambda_{v_{max}}$). The ratio of the signals at these two wavelengths is then calculated as a function of time across the peak for all points at which the signal is greater than an experimentally determined cut-off value. The exclusion of any signal of low value concurs two benefits on the final profile. Firstly any distortion due to the effects of noise on the ratiogram is

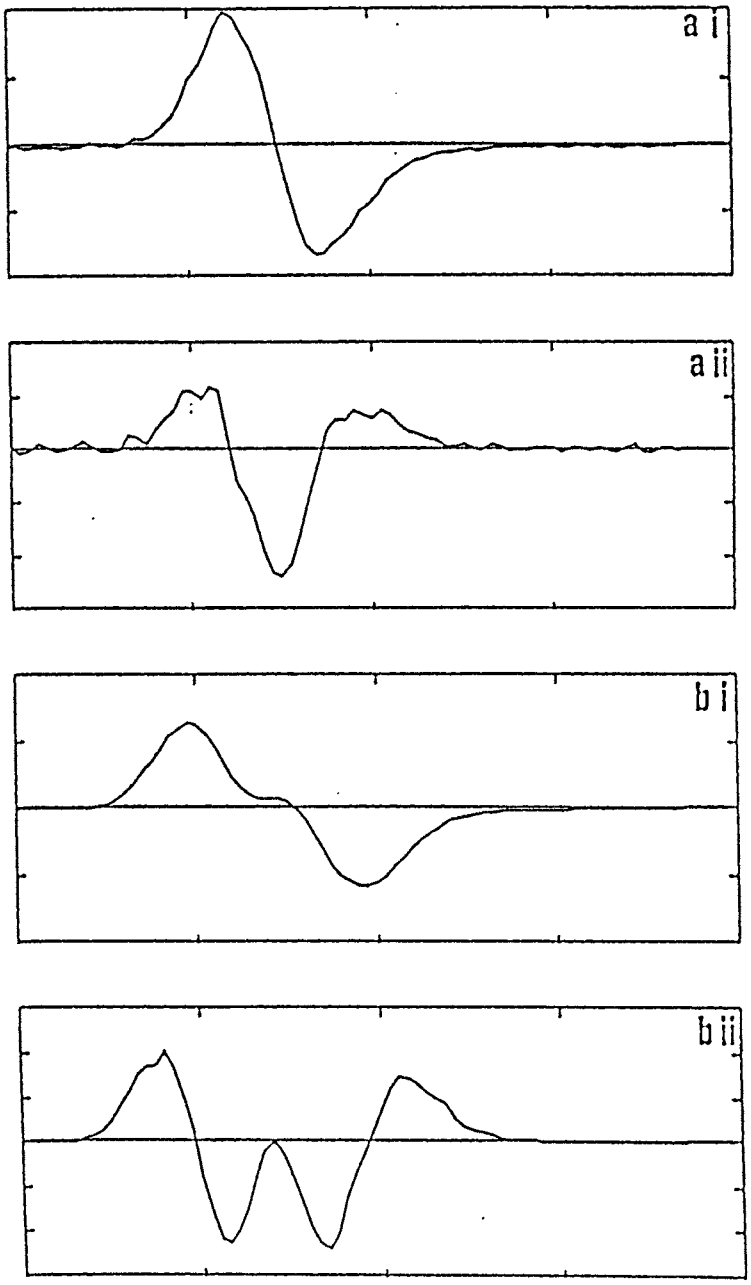


Figure 2: First (i) and second (ii) time-derivative chromatograms of a pure (a) and a composite (b) peak.

reduced, and secondly the chance of an inadvertant division by zero is eliminated. The RSS deviation from the value of the ratio at t_{max} is determined and reported. Ideally this should be zero, or for a real system not exceed the variation in signal due to detector noise. The presence of additional components distorts the ideal square-wave profile of the ratiogram (Figure 3a), as illustrated by Figure 3b.

Determination of the Apparent Retention Time as a Function of Wavelength

The retention time for a given component will be independent of the wavelength used to monitor the chromatogram. Where a peak is actually due to the co-elution of two or more components, a variation in apparent retention time with detector wavelength may be observed (18). To quantify this variation the value of t_{max} is determined for each detector channel and the difference between the largest and smallest values reported. To avoid reporting of noise signals for wavelengths at which the solute does not absorb, no retention time is recorded for wavelengths with a maximum signal of less than 0.1 AU.

Examination of the Ratio of Time-Domain Derivative Chromatograms

The idea of absorbance ratio evaluation may be extended into the derivative domain (19). This routine extracts chromatograms at λ_{max} and at $\lambda_{\prime max}$, normalises them at t_{max} and then calculates the time derivative for each chromatogram. The ratio of the chromatogram at $\lambda_{\prime max}$ to that at λ_{max} (which should be unity) is then calculated and the RSS deviation computed and reported, as for the absorbance ratio chromatogram.

Since the accuracy of the above tests is affected to different degrees by a number of factors (e.g. changes in the peak profile, the signal-to-noise ratio of the detector and the degree of similarity between the component spectra) the interpretation of the information provided by the tests requires consideration of these factors, when they can be determined. These factors are considered by rules contained within the expert system, which directs the interpretation of these test results and provides information on their reliability. The modular nature of the expert system facilitates the incorporation of additional homogeneity tests at any stage during the development of the system.

IMPLEMENTATION OF THE EXPERT SYSTEM

As implied above, the expert system is constructed from a number of modular routines, the internal operation of each module being independent of the rest of the system. The decision-making modules are written in microProlog (17), to which an extension has been provided (18) enabling it to call data capture and numerical processing modules written in other languages, such

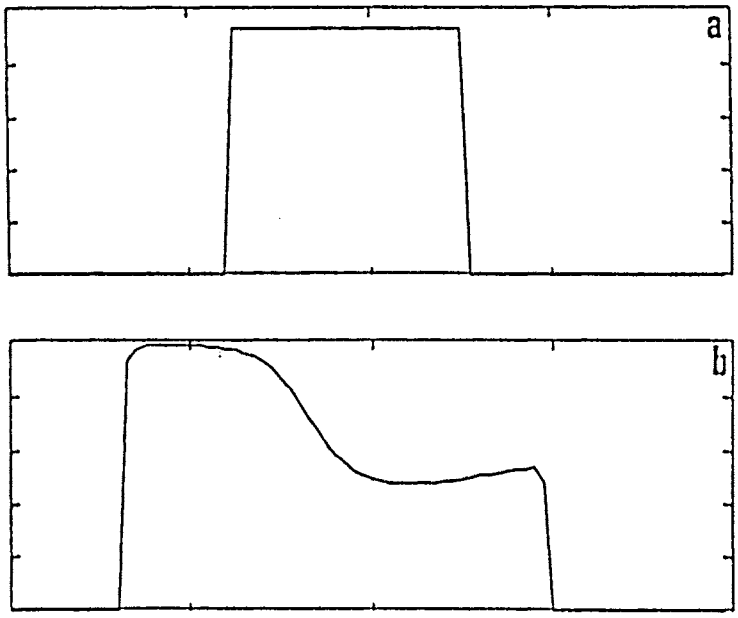


Figure 3: Absorbance ratiograms of a pure (a) and a composite (b) peak.

as Pascal, Fortran and Basic. The system is hosted on an IBM PS/2 microcomputer, with data from the photodiode array detector passed to the computer via an RS232C serial data link.

The present system requires the column to be preselected, and an operator to prepare the mobile phases, operate the pumps and perform the injections. It is our intention to automate these tasks under the direct control of the computer at a later stage in the development of the optimisation system.

MATERIALS AND EQUIPMENT

HPLC-grade methanol, acetonitrile and tetrahydrofuran (Rathburn Chemicals, Walkerburn, UK) were used as received. All eluents were filtered through Millipore® 0.45 µm filters in all glass apparatus and degassed under reduced pressure in an ultrasonic bath for 10 minutes.

Eluent delivery was achieved by the use of two PU4010 pumps under the control of a PU4850 chromatography data station (Pye Unicam Ltd., Cambridge, UK). The detector employed was a Pye-Unicam PU4021 diode array detector, connected to the microcomputer via an RS232C serial interface. The 100 x 5 mm column was packed with 5-µm Hypersil ODS.

EXPERIMENTAL

The operation of the expert system has been demonstrated for the optimisation of the separation of a synthetic mixture. The test mixture contained diethyl phthalate, 3-hydroxybenzaldehyde, n-propyl-p-hydroxybenzoate and phenol dissolved in methanol at a concentration of 20mg/ml.

The initial gradient elution experiment indicated that the required eluting power would be provided by a mixture of methanol and water (60:40 v/v). Equivalent combinations of acetonitrile - water (52:48 v/v) and tetrahydrofuran - water (42:58 v/v) were also prepared. The sample was injected onto the chromatograph using these three isoeluotropic mobile phases and each detected peak examined by the suite of purity checks described above. Retention times were established for each of the four components in each of the three isocratic chromatograms. These data were used to construct a mathematical model of the response surface, as described in the foregiong sections. A contour plot of the lines of equal separation for the worst resolved pair of solutes at each eluent combination is illustrated in Figure 4. In addition to the predicted global optimum composition, with a separation of over 40 arbitrary units, the system predicts the existence of two local maxima: one located at the apex representing 100% methanol - water, with a separation of 32 arbitrary units; the other is located at the apex for 100% THF - water and has a separation of under 5 arbitrary units.

A chromatogram was then carried out under the conditions predicted by this model and the retention data derived from this chromatogram was used to modify the model to produce a refined estimate of the required mobile phase composition. The process of continual refinement was continued until the new predicted mobile phase composition converges with the current actual composition.

DISCUSSION

The system described in this paper illustrates some of the potential benefits which may be derived from the incorporation of a degree of artificial intelligence into an optimisation strategy. To enhance the utility of the system, a number of additional facilities are intended. These include additional tests for peak homogeneity; improved rules to guide the interpretation of these tests; direct computer control of the chromatographic hardware; and rule-based modules to assist in the initial selection of column type and organic modifiers.

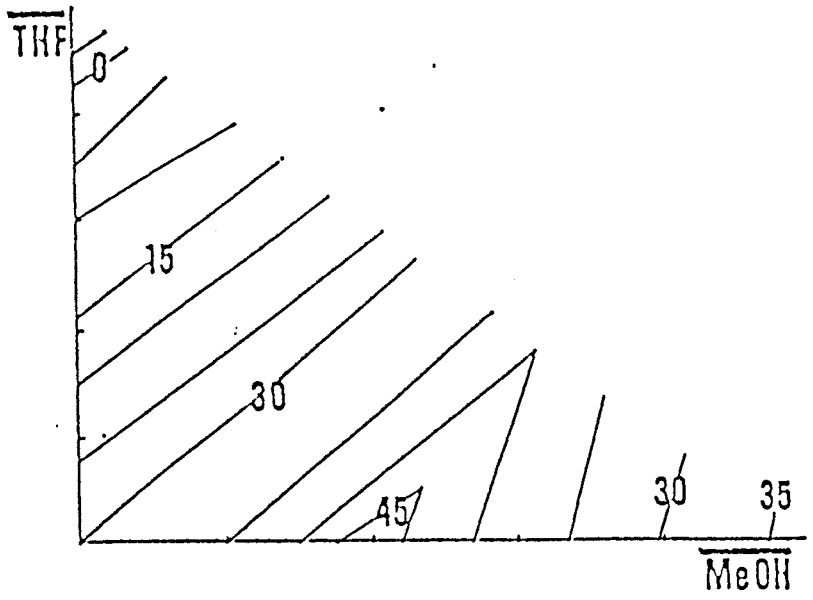


Figure 4: Contour plot of predicted resolution between the worst resolved pair of components.

The contours indicate the separation, in arbitrary units, of the least resolved pair of solutes at all points in the solvent triangle, as predicted by the first stage of the iterative regression optimisation routine. A value of zero indicates mobile phase compositions under which coelution of two components occurs. The best resolution is predicted to occur in a mobile phase containing 40% methanol-water with 0% tetrahydrofuran-water, and hence 60% acetonitrile-water.

ACKNOWLEDGEMENT

The Authors acknowledge receipt of a Post-Doctoral Research Assistantship from the Science and Engineering Research Council for the support of this work.

REFERENCES

1. Bridge, T. P., Williams, M. H. and Fell, A. F., Analytical Chemistry and Expert Systems, Chem. Brit., 23, 1085, 1987

2. Milne, K. T., An Investigation into IKBS Techniques for Analytical Spectrofluorimetry, PhD Thesis, Heriot-Watt University, Edinburgh, 1986.

3. Woodruff H. B. and Smith G. M., A Computer Program for the Analysis of Infra-red Spectra, Anal. Chem., 52, 2321, 1980

4. Detaevernier, M. R., Michotte, Y., Buydens, L., Derde, M. P., Desmet, M., Kaufman, L., Musch. G., Smeyers-Verbeke, J., Thielemans, A., Dryon, L. and Massart, D. L., Feasability Study Concerning the Use of Expert Systems for the Development of Procedures in Pharmaceutical Analysis, J. Pharm. Biomed. Anal., 4, 297, 1986

5. Gunasingham, H., Srinivasan, B. and Anada, A. L., Design of a Prolog-Based Expert System for Planning Separations of Steroids by High-Performance Liquid Chromatography, Anal. Chim. Acta., 182, 193, 1986

6. Berridge, J. C., Techniques for the Automated Optimisation of HPLC Separations, Wiley, Chichester, 1986

7. Bridge, T. P., Williams, M. H., Seaton, G. G. R. and Fell, A. F., An Expert system for Eluent Optimisation in Liquid Chromatography with Photodiode Array Detection, Chromatographia, 24, 691, 1987

8. Bridge, T. P., Williams, M. H. and Fell, A. F., Expert Systems Design in High Performance Liquid Chromatography, Anal. Proc., 25, 43, 1988

9. Fell, A. F., Scott, H. P., Gill, R. and Moffat, A. C., Computer-Aided Multichannel Detection in HPLC, Chromatographia, 16, 69, 1982

10. Balke, S. T., Quantitative Column Liquid Chromatography - a Survey of Chemometric Methods, Journal of Liquid Chromatography Library Volume 29, Elsevier, Amsterdam, 1984

11. Schoenmakers, P. J., Billiet, H. A. H., de Galan, L., Use of Gradient Elution for Rapid Selection of Isocratic Conditions in Reversed Phase HPLC, J. Chromatogr., 205, 13, 1981

12. Drouen, A. C. J. H., Computerised Optimisation and Solute Recognition in Liquid Chromatographic Separations, PhD Thesis, Technical University Delft, 1985

13. Berridge, J. C., Automated Multiparameter Optimisation of HPLC Separations using the Sequential Simplex Procedure, Analyst, 109, 291, 1984

14. Watson, M. W. and Carr, P. W., Simplex Algorithm for the Optimisation of Gradient Elution HPLC, Anal. Chem., 51, 1853, 1979

15. Seaton, G. G. R. and Fell, A. F., Multivariate Analysis of Non-Homogemeous Peaks in Liquid Chromatography, Chromatographia, 24, 208, 1987

16. Bridge, T. P., Fell, A. F. and Wardman, R. H., Perspectives in Derivative Spectroscopy 1: Theoretical Aspects, JSDC 103, 17, 1987

17. Fell, A F., Scott, H. P., Gill, R. and Moffau, A. C., Novel Techniques for Peak Recognition and Deconvolution by Computer-aided Photodiode Array Detection in HPLC, J. Chromatogr., 282, 123, 1983

18. Wright, A. G., Fell, A. F. and Berridge, J. C., Sequential Simplex Optimisation and Multichannel Detection in HPLC: Application to Method Development, Chromatographia, 24, 533, 1987

19. Bridge, T. P., Williams, M. H. and Fell, A. F., Mobile Phase Optimisation in HPLC: An Expert System Approach J. Chemom. Int. Lab. Syst., *submitted for publication*

20. Clark, K. L. and McCabe, F. G., micro-PROLOG: Programming in Logic , Prentice-Hall International, London, 1984

21. Milne, K. T., Williams, M. H., Clark, B. J. and Fell, A. F., Expert Systems in Luminescence Analysis, Anal. Proc., 23, 157, 1986

79. T.P.BRIDGE, M.H.WILLIAMS and A.F.FELL, Comparative evaluation of methods for assessing peak homogeneity in liquid isocratic reverse-phase chromatography, Anal. Chimica Acta, vol. 223, 175-182 (1989).

Analytica Chimica Acta, 223 (1989) 175–182
Elsevier Science Publishers B.V., Amsterdam — Printed in The Netherlands

COMPARATIVE EVALUATION OF METHODS FOR ASSESSING PEAK HOMOGENEITY IN LIQUID ISOCRATIC REVERSED-PHASE CHROMATOGRAPHY

T.P. BRIDGE and M.H. WILLIAMS

Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ (Great Britain)

A.F. FELL*

Analytical Research Group, Department of Pharmaceutical Chemistry, University of Bradford, Bradford, West Yorkshire BD7 1DP (Great Britain)

(Received 18th May 1988)

SUMMARY

A significant problem in the development of an automated system for the interpretation of chromatographic profiles is the detection of cases of co-elution without the use of a considerable amount of computer time or intervention by a human operator. A partial solution to this problem is offered by a range of tests which can be implemented on a suitable microcomputer to provide an estimate of peak homogeneity. Several such tests are described and their application to the three-dimensional data sets produced by a photodiode-array detector is illustrated. The sensitivity of each of the different tests to the presence of a second component varies according to the relative amplitude, separation and spectral similarity of the two components. The extent of this variation is assessed for each of the tests and an indication is given of the reliability of each method under typical chromatographic conditions.

The authors have previously described [1,2] an expert system to assist in the development of an optimum separation for a sample by isocratic reversed-phase liquid chromatography. The expert system determines the appropriate solvent strength by means of a gradient elution experiment, and subsequently establishes the mobile-phase composition to achieve optimum selectivity. The selectivity optimization is achieved by adjusting the relative concentrations of three organic modifiers according to either a modified simplex [3] or an iterative regression optimization [4] technique.

The choice of optimization method is governed by the amount of reliable information which can be extracted from the iso-eluotropic chromatograms. The iterative regression method relies on the ability of the system to establish capacity factors for each solute under a number of different elution conditions

0003-2670/89/$03.50 © 1989 Elsevier Science Publishers B.V.

in order to provide a basis from which a mathematical model of the effect of changes on mobile-phase composition can be produced, so that the optimum composition can be selected. The basis for identifying each solute in each chromatogram, and hence allowing the variation in capacity factor with mobile-phase composition to be "tracked", is the spectral information derived from a linear photodiode-array detector. The system compares the spectra associated with each component in one chromatogram with the spectra derived from other separations of the same mixture. If the matching process is successful, the variation in retention time, and hence capacity factor, can be evaluated for each detected solute.

This matching process may fail, however, for reasons which include the variation in spectral profile caused by the change in solvent, the inability to discriminate between two or more components with similar spectral profiles, or the masking of components as a result of co-elution. In such cases, the optimization is detected by a modified simplex procedure, which requires no spectral information, until sufficient additional information has been acquired to allow the iterative regression procedure to take control of the optimization process.

The reliance of the overall optimization on the simplex method will be reduced if spectra can be extracted from peaks resulting from the co-elution of two or more components. This can be achieved by the use of appropriate chemometric tools, such as factorial analysis [5]. A disadvantage of applying such techniques is the amount of computer power required to produce meaningful spectral profiles. The overall computing requirement can be reduced significantly if factorial analysis is applied only to heterogeneous peaks, where the spectral information is only available after deconvolution, and not to homogeneous peaks, where no additional information can be derived.

As a precursor to the development of a knowledge-based module to assist in the evaluation of peak homogeneity, several rapid screening tests were implemented on a microcomputer. This paper describes the operation of these tests on peaks of controlled complexity and comparison is made of the apparent sensitivity of the tests.

ASSESSMENT OF PEAK HOMOGENEITY

In all, six methods were used for assessing peak homogeneity in this study. All were coded in Pascal on an IBM PS/2 microcomputer. A brief description of each method is given below.

Difference in spectral profile

The normalized profiles of spectra taken from the leading and trailing edges of the peak are compared. Spectra are extracted from the data matrix at times before and after the peak maximum, when the amplitude of the total absorbance signal is one third of that recorded at the maximum. These spectra are

then normalized at the wavelength of maximum absorption in the spectrum recorded on the leading edge of the peak. The square root of the sum of squares (RSS) of the differences between the two profiles is calculated and this value is reported.

First-derivative total absorbance chromatogram

The profile of a differentiated curve is often a more sensitive indicator of the existence of a minor component than is the original zero-order curve [6]. To make use of this effect, an approximation to the differential of the total absorbance chromatogram with respect to time is obtained by a simple difference formula:

$$dA/dt\ (i) = (A_{(i+1)} - A_{(i-1)})/2$$

The number of zero crossings in the derivative profile is counted. For a pure peak, one zero crossing would be expected. To reduce the effects of noise in the derivative signal in estimating the number of zero crossings, a crossing is accepted as valid only if the value of the total absorbance chromatogram at that time is above a defined threshold.

Second-derivative total absorbance chromatogram

As a consequence of the band-sharpening produced by differentiation, the second derivative profile, d^2A/dt^2 is more sensitive to the presence of minor components than is the first derivative. To produce the second-derivative profile, the first-derivative chromatogram is further differentiated numerically. As for the first-derivative profile, the number of zero crossings is used as an indicator of homogeneity. Two such crossings are expected for a pure peak.

Absorbance ratio chromatogram

The ratio of absorbances measured at two wavelengths should be invariant with time for an homogeneous peak [7]. This test locates the time at which maximum absorbance occurs (t_{max}) and identifies the wavelength of maximum absorbance (λ_{max}) and the wavelength which corresponds to half this absorbance value at this time ($\lambda_{max/2}$). The ratio of the signals at these two wavelengths is then calculated as a function of time across the peak. As the ratio is not evaluated for signals below a pre-determined threshold, to reduce the effects of noise on the final ratiogram, and as the ratio of the absorbance values of the two wavelengths is invariant with time, a pure peak produces a ratiogram with a square-wave profile. To assess the variation from this ideal profile, the RSS deviation from the value of the ratio at t_{max} is calculated and reported. Ideally this should be zero, or for a real system it should not exceed the variation in signal caused by detector noise.

Variation of apparent retention time with detector wavelength

The observed retention time for a pure component will be independent of the wavelength used to monitor the chromatogram. In the case of a peak which results from the co-elution of two or more components, which have sufficiently different spectra, a variation in apparent retention time with detector wavelength may be observed [8]. To quantify this variation, the value of t_{max} is measured for each detector channel and the difference between the largest and smallest values is reported. To avoid reporting of artifact signals caused by noise for wavelengths where the solute does not absorb, no retention time is recorded for wavelengths with a maximum signal of less than a defined threshold.

Ratio of derivative chromatograms

The idea of absorbance ratio evaluation, as described above, can be extended to the derivative domain [9]. This routine extracts chromatograms at λ_{max} and at $\lambda_{max/2}$ for the spectrum recorded at t_{max}, normalizes them with respect to each other at t_{max} and then calculates the time derivative, dA/dt, for each of the two chromatograms. The ratio of the derivative chromatogram at $\lambda_{max/2}$ to that at λ_{max} (which should be unity) is then calculated and the RSS deviation computed and reported, as described for the absorbance ratio chromatogram.

EXPERIMENTAL

Methanol (HPLC grade; Rathburn Chemicals, Walkerburn, Gt. Britain) was used without further purification. All solutes (1,3-xylenol, 3,4-xylenol, 2,3-xylenol, 2,5-xylenol, fluoranthrene, biphenyl, fluorene and pyrene; BDH) were used as received. The chromatographic hardware comprised a PU4010 pump controlled by a PU4850 chromatography controller, with a PU4021 diode-array detector (all Pye Unicam). The simulated column consisted of 19 m of 0.010-in. i.d. stainless steel tubing. Injection was achieved by means of a Rheodyne 7125 valve with a 20-μl loop.

The procedure used to generate the chromatographic data was to inject a single component and then run the pump at a low flow rate (0.1 ml min^{-1}) for a specified time interval, to sweep the injection into the tube. The flow was then stopped and the next component injected. This procedure was repeated for each component under investigation. The pump speed was then increased to 0.5 ml min^{-1} and data collection was started after a delay of 10 s to allow any pressure surges to subside. Pure methanol was used as the solvent for all experiments. The solutes were dissolved in methanol at 2 μg ml^{-1}. Injections were in the range 1–2 μl, adjusted to produce the appropriate detector response.

The chromatograms recorded fell into four different classes: (a) single-component chromatograms to serve as standards for the homogeneity tests; (b) chromatograms consisting of 2, 3 and 4 overlapping components, with an over-

TABLE 1

Composition of the test chromatograms

Run no.	Component	Relative amplitude	Separation (s)	Run no.	Component	Relative amplitude	Separation (s)
5	Fluoranthrene	1	-	1	1,3-Xylenol	1	-
6	Biphenyl	1	-	2	3,4-Xylenol	1	-
7	Fluorene	1	-	3	2,3-Xylenol	1	-
8	Pyrene	1	-	4	2,5-Xylenol	1	-
11	Fluorene	1		9	2,3-Xylenol	1	
	Pyrene	1	20		3,4-Xylenol	1	20
12	Fluoranthrene	1		10	1,3-Xylenol	1	
	Biphenyl	1	20		2,5-Xylenol	1	20
14	Pyrene	1		13	2,3-Xylenol	1	
	Biphenyl	1	20		2,5-Xylenol	1	20
	Fluorene	1	20		1,3-Xylenol	1	20
16	Fluoranthrene	1		15	2,3-Xylenol		
	Biphenyl	1	20		3,4-Xylenol	1	20
	Fluorene	1	20		2,5-Xylenol	1	20
	Pyrene	1	20		1,3-Xylenol	1	20
17	Biphenyl	1		21	1,3-Xylenol	1	
	Pyrene	1	40		2,5-Xylenol	1	40
18	Biphenyl	1		22	1,3-Xylenol	1	
	Pyrene	1	30		2,5-Xylenol	1	30
19	Biphenyl	1		23	1,3-Xylenol	1	
	Pyrene	1	10		2,5-Xylenol	1	10
20	Biphenyl	1		24	1,3-Xylenol	1	
	Pyrene	1	5		2,5-Xylenol	1	5
25	Fluorene	1					
	Biphenyl	1	20				
26	Fluorene	1					
	Biphenyl	1	10				
27	Fluorene	1					
	Biphenyl	1	5				
28	Fluorene	1					
	Biphenyl	1	20				
29	Fluorene	1					
	Biphenyl	1	10				
30	Fluorene	2					
	Biphenyl	1	5				
31	Fluorene	4					
	Biphenyl	1	20				
32	Fluorene	4					
	Biphenyl	1	10				
33	Fluorene	4					
	Biphenyl	1	5				

lap of ca. half peak width; (c) chromatograms containing two components at differing degrees of overlap; and (d) chromatograms containing two components at differing relative concentrations. With the exception of series (d), the tests were conducted on compounds exhibiting similar spectra (the xylenols) and on those with different spectral profiles (the aromatic hydrocarbons). The amplitude of each peak in (b) and (c) was the same. Details of the solutes and data for each experiment are given in Table 1.

RESULTS AND DISCUSSION

The six tests described above were each applied to the 33 test chromatograms, and the results are given in Table 2.

The spectral difference test clearly distinguishes between homogeneous and heterogeneous peaks. Only one pure peak returned an error value greater than 3.0 (from a population of eight pure peaks); and two heterogeneous peaks (from a population of 25) gave an error below this value.

In the first-derivative total absorbance test, a peak is assumed to be homogeneous if only one zero crossing is observed. On this basis, all the pure peaks were correctly classified. However, most of the composite peaks were also classified as homogeneous. In this test, only 5 of the 25 heterogeneous peaks were correctly classified.

The second-derivative total absorbance test is based on the second-derivative profile. thus it would be expected to be more sensitive than the first-derivative test. When two crossings were used as the criterion of purity, the pure peaks were again correctly classified. However, only 12 of the composite peaks were correctly indicated to be heterogeneous, including all those identified by the examination of the first-derivative profile; 13 of the 25 heterogeneous peaks were incorrectly classified as pure.

For the absorbance ratio test, the spread of values obtained was so great that it was not possible to assign an unambiguous value of the error in the absorbance ratio to discriminate between homogeneous and heterogeneous peaks. When the upper limit for homogeneous peaks was set to 0.6, then only one of the pure peaks was incorrectly classified as heterogeneous, but 19 of the composite peaks were classified as pure. Reducing the threshold to 0.5 reduced the number of composite peaks classified as pure to nine but also classified four of the eight pure peaks as heterogeneous.

With the retention-time test, although there should, in theory, be no variation in the apparent retention time with detector wavelength for a pure peak, in practice some variation can be observed as a consequence of noise or lack of sensitivity. Allowing a "window" of ± 1 s for this variation enabled all the pure peaks to be correctly classified. Only three of the composite peaks were incorrectly assessed as pure.

In the derivative ratio test, the use of a threshold value of 0.5, as assigned

TABLE 2

Results of homogeneity tests

Run no.	Spectral error	Derivative zero crossings		Absorbance ratio error	Derivative ratio error	Change in t_{max} (s)
		D1	D2			
1	5.42659	1	2	0.32500	0.23750	0
2	1.90799	1	2	0.41361	0.35820	0
3	2.71494	1	2	0.35000	0.26250	1
4	1.89673	1	2	0.40000	0.31250	1
5	0.24928	1	2	0.55000	0.43750	1
6	0.21886	1	2	0.59996	0.45906	0
7	0.18351	1	2	0.58180	0.44921	0
8	0.19910	1	2	0.75041	0.49021	1
9	5.91527	1	4	0.52500	0.43750	2
10	10.9798	1	4	0.51250	0.41250	8
11	23.2015	1	4	0.19698	-	9
12	18.5441	1	2	0.67660	0.81526	9
13	6.66588	1	6	0.33809	0.24666	2
14	23.5110	1	4	0.84762	0.88098	20
15	2.86176	1	4	0.68982	0.57174	11
16	13.4822	49	52	0.69509	0.67568	30
17	17.2126	4	4	0.56724	1.38595	20
18	17.0821	3	4	0.73128	0.65286	15
19	11.4358	1	2	0.51556	0.53164	4
20	6.25477	1	2	0.43851	0.62723	2
21	6.72012	3	4	0.43339	0.33771	1
22	7.15773	3	4	0.50867	0.37979	1
23	4.65249	1	2	0.42193	0.29398	2
24	2.36396	1	2	0.48750	0.35000	1
25	13.4454	1	2	0.54586	0.59699	9
26	10.1793	1	2	0.44760	0.37122	4
27	5.67206	1	2	0.34642	0.52921	2
28	14.1471	1	7	0.50261	0.32012	10
29	9.48812	1	2	1.02328	1.13859	4
30	5.26361	1	2	0.43268	0.32894	2
31	10.5586	1	2	0.59486	0.44141	9
32	6.57768	1	2	0.50069	0.37220	4
33	4.02878	1	2	0.55058	0.43595	2

for the absorbance ratio test, correctly classified all the single-component peaks as pure, but also included 12 of the composite peaks in this classification. Using a lower threshold value did not improve the discrimination, but did result in a change in the classification. For example, a threshold value of 0.4 classified ten of the composite peaks as homogeneous, but excluded four of the pure peaks from this category.

On the basis of the chromatograms examined in this study, the following

values may be used for discrimination between homogeneous and heterogeneous peaks:
Spectral error — homogeneous if error less than 3.0 arbitrary units.
First derivative — homogeneous if 1 zero crossing in profile.
Second derivative — homogeneous if 2 zero crossings in profile.
Absorbance ratio — homogeneous if error less than 0.5 arbitrary units.
Derivative ratio — homogeneous if error less than 0.5 arbitrary units.
Retention time — homogeneous if variation less than 2 seconds.
It appears that none of the tests can be used in isolation to discriminate reliably between homogeneous and heterogeneous peaks. The situation improves, however, if the combined results are considered. If a peak is assumed to be homogeneous unless two or more tests indicate heterogeneity, then with the discrimination values given above, only one peak is incorrectly classified as homogeneous, with no false indications of heterogeneity.

This enhancement of the overall sensitivity of the homogeneity assessment derives from the reliance of the individual tests on different parameters. Thus whilst the first- and second-derivative total absorbance chromatographic profiles are insensitive to spectral profile but sensitive to the the degree of overlap of the contributing components, the absorbance ratio test is dependent on the differences between the individual spectra. The observed variation in retention time, as a function of detector wavelength, is dependent on both spectral differences and chronological separation. It is intended to incorporate information on the relative sensitivity of the various tests to these factors in a knowledge-based system for peak homogeneity assessment, which is currently under development [10].

The foregoing discussion applies to isocratic systems only. If gradient elution is used the various tests can still be utilised, but interpretation of the results must make allowance for the changes in spectral profile caused by the change in mobile-phase composition during the elution of each peak.

The authors acknowledge receipt of a Post-Doctoral Assistantship from the Science and Engineering Research Council for the support of this work, the assistance of M. Girling (Philips Scientific) in providing the chromatographic data and the generous loan of equipment by Philips Scientific, Cambridge.

REFERENCES

1 T.P. Bridge, M.H. Williams, G.G.R. Seaton and A.F. Fell, Chromatographia, 24 (1987) 691.
2 T.P. Bridge, M.H. Williams and A.F. Fell, Anal. Proc., 25 (1988) 43.
3 J.C. Berridge, Analyst, 109 (1984) 291.
4 A.C.J.H. Drouen, Ph.D. Thesis, Technical University of Delft, 1985.
5 G.G.R. Seaton and A.F. Fell, Chromatographia, 24 (1987) 208.
6 T.P. Bridge, A.F. Fell and R.H. Wardman, J. Soc. Dyers Colour., 103 (1987) 17.
7 A.F. Fell, H.P. Scott, R. Gill and A.C. Moffat, J. Chromatogr., 282 (1983) 123.
8 A.G. Wright, A.F. Fell and J.C. Berridge, Chromatographia, 24 (1987) 533.
9 A.F. Fell, T.P. Bridge and M.H. Williams, J. Pharm. Biomed. Anal., 6 (1988) 555.
10 A.F. Fell, T.P. Bridge and M.H. Williams, J. Chromatogr., 465 (1989) 59.

80. T.P.BRIDGE, M.H.WILLIAMS and A.F.FELL, Artificial intelligence techniques for peak homogeneity validation in liquid chromatography, Journal of Chromatography, vol. 465, 59-67 (1989).

Journal of Chromatography, 465 (1989) 59–67
Elsevier Science Publishers B.V., Amsterdam — Printed in The Netherlands

CHROMSYMP. 1534

ARTIFICIAL INTELLIGENCE TECHNIQUES FOR PEAK HOMOGENEITY VALIDATION IN LIQUID CHROMATOGRAPHY

T. P. BRIDGE and M. H. WILLIAMS
Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ (U.K.)
and
A. F. FELL*
School of Pharmaceutical Chemistry, University of Bradford, Bradford, BD7 1DP (U.K.)

SUMMARY

A problem area in the interpretation of chromatographic data is the determination of the purity of a detected peak. The problem arises in deciding whether a peak is due solely to one component, or is caused by the simultaneous elution of two or more components. Whilst a number of multivariate statistical techniques are available to assess the number of components contributing to a given signal, they may require a significant amount of computer power to provide an answer, and are prone to noise-induced errors. As part of a programme of research aimed at developing an intelligent system for mobile phase optimisation, we have developed a knowledge-based systems approach to the problem of peak homogeneity assessment. The expert system applies a number of simple numerical tests to the three-dimensional dataset produced by a photodiode-array detector and assess the results in light of the chromatographic signal.

INTRODUCTION

The authors are engaged in the development of an expert system to assist in the development of an optimum separation for a mixture in reversed-phase liquid chromatography[1,2]. The expert system determines the appropriate solvent strength by means of a gradient elution experiment, and subsequently establishes the mobile phase composition to achieve optimum selectivity. The selectivity optimisation is achieved by adjusting the relative concentrations of three organic modifiers according to either a modified simplex[3] or iterative regression optimisation strategy[4].

The choice of optimisation strategy is guided by the amount of reliable information which can be extracted from the isoeluotropic chromatograms. The iterative regression method relies on the ability of the system to determine capacity factors for each solute under a number of different elution conditions, to provide a basis from which a mathematical model of the effect of changes in mobile-phase composition on selectivity, can be produced. The optimum composition for the mobile

0021-9673/89/$03.50 © 1989 Elsevier Science Publishers B.V.

phase may then be identified by examination of the modelled response surface. The basis for identification of each solute in each chromatogram, and hence for allowing the variation in capacity factor with mobile-phase composition to be tracked, is the spectral information derived from a linear photodiode-array detector. The system compares the spectrum associated with one component with the spectra derived from other separations of the same mixture. If the matching process is successful, the variation in retention time, and hence capacity factor, can be determined for each detected solute.

However, this matching process may fail for reasons which include the variation in spectral profile caused by the change in solvent, the inability to discriminate between two or more components with similar spectral profiles, or the masking of components as a result of simultaneous elution. In such cases, the optimisation process is directed by a modified simplex algorithm, which requires no spectral information until sufficient additional information has been acquired to allow the iterative regression procedure to resume control of the optimisation process.

The reliance of the overall optimisation on the simplex method will be reduced if spectra can be extracted from peaks resulting from the simultaneous elution of two or more components. This may be achieved by the use of appropriate chemometric tools, such as factorial analysis[5]. A disadvantage of employing such techniques is the amount of computer power required to produce meaningful spectral profiles. The overall computer power requirement can be reduced significantly if the use of factorial analysis is restricted to heterogeneous peaks, where the spectral information is only available after deconvolution, and not on homogeneous peaks, from which no additional information can be derived by the application of such techniques.

To this end, the knowledge-based system is provided with the means to call upon a number of rapid homogeneity screening tests, the results of which are used in combination to assess the need for applying full matrix-based deconvolution methods.

METHODS FOR ASSESSING PEAK HOMOGENEITY

The peak homogeneity tests provided for use by the knowledge-based system were selected on the basis of their rapidity and reliability[6]. Whilst no single test is capable of detecting all cases of simultaneous elution, the use of a number of different tests in combination will increase the probability of correctly identifying heterogeneous peaks. The tests available within the system are described briefly below.

Determination of difference in spectral profile

The normalised profiles of spectra taken from the leading and trailing edges of the peak are compared. Spectra are extracted from the data matrix at times before and after the peak maximum, where the amplitude of the total absorbance signal is one third of that recorded at the peak maximum. These spectra are normalised at the wavelength of maximum absorption of the leading edge spectrum. The square root of the sum of the squares (RSS) of the differences between the two profiles is calculated, and this value is passed to the logic-processing element of the knowledge-based system.

Derivative total absorbance chromatograms

The profile of a differentiated curve is often a more sensitive indicator of the

existence of an otherwise hidden minor component than is the original zero-order curve[7]. To make use of this effect, an approximation to the time-domain derivative of the total absorbance chromatogram is calculated, and the number of zero crossings in the derivative profile is determined. For a pure peak, only one zero crossing will be produced. To reduce the effect of noise in the derivative signal in determining the true number of such crossings, a crossing is only accepted as valid if the value of the total absorbance chromatogram at that time is above a predefined threshold. First-derivative chromatograms of pure and composite peaks are illustrated in Fig. 1.

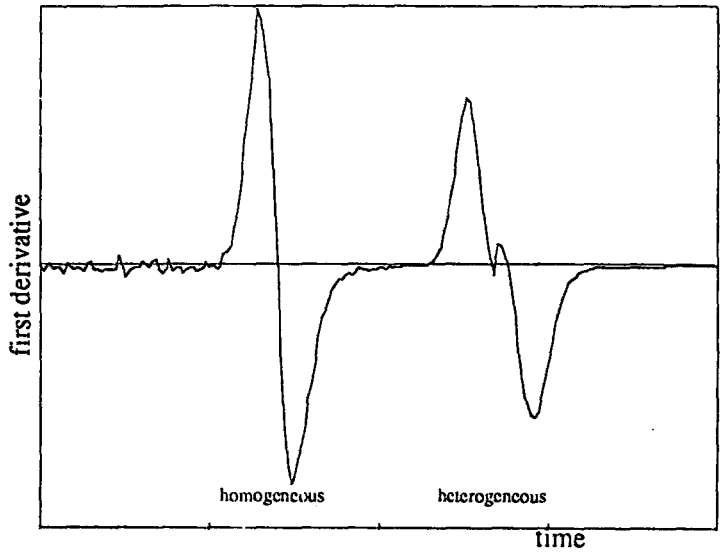


Fig. 1. First-derivative chromatogram.

As a consequence of the band-sharpening produced by differentiation, the second-derivative profile is more sensitive to the presence of minor components than is the first derivative. To produce the second-derivative profile, the first-derivative chromatogram is further differentiated numerically. As for the first-derivative profile, the number of zero crossings is used as an indicator of homogeneity. Two such crossings will be produced by a pure peak. Second-derivative chromatograms of pure and composite peaks are illustrated in Fig. 2.

Absorbance ratio chromatogram

The ratio of absorbances measured at two wavelengths should be invariant with time for a homogeneous peak[8]. This test locates the time at which maximum absorbance occurs in the total absorbance chromatogram and identifies the wavelength of maximum absorbance as well as the wavelength associated with an absorbance of half the maximum, at this time. The ratio of the signals at these two wavelengths is then calculated as a function of time across the peak. As the ratio is not evaluated for signals below a pre-determined threshold, to reduce the effect of noise on the output ratiogram, and as the ratio of the absorbances is independent of

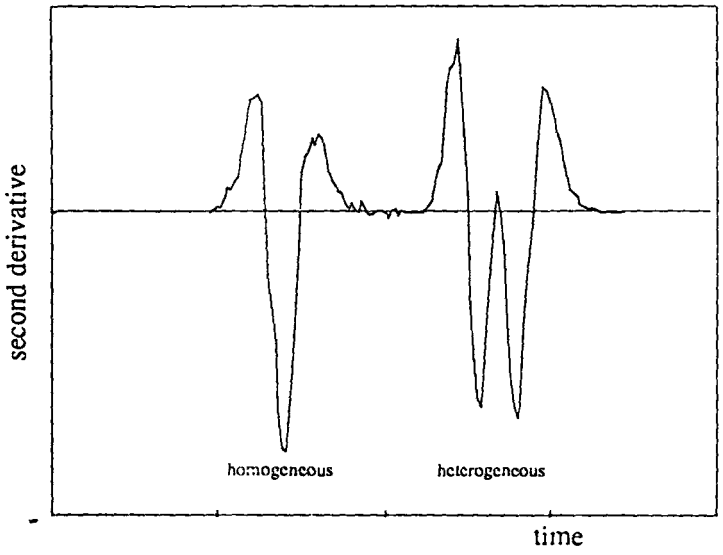


Fig. 2. Second-derivative chromatogram.

concentration, and therefore time, a pure peak produces a ratiogram with a square-wave profile. To assess the variation from this ideal profile, the RSS deviation from the value of the ratio at the midpoint of the peak is determined and reported. For a pure peak this should be zero, or for a real system it should not exceed the variation in signal due to detector noise. Ratiograms derived from homogeneous and heterogeneous peaks are illustrated in Fig. 3.

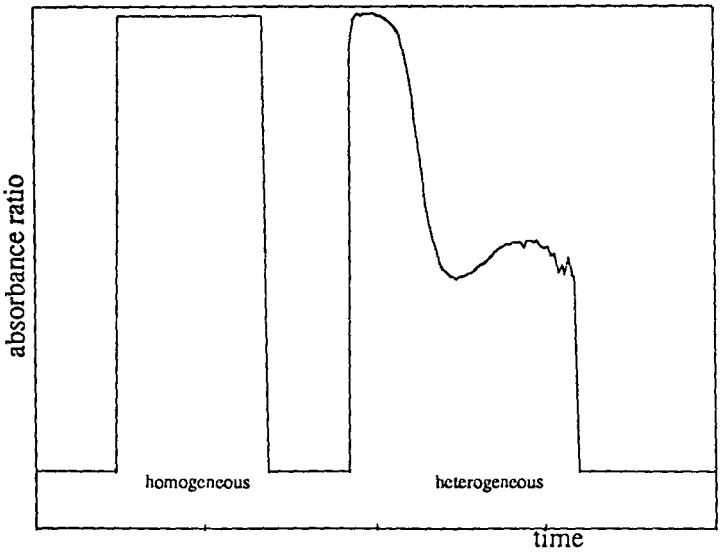


Fig. 3. Ratiogram of pure and impure peaks.

The idea of absorbance ratio evaluation may be extended into the derivative domain[9] if the chromatograms extracted, as above, at the wavelengths of maximum absorbance and half-maximum absorbance respectively, are differentiated before the ratio is evaluated. This may offer improved sensitivity towards certain cases of peak overlap, but at the cost of reduced indifference to noise in the spectrum.

Variation of apparent retention time

The observed retention time of a pure component will be independent of the wavelength used to monitor the chromatogram. However, in the case of a peak resulting from the simultaneous elution of two or more components, a variation in apparent retention time may be observed[10]. The degree of variation depends on the difference in the spectral profiles of the components, their separation and relative amplitudes. To quantify this variation for a single peak, the times of maximum absorbance are determined for each detector channel, and the difference between the largest and smallest values reported. Plots of retention time as a function of wavelength for two peaks are given in Fig. 4.

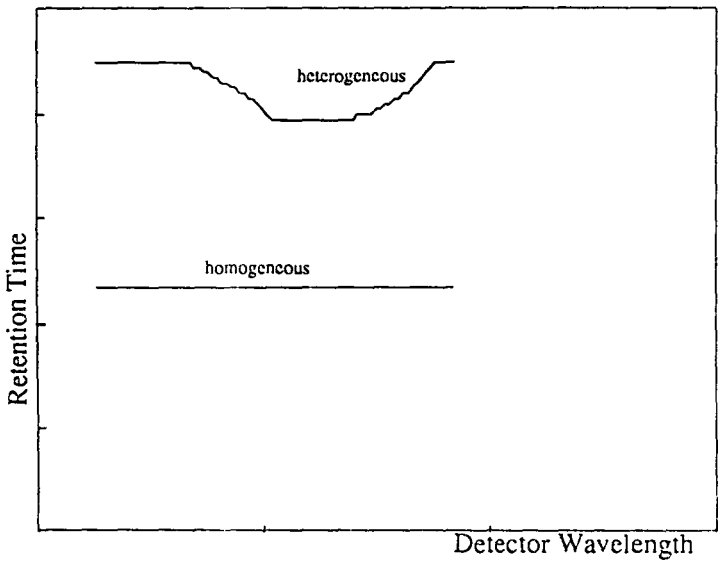


Fig. 4. Variation of retention time for pure and impure peaks.

THE EXPERT SYSTEM

The software forming the knowledge-based system is hosted on an IBM PS/2 micro-computer, operated with PC-DOS 3.30. The software is arranged in a highly modular fashion to facilitate development. Such a structure also allows the use of languages appropriate to the local task. Thus, whilst the logic processing is performed by modules written in microProlog, the homogeneity tests are coded in Pascal. The selection and execution of the various modules is controlled by a supervisory

Prolog Code for Peak Evaluation

```
The query is: ?((homogeneity assessed))

((homgeneity assessed)
 (peak_is_pure yes))

((homgeneity assessed)
 (peak_is_impure yes))

((peak_is_impure yes)
 (NOT derivative_check ok)
 (NOT spectrum_check ok))

((peak_is_impure yes)
 (NOT derivative_check ok)
 (NOT ratio_check ok))

((peak_is_impure yes)
 (NOT derivative-ratio_check ok)
 (NOT derivative_check ok))

 ...

((derivative_check ok)
 (first_derivative_test ok)
 (second_derivative_test ok))

((derivative-ratio_check ok)
 (Evaluate derivative_error x)
 (x LESS limit_deriv_ratio))

((first_derivative_test ok)
 (Evaluate d1_zero_crossings x)
 (x LESS 2)
 (0 LESS x))

 ...

((Evaluate x y)
 (EXECUTE x)
 (OPEN "result_file")
 (READ "result_file" y)
 (CLOSE "result_file"))
```

Fig. 5. Extract from the PROLOG code used to interpret the homogeneity tests.

microProlog program. A section of the logic routine used to interpret the results of the homogeneity tests is reproduced in Fig. 5. The logic employed by the expert system to evaluate peak homogeneity is illustrated by the flowchart of Fig. 6.

EVALUATION

The operation of the expert system was evaluated by assessing the homogeneity of 33 chromatographic peaks of known composition. The composition of each of these

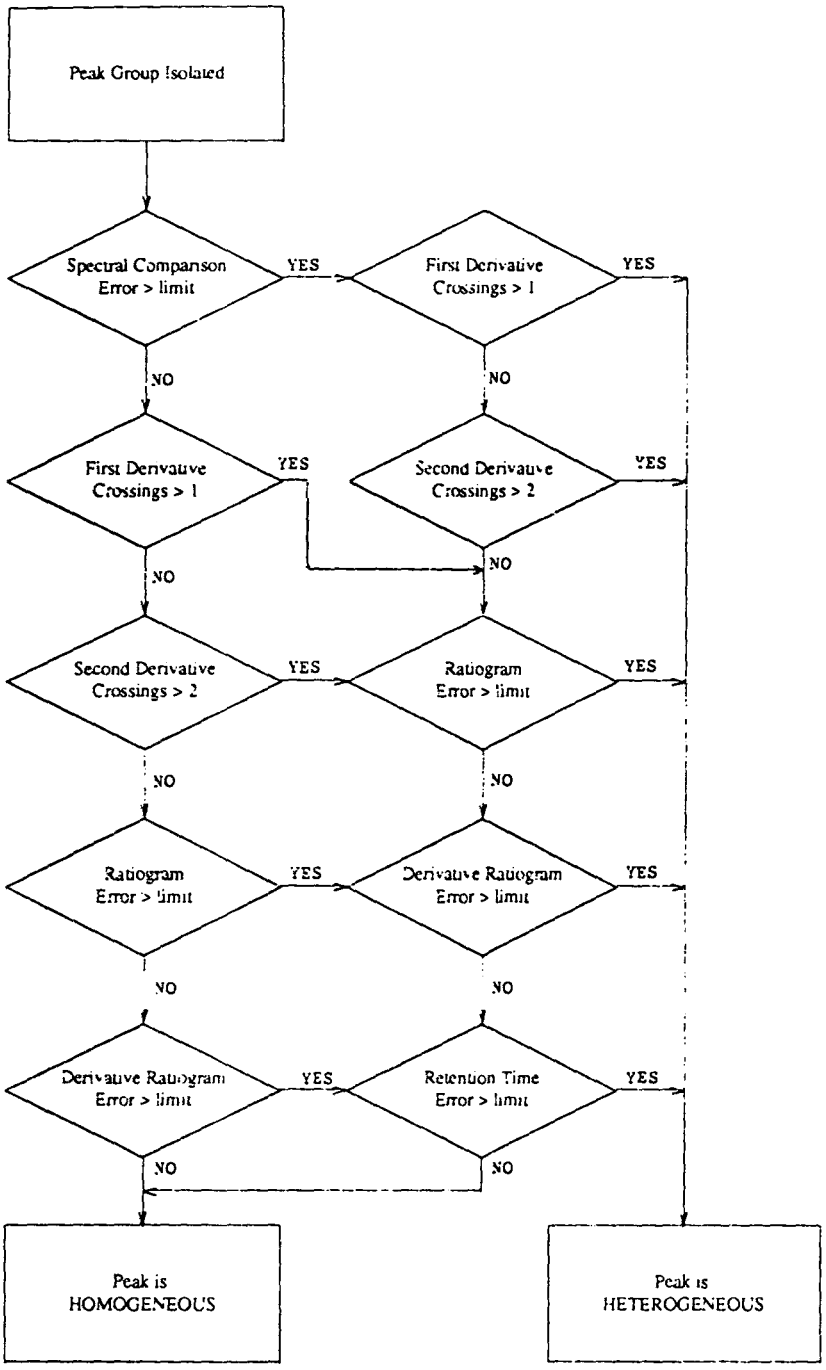


Fig. 6. Flowchart of the logic used to assess peak homogeneity.

TABLE I

COMPOSITION OF THE TEST CHROMATOGRAMS

No. of components	*Component*	*Relative amplitude*	*Separation (s)*	*No. of components*	*Component*	*Relative amplitude*	*Separation (s)*
1	Fluoranthrene	1		1	1,3-Xylenol	1	
1	Biphenyl	1		1	3,4-Xylenol	1	
1	Fluorene	1		1	2,3-Xylenol	1	
1	Pyrene	1		1	2,5-Xylenol	1	
2	Fluorene	1		2	2,3-Xylenol	1	
	Pyrene	1	20		3,4-Xylenol	1	20
2	Fluoranthrene	1		2	1,3-Xylenol	1	
	Biphenyl	1	20		2,5-Xylenol	1	20
3	Pyrene	1		3	2,3-Xylenol	1	
	Biphenyl	1	20		1,5-Xylenol	1	20
	Fluorene	1	20		1,3-Xylenol	1	20
4	Fluoranthrene	1		4	2,3-Xylenol	1	
	Biphenyl	1	20		3,4-Xylenol	1	20
	Fluorene	1	20		2,5-Xylenol	1	20
	Pyrene	1	20		1,3-Xylenol	1	20
2	Biphenyl	1		2	1,3-Xylenol	1	
	Pyrene	1	40		2,5-Xylenol	1	40
2	Biphenyl	1		2	1,3-Xylenol	1	
	Pyrene	1	20		2,5-Xylenol	1	30
2	Biphenyl	1		2	1,3-Xylenol	1	
	Pyrene	1	10		2,5-Xylenol	1	10
2	Biphenyl	1		2	1,3-Xylenol	1	
	Pyrene	1	5		2,5-Xylenol	1	5
2	Fluorene	1					
	Biphenyl	1	20				
2	Fluorene	1					
	Biphenyl	1	10				
2	Fluorene	1					
	Biphenyl	1	5				
2	Fluorene	2					
	Biphenyl	1	20				
2	Fluorene	2					
	Biphenyl	1	10				
2	Fluorene	2					
	Biphenyl	1	5				
2	Fluorene	4					
	Biphenyl	1	20				
2	Fluorene						
	Biphenyl	1	10				
2	Fluorene	4					
	Biphenyl	1	5				

peaks is indicated in Table I. As a comparative exercise the same chromatographic data were interpreted by a number of chromatographers in our laboratories. The results of the evaluation are expressed in Table II.

TABLE II

COMPARISON OF PEAK HOMOGENEITY ASSESSMENT BY THE EXPERT SYSTEM AND HUMAN EXPERTS

Analyst 1	*Analyst 2*	*Analyst 3*	*Analyst 4*	*Expert system*
Homogeneous peaks correctly assessed (from a total of 8)				
2	4	2	5	8
Heterogeneous peaks correctly assessed (from a total of 25)				
25	24	21	24	22

DISCUSSION

In the limited evaluation exercise reported above, the knowledge-based system out-performs the human analysts in the recognition of homogeneous peaks, and is of comparable performance in the detection of heterogeneous peaks caused by the simultaneous elution of two or more components. In such an exercise the human analysts might be expected to out perform the computer system, since they are using a visual interpretation of the complete chromatographic profile, rather than the limited numerical sample available to the expert system.

The use of a limited degree of artificial intelligence in the interpretation of rapid numerical tests offers a method for improving the reliability of peak homogeneity assessment in cases where the resources are not available for more comprehensive analysis by chemometric methods, such as iterative target testing factor analysis.

REFERENCES

1 T. P. Bridge, M. H. Williams, G. G. R. Seaton and A. F. Fell, *Chromatographia*, 24 (1987) 691.
2 T. P. Bridge, M. H. Williams and A. F. Fell, *Anal. Proc.*, 25 (1988) 43.
3 J. C. Berridge, *Analyst (London)*, 109 (1984) 291.
4 A. C. J. H. Drouen, *Ph.D. Thesis*, Technical University of Delft, Delft, 1985.
5 G. G. R. Seaton and A. F. Fell, *Chromatographia*, 24 (1987) 208.
6 T. P. Bridge, M. H. Williams and A. F. Fell, *Anal. Chim. Acta*m (1988) in press.
7 T. P. Bridge, A. F. Fell and R. H. Wardman, *J. Soc. Dyers Colour.*, 103 (1987) 17.
8 A. F. Fell, H. P. Scott, R. Gill and A. C. Moffa, *J. Chromatogr.*, 282 (1983) 123.
9 A. F. Fell, T. P. Bridge and M. H. Williams, *J. Pharm. Biomed. Anal.*, 6 (1988) 555.
10 A. G. Wright, A. F. Fell and J. C. Berridge, *Chromatographia*, 24 (1987) 533.

81. B.J.CLARK, A.F.FELL and M.H.WILLIAMS, Biomedical applications of expert systems in analytical spectroscopy, Advances in Multidimensional Luminescence, vol 1, Eds. I.M.Warner and L.B.McGown, JAI Press Inc., Connecticut, 55-72 (1991).

BIOMEDICAL APPLICATIONS OF EXPERT SYSTEMS IN ANALYTICAL SPECTROSCOPY

Brian J. Clark, Anthony F. Fell, and
M. Howard Williams

1. Introduction 56
2. Expert-system Development 58
   2.1. The Major Components of an Expert System 58
   2.2. Shell-Based Expert Systems 60
   2.3. Programming Languages 60
3. An Expert System for Luminescence Spectroscopy 61
   3.1. Design 61
   3.2. Instrumentation 61
   3.3. Compounds Stored in the Data Base 62
4. Implementation of the Expert System 63
   4.1. Drug Component Identification 64
   4.2. Multicomponent Samples 68
5. Conclusions 70
   References 71

**Advances in Multidimensional Luminescence,
Volume 1, pages 55–72.**
**Copyright © 1991 JAI Press Inc.
All rights of reproduction in any form reserved.**
ISBN: 1-55938-172-8

1. INTRODUCTION

In the short history of computing, the ideas of artificial intelligence have been circulating for some time. In the early days it was widely believed that systems could be designed that would to some degree model the human brain. However, this has been somewhat modified, and artificial intelligence techniques are now more concerned with the design of computer programs that undertake activities thought to require human intelligence. Very simple examples of this are the programs designed to play chess.

Within the discipline of artificial intelligence considerable interest has centered on *expert systems*. An expert system can be regarded as a high-performance program in which translation of human expertise in a particular domain is carried out in a form that the computer can manipulate. These programs can then be used to solve problems that previously could only be resolved by a human expert. A useful definition of an expert system is given by Goodall [1], who states that "an expert system is a computer system that uses a representation of human expertise in a specialist domain in order to perform functions similar to those normally performed by a human expert in that domain."

As working examples of artificial intelligence, expert systems are now widely used in many diverse application areas. Diagnostic-based expert systems have been particularly successful, especially when applied in the fields of medicine, finance, crop disease control, engineering and analytical chemistry.

In medicine and health care the prevalence of patient-oriented expert support systems stemming from MYCIN, an infection diagnostic system developed at Stanford in 1972 [2], has steadily increased. Systems are numerous and varied and include the medical consultant-based systems INTERNIST, PUFF [3], and GLUCON [4].

The considerable current activity in the medical field suggests a flourishing future for medical-based expert systems. From this it is possible to postulate that future developments may encompass all aspects of health care. Total hospital information systems may cover quality assurance of drug product and patient records through to biomedical training. This latter point opens up further possibilities where expert systems may flourish in the future, as self-teaching and simulation packages. Early developed systems that are well known in this area are MEES (economics) and STREAMER (naval engineering) together with numerous smaller packages introduced in specific workplaces.

The spread of interest in expert systems is, therefore, substantial, and the analytical chemist has not been slow to appreciate the advantages accruing from exploiting this technology. It has to be acknowledged that one of the first and arguably the most successful expert systems, DENDRAL, was developed to solve an analytical problem [5]. Stanford University was again responsible for the introduction of DENDRAL, which represented the product of a heuristic programming project started in 1965. The design strategy was to allow elucidation of

molecular structure from mass spectroscopic data. DENDRAL has since formed a building block for many similar expert systems based on organic molecular fragmentation pattern [6–8].

Other areas in analytical chemistry have not been neglected, and it is not surprising that high-performance liquid chromatography (HPLC), one of the fastest developing analytical techniques, should come under close scrutiny. High-performance liquid chromatography offers high separation efficiency for a wide range of multicomponent mixtures on small particle chromatographic columns. However, these separations are not achieved without close attention to the many parameters involved in method development [9]. This is a time-consuming procedure in which all aspects of the method have to be considered, from the instrumentation and the optimized separation itself through to method validation. One aspect of this process, the optimization strategy, can be regarded as a discrete activity in itself [10]. In this area there has already been a movement from chromatographic parameter selection based on experience and intuition to development of the conditions by computer-controlled optimization. The choice of parameters for the best separation includes column material, mobile phase, and detector selection, and requires considerable input from the chromatographer. It became apparent that the multiparameter nature of HPLC should benefit from the application of an expert system in the process of designing a suitable chromatographic procedure.

The early expert systems reported by Musch et al. [11,12] have concentrated their efforts on particular sections of the method-development process or on specific compound classes [13]. A singularly ambitious attempt at assembling an expert system to incorporate experimental design, parameter optimization, results interpretation, and a troubleshooting facility was proposed in a system known as ECAT [14]. This system is, however, isolated at present, as most of the recent literature on expert systems for HPLC has concentrated on a limited number of the facets of method development [15,16]. Nevertheless, the continually increasing computer power of the bench-level microcomputer suggests that the future of expert systems for such an important subject area as HPLC is assured. It will therefore be interesting to chart the progress through to a totally automatic expert system in a format exemplified by the ECAT program.

One section of the analytical community that views the developments in expert systems with extreme interest is the pharmaceutical analyst. Over the last decade this group has fully appreciated the power of HPLC for multicomponent separations and has become a major user of the technology. The pharmaceutical analyst's interest in expert systems is not solely confined to HPLC, however; it extends to spectroscopy, where increases in computing power have led to improvements in data storage and computer-aided manipulation methods [17,18]. Leading on from this there is an increasing awareness that HPLC does not always present a robust and transportable analytical method for multicomponent drug systems. In this respect classical methods of spectroscopy with computer-aided techniques can offer rugged alternative possibilities. Luminescence methods have profited in this

area [19], despite the problems facing the analyst when considering drug compounds that are generally characterized by broad-band featureless spectra. This often leads to considerable spectral overlap, especially when analyzing for drugs, degradation products, and metabolites, often at the trace level.

The interfacing of a spectrofluorometer with a microcomputer has enabled the capture of total-luminescence data as an emission-excitation matrix (EEM) [20]. Data acquisition can be in the multichannel mode through utilization of vidicon or photodiode-array technology for rapid spectral collection [21], or through conventional collection of emission spectra at increments of excitation wavelength. As a result, numerous possibilities exist for the application of deconvolution routines in multicomponent analysis. Nevertheless, preliminary attempts at interrogating the data stored in the EEM to identify specifically useful areas of the matrix for deconvolution methods will illustrate how formidable this task is, even for the experienced analyst.

An expert system applied to total-luminescence data could open up new possibilities for handling and interpreting large matrices of luminescence data. Moreover, the capability could be developed, using digital deconvolution algorithms, to allow rapid interpretation of the data for component identification in multicomponent analysis.

An example of an early development of an expert system designed to run on a bench-top microcomputer is discussed in the next section. The expert system allows instrument control, data collection, manipulation, and a facility for explanation of sample identification [22,23].

2. EXPERT-SYSTEM DEVELOPMENT

If by definition an expert system carries out functions comparable to those of a human expert, it therefore follows that specialist expertise must be installed within the computer program. The accumulation of this expert information can be acquired through standard text or through input from one or more experienced workers in the field. The communication of knowledge obtained by interview or through interaction with the computer software is the foundation of a set of statements commonly known as the *knowledge base* and forms one of the key elements of an expert system.

2.1. The Major Components of an Expert System

Essentially there are three main components of an expert system: (1) a knowledge base, (2) an inference engine, (3) an explanation facility. The system (Figure 1) operates on a data set on the basis of a group of rules, networks, or information frames resident within the body of knowledge and applied through selection by the inference engine. Each step within the reasoning process should ideally be able to be accessed for information on progress, before moving off toward the final statement produced from the data handling.

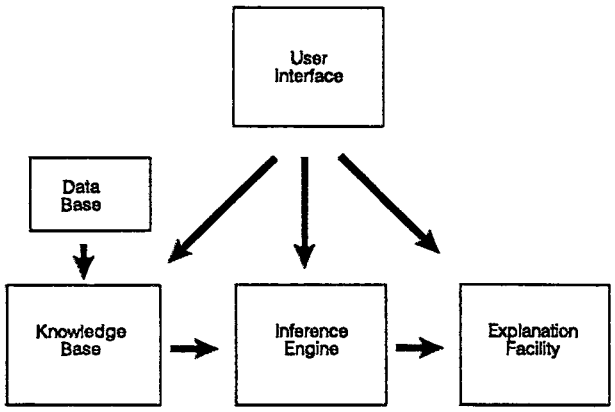


Figure 1. A typical expert system showing the main components.

Setting up a knowledge base can, however, be a laborious and time-consuming process. In interviews with those regarded as experts, the expression of particular reasoning decisions disseminated from large data sets is often difficult. Regularly, an empirical method is used, and there is, in general, a reliance on past experience. In addition to extraction of information, a decision has to be taken on the format for inclusion in the knowledge base.

Most frequently, a rule-based approach is operated, commonly known as a production system [24]. In this format data-directed forward chaining or target-directed backward chaining is operated, based on the logical structure: IF a situation arises, THEN action. Both methods use the same information, but the strategy is slightly different. Forward chaining is data driven and reasons from the available information, in this case luminescence data. It is founded on obtaining a final result by forward movement from one set of information to another until a particular objective is reached. This arises from the basis of satisfying a conditional proposition on which a final result depends.

By contrast, backward chaining is objective driven and looks for data to prove or disprove a given premise. The choice of format, whether forward or backward chaining, depends largely on the information available. In luminescence studies the requirement may be in two forms: in compound identification from a multi-component mixture, forward chaining would be most suitable; whereas in a quality control environment checks on identity may be the objective, and backward chaining would suffice.

In addition to a rule-based system for structuring knowledge, 'semantic nets' have been utilized. Their basis is the association of a number of facts in a netlike structure. In operation net systems are not totally devoid of a rule notation, which functions alongside the net format. A typical example is PROSPECTOR [25], a

successful expert system designed to seek out mineral deposits. A certain degree of rule introduction is also applicable by means of 'frames,' another of the knowledge representation schemes, where particular aspects of data information are stored, and such frames are compared with data in the data base.

Whichever format is established for dealing with the expert knowledge, an essential element is the *inference engine*, or interpreter, for controlling the sequence of operations. In a rule-based system this would be carried out through rule selection, and in semantic-net or frame systems, control would be based on inference decisions. For luminescence there is also the possibility of a considerable degree of instrument control, based on the information contained within the knowledge base.

In carrying out all these procedures within the confines of an expert system there is a continuing need to supply information to the user by way of explanation of the reasoning for following a particular pathway through the knowledge base for a data set. This user interface, or *explanation facility*, has a number of functions. In the initial stages it can be used to detect, diagnose, and correct errors that are present within the programs. But its major use is in the explanation of the action of the system, e.g., why a conclusion was reached and an alternative rejected. In answering a request of this nature a previous step is examined from the knowledge base, and the inference drawn at that stage is transmitted to the user.

2.2. Shell-Based Expert Systems

It can therefore be seen that the design of an expert system requires the close integration of these key elements. However, in construction it is often possible to circumvent the need for total development of all the elements. This can be carried out by utilizing an existing inference engine known as an *expert system shell*, which in certain circumstances can allow a more rapid route toward a working system. A typical example of a shell base is EMYCIN, which comes from MYCIN, a system developed for diagnosis of bacterial infection. EMYCIN has been successfully exploited as a base for numerous diagnostic systems. A shell system is generally constructed on a rule-based principle, which allows the addition or adjustment of the rules for different tasks.

Expert system shells are continually being introduced and updated for a wide range of computers, which extends the possibilities for finding a suitable shell for an application of interest. Nevertheless, there are certain drawbacks with regard to flexibility; in particular they are mainly rule based and backward chaining, and can be too specific. For applications in luminescence where unknown components are present, an expert system shell was not considered to be suitable.

2.3. Programming Languages

If an expert system is to be built from scratch, a decision has to be taken on a suitable programming language. This becomes a choice between the traditional

languages of BASIC, FORTRAN, etc., and the high-level artificial-intelligence languages, of which the prime members are LISP and PROLOG. Although traditional languages have the advantage of convenience, they have very little else to commend them and particularly suffer from the inability to handle logical programming requirements. Alternatively, the reasoning or logic programming languages of PROLOG and LISP are able to handle the multifaceted requirements of expert systems, from knowledge base to explanation facility.

Given that these languages are the most useful, it becomes a question of which one to use. LISP [26] was the first developed of these languages and was reported in the late 1950s. By contrast, PROLOG has only been available since the early 1980s and has the advantage of greater flexibility together with a number of other advantages [27].

3. AN EXPERT SYSTEM FOR LUMINESCENCE SPECTROSCOPY

The objective in constructing an expert system for luminescence spectroscopy was to access the large amount of data present in the emission-excitation matrix (EEM) (I_{em}, λ_{em}, λ_{ex}) for spectral data interpretation. Artificial intelligence techniques are well suited to handle the complexity of the EEM as it occurs when considering overlapping spectra from drug compounds. An expert system would permit the encapsulation of expert knowledge in digital form together with a number of deconvolution techniques for the extraction of characteristic features from the EEM. Any such system would have to incorporate a degree of automatic instrumentation control, data acquisition and storage facilities, and intelligent data-analysis techniques. Finally, it was hoped that the overall performance of an expert system in this area would exceed that of a human expert.

3.1. Design

The initial consideration in designing an expert system for luminescence data was the implementation of a suitable programming language. This presented an immediate problem in that although logic programming languages are regarded as most relevant for general control and decision making, they are not considered suitable for the large-scale numeric processing expected with luminescence data.

It was therefore concluded that two languages were required to handle the EEM data. Micro-PROLOG [28], a derivation of PROLOG designed specifically for microcomputers, was used to access the data in a compiled Pascal environment where numeric processing was completed.

This, however, generated a further problem in that it was not possible to link these languages together. One solution was to develop a novel 'load and execute' primitive, where a micro-PROLOG program loaded a compiled Pascal program into memory and after execution was completed returned to the micro-PROLOG environment. This allowed fast numeric processing of the EEM data.

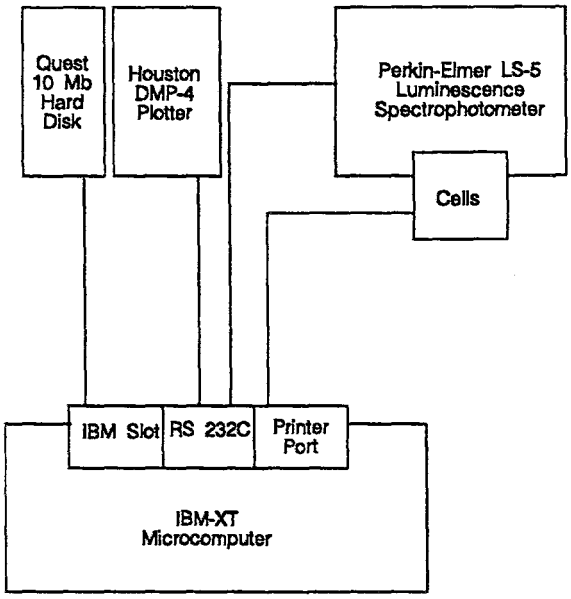


Figure 2. The hardware configuration used in the development of the expert system for luminescence data.

3.2. Instrumentation

An IBM-XT microcomputer was utilized with a 5¼-inch disk drive and a 10-megabyte hard disk for storage of the large amount of luminescence data generated to form the data base. The microcomputer controls a Perkin-Elmer LS-5 spectrofluorometer (Perkin-Elmer, Beaconsfield, U.K.) through an RS-232 C interface. The LS-5 was fitted with a four-position sample turret, which allows sample selection through stepper-motor control from the microcomputer. For graphics plotting a Houston DMP-4 Hiplot digital plotter (Bausch and Lomb, Texas, U.S.A) was used through a modified Centronics printer port.

3.3. Compounds Stored in the Data Base

The performance of the expert system was characterized with respect to a small repertoire (n = 10) of naturally fluorescent drug compounds. These consisted of warfarin, frusemide, atenolol, riboflavin, trimethoprin, chlorpromazine, estriol, chloroquine, propranolol, and disopyramide, which represent a wide range of classification of drug compounds. In addition a number of polynuclear aromatic hydrocarbons (PNAH) of toxicological interest were examined both as single substances and as multicomponent mixtures.

4. IMPLEMENTATION OF THE EXPERT SYSTEM

An initial consideration in setting up the expert system was the linking together of the various hardware components and the introduction of a necessary substructure to direct the spectrofluorometer control. The Perkin-Elmer LS-5 is a conventional dual monochromator spectrofluorometer from which an EEM can be generated by the collection of emission spectra at increments of excitation wavelength. A command structure was therefore required to set instrumental parameters, control the data capture, and handle storage to disk. To accomplish this a knowledge base was set up using input from human experts and also incorporating the generally accepted characteristics from fluorescence spectra.

From this it was apparent that data collection should consist of intensity information as integer values within the wavelength ranges; excitation, 200–400 nm, and emission, 220–600 nm. It must be said, though, that total data collection within these ranges is not always required, and single scans in emission or excitation or both would often be sufficient for single compound identification, such as in raw drug material identification. It was also important to consider the variables inherent in luminescence spectra due to temperature, concentration, solvent effects, and instrumental fluctuations.

Within certain limitations the system handles these difficulties by including a knowledge acquisition system to direct the collection of fluorescence data. In addition a facility within the knowledge base was used to investigate the data on the basis of using suitable identification routines. An essential part of this facility is the user interface.

Three identification routines were implemented;

1. single emission or excitation scans through the maximum intensity point, together with synchronous luminescence scans (known as fingerprints)
2. the presence of topographical features in the EEM plots of peaks, saddles, and valleys (Figure 3)
3. the use of reduced data matrices (Figure 4)

The incorporation of all these concepts and of relevant spectral information on candidate compounds into the knowledge base was achieved by developing a subprogram. This was designed to allow a human expert to interact with the acquisition program and set up the foundations for the ground rules for later inclusion in the knowledge base. Therefore, a 'learn system' was established and consisted of eight individual parts that could be accessed in isolation.

An important section of the learn system involved data and information checking by the human expert in order to obtain suitable information for inclusion in a library of candidate compounds. Once these aspects have been verified within the learn system, routine data can be introduced into the knowledge base.

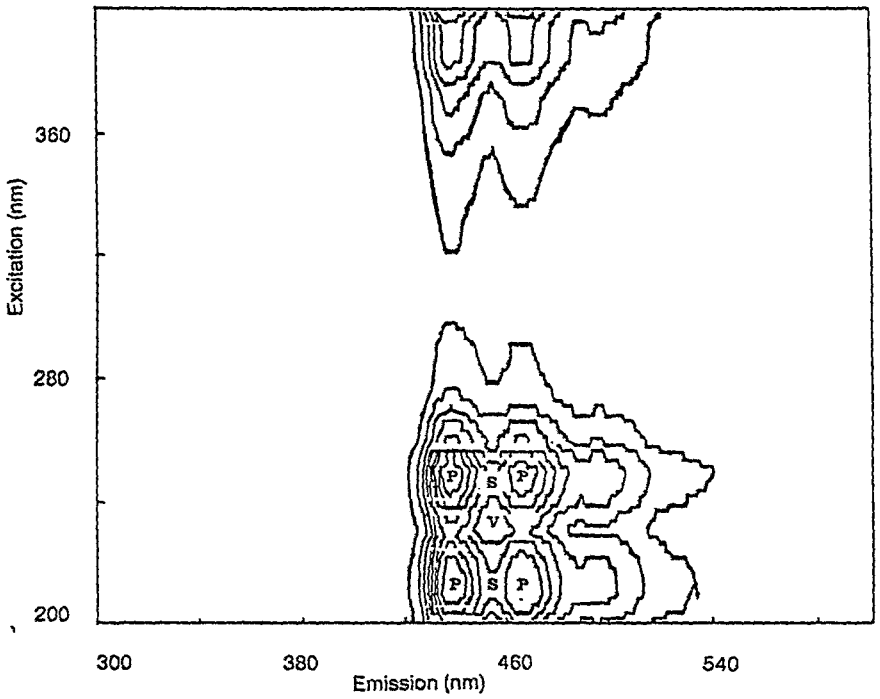


Figure 3. Characteristic topographical features used in the identification routines for perylene are (*P*) peaks, (*S*) saddles, and (*V*) a valley.

4.1. Drug Component Identification

The first step in any analysis is the sample preparation, which is essentially a manual operation. To simplify this step a standard solvent mixture of methanol–water (50 : 50, v/v) was used whenever possible, with variation in the proportions to suit the particular compound of interest.

A small quantity of the sample (<5 μg/mL) was dissolved in a suitable solvent, transferred to silica cells, and inserted into the sample compartment. From this point the expert system takes over and sends commands to the spectrofluorometer to begin operation in rapid-scan mode for a single excitation and emission spectrum known as a 'prescan.' The primary function of this procedure is to obtain a rapid fix on the maximum intensity in the EEM for scaling purposes. If the intensity level is too high, a message to that effect is generated for the operator's convenience. In addition to scaling to 90% full scale, scan speed and filter response are set at this stage.

Emission spectra can then be accumulated at specified increments of excitation wavelength. Intensity data are sampled every 0.5 nm to give an EEM over the

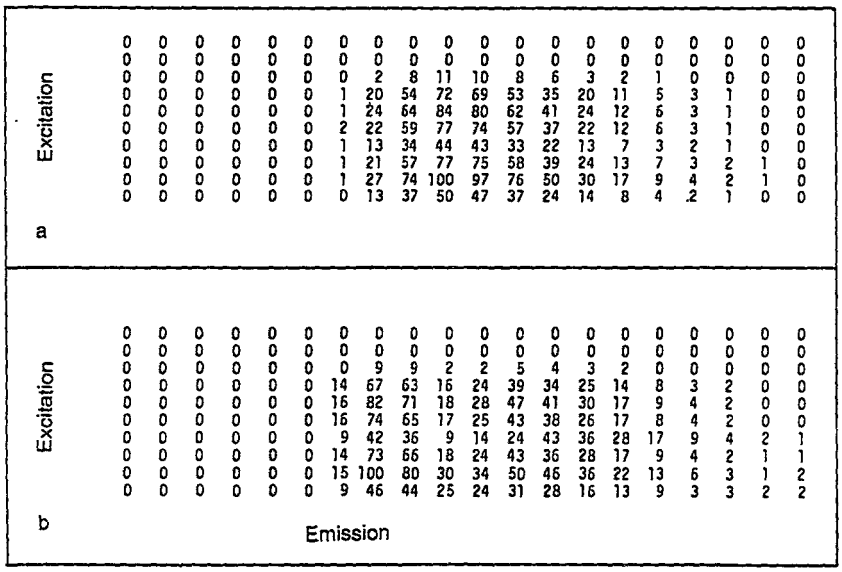


Figure 4. Reduced data matrix types for the drug warfarin; (*a*) density matrix, (*b*) activity matrix.

wavelength ranges stated previously and consisting of a 761 × 41 matrix of intensity values. After completion of sample data collection the sample-cell turret is rotated automatically by stepper motor to the reference position and background data are collected at each point. Information at each spectral point for the reference is then subtracted from the equivalent sample data. The spectral information for the compound under test can then be permanently stored on the hard disk (Figure 5).

The next stage in the process involves the expert system's characterizing suitable techniques for effectively assisting in the unknown substance identification. It is generally desirable that the methods chosen rapidly estimate possible candidates in order to reduce data collection to a minimum. Typical examples in which speed of identification is a prerequisite are drug identification in overdose cases or rapid checks of the identity of raw drug materials on receipt from an industrial supplier. In such cases an initial estimate of a candidate compound can be obtained by comparing data from a scan through the spectral maximum (either in the plane of the excitation or of the emission wavelength) with that from the data-base spectra. To confirm identity the methods of single scan, distinctive features, and reduction of data matrix (EEM) are introduced.

Although the EEM gives a global view of intensity data, for fast processing an identification can be achieved by a reduction of the data to a more manageable size. The data as collected in a 741 × 41 matrix can be reduced to each of four matrix

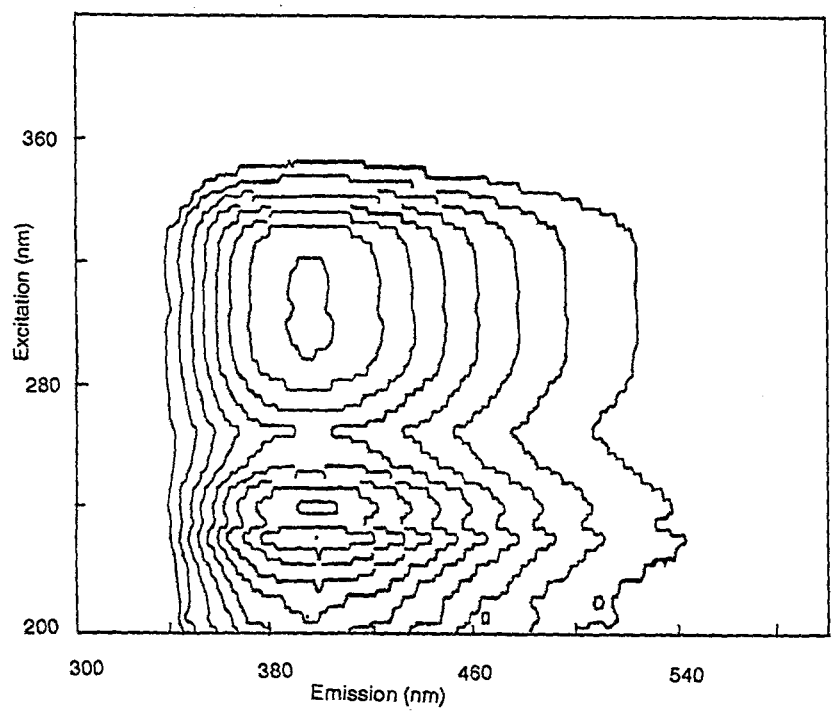


Figure 5. Emission-excitation matrix acquired for warfarin and stored in the data base.

sizes (80 x 40, 20 x 10, 10 x 5, or 5 x 5). However, it has to be accepted that any reduction in matrix size is likely to lead to a loss of detail. In the 80 x 40 matrix a representation of a 5-nm square is presented and therefore very little information is lost, whereas a 5 x 5 matrix represents a gross reduction in size and is only used for a rapid comparison. To maximize the differences within these matrices two different matrix forms were introduced, a 'density' and an 'activity' matrix, which are utilized in normalized format. The former is simply a reduction of the larger data matrix such that each matrix value is the mean of all the intensities within a particular square or rectangular area (**A**, Figure 4), whereas the activity matrix represents the difference between the minimum and maximum intensity values (**B**, Figure 4). In other words it comprises a range of values within a particular area. The density matrix is complementary to the activity matrix, and when used together (as illustrated in Figure 4) they allow the identification of warfarin from the data base.

In operation there is a progression from the smallest matrix size using the two formats through to the largest. The number of possible matches of an unknown with

the data base is subsequently reduced until the best possible match is given. Together with the reduced format of the EEM the data matrix can also be interrogated for characteristic features of peaks, valleys, and saddles, as shown in Figure 3. A peak is recognized by the expert system as a point where excitation and emission scans locate a turning point in both directions around a maximum. A saddle is represented by a point of maximum intensity in one direction and minimum intensity in the other. In contrast, valleys are slightly less common and can be viewed as turning points at minimum intensity or simply as a hole in the data.

To reduce the number of comparisons required in examining for these features an 'importance value' was assigned by a human expert. Such a value is effectively a weighting, such that if a certain feature in an unknown matches with a feature of a known the importance value will reflect its significance.

The last of these methods is the use of fingerprints, which are single scans, similar to those completed in an initial estimate but with the exception that they either slice through the data matrix at an angle to give a characteristic spectral profile or by differentiation, which yields a derivative spectrum in the emission or excitation planes [29]. In cutting through the data matrix at any angle a variable-angled synchronous luminescence scan is effectively obtained [30], in which the angle of trajectory through the data matrix can be chosen to give the most significant synchronous spectrum for the compound of interest.

Often, to make suitable comparisons between test data from an unknown compound and information in the data base, a number of the methods highlighted are introduced by the expert system. One proviso in carrying out these comparisons by a least squares best-fit measurement is the allowance of suitable error limits between the spectral data by accommodation of shifts in the positions of features. These shifts can result from a number of variations and include differences in concentration resulting in the inclusion of erroneous factors such as instrument noise. In this respect a limit of up to 10 nm was imposed within the knowledge base. This limit, although apparently large, should not have a significant bearing on compound identification, as it is unlikely that a single feature would be used in isolation for component identification.

For a single substance the expert-system process develops from the first stage of an initial estimate of identity through to the identification methods for distinguishing characteristic features.

To assess the effectiveness of the expert system for single-component identification the ten drugs in the data base were treated in turn as unknowns. In all cases a degree of certainty in identification was achieved and this applied even when similar spectral properties existed. This test is adequate to illustrate the usefulness of the expert system for single-component identification, which could be applied in a quality assurance laboratory, where raw materials and bulk drugs are examined.

4.2. Multicomponent Samples

In addition to considering single-component identification, it is important to examine the applicability of the expert system to multicomponent mixtures. Here, the advantages of an expert system with regard to substance identification are increased when compared with deliberations made by a human expert from the complexity of the data matrix. Fundamentally varying degrees of spectral overlap come into play, in which swamping of the emission intensity of a component can easily be observed, due either to concentration or quantum yield differences in the components. These factors can present major problems to a human expert not only in identifying the sample but in deciding the number of substances present in a mixture.

Therefore, to handle mixtures there is a requirement for an expansion of the search and identification strategies. The data collection step progresses as previously, but it is no longer sufficient to apply the three different identification techniques without modification. This is primarily due to the reduction in effectiveness of the information gained as a result of spectral overlap. The identification methods are not, however, discharged, with the exception of the initial estimate, which is no longer feasible.

The initial step after data collection is to examine the topographical features of the matrix and classify the candidates into 'probable' and 'possible' components. The expert system then moves onto a matrix manipulation process in which the probable and the possible candidates suggested from feature detection are examined. The largest reduced matrix is utilized at this stage with the inclusion of the activity and density matrix formats. The purpose of these formats is to assess specific areas of the reduced EEM where the candidate components give fluorescence intensities. It could be suggested at this point that a more successful method for handling the EEMs would involve building a matrix of individual EEMs and comparing it with the unknown data matrix. However, although this method has a certain degree of elegance, it is not at present feasible within the confines of the knowledge base. The matrix manipulation should yield a further set of probable and possible assignations, which are considered at the next stage. Additional evidence is then obtained by using characteristic spectral profiles as fingerprints in the form of variable-angled synchronous scans, which are compared with data-base spectra. The information gained in these methods can be assimilated to provide a final set of probable and possible candidates. These are then reported to the operator together with supporting explanations for the choice of candidates.

As an example to test the performance of the expert system, a number of polynuclear aromatic hydrocarbons (PNAH) were mixed, and after data collection and storage, the deconvolution processes were applied by the expert system (Figure 6). The first step was to examine for topographical features, which immediately illustrates the problems faced in extracting suitable information from the data matrix by the expert system (Table 1). For a six-component PNAH mixture the

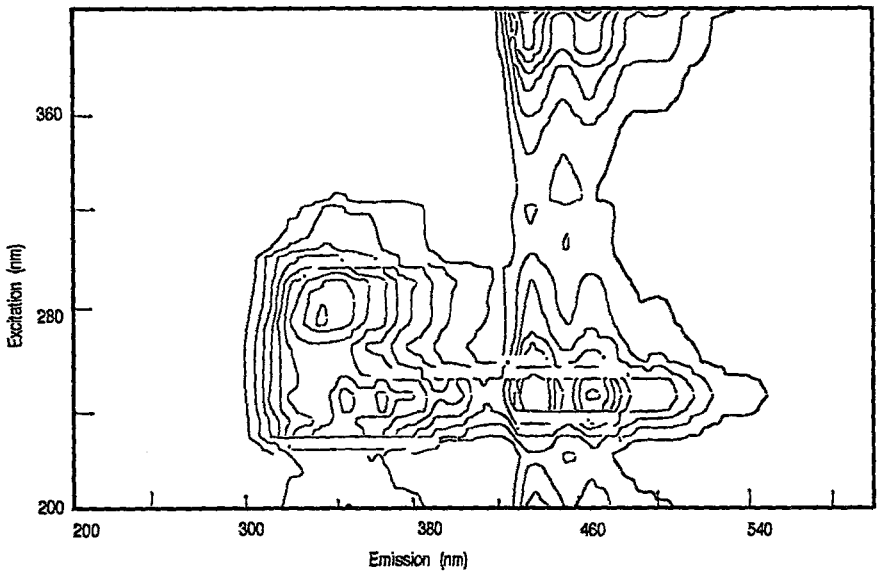


Figure 6. Total luminescence plot for the six-component PNAH mixture of anthracene, perylene, diphenylmethane, fluorene, carbazole, and phenanthrene.

expert system was unable to locate more than a few of the expected features. However, on progressing through the different identification routines, improvement in the matching of the mixture components with those stored in the knowledge base was achieved. On completion of these methods all the expected substances in the mixture were identified as probable or possible candidates without any additional spurious components included. In contrast, visual examination by human experts revealed two probable candidates (diphenylmethane and perylene), three possible substances, and two substances that were not present in the initial sample preparation (biphenyl and indene).

In addition to the six-component mixture which was considered the limit of capability for the expert system, a number of other mixtures containing fewer components were assessed. The results obtained by the expert system were favorable in most cases, whereas the comparison with the manual method of assessing contour presentation, feature detection, and fingerprinting data gave poorer results in all the mixtures examined. In investigating the mixture data the comment must be made that the mixtures represent an idealized system in which individual concentrations were adjusted to give relatively large intensity values for each of the components, a situation not generally found in practice.

Table 1. Comparison of the results from the analysis of the six-component PNAH mixture carried out manually by a human expert and by the developed expert system.

	Probable Components	*Possible Components*
Manual analysis by human expert	Diphenylmethane Perylene	Biphenyl Anthracene Fluorene Indene* Phenanthrene
Expert system	Anthracene Diphenylmethane Fluorene Perylene Phenanthrene	Carbazole

*Additional components found, but not present in the six component mixture.

5. CONCLUSIONS

The expert system as developed was introduced to perform component identification on drugs and related compounds that exhibit significant natural fluorescence. The philosophy behind the expert-system design was to form the basis for the establishment of a much larger expert-system program for application both in classical spectroscopy and in chromatography.

It was considered important to base the expert system on a bench-top microcomputer for convenience. Continuous developments indicate greater memory and storage capacities in the future.

It has been possible to demonstrate with the limited range of compounds examined that considerable advantages exist in terms of the speed of data interpretation and data deconvolution by using an expert system. At present the data-matrix deconvolution methods in operation could be considered fairly fundamental. This leads to the possibility of limited use when applied to authentic samples, in which fluorescence intensities will vary considerably. In this respect it would be reasonable in the future to extend the system to the more complex matrix mathematical manipulations of the EEM data. Specifically, this would involve consideration of the methods of principal-component analysis and rank annihilation [31], which have been shown to be extremely powerful data-manipulation methods for the deconvolution of multicomponent spectral data [32]. In the present system, however, the introduction of these methods is precluded until an upgrading

of the microcomputer is carried out to increase memory capacity and computer speed.

A further advance would entail the introduction of a rapid-scanning spectrofluorometer of the vidicon or photodiode-array type, with particular emphasis on the advantages of a two-dimensional configuration. In this respect the vidicon or possibly the forthcoming charge-coupled devices would effectively offer real-time data acquisition in the emission and excitation planes.

The expert system as implemented does present a useful tool for the analysis of luminescence information obtained from drug and related compounds. Although the expert system for luminescence is some way from solving the general problem of identification of any particular drug that fluoresces, or for the identification of drugs in accidental or overdose poisoning cases, it does form a useful basis for further development in the pharmaceutical field.

REFERENCES

1. Goodall, A., *The Guide to Expert Systems*, Learned Information, Oxford, 1985.
2. Winston, P. H., *Artificial Intelligence*, 2nd ed., Addison-Wesley, Reading, Mass., 1984.
3. Feigenbaum, E. A., Proc. 5th Intl. Conf. Art. Intel., 1977, 1014.
4. Kiesewetter, M., Lemke, K., Wernstedt, J., and Moericke, R., Int. Wiss Kollog-Tech. Hochsch. Ilmenay, 30, no. 2 (1985), 149.
5. Buchanan, B. G., and Feigenbaum, E. A., Artif. Intel., 11 (1978), 5.
6. Ting, K. H., Lee, R. C. T., Milne, G. W. A., Shapiro, M., and Guarino, A. M., Science, 180 (1973), 4084, 417.
7. Gray, N. A., Buchs, A., Smith, D. H. and Djerassi, C., Helv. Chim. Acta, 6, no. 2 (1981), 458.
8. Gribov, L. A., and Elyashberg, M. E., CRC Crit. Rev. Anal. Chem., 8, no. 2 (1979), 111.
9. Drouen, A. C., Billiet, H. A. H., and De Galan, L., J. Chromatogr., 352 (1986), 127.
10. Berridge, J. C., *Techniques for the Automated Optimisation of HPLC Separations*, Wiley, Chichester, 1985.
11. Musch, G., De Smet, M., and Massart, D. L., J. Chromatogr., 348 (1985), 97.
12. Musch, G., and Massart, D. L., J. Chromatogr., 370 (1986), 1.
13. Gunasingham, H., Srinivasan, B., and Anauda, A. L., Anal. Chim. Acta, 182 (1986), 193.
14. Bach, R., Karnicky, J., and Abbott, S., ACS Symp. Ser., Artif. Intel. Appl. Chem., 306 (1986), 278.
15. Bridge, T. P., Williams, M. H., and Fell, A. F., Chemistry in Britain, 23, no. 11 (1987), 1085.
16. Lu, P., and Lu, X., J. Chromatogr., 292 (1984), 169.
17. Gribov, L.A., Anal. Chim. Acta, 122 (1980), 249.
18. Janssens, K., and Van Espen, P., Anal. Chim. Acta, 184 (1986), 117.
19. Miller, J. N., Analyst, 109 (1984), 191.
20. Giering, L. P., and Hornig, A. W., Amer. Lab., 11 (1977), 113.
21. Talmi, Y., Baker, D. C., Jadamec, J. R., and Saner, W. A., Anal. Chem., 50 (1978), 936A.
22. Milne, K. T., Williams, M. H., Clark, B. J., and Fell, A. F., Anal. Proc., 23 (1986), 157.
23. Milne, K. T., Williams, M. H., Clark, B. J., and Fell, A. F., Proc. 6th Intl. Workshop, Expert Systems and Their Applications (Avignon), 2 (1986), 1203.
24. Davis, R., and King, J., in E. W. Elcock and D. Michie (eds.), *Machine Intelligence*, Vol. 8, p. 300, Ellis Horwood, Chichester, 1977.
25. Duda, R., Gaschnig, J., and Hart, P., in D. Michie (ed.), *Expert Systems in the Microelectronic Age*, p. 153, Edinburgh University Press, Edinburgh, 1979.

26. MacLennan, B. J., *Principles of Programming Languages*, Holt, Rinehart & Winston, New York, 1983.
27. Warren, D., in D. Michie (ed.), *Expert Systems in the Microelectronic Age*, p. 112, Edinburgh University Press, Edinburgh, 1979.
28. Clark, K. L., and McCabe, F. G., *Micro-Prolog Programming in Logic*, Prentice-Hall, London, 1984.
29. Miller, J. N., Ahmad, T. A., and Fell, A. F., Anal. Proc., 19 (1982), 37.
30. Clark, B. J., Fell, A. F., Milne, K. T., Pattie, D. M. G., and Williams, M. H., Anal. Chim. Acta, 179 (1985), 35.
31. Warner, I. M., Christian, G. D., Davidson, E. R., and Callis, J. B., Anal. Chem., 49 (1977), 564.
32. Ho, C. N., Christian, G. D., and Davidson, E. R., Anal. Chem., 53 (1981), 92.

82. J.L.MURRAY and M.H.WILLIAMS, Knowledge based systems in process planning and assembly design, in 'Geometric Reasoning', Ed. J.Woodwark, Clarendon Press, Oxford, 217-236 (1989).

13 Knowledge-based systems in process planning and assembly design

J. L. MURRAY and M. H. ~~MILLER~~ WILLIAMS

13.1 Introduction

Engineers are concerned with the creation and manufacture of products. In general a product consists of a number of components and sub-assemblies linked in a unique manner to satisfy a particular consumer need; each component can be described unambiguously in terms of its geometry and material. Currently there is considerable interest in the development of knowledge-based, or expert, systems to assist in both the design and manufacturing phases of products.

Although some of the simpler decision-making processes, such as material selection, may be handled by knowledge-based systems with no geometric capability, much of the decision making requires some knowledge of, and reasoning about, the geometry of the artefacts to be produced. This affects the design phase in such areas as component selection and tolerance determination and the manufacturing phase in aspects including process planning and robotic assembly.

The obvious way of handling geometric information is through the use of three-dimensional modelling systems. There are three fundamentally different approaches used by modellers, wire-frame, surface and solid; of these solid modelling is the only one which provides an unambiguous representation of the geometry.

Unfortunately, the capabilities of existing modelling systems are limited from the point of view of linkage to and use by a knowledge-based system. For example, suppose that one wishes to construct a system which needs to interrogate and manipulate a solid model via a three-dimensional modelling system. The potential for such interaction provided by current modelling systems is very limited.

Clearly if knowledge-based systems are to be developed for effective decision-making in CADCAM, it is desirable that three-dimensional modelling systems should change. In particular, it is important that

more of the lower-level geometric knowledge is moved into the modeller so that it can reason about aspects of the geometry of the model, such as the classification of features and the storage and manipulation of the relationships between components of an assembly. It is also essential to provide a more adaptable interface by which knowledge-based systems can communicate with such modellers.

This paper looks at two applications with requirements for more intelligent three-dimensional modelling systems. The first is in the area of process planning and the second in the design of assemblies.

13.2 A knowledge-based system for process planning

Background

Operations planning is the task of selecting and sequencing machines and processes whereas **process planning** involves the detailed technological selection of individual operations on each machine. The two together are fairly complex tasks which until now have been performed exclusively by human planners. However, for several reasons there is growing interest in creating knowledge-based systems to assist the planner in these tasks. For example:

- Van't Erve and Kals [5] report that, as a result of decreasing product lifetimes and increasing variety of products, batch sizes in part manufacturing have been falling steadily. This places increasing pressure on planners in order to maintain the level of production.

- Davies et al. [3] note that in many companies the average age of process planners is over fifty and that soon there will be a shortage of process planners with the necessary knowledge.

- The knowledge used by process planners represents years of accumulated experience. As planners retire their expertise will gradually disappear. The capture and formalization of such knowledge could stop this erosion and enable the knowledge to be extended.

In view of this and the fact that these processes can be costly and prone to error the authors are collaborating with Ferranti Infographics and Ferranti Defence Systems in the development of knowledge-based systems which will assist planners in these processes. The first phase of this venture involved the development and evaluation of a fully automatic machining system for the data generation and manufacture

of precision 2½D milled parts [1]. The current programme of work is aimed at extending this system through a knowledge-based approach which involves linking a knowledge-based system with a solid modeller.

Design discipline

The planning and programming systems outlined in the following sections must been seen against the design and manufacturing environment of Ferranti Defence Systems.

Within the company a strict code of design discipline is adhered to. Standardization is regarded as essential and plays a key role in facilitating the design of components capable of performing complex functions. Wherever possible, the parts are constrained to be manufactured from cuboidal blanks of pre-stretched aluminium alloy machined from one or two opposing faces using cutters limited to four diameters and three lengths. Non-standard machining operations are strongly discouraged although in some cases they are essential. Complex structures or components may be built by riveting or dip-brazing an assembly of 2½D parts. In manufacture the range of machine tools, cutters and clamping systems are all minimized.

Many of the components manufactured are complex in shape, produced to tight tolerances and made in small batches. A cellular or group technology approach has been effectively implemented. During the development stage of a product, design changes are frequent and the lead time short. This places exacting demands on the planners who need to respond quickly and accurately to changes in design.

13.3 The automatic machining program (AMP)

Part-programmers generally set about their task by examining the detailed drawing, deciding on the method of holding and the machining sequence, and then extracting the dimensional information required to determine the cutter path. Computer programs which assist part-programmers will generally automate the geometry extraction phase, leaving the other decisions to be taken by the programmer on the basis of experience or calculations. The resulting solution is very much dependent on the part-programmer's ability from the point of view of both accuracy and efficiency.

With the strict enforcement of standard approaches, the decision regarding the methods of holding is simplified as the work-holding strategy reduces to a choice between elementary clamping and bolting to a standard platen. Some ten years ago it was decided to try to

capitalize further on the improved design discipline to develop a package which automates the planning and part-programming processes for 2½D parts. At the time it was perceived that, although there had been a number of attempts (e.g. the CUBIC system developed by Stolenkamp et al. [4] and the COBASH system of Bosch [2]), no suitable commercial system existed for dealing with prismatic parts. The intention was to develop a fully automated planning and programming package which would generate complete manufacturing information from an input file including part geometry, material and inspection data. The package would derive:

- The group technology cell most appropriate for the manufacture of the part.
- The blank size selection from a standard range.
- The sequence of manufacturing and inspection operations required together with the estimated completion times for each operation.
- The best method of component clamping of the blank and its location on the platen.
- A full package of information for the validation of the selected process and for issue to the shop-floor for part manufacture and inspection.

Although it was hoped that the geometry of the component could be captured from design layouts, in general a detailed drawing provided a more suitable starting point. the geometry is expressed as a series of closed or open 2D profiles, starting with the outer boundary of the part. With each profile is associated a pair of depths (Z coordinates) which determine its vertical extent. In addition the position of each hole or group of holes is specified.

Tolerance data and notes can be appended. In order to verify that the input data are correct, check plots are generated for each face of the component. An example of typical output produced by the system is given in Fig. 13.1. If requested by the user, inspection features can be highlighted on the plots.

13.4 Machining strategy used in AMP

A modular approach has been followed in developing the machining strategy for components. This can be regarded as being made up of six modules.

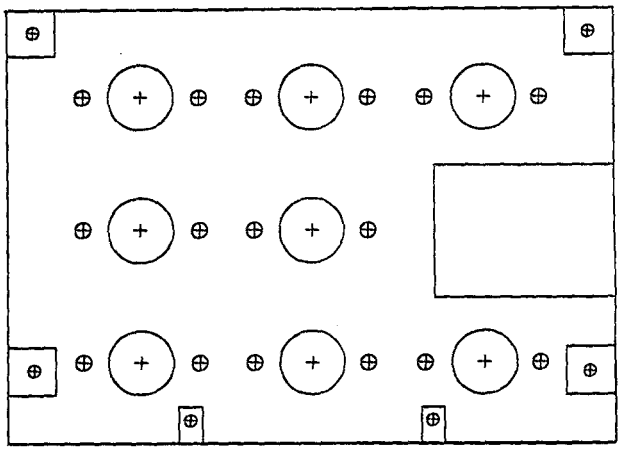


Fig. 13.1

Method of holding

Three methods of blank and workpiece holding were investigated: machine clamps, vice and bolting. Of these, bolting to standard platens incorporating a grid of tapped and reamed holes was generally preferred with machine clamps as a permitted alternative. Although the former approach necessitates larger blanks, it removes the problem of clamp relocation and avoidance.

Blank sizing

When blank dimensions are computed, allowances are made for the method of holding selected, together with the most appropriate cutter diameter for parting-off the finished component.

Profile division

The initial task of this module is to determine the location of the attachment points of the finished machined component to the blank frame. In the case of a complex part, different strategies may apply giving rise to different solutions to the problem. The choice is then made on the basis of ease of access and the rigidity of the workpiece during the machining operations. As a general heuristic the spread of the attachment points should be as large as is practicable.

Bolt positioning

If bolting is the selected method of attachment, this module identifies the tapped holes in the platen most suitable for blank attachment. The

hole locations are chosen with a view to minimizing the blank size together with the provision of maximum rigidity during the machining process.

In machining, detachable waste can be a serious safety hazard. This is particularly so where the component is machined from both sides or recesses occur in the outer profile. This module aims to identify such occurrences before making a decision to between bolting or area clearance of the offending waste.

Dowelling

Double-sided components demand accurate location of the two machining sequences. This is achieved by the insertion of two dowels in an area of waste material which locate in reamed holes in the platen. This module contains the rules for the selection of the dowel position.

The machining strategy is based on using the largest possible standard cutter to remove the bulk of the metal before reverting, if necessary, to smaller cutters for finishing or to provide the finer details. For double-sided components the second face is machined after turn-over.

The output from the AMP is in a neutral data format which is post-processed to the chosen machine tool. The post-processors used incorporate the necessary technological knowledge to select the feeds and speeds appropriate to the to the selected material, cutters and machine tool.

Verification and planning documentation

A range of check plots of component geometry and cutter centre paths are output (Fig. 13.2). Both line drawings and ink-jet coloured plots are used as aids to verification.

In addition AMP produces a comprehensive set of planning documentation to control the various stages of the manufacturing process. The set includes:

- The master planning sheet detailing the component identification, material, group technology cell selected together with a list of the blank preparation and machining operations (Fig. 13.3).
- Plots showing the blank preparation details (Fig. 13.4) and the set-up procedures for each side to be machined, including dowelling locations for the second side.

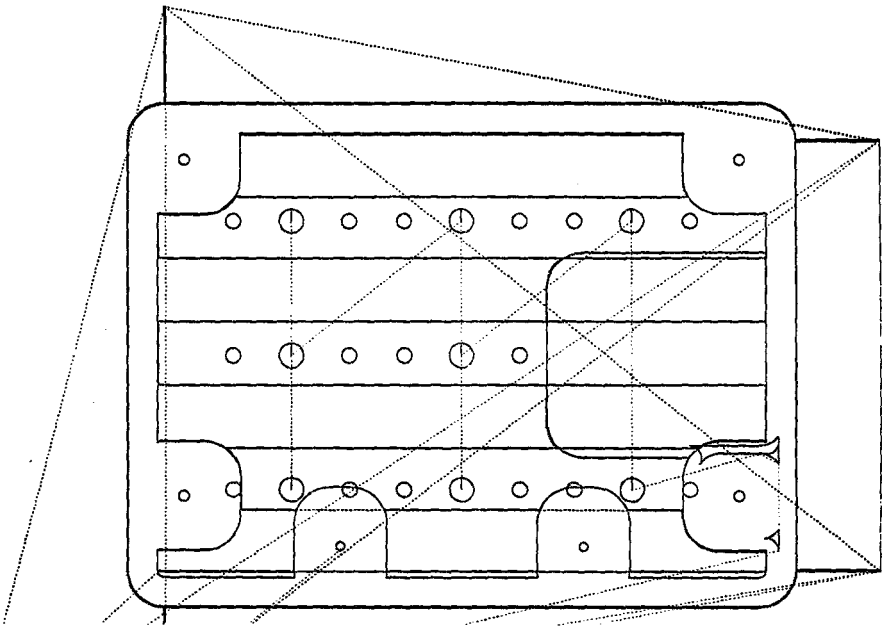


Fig. 13.2

- Operator instruction sheets giving machining datums, tool descriptions, minimum tool projections, drive tape sequences and the tape running time (Fig. 13.5).
- The part-off sheet showing the partly machined component in location on the platen and providing instructions on removing the bolts or clamps during the part-off sequence (Fig. 13.6).

Appraisal of AMP

AMP has clearly demonstrated the feasibility of developing a computer package to perform the functions of a part-programmer for the manufacture of 2½D parts from cuboid blanks. The program has a number of limitations including the following:

- Since the knowledge used has been encapsulated in a FORTRAN program rather than being stored as a separate body of rules, it is difficult to update and extend the system to cater for new knowledge or to deal with a wider range of components.
- The geometry of the parts is held in a relatively simple form, which although adequate for a wide range of 2½D components manufactured from blanks, is not suitable for extension to preformed components, such as castings and forgings.

MACHINE SHOP COPY

MASTER PLANNING CONTINUED		
OP NO.	OPERATION DETAILS	TAPE/TOOL DESCRIPTION
10	CUT BLANKS TO SIZE	NON FERROUS SAW
	9.125" X 7.125" TOL +-1/8"	
20	FLYCUT BOTH SIDES TO THICKNESS	MASERATI/CINCI 2V
	1.580" +-.003"	
	FLAT AND PARALLEL TO .003 F.I.M.	
30	DRILL BLANK AS PS 3	BLANK DRILL
40	SET UP AS PS 4	MATCHMAKER CNC
	MACHINE AS PS 5	11259/101BT10
50	SET UP AS PS 6	MATCHMAKER CNC
	MACHINE AS PS 7	11259/101BT20
60	DEBURR	DEBURRING BENCH
70	INSPECT OPERATIONS 10 TO 60	INSPECTION DEPT
80	TAP ALL HOLE(S) AS DRG	ESSEX TAPPER 2BA
90	EXAM ALL OPERATIONS COMPLETE	INSPECTION DEPT
	PS 11259/101B SHEET 2	ISSUE 1

Fig. 13.3

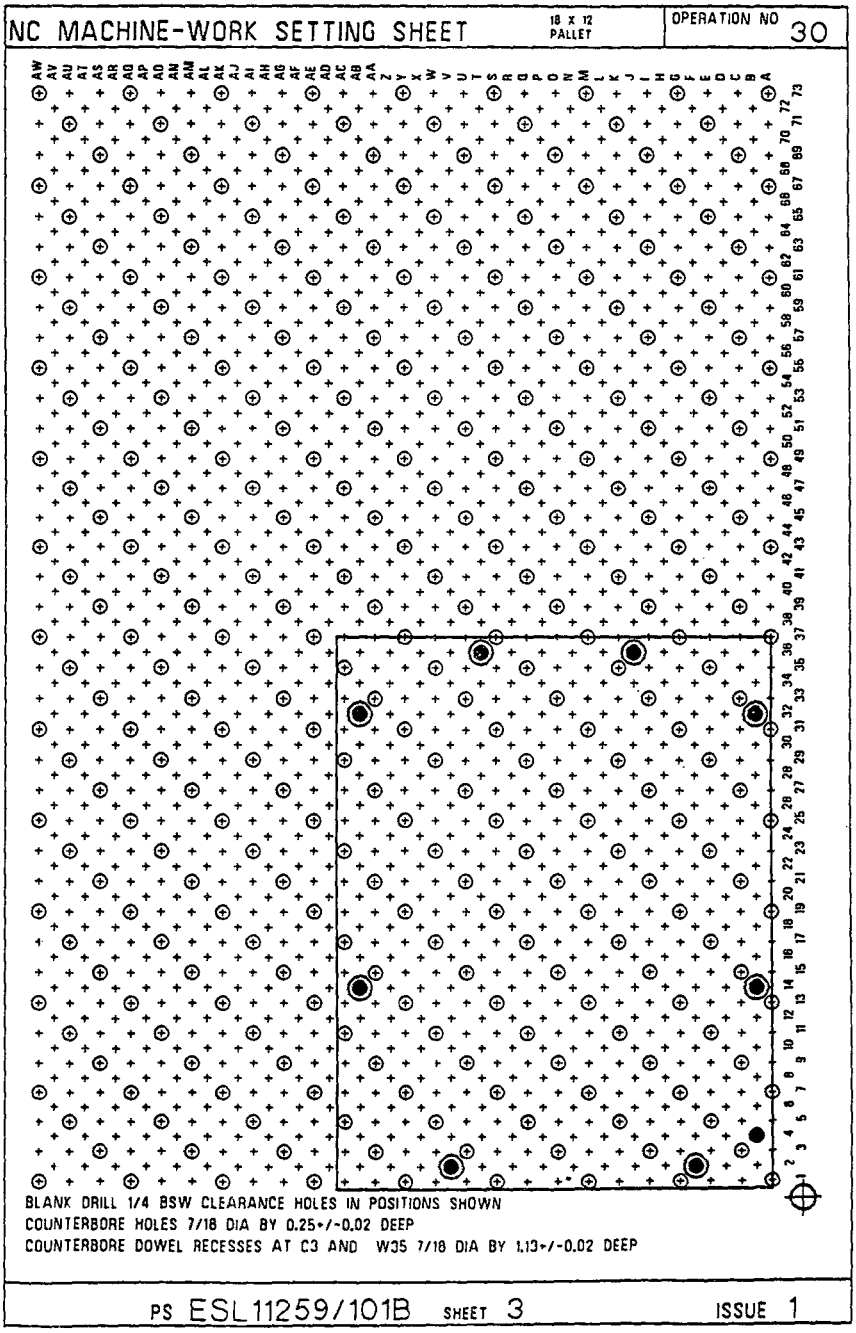

NC MACHINE-WORK SETTING SHEET
18 X 12 PALLET
OPERATION NO 30
BLANK DRILL 1/4 BSW CLEARANCE HOLES IN POSITIONS SHOWN
COUNTERBORE HOLES 7/18 DIA BY 0.25+/-0.02 DEEP
COUNTERBORE DOWEL RECESSES AT C3 AND W35 7/18 DIA BY 1.13+/-0.02 DEEP
PS ESL11259/101B SHEET 3 ISSUE 1

Fig. 13.4

JOB NUMBER : ESL)1259/1018T2D 1550 CREATED : 4-JUN-1985 11:11:11
OPERATION NO. 50 MACHINE : POSIDATA CNC 2800
MATERIAL : ALUMINIUM PLANNER : D ROBERTSON

XY DATUM : STANDARD MACHINE SET POINT

Z DATUM : 3 IN ABOVE PLATTEN/BACKING SHEET

MINIMUM PACKING THICKNESS : 0.087

MAXIMUM Z MOVEMENT : 2.993 X LIMITS : 0.000 9.300 Y LIMITS 0.000 7.625

ESTIMATED RUN TIME : 46.39 MINUTES

TLC NO.	DIA NO.	BLOCK NO.	TOOL DESCRIPTION	MIN.PROJ. INS.	SPEED R.P.M.	TIME MINS.	NO.OF HOLES	HOLE REF.	LOW LIMIT	HIGH LIMIT
1	-	5	0.750 IN. SLOT DRILL	1.50	2292	13.11				
2	-	2305	0.500 IN. SLOT DRILL	1.13	3438	2.34				
START OF DRILLED HOLES SEGMENT										
3	-	3060	NO.1 CENTRE DRILL		3600	2.99	22			
4	-	3550	4.00 MM. DRILL		3600	1.41	16	C	0.1567	0.1626
5	-	3920	2.80 MM. DRILL		3600	1.50	4	B	0.1094	0.1154
6	-	4290	2.30 MM. DRILL		3600	0.92	2	A	0.0898	0.0957
1	-	4530	0.750 IN. SLOT DRILL	1.50	2292	4.77				
7	-	4980	0.750 IN. LONG SERIES SLOT DRILL	3.22	2292	6.84				
RECLAMP AS PER PLANNING SHEET B										
1	-	5785	0.750 IN. SLOT DRILL	1.50	2292	2.83				
7	-	6110	0.750 IN. LONG SERIES SLOT DRILL	3.22	2292	7.89				

Fig. 13.5

- Since there is no in-built explanation facility within AMP, it is difficult to obtain meaningful feedback from users on improvements to be embodied as a result of experience.
- It is difficult to assess the effect of aspects of the geometry, for instance thin walls and detachable waste, on the cutting and clamping strategy.
- The success achieved and the identifiable potential for improvement led to the decision to proceed to the second phase of the work.

An intelligent knowledge-based system approach

In an attempt to overcome the major limitations of AMP, the second phase of the work is concentrating on the creation of a part-programming knowledge-based system which will link directly to and will communicate with a solid modeller. The knowledge about the existing part-programming process is being reformulated as a separate set of rules in a form which is understandable to the user. In the light of experience gained, the rule base will be updated and extended. An important feature of the new system will be an explanation facility which, when challenged, will explain to the user the rules used to reach a decision. This is essential for the refinement of the rule base.

The solid modeller provides a well structured and easily modifiable representation of the geometry of the workpiece and other associated hardware, including jigs, fixtures and cutting tools. Current software available can be used to modify the models as required. A major part of the project will concentrate on creating an effective method of

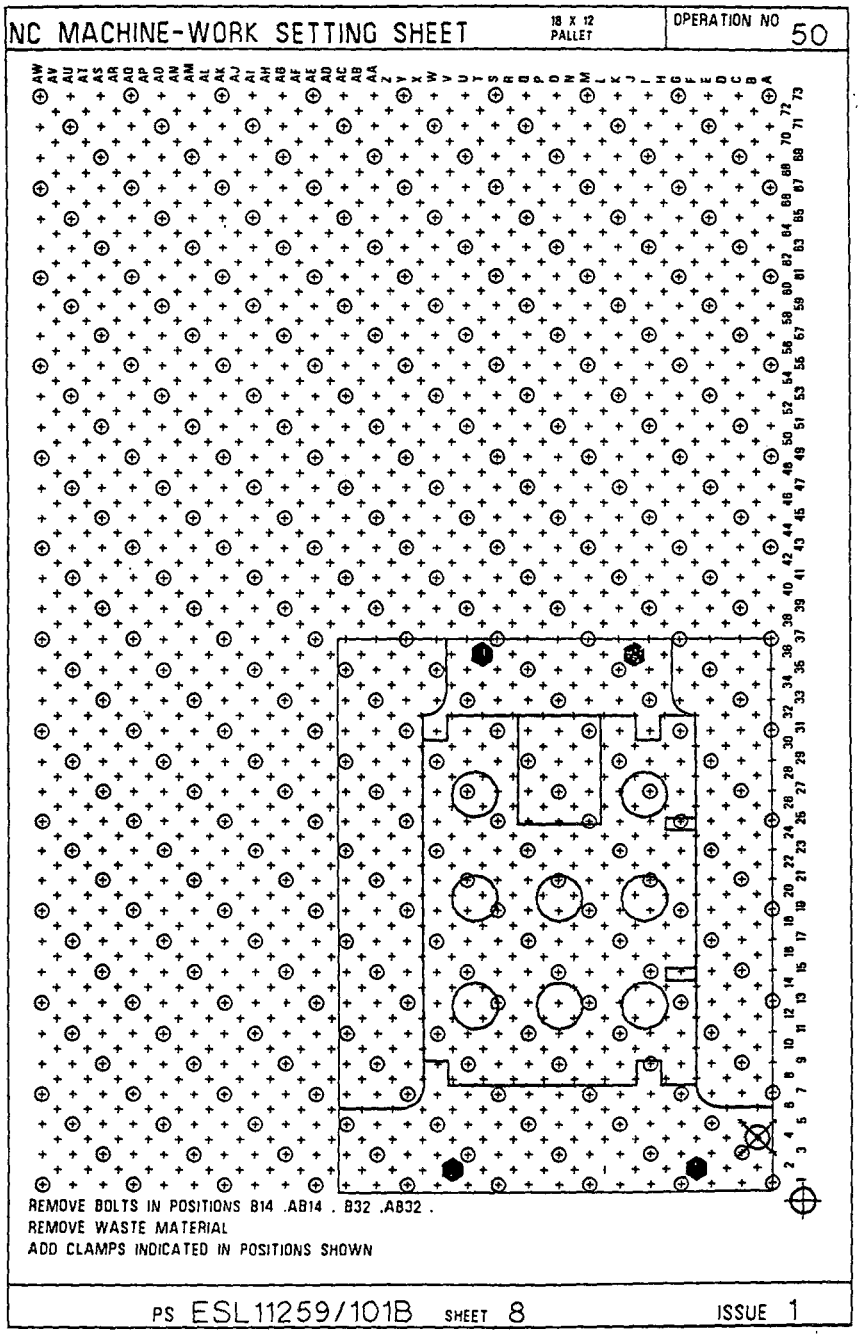

NC MACHINE-WORK SETTING SHEET
18 X 12 PALLET
OPERATION NO 50
REMOVE BOLTS IN POSITIONS B14 .AB14 . B32 .AB32 .
REMOVE WASTE MATERIAL
ADD CLAMPS INDICATED IN POSITIONS SHOWN
PS ESL11259/101B SHEET 8
ISSUE 1

Fig. 13.6

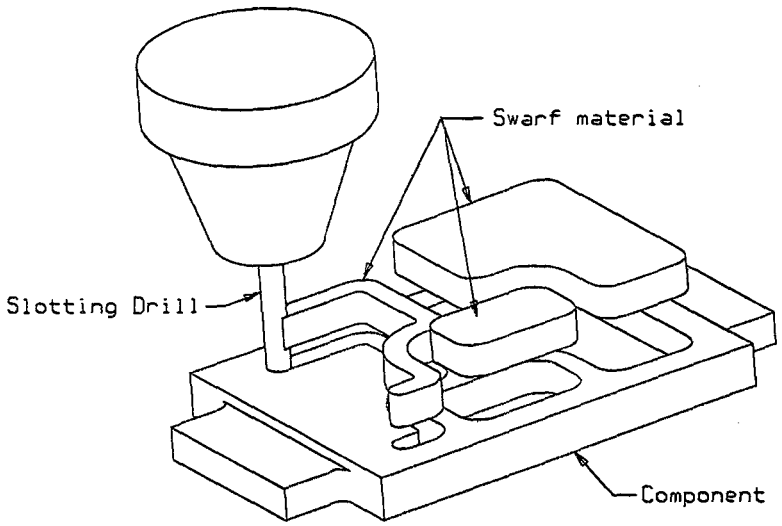


Fig. 13.7

interrogating the model for information relating to geometric relationships and features.

Fig. 13.7 shows an example of how the modeller would subtract cleared columns from the blank form. The strategy for updating the part-machined model has not been fully established. A compromise must be struck between holding a model compatible with the current stage of machining and the computer power required to achieve this objective. Initial investigations would point to updating the model prior to any interrogation requirement.

Detachable waste parts are a serious safety hazard. Their creation during the machine process can be detected by the modeller, as a new body will be produced if a part becomes detached from the parent body. This is illustrated in Fig. 13.8.

One of the most difficult tasks in this project is the extraction of data from the modeller. A hierarchical interrogation of the model is envisaged, starting with the gross shape and working progressively through specific features down to the detailed geometry. The modeller must provide the necessary strategy validation as well as geometric details for cutter path determination. Currently, the extraction of 'implicit' knowledge from the model can only be undertaken through user interaction.

The first objective of the project is to reconstruct AMP using knowledge-based system techniques. Currently, the existing decision making processes of AMP are being reformulated as a set of rules in Prolog. Because of the need to manipulate three-dimensional data, this system will interface directly with a solid modeller.

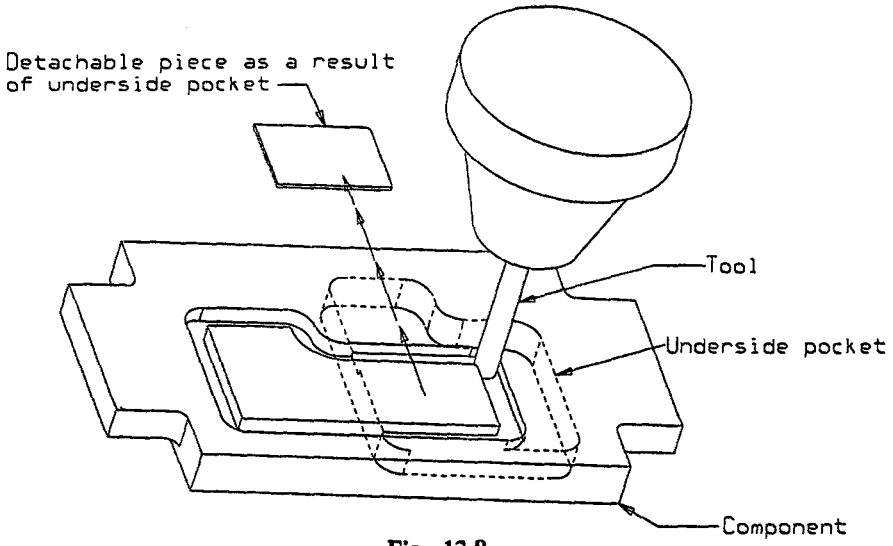


Fig. 13.8

For this purpose the necessary links to the modeller are being defined and developed to provide the facilities required by the knowledge-based system. This will enable the system, in developing its manufacturing strategy for the component, to query the modeller as required. This process will start with a high degree of user interaction with the aim of moving towards increased automation as the project progresses.

A knowledge-based system to aid assembly design

Approach

In designing an assembly a designer is inevitably aware of a number of relationships which exist between the different components of the assembly. If a CAD system is used to record the design, these relationships are captured implicitly in the model. However if the model needs to be changed, it is not always easy to keep track of the consequential changes that must be made to other components in the assembly.

Some systems go part way towards solving this problem by providing parametric design aids for the design of 'families of parts'. These design aids are normally implemented as computer programs in some standard programming language. Programs such as these are generally large and complex and very difficult to adapt if the design rules change.

Ideally one would like a flexible system in which knowledge about relationships between components and the dimensions of individual components is stored as a set of rules which may be easily

comprehended and modified as and when required. For this reason a knowledge-based system has been developed as an aid to the designer responsible for the design of assemblies. In particular it is intended for use in the design of families of parts where the gain in productivity which can be achieved, is likely to be significant. The system maintains a base of knowledge comprising factual information about components of the assembly and their inter-relationships together with a set of design rules. It provides two basic modes of operation:

- **Edit mode**, in which rules and facts about a design may be entered or updated.
- **Design mode**, in which, by assigning a set of values to the parameters of an assembly, an instance of the assembly may be generated which satisfied all the design constraints. This may be used to follow the effects of changes in the design parameters as well as to investigate the effects produced by changes in the design rules.

Knowledge representation

The information/knowledge which is stored in the knowledge base is declared as a sequence of modules. Each module is one of two types:

Part modules

Each part module declares the attributes associated with a component. These include the name of the component, its shape, dimensions, etc. Some of these might be specified by the user, others generated by rules. For example, an elementary part module with user defined attributes might be:

```
Part module : cylinder
att (len)
att (diam) .
```

A complex component may be built up from more elementary ones using the system options 'apartis' (which defines one module to be part of another) and 'atypeof' (which defines a part to be of a type previously declared). An example of this is given in the definition of a stepped shaft which is made up of a set of three cylinders, for which a definition might be:

```
Part module : steppedshaft
apartis (end1, cylinder)
apartis (end2, cylinder)
apartis (middle, cylinder).
```

The values of attributes may be determined by means of assignment rules in the body of the module definition, for instance:

```
Part module : steppedshaft
apartis (end1, cylinder)
apartis (end2, cylinder)
apartis (middle, cylinder)
end1 : len : = 1.5*end1 : diam
end2 : len : = end1 : len
end2 : diam : = end1 : diam.
```

The value which an attribute might take may also be limited by constraint rules, such as:

```
Part module : steppedshaft
apartis (end1, cylinder)
apartis (end2, cylinder)
apartis (middle, cylinder)
end1 : len : = 1.5*end1 : diam
end2 : len : = end1 : len
end2 : diam : = end1 : diam
middle : diam > 1.5*end1 : diam.
```

Or one may define a gearwheel as follows:

```
Part module : gearwheel
apartis (blank, cylinder)
apartis (hole, cylinder)
hole : len : = blank : len
att (noofteeth)
att (toothdepth)
blank : len > = 1.2*hole : diam
blank : diam > 3*hole : diam.
```

Relation modules

A relation module defines a type of relationship which may exist between components. For example, the relationship 'isforcefitin' might be defined as

```
Relation module : isforcefitin
parameter1 : shaft type steppedshaft
parameter2 : wheel type gearwheel
wheel : hole : diam : =0.999*shaft : middle : diam.
```

Such a relationship may be involved from within a part definition. The most obvious way of doing this is in a production rule. For example:

```
Part module : gearassembly
apartis (shaft, steppedshaft)
apartis (gwheel, gearwheel)
shaft : end1 : diam : = torque/100
torque > 100 ⇒ isforcefitin (shaft, gwheel).
```

The user interface

The user may interact with the system via an editor which enables the insertion, deletion and update of facts and rules in the knowledge base. The editor is a menu-driven sub-system which operates at two levels, the module level and the rule level. At the module level whole modules may be inserted, deleted or renamed; at the rule level facts and rules within a module may be inserted, deleted or changed. The editor checks the syntax of the rules entered; it also performs certain basic checks on the operations performed and warns the user against such problems as insertion of a rule that gives a value to an attribute which is already defined in another rule, deletion of a module that is referenced by another module, etc.

Once a complete assembly has been created the user may generate instances of the assembly by entering the design subsystem. This part prompts the user for values for the design parameters and then uses the design rules to produce a complete design description.

13.5 Conclusions

In order to build more powerful knowledge-based systems to aid in various aspects of the processes involved in the design and manufacture of products, it will be necessary to develop more advanced three-dimensional modelling systems. Besides the pure geometric description of the shape of the artefact, such systems will need to store various types of knowledge concerning the features, functions and relationships relevant to it. It is also desirable to provide an interference capability to enable the modeller to use this knowledge in order to answer queries at a higher level than is currently possible. The two examples which are described in this paper illustrate different aspects of this need.

Acknowledgements

The authors would like to thank J.Y. Alexander and K. Edmonstone of Ferranti Defence Systems Ltd., who undertook much of the original work on the AMP and also the ACME Directorate, FDSL and Ferranti Infographics Ltd. for supporting the current research programme. Thanks are also due to I.E. Aitchison and G.A.R. Wilkie of Heriot-Watt University who developed the knowledge-based system for the design of engineering assemblies.

References

1. J.Y. Alexander, 'The automatic generation of control and planning data for the manufacture of 2½D milled components', BNCS Annual Technical Conference, Brighton, 1982.
2. G. Bosch, 'NC programming developments', BNCS Annual Technical Conference, Wembley, 1981.
3. B.J. Davis, I.L. Darbyshire, A.J. Wright and M.W. Park, 'The integration of process planning with CADCAM including the use of expert systems', Proceedings of an International Conference on Computer-Aided Production Engineering, Edinburgh, MEP (35-40), 1986.
4. H. Stolenkamp, W.J. Oudolf, H.J.J. Kals, 'Cubic—a highly automated system for programming NC machining centres', SME Technical Paper Series MS n MS79-167 (22) 1979.
5. A.H. Van't Erve and H.J.J. Kals, 'XPLANE: a knowledge base driven process planning expert system', Proceedings of an International Conference on Computer-Aided Production Engineering, Edinburgh, MEP (41-46), 1986.

Discussion

(Presented by Murray and M. Williams).

Middleditch: First of all, a comment: I agree with you thoroughly about this business of part-program proving and looking at plots of cutter centre-lines and so on. We've tried lots of different ways of doing it, they're all useless, and the only way I have found practicable was to watch a volume model transform itself from a blank into a component on the screen, and that's a very good way to know that you're doing the right job, and that's the one we've locked into.

Another thing about this 2½D question: the handwheel that Bowyer showed us (Fig. 1.2) was made by 2½D machining. Nobody in his right mind

83. D.WILLIS, I.A.DONALDSON, A.D.RAMAGE, J.L.MURRAY and M.H.WILLIAMS, A knowledge-based system for process planning based on a solid modeller, Computer Aided Engineering Journal, vol. 6, no. 1, 21-26 (1989).

A knowledge-based system for process planning based on a solid modeller

by D. Willis, I. A. Donaldson, A. D. Ramage, J. L. Murray and M. H. Williams
Heriot-Watt University

The generation of planning and control data for the machining of mechanical components is a complex task which requires the application of skill and experience. With current interest in applying expert system (ES) techniques to problems in the area of CAD/CAM, this particular aspect of process planning appears to be well suited to a knowledge-based (KB) approach. However, the knowledge used to solve such problems involves the interpretation of three-dimensional (3-D) geometry. This paper outlines the development of a process planning system using knowledge-based techniques with derivation of component geometric information from a three-dimensional solid modeller. The knowledge-based components of the system are centred on a production rule based expert system, coexisting with a nonlinear knowledge-based planning system. These are in turn coupled to a solid modeller via a geometric interface. The knowledge resident within the system operates on the geometry of the component to produce the planning and control data. The system is targetted at planning the manufacture of components with 2½-D geometries, using a three-axis machining centre, from rectilinear blanks.

Introduction

Process planning is that function within manufacturing which is concerned with the selection of processes to allow realisation of component form. In the case of three-dimensional machined components, it entails decisions about the shape and size of the initial workpiece, the securing method, and its configuration, machining strategy etc.

Some work has already been done on applying knowledge-based (KB) techniques, generally in the guise of expert systems (ES), to process planning and related engineering domains. Systems such as GARI (Ref. 1), OPGEN (Ref. 2) and SIPS (Ref. 3) have been described in the literature. These systems cover various planning domains and range from high level operations planners (for mechanical assembly), to those at a lower level (for the definition of individual material cuts, in the machining domain).

However, these systems allow limited ability to re-use the geometric and topological information derivable from the model of the component to be manufactured.

The current work focuses on the manufacture of 2½-D components using three-axis machining centres. By 2½-D we mean a class of components which have planar and cylindrical faces as orthogonal sets, such that all features are a product of a 2-D profile swept through a linear distance along an axis normal to their plane.

The ideas were originally developed within an algorithmic planning system written in FORTRAN77, the automatic machining program (AMP), a product of collaborative work between Ferranti Defence Systems Limited, Edinburgh and Heriot-Watt University. However, as the size of the code grew, it became increasingly difficult for anyone but the system's authors to develop and extend AMP, and hence a KB approach appeared to offer a promising solution. By developing a system in which planning knowledge is held separately from other system code and in a form which is understandable to the average planning engineer, it is hoped that the system will be more easily maintained and extended.

In addition, a solid modeller is used to store the geometry of the component to be manufactured. This will be interrogated by the KB system as and when it requires geometric information in the course of applying the production engineering knowledge. In this way, it is hoped to increase the flexibility and extendability of the system.

Definition of domain knowledge

The prime task in the development of such a KB system is the identification and formalisation of the knowledge which is applied by an expert in the solution of a problem. In this case, an obvious starting point was the body of knowledge used in the decision making within AMP.

AMP takes as input a 2½-D geometry description language called 'Simplified Cutting Sequence (SCS)', developed by Ferranti in the move towards automation of the part programming/process planning function. This language facilitates the expression of component geometry, machining instructions and tool selection details.

With a description in SCS of the component to be manufactured, AMP can determine the planning to produce the component from a flat plate using a three-axis machining centre with a standard set of tools. The component blank is set up on a standard platen, which has an array of

holes to allow securing by bolts or clamps (Fig. 1). Dowel holes are also incorporated to facilitate the accurate location of the component for machining its second side. Certain decisions are left to the user, such as the side to machine first, the method of securing to be employed and the positioning of any clamps required.

Although the knowledge in AMP provided a useful starting point, it is limited in a number of respects, and some extensions which suggest themselves include:

- operations on more than two parallel sides
- the use of alternative securing methods and extension of those currently employed
- a more efficient method of handling waste material which may reduce any safety hazards
- consideration of the consequences of machining a given feature in different ways.

The knowledge derived from AMP can be divided into two types:

- low level procedural knowledge, which comprises the bulk of the knowledge derived and which is retained in its original format
- high level decision rules, which can be formalised and stored in a KB.

Once knowledge had been extracted from AMP, the next stage required the elicitation of knowledge from experienced production personnel. Three methods for deriving such knowledge manifested themselves

- Questions were derived from information contained in production standards and the documentation of AMP knowledge.
- Typical planning examples were used to highlight the general and specific reasoning applied to the planning of a component.
- Documentation produced from initial discussions provided a focal point for further extraction and development of knowledge.

Interviews with experts were recorded using audio tape, the transcripts of which served two purposes; first, they provided the seed for further analysis, and, secondly, they formed the basis for the development of rules to model the planning process.

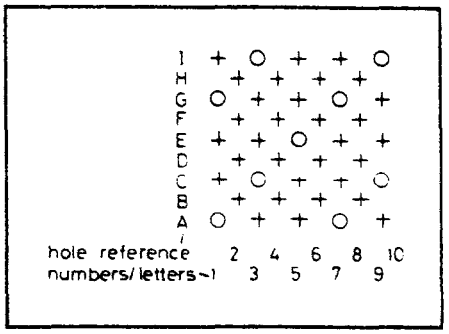


Fig. 1 Configuration of standard platen
○ dowell position
\+ bolt position

The two main areas of the planning problem which have been investigated in this way are machining strategy and securing/set-up strategy. When combined, these two elements of the problem solution cover the essence of process planning for this class of components and allow manufacture to be effected.

Machining strategy

The knowledge derived from sessions with production personnel concentrated on the machining strategy, i.e. sequencing of features and machine cuts within features. A human planner looks for familiar or unfamiliar component geometry/topology, and from this high level analysis he develops a notion of a likely method of manufacture and how this will manifest itself in the final plan. This 'look ahead' approach is the system model. The component *approach directions* (AD, characterised by the axis mentioned as part of our definition of 2½-D) have their features planned top down, i.e. in the order that they occur in, moving towards the component along the AD. Resequencing of this plan is carried out when the constraints listed in Table 1 are present.

The inclusion of knowledge about these constraints is limited to the first three types listed. The effects of surface finish and the various tolerancing possibilities are left for later work (any implications that they may have on planning knowledge expression and geometry manipulations is considered throughout the system's development).

Set-up strategy

To select the best strategy for set-up and securing, it is necessary to consider the global geometry of the component and the implication of any geometric constraints propagated from the development of the machining part of the plan (the first three constraints in Table 1).

From consideration of the global geometry, the planning system can reason on

- the prospect of having a stable platform, formed by geometry of a component side, on which to support the component. The order of sides to be machined may be implied from the relative stability of the various sides.
- alternative sites for bolts (where this is the method of securing employed), in order to ensure as small a blank as possible, which thereby reduces wastage, without affecting safety.
- the distribution and suitability of sites for clamps (both where the securing method is clamping and during the parting-off phase when bolts are used).

The sequencing of machining cuts may be constrained, as mentioned previously. In addition, the consideration of features for machining may suggest an order which conflicts with that derived from consideration of the geometry, made as part of component set-up. This conflict is resolved by checking for the sensitivity of the two sequences to alteration of their order and definition of the final plan, in terms of the order which offers least disruption. If the system fails to resolve the conflict, then the user is prompted for a solution to the problem. The resequencing of the plan is a simple task within the nonlinear planner.

The planning system

An advantage of KB techniques is the clear separation between the body of domain knowledge and the inferencing strategy employed to form deductions from it. This aids system expansion and maintenance of the knowledge base but, additionally, it allows an explanation text to be attached to production rules or schemata. Even a simple explanation text is useful in providing a trace of the chain of inference which has built up in the solution of a user's query. Accordingly, a production rule system was selected to represent the higher level knowledge of the domain.

However, lower level planning involves the manipulation of the sequence of machining steps in response to feature interactions or geometry constraints, and this reordering action was found to be difficult to express in a production rule format.

Therefore, the present system comprises a hybrid of representations in which a production rule system co-operates with a nonlinear planner.

Table 1 Constraints on machining strategy

- machining of thin floors
- machining through thin floors
- machining thin walls
- surface finish
- tolerances

Expert system

An ES uses a body of knowledge, usually associated with a human expert, to infer a solution to a problem.

Knowledge may be represented in different formats by different expert systems, e.g. frames, semantic nets, production rules etc. The most common of these is the production rule, and it is this format which was chosen for the current work. In a production rule system, an inference mechanism is used to access rules about the domain and it attempts to chain the rules together to form a solution to the problem.

The basic format of a production rule is a statement of the form

IF <condition(s)> THEN <action>

For the purposes of the current system, a format was defined which includes the following facilities:

- *Assignment*

LET entity BE value
IF conditions_hold

This structure constitutes the 'classical' action of a production rule, the conditional attachment of a value to an entity.

Example:

LET side one BE thin for area clearance of side one
IF <component thickness – z bottom one> is = <0.080;

This rule is a simple one which defines the presence of a thin floor on the first side of machining. A single value defining a thin floor was used in the early stages. It was found to be too simplistic and has now been replaced by a function which relates to material type, section support arrangements, and section aspect ratio.

Area clearance is the term given to the process of removing material from a large area in the component geometry during rough machining.

- *Procedure call*

CALL procedure_id(parameter_list)
TO FIND entity

This structure affords the ability to call procedural code (e.g. FORTRAN, C etc.) from the ES, thus allowing the low level, procedural knowledge to be left in a form similar to its statement in AMP.

- *User query*

ASK "question" ABOUT entity

This structure allows the system to seek advice from the user to resolve a conflict, a very simple form of system/user interaction. This is necessary because components passed to the system may cause violation of current rules, or the system may have no knowledge of the current facet of the problem.

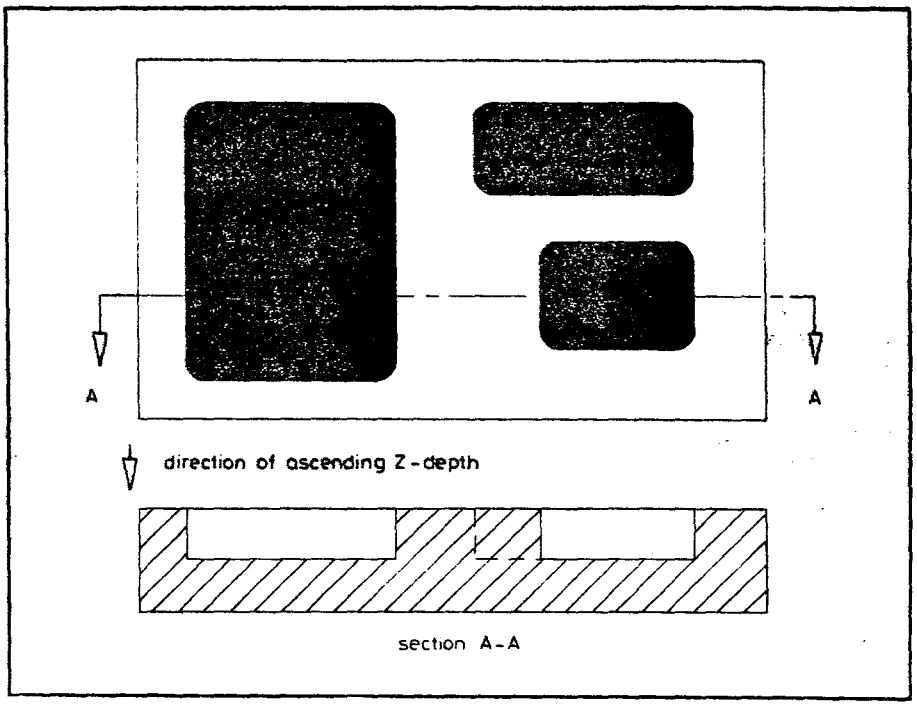


Fig. 2 Three features accessible at same Z

- *User information*

REPORT "message" ABOUT entity

This structure allows the system to give information on the status or progress of inference concerning an entity.

The ES is implemented in Quintus Prolog and uses a backward chaining inference strategy (i.e. goal directed). The user provides a most-general question (e.g. can this component be planned successfully?), and the system will then try to satisfy this goal by generating sub-goals, information about which resides in the planning knowledge base.

Nonlinear planning system

Suitability of artificial intelligence approach

The main task of process planning is to specify an ordered sequence of machining processes which will form the product from stock material, giving regard to local criteria such as costs and resource utilisation. Several features of the process planning task, particularly at its more detailed stages, make it suitable for manipulation by the type of 'nonlinear' planning system derived from artificial intelligence (AI) applications such as robot planning. 'Nonlinear' planners are characterised by the requirement to achieve several goals simultaneously, and do so by finding a partially-ordered (nonlinear) network of tasks to achieve the set of conjunctive goals.

First, there is a substantial, inherent parallelism to the time order of machining steps, and also to the order in which the various set-up phases may be scheduled. This is clearly evident in components which have several features at each level of accessibility, as in Fig. 2. Therefore, when a human develops a process plan, he finds considerable freedom in ordering these steps and is guided more by optimality of his chosen plan sequence than by absolute constraints on geometry. It is important to retain this 'least commitment' to time order in the plan in order to provide flexibility for plan search and any subsequent plan refinement.

Secondly, process planning can be viewed at several levels of abstraction. The selection of set-ups, the ordering of machineable features within each set-up, and the planning of machining cuts within each feature, represents one such hierarchical organisation. It finds a parallel in hierarchical, domain-free AI planners which proceed from a high-level definition of a task to a lower level description, which descends in turn to some arbitrary level of primitive action which will perform the task.

Thirdly, it is possible to distinguish between the superficial and the deeper knowledge of process planning. We typify superficial knowledge as the expertise which will specify some valid plan, and the deeper knowledge of the domain will refine such a plan to a near-optimal state.

These three similarities encouraged us to adopt AI planning as an approach.

AI planning systems enforce a strict separation between the process which generates a network of plan steps and the subsequent process of plan search or plan

refinement. The modularity which this provides can be equated to the categories of superficial (plan generation) and deeper (plan refinement) process planning knowledge, mentioned above. AI planning systems are also characterised by a constraint-posting technique (Ref. 4) in which the plan is defined by successively specifying more partial descriptions which it must fit. This amounts to a gradual reduction in the search space as constraints (or preconditions) rule out alternative plan strategies. The planning system, implemented in Prolog, is modelled on the NONLIN planner described by Tate (Ref. 5).

Knowledge to be represented to the planner

The planner is supplied with two categories of knowledge, generic and specific, which are described below. This scheme is not yet complete as work is continuing on the extraction and specification of deeper generic knowledge.

A generic description of the process planning task is derived from the expertise of planning engineers. It is considered to be invariant and includes the five forms of knowledge listed in Table 2, each of which has an example in the natural language of the planning engineer. The five classes can be summarised as:

- task elaboration
- task primitives
- task purpose
- task constraints
- deeper expertise.

This classification of generic knowledge was influenced by the description of the 'task formalism' described for the NONLIN system (Ref. 5); the first four classes in Table 2 (which define the knowledge for generation of a plan network) can be represented as schemas, whereas the fifth class describes the subsequent stage of plan refinement and is expressed in the form of production rules.

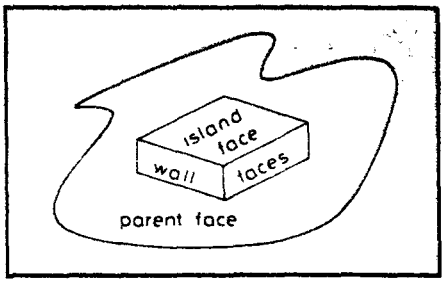


Fig. 3 Parent, wall and island/pocket faces

The specific description uniquely depends on the product which is to be planned. It comprises a hierarchical description of the machineable features which form the product, in addition to its geometry and material properties. Three classes of machineable feature are recognised: approach direction, protrusion and recess. Machineable features are ordered in z-depth from their approach facet to produce a feature hierarchy, which is adopted as a default order for the machining sequence. This is supplemented by geometric and topological data, available on demand, from the analytical interface to the solid modeller, and its acquisition is discussed further in the section on geometry and shape representation.

The class of constraints or preconditions which characterises the generic description of process planning can take one of three forms, according to the latency of the condition. By latency we mean the 'urgency' of satisfaction of a condition, and the three forms allow specification of conditions

- prior to the expansion of a task, or
- by the completion of an expansion, or
- by the time of completion of the full plan.

This aids expressiveness and is of prime importance in maintaining the consistency of the plan as it is elaborated.

Table 2 Types of generic knowledge represented by the planning system

• **A definition of high-level tasks and how to expand each task into a network of sub-tasks:**	**'to machine an approach facet, machine each feature accessible from that facet'**
• **A definition of low-level (primitive) actions in a form which may be used for planning output:**	**'countersink the hole'**
• **The effects and purpose of each action whether at a high or at a low level:**	**'dowel location on turnover of the workpiece provides precise registration of the component'**
• Pre-conditions (constraints to the achievement of an expansion or a primitive action:	'depth of finishing cut must not exceed 0.1 inches'
• Strategies for plan refinement:	'aggregate all drilling operations to the start of an approach facet sequence'

Consistency of the plan

An action in the plan may interfere with other tasks if its effects destabilise necessary preconditions of a subsequent task, and the planner is designed to control such goal interactions. This interference arises, most commonly, when machineable features 'interact' and impose constraints on the machining strategy which would otherwise be adopted — in cases of close proximity between features, for example. The planner is aware of the purpose of each task and of the contributors which are required to satisfy any precondition on a task. This internal knowledge is used to maintain a 'protection range' between a task and its contributors, and the planner attempts to alter the plan network to remove the interfering action from this range. A check on consistency is performed as each task is expanded, and if it proves impossible to relocate an interfering action, then the expansion is rejected and an alternative expansion schema is sought from the generic knowledge.

Advantages of the planning approach

The constraints on machining strategy which were recognised (Table 1) are managed by re-ordering the machining sequence. This is represented explicitly by disconnecting and reforming links in the network (which is a partial order in time on the steps of the process plan) in a way which could not be formalised adequately in a production rule system such as the expert system described above. An additional advantage of the approach is the uniform format by which the description of each task expansion is held as a miniature plan network, identical in form to the main plan network. Thus the system will accept a partially complete plan as input. This might be exploited, in future, as the basis of a variant process planning facility, but we are not aware of any planning system which permits the splicing of subplans with consistency.

Three significant levels of abstraction were recognised. These correspond to the planning of

- approach directions within the body
- features in each approach direction
- machining cuts within each feature.

This abstraction is reflected in the specification of schemas for generic knowledge and provides a concise, structured description of the process planning task. The abstraction property is also presently being exploited to provide strategic explanation of task detail.

Searching the plan for a near-optimal plan

Work is at present underway on the defini-

tion of suitable cost metrics which may be attached to primitive actions in the plan network. These will be used to guide search for a near-optimal linearisation of the process plan.

Geometry and shape representation

An essential feature of the current work is the re-use of geometric and topological information which is present in a boundary representation (B-Rep) solid model of the component. This is facilitated by the use of an extended interface to the data structure of the solid modeller ('Romulus' embedded in the InfoSOLID modelling system from Ferranti Infograpics Limited), supporting analytical queries of geometry and topology. Although the solid model holds an explicit representation of geometry relations, it does not have the facilities to implement the retention of shape relations, such as feature organisation. This level of abstraction must be added to the model by one of two basic means — either manually or by automatic feature recognition:

- manual labelling of facesets to define features, at or near their time of input — this method is effected by interpretation of modelled features by the designer or other user. Facesets are given a label as to their type and a unique identifier (e.g. pocket6). Subsequent queries of a given feature will then use the analytical interface to derive geometric and topological information.
- automatic feature recognition — the extraction of information on component features, without any prior knowledge of the model, is a very attractive means to define the necessary information required for process planning. Work on this approach has been published by Henderson (Ref. 6), Kung (Ref. 7) and Kyprianou (Ref. 8).

At the present time, the latter approach is still very much a subject of research and does not offer a realistic solution for inclusion in a commercially usable system. For this reason, the approach of faceset labelling was adopted. However, as feature-based design systems proliferate, more model information (such as facesets, tolerances etc.) will be included in the model data structure, thereby reducing the need for feature recognition.

The main task in defining the extraction of geometric and topological information is one of developing a consistent feature description formalism, allowing unambiguous definition of the faces which constitute a given feature. Development of such a formalism centres around the identification of two elements of the problem.

The first of these is the perceived need to have facesets constructed which represent features. This labelling of facesets provides easy identification of parts of the model and allows more efficient searching within the model data structure (i.e. search by feature rather than search by face). Features in isolation are insufficient for the synthesis of a machining plan for the whole body (e.g. adjacent islands may interact to dictate a tool size for their manufacture which is not implied by consideration of either individually). The second element which allows the production of a complete solution depends on the fact that any 2½-D geometry can be represented as a set of 2-D profiles, each with an associated pair of limiting positions. The construction of feature facesets for 2½-D components is based on the definition of a parent face, wall faces and island or pocket faces (Fig. 3).

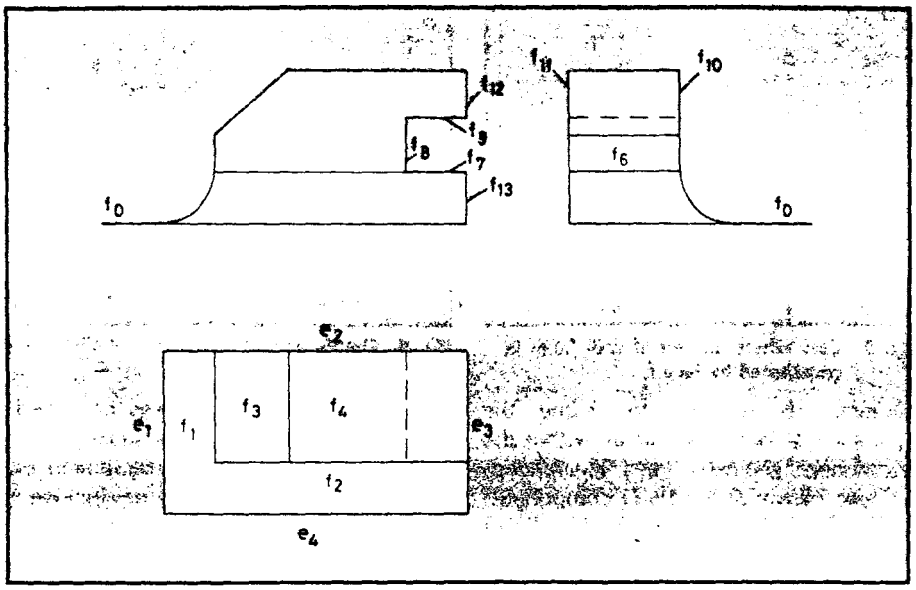


Fig. 4 Description classes used and their application to a simple island

parent face(s)	= $[f_0]$	internal form faces	= $[f_7, f_8, f_9]$
edge loop	= $[e_1, e_2, e_3, e_4,]$	upper form faces	= $[f_3]$
lower form faces	= $[f_1, f_2]$	island face	= $[f_4]$
wall faces	= $[f_6, f_{10}, f_{11}, f_{12}, f_{13}]$	island form faces	= [ ]

However, the assumption of having pure 2½-D is unnecessarily restrictive. To allow the processing of a much wider range of components, this basic list was extended to include lower-form faces, intermediate-form faces, upper-form faces and island/pocket-form faces. These form-feature faces allow inclusion of 3-D geometry, such as fillets, chamfers, undercuts and slots. The example in Fig. 4 shows the description classes used and their application to a simple island. Feature facesets are constructed such that, when moving towards an approach direction (AD), accessible faces, whose normals are anti-parallel to the AD direction, delimit themselves locally. This means that one set of wall faces are traversed in reaching the parent or pocket/island face from the other (i.e. only one set of wall faces are traversed to reach the features parent face, which also has a normal anti-parallel to the AD direction). Consider Fig. 5, where one set of wall faces is constituted by face f_1. This means that complex islands or pockets are expressed in terms of their most basic facesets.

The facesets constructed are given a descriptive label which allows them to be referenced and found more easily and efficiently. The base and parent faces of the features allow 2-D profiles to be defined which can also include reference to any form features present. As an example of this, in the case of a fillet around the bottom of an island, it is necessary to offset the rough cutting tool from the island to allow the fillet to be formed through the use of a radiused finish cutting tool.

The feature description formalism is exploited to answer queries of the solid model, posed by the planning system. An example of its use is the derivation of 2-D profiles from the parent and island/pocket faces which are, by definition, normal to the AD direction.

At present two major classes of modeller query have been identified:

- return geometry of a face set(s) to furnish external procedures with the 2-D profiles which allow the definition of, for example, the sizes of tool to rough machine an area, as outlined above.
- feature interactions, of the types listed in Table 1, which may infer geometry that influences the way in which the manufacturing plan is synthesised. Such cases will have to be identified and recorded to allow effective planning.

These types of queries are best understood with the help of examples.

Example 1

One of the objectives of the planning system is to define geometry for the generation of NC code to drive a three-axis machining centre. To this end, 2-D

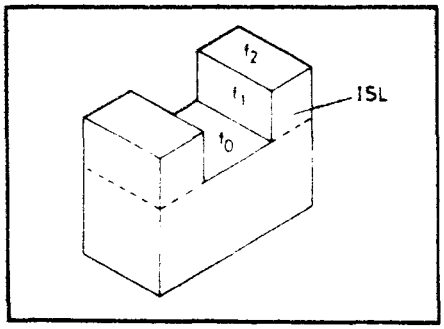


Fig. 5 Case where one set of wall faces is constituted by face f_1

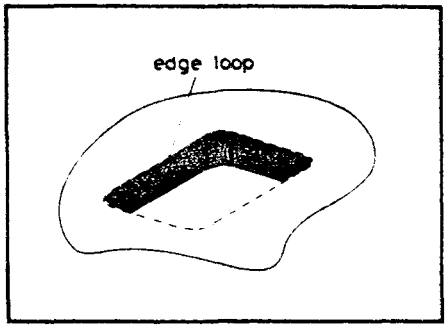


Fig. 6 Simple pocket

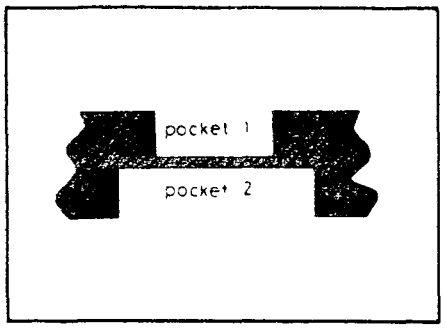


Fig. 7 Component section

geometry is required to define the clearance of large open areas within the component geometry (area clearances). An area clearance for 2½-D components is defined by profile geometry (edge list of polygon), profile depth and the diameter of a suitable tool. The polygon representing the profile of a feature is derived from the modeller feature(s) by analytical enquiry procedures, exploiting the wall/base structure imposed by the feature formalism.

Consider the simple pocket shown in Fig. 6. The profile of the feature is defined by an edge loop in the data structure of the modeller. Enquiry routines will extract the detail of the edge loop to furnish the planning system with the profile and its depth. Further, the size of the tool(s) necessary to machine the given profile may be constrained by the geometry. For example:

- The minimum internal radius may imply a diameter of tool which is small in comparison with the large area pocket to be machined. The use of two tools may offer a more efficient planning solution.
- Narrowing of profile geometry may constrain the size of the tool and imply the need to divide an area into subareas, to facilitate efficient clearance.

Example 2

Component features interact with each other in all but the simplest of cases. Such interactions are identified within the model, so that contingency can be made when planning the manufacture of a component. No information on feature adjacency or connections between faces is included in the formalism used, yet it is necessary that these properties are derivable if interactions are to be handled.

In Fig. 7, pocket 1 and pocket 2 interact to form a thin floor, interposed between the two features. The manufacturing planning of such a floor may, depending upon thickness, extent of area and material type, require reduction in cut depth or alteration of tool feed/speed. These interactions are identified within the component model at the outset, and so manufacturing intelligence can be applied to make contingencies for their resolution. The identification of interactions prior to planning the component corresponds to the 'look ahead' planning strategy described in the section on machining strategy.

Outstanding areas for development of geometry manipulation routines include

- selection of optimal tool sets for area clearance
- selection of the order of component set-ups to ensure stability under machining forces
- selection of applicable securing methods and the siting of their mechanisms.

Conclusion

This paper has given a brief overview of the KB system which is being developed for process planning. Novel features of the current approach lie in the linkage of an AI planning system to a solid modeller and the way in which feature interactions are detected and handled.

The solid modeller provides geometric and topological information used by the AI planner to determine the manufacturing strategy. The procedural interface provides an intimate linkage between modeller and planner which can be exploited during planning to resolve plan conflicts.

In addition, the detection of feature interactions is a non-trivial problem, but this has been overcome by using the procedural interface to the modeller. Interactions involving the proximity of features, the first three items in Table 1, can be detected by considering the minimum separation between sets of parallel faces along the component's approach directions.

There are a number of unresolved problems in the automation of process planning. Principal among these is automatic feature recognition. This problem has been avoided in the current project by adopting a strategy of explicit labelling of facesets which constitute features. The future development of feature-based design systems will avoid the need to use manual labelling.

The development of a feature grammar, a formalised definition of set-up knowledge and an extension to the task formalism and modeller queries, offers the basis of an ongoing programme of work. The resulting prototype will demonstrate the feasibility of combining an AI planning system with a solid modeller in order to produce a more intelligent tool for process planning.

Acknowledgments

This work has been funded by the ACME directorate of the SERC under grant number GR/D/68900, Ferranti Defence Systems Ltd. and Ferranti Infographics Ltd.

References

1 DESCOTTE, Y., and LATOMBE, J. C.: 'GARI: a problem solver that plans how to machine mechanical parts'. Proc. of 7th International Joint Conference of Artificial Intelligence, Vancouver, B.C., Canada, 1981
2 FREEDMAN, R. S., and FRAIL, R. P.: 'OPGEN: The evolution of an expert system for process planning' *AI Mag.*, Winter 1986, pp. 58 – 70
3 NAU, D. S., and GRAY, M.: 'SIPS: An application of hierarchical knowledge clustering to process planning', PED-Vol. 21, Integrated and intelligent manufacturing. Annual meeting of the American Society of Mechanical Engineers, 7-12 December 1986, Anaheim, California, USA, pp. 219 – 226
4 CHAPMAN, D.: 'Planning for conjunctive goals', *Artif. Intell.*, 1987, vol. **32**, pp. 333 – 377
5 TATE, A.: 'Project planning using a hierarchic non-linear planner'. DAI Research Report No. 25, University of Edinburgh, UK, 1976
6 HENDERSON, M. R.: 'Extraction of feature information from three-dimensional CAD data'. Ph.D. Thesis, Purdue University, West Lafayette, Indiana, USA, 1984
7 KUNG, H.: 'An investigation into development of process plans from solid geometric modelling representation'. Ph.D. Thesis, Oklahoma State University, USA, 1984
8 KYPRIANOU, L. K.: 'Shape classification in computer-aided design'. Ph.D. Thesis, Christ College, University of Cambridge, UK, 1983

The authors are with the Department of Mechanical Engineering, Heriot-Watt University, James Nasmyth Building, Riccarton, Edinburgh EH14 4AS, Scotland.

84. J.H.DRIPPS, S.SALVINI, M.H.WILLIAMS, W.A.FULTON, K.BODDY and G.A.VENTERS, Telemedicine in obstetrics, Proceedings of Medical Informatics Europe '90, Glasgow, 501-505 (1990).

TELEMEDICINE IN OBSTETRICS

James Dripps*, Steven Salvini+, Howard Williams+, William Fulton%, Kenneth Boddy* & George Venters*
***University of Edinburgh +Heriot-Watt University %Ferranti International**

1. Introduction

Telemedicine, Project A1032 of the Advanced Informatics in Medicine (AIM) programme of the CEC, is the largest project funded under AIM with partners in seven of the member states each involved in the development of demonstrators projects at five different sites. These demonstrator are in the areas of hypertension, radiology, neurology, renal medicine and finally, the subject of this paper, a system for the monitoring of at-risk pregnancies [1].

2. Background to the Obstetrics Demonstrator Scenario

A mother with an at-risk foetus may have be admitted for observation for a period of days to weeks. Typically, these patients are monitored twice per day for about 20 minutes using a cardiotocograph to record foetal heart rate and maternal contractions. This can apply to some 10% of all obstetric patients with the cost of hospitalisation currently approximately 200 pounds per 24 hours (1989 figures). In addition to the high cost of this type of provision of care, there are the problems for the patient who has, for example, to make provision for the care of previous children over the period of hospitalisation.

It has been demonstrated that for many patients home based obstetric care is the most preferable and cost-effective means of health care delivery. This demonstrator aims to show that by providing adequate monitoring and access to expert obstetric opinion from the home, the proportion of patients who can be so managed is increased.

Telemedicine is the assessment and management of patients at a distance from the specialist clinician and involves a remote sensor linked via a telecommunications network to a display at a central site. Many opportunities exist in such a system to employ recent developments in information technology such as knowledge-based systems and other decision-support tools. In this project the patient is managed in her own home with central expertise at the local clinic at Sighthill Health Centre and at the Simpson Memorial Maternity Pavilion in which are situated the central control computers.

3. The Obstetric Demonstrator Scenario

To illustrate how the system is used consider the case of a midwife visiting a pregnant woman in the patient's home. She takes with her a portable telemonitor station which she connects to the telephone system. She then observes the patient's condition by attaching a belt containing a battery of phono transducers to the abdomen and views both the traces generated and the interpretation offered by the expert system. If she is satisfied that all is well and her interpretation of the signals agrees with that of the system she can go on and visit her next patient. If, however, the interpretations differ or the midwife feels that a second opinion is required for any other reason, the system will alert the expert on call at the hospital. The expert can then review the traces in question along with any other information that may be required from a database of patient information. Once the expert has evaluated the case, she can then send a message to the remote site giving the requested "second opinion".

4. The Demonstrator Structure

The foetal monitoring system is shown in Figure 1 below.

In the diagram 1 - 9 denote dumb terminals attached to the original Simpson Memorial Maternity Pavilion database system which comprises all the components above the dotted line. The Laptop - Signal Processing - Modem groups in the REMOTE CLINIC and the PATIENT'S HOME represent the remote telemonitor stations.

4.1. The Remote Telemonitor Station

The remote telemonitor station consists of a belt of transducers and the associated signal processing hardware and software to extract the biophysical data, and a modem and an RS232 port for the connection of a laptop terminal. The transducers detect foetal heart and breathing activity and maternal contractions and blood pressure. The simultaneous collection of this comprehensive package of biophysical parameters and derived timeseries has not previously been demonstrated in a remote monitoring scenario or even in a hospital setting. Further information on the type of phono transducers used can be found in [2].

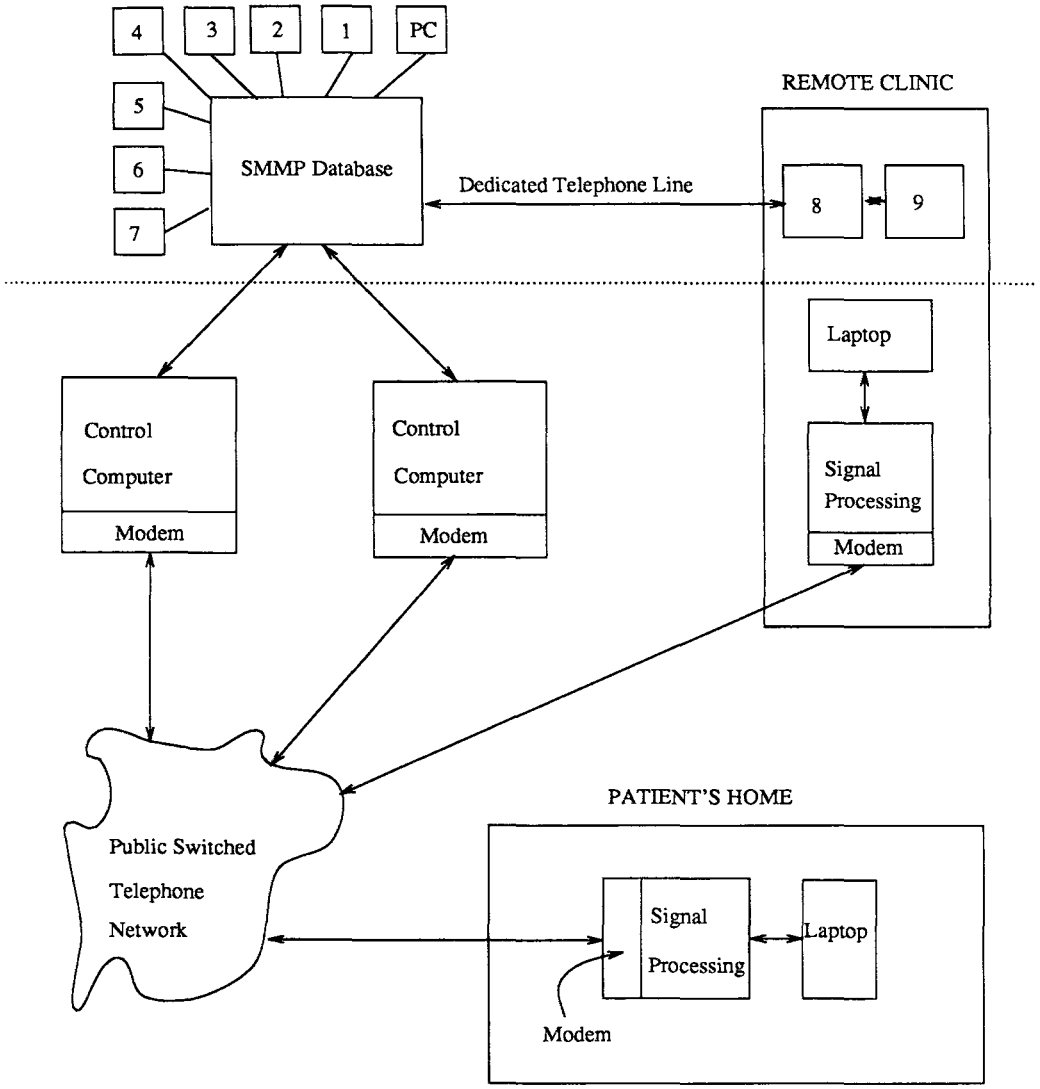


Figure 1: The Obstetric Telemedicine Demonstrator

4.2. The Central Control Computer

To minimise costs and thus facilitate exploitation of the system, PC compatibles have been used for both the control computers and the laptop terminals. The front-end to the system is provided by means of MS-Windows to allow a clean and user-friendly human computer interface.

The central control computer has two main roles. One concerns the interrogation of the database associated with the system. In keeping with the aim of minimising costs and maximising the use of currently available resources, the database used in the demonstrator is the existing Simpson Memorial Maternity Pavilion database system which is Pick based and runs on ADS Mentor hardware. This contains the clinical records of over 29,000 admissions (1988 figures) in the area of the Lothian Health Board since 1983 [3]. The original database has been extended to allow the storage of timeseries data collected at the remote sensors.

The other main role is to provide expert advice about the biophysical parameters currently being monitored at the remote site. This facility will be provided by means of a rudimentary expert system with a knowledge base developed in conjunction with a clinician who is expert in both the interpretation of the biophysical parameters being monitored and in risk assessment in pregnancy [4]. This rule-based expert system will make intensive use of the database and is being developed using Leonardo [5] based on previous experience linking expert systems to a database system [6, 7].

5. Conclusions and Projected Further Work

As noted in the introduction, hospital admission for at-risk pregnancies consumes a sizable proportion of obstetric care resources. By developing a telemedicine system to allow a greater number of mothers to stay at home, considerable savings should accrue in both pure financial and social terms. Cost-effectiveness may be further increased through the extension of the expert system component by filtering out the majority of cases where there is little difficulty in interpreting the signals monitored but still providing expert diagnostic help when required.

The early experience gained with this demonstrator system is encouraging and it is hoped that the formal assessment of its impact in a related workpackage of the current project will confirm that considerable savings can accrue from the use of the telemedicine approach to patient management.

The next stage of this work is to collate and analyse the experience gained from this demonstrator system with those of the other four demonstrator projects being developed by our partners. It is intended that this should lead to the design and implementation of a general telemedicine system. This will comprise a standard telemonitor station for use at the remote site which will be able to accept any of a number of sensors designed using agreed standards based on the experience with the demonstrator systems. For the central control computers, a standard software system will provide general interfaces to database and knowledge-based systems with facilities to handle multi-media medical records.

6. Acknowledgements

This work is currently being funded under the AIM programme of the CEC (project A-1032) and the authors wish to acknowledge this generous support. They should also like to thank their partners from the other demonstrator projects and the Lothian Health Board for their continuing support and encouragement.

7. References

[1] AIM Telemedicine Project A 1032, (1989) "Telemedicine Requirements, Standards and Applicability to Remote Care Scenarios in Europe", Technical Annex (Annex 1), CEC, Brussels.

[2] Goovaerts, H.G., Rompelman, O., and van Geijn, H.P., (1989) "A Transducer for the Detection of Foetal Breathing Movements", IEEE Transactions on Biomedical Engineering, volume 34, number 4.

[3] MacDonald, I, (1988) "MADAM - An Intelligent Data Monitoring System to Assist in Ante-natal Care", M.Sc. Dissertation, Department of Computer Science, Heriot-Watt University, Edinburgh.

[4] Boddy, K., Parboosingh, I.J.T., and Shepherd, W.C., (1984) "A Schematic Approach to Prenatal Care", Lothian Health Board, Edinburgh.

[5] Creative Logic Limited, (1989) "Leonardo", Technical Documentation, Creative Logic Limited, Uxbridge.

[6] Salvini, S., and Gammerman, A.J., (1987) "PROTEST: a Prototyping Tool for Knowledge Engineering", Proceedings of Medical Informatics Europe '87, Roma, Italia.

[7] Salvini, S., and Williams, M.H., (1990) "Knowledge Management for Expert Systems", IEE Colloquium on Knowledge Engineering, London.

85. M.H.WILLIAMS, B.MAHR, G.VENTERS and D.LUTZEBACK, A framework for telemedicine services, Proceedings of Medical Informatics Europe 93, to appear in April 1993.

A FRAMEWORK FOR TELEMEDICINE SERVICES

Williams M. H.[a] ,Mahr B.[b], Venters G.[a], Lutzebäck D.[b]

[a] Department of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh

[b] Fachbereich 20 Informatik, Technical University of Berlin, Berlin

Abstract

With the growing interest in Telemedicine and in Telemedicine services, there is a need for some mechanism to provide assistance in taking decisions relating to the development or introduction of such a service. This mechanism could be provided by the development of a rational framework for describing and implementing Telemedicine Services. This paper describes such a framework which is being developed as part of the FEST project.

Introduction

The introduction of computing and telecommunications technology to medicine through the notion of Telemedicine (TM) offers a number of potential benefits for health care. These include:

(a) Improvement of the level of health care offered to patients. Through applications such as telemonitoring, health care professionals can obtain a more comprehensive picture of each patient's progress and can be quickly alerted to problems. Through tele-consultation, the experience of remote experts can be called upon to assist in arriving at a correct diagnosis and treatment of patients. In general TM offers considerable benefits to patients.

(b) Improvement of the cost-effectiveness of health care. The nett effect of TM is a reduction in the time required for treatment of a patient, in the bed occupancy in hospitals and in the amount of health care professionals' time required for handling patients. This can result in improvements in the cost-effectiveness of health-care to those responsible for its provision.

(c) Uniformity in the level of medical provision across all countries in the European Community. Given the current heterogeneity of medical provision across Europe, the geographical and demographic diversity and the different medical cultures which are present, it is one of the objectives of the EC to achieve a greater level of uniformity in health care provision across Europe. TM technology offers the potential to provide health care which is 'geographically transparent' so that the same services can be used throughout Europe regardless of national boundaries.

Such are the advantages of Telemedicine that there is a strong interest in the establishment of services based on Telemedicine applications (Telemedicine Services) and it is anticipated that such services will proliferate during the next decade or so. Such services can be distributed with respect to geography while at the same time integrated into a unified organisational structure. The benefits of integration and distribution can be realised regardless of the scale of TM service, whether it be at the level of a local service giving improved communication between individual health care personnel or at the level of an international service that offers uniformity of service across national boundaries.

The development and introduction of a TM service will involve consideration of a large number of inter-related factors in order to help ensure its success. To enable the producers and operators of such services to take full account of these many and complex factors, a framework for developing

European TM services is envisaged. Such a framework should ensure that applications are not developed in isolation and that they are compatible with the health services of the different European nations. For the development of a given Telemedicine project, the framework should offer support in analysing the possible consequences resulting from the development and advise on its feasibility and advisability. The support offered should be configured according to the type and scale of service required and to the roles of the individuals conducting the investigation.

This paper describes the approach being taken to the development of a framework for TM services in the FEST project, a project funded under the EC Telematics programme.

Components of the Telemedicine framework.

The framework is being developed by following two complementary approaches. These can be characterised as top-down or bottom-up. The top-down component provides an overall generic model of Telemedicine services whereas the bottom-up component provides guidance on the specific factors and considerations relevant to the design, development and provision of TM services and applications.

Both of the approaches make use of information and expertise on a variety of topics identified as relevant to TM. Besides a range of technological topics, they include a number of non-technological topics such as political, legal, ethical and economic issues etc. The total volume of information which must be considered is very large. The two approaches to the framework are concerned with making this information usable to the different types of individual interested in different TM services.

The top-down approach is used to 'distil' the essential features and elements of TM services into an overall model. The bottom-up approach is concerned with providing the means whereby an individual can be guided through the total set of information to the subsets of information relevant to the application in question.

The top - down approach.

The TM model being developed will provide a definition of the notion of a TM service that will allow such a service to be defined in a rational and complete way. The model is based on the principles of Open Distributed Processing (ODP) but with such modifications as are appropriate to the particular area of Telemedicine. ODP provides a modelling approach which is suited to describing distributed systems that are open in the sense that service user and service provider do not require special knowledge about each other in order to function co-operatively. Such open systems will, nevertheless, provide mechanisms for security and access control. The ODP approach includes a glossary in which concepts and terms for distributed systems are clearly defined together with a generalised model of distributed processing using these concepts and terms.

One of the fundamental ideas on which the ODP model is based is that any system can be described from a number of different viewpoints. These make the description of a system both more comprehensive and more comprehensible by partitioning it into generic areas of concern and providing an appropriate terminology for describing the system within each area of concern. In applying the ODP approach to TM, the viewpoints have been adapted slightly and currently three are being investigated, viz.

(a) the Enterprise Viewpoint which is concerned with the view of the organisational structure, policies, human roles, enterprise information and activities.

(b) the Information and Processing Viewpoint which is concerned with the structure and flow of information, presentation of information and the communication and processing of information.

(c) The Engineering and Technology Viewpoint which is concerned with the technical development and validation of a TM service, its installation and its maintenance.

Besides looking at a TM service from different viewpoints, it may also be analysed in terms of a number of different aspects. Aspects are orthogonally related to the viewpoints and are applicable (in varying degrees) to each viewpoint, providing the additional terminology by which the model of a service as viewed from a particular viewpoint may be further refined. Aspects are used to express the specific concerns of TM services and as such are specific to the needs of Telemedicine services rather than to open systems in general. The aspects are:

- the Agent Aspect which is concerned with roles involved in the TMS - both humans and machines and describes them in terms of their identification, behaviour, autonomy
- the Communications Aspect which is concerned with information exchange and interpretation and includes areas such as language, interfaces, and coding;
- the Construction Aspect which is concerned with planning and establishing a TMS;
- the Persistency Aspect which is concerned with the means and material used to establish a TMS and includes the concepts of knowledge, storage and resources;
- the Consistency Aspect which is concerned with the adherence of the TMS to its specification and utilises the concepts of synchronisation, co-ordination, verification and legal issues;
- the Trust Aspect which is concerned with the confidence that can be had in the TMS relating to requirements. It includes the concepts of responsibility, validation, security and ethical issues.

The viewpoints and aspects form a grid within which the specific knowledge on Telemedicine processed by the bottom-up approach may be classified and located. The structures and terminology provided by the TM model will allow a description of a TMS to be made that will support its planning, design, specification, installation, documentation, maintenance and validation.

The bottom-up approach

Whereas the top-down approach represents the 'distillation' of generalisations about TM services from a mass of information, the bottom-up approach is concerned with the management and organisation of the information in its entirety. It provides a means whereby interested parties can be directed through the mass of data to information which is relevant to their current situation and hence will support the development of Telemedicine services by extracting and presenting specific guidance on their analysis, design, implementation and evaluation. This is done by using the concept of a programmed text to present a series of "guided tours" through the process of Telemedicine Service production.

Given the volume of information which could be considered relevant to the setting up of a TM service, determining and locating what is useful is not a trivial task. It is further complicated by the fact that the framework will have to support a variety of users, using the framework for different

purposes. This means that the structure and organisation of the information space must allow for flexible and changing access.

The need for flexible navigation along many different interwoven paths through a large and complex information space suggests the use of a hypertext structure. Certainly, the facility of hypertext to offer non-linear and unbounded navigation through information, using inter- and intra-document links will be a necessary feature of the bottom-up approach. In addition extensive use will be made of the navigational structures of traditional documents, such as multiple tables of contents, lists of illustrations, extensive rubrics and indices, bibliographies, glossaries, etc. The navigational elements will reflect both the structure of the TM subject areas and the concepts developed as part of the top-down approach.

These navigational structures can be used to adapt the documents (which comprise the information content of the framework) to the needs of the different types of user. A simple example of using navigational structures in such a way would be a report that has two different tables of contents: one intended to be used by technical actors and one by non-technical actors. The former would refer to those section, chapter and paragraph headings containing a level of detail appropriate to a technical user, the latter would contain references to only those parts suitable for a non-technical user. The underlying text is the same in both cases but the view of the document obtained by using the two tables would be substantially different. The superimposition of several structures on to a single mass of information in this way effectively transforms it from one text into many and provides effective support for the notion of different user types making use of the same underlying information.

The potential that hypertext offers for links to sources of information outwith the framework will also be an important way of making the framework more extensive.

However, the provision of a hypertext representation of the TM information space on its own will not be sufficient. The provision of elements that support flexible navigation will only provide a **passive** mechanism by which the reader can easily get to information that he already knows about. This is adequate where the user has a clear definition of his information requirements. In most cases, however, such a clear definition of what has to be considered will not exist. In such circumstances, the complexities of Telemedicine services are such that it is likely that important information will be missed. The bottom-up part of the framework must provide guidance on the specific factors that will have to be taken into account in producing a particular TMS.

The bottom-up component will provide an **active** mechanism for directing readers to the relevant information[1]. The active directions to information will be provided by incorporating the features of a programmed text. Programmed texts have been used as teaching aids for several decades and combine a way of directing navigation with the representation of rulesets which can be used to offer advice. This technique will be used to direct a reader through a progression of advice considered appropriate to particular TM topics or parts of the model. Rulesets based on the underlying TM

---

[1]An analogy of the distinction between the active and passive mechanisms is the distinction between a city map and a tourist guide. The map gives a set of possible routes from site A to site B but a guidebook advises on which sites are of interest and the route to take.

information will be used to establish the next logical location in the program text sequence. Since these locations may not be physically sequential, hypertext links will be used to facilitate these jumps between physical locations.

Adding the expertext features (Rada 1991) which the concept of programmed texts bring to information in hypertext form will provide a useful tool for TM service developers.

A further aspect to be considered, is the need for the framework to be adaptable to developments and innovations as they occur in TM. This means that the framework and in particular the guidance part of the framework, must be readily maintainable. In order to meet this requirement, it must be possible to alter the information contained in individual components with the minimum amount of impact on other components. Links between components should be treated in such a way as to be independent of the actual information content.

This will be implemented by establishing links between sections of information encapsulated in the bottom-up approach which represent stable sub-topic areas within the information domain. Stable entry points to the information will be provided by establishing a set of questions concerning the development or installation of a TM service.

Integrating the top-down and bottom-up components

The stable structures provided by the classification of input areas into sub-topics and the establishment of a set of top-level questions will provide a means of ensuring that the bottom-up and top-down approaches are compatible. It can be used to establish, validate and refine the content of the framework, and in particular to decide what information is relevant to the provision (including planning, implementation and ongoing operation) of a Telemedicine Service (TMS). This will help to pin-point areas in which the TM model may require modification.

Operating at the highest level, the question set will provide a series of preliminary entry points to the paths by which the TM information set can be traversed. Establishing a top level question set will give a focus by which to clarify the input information and allow a series of mappings to be developed between information in the input area and the grid provided by the TM model.

Conclusion

It is anticipated that TM services will proliferate in the next decade. The dual approach to a framework described here will help facilitate the systematic development of TM services. As well as allowing TM service producers to make a rational and comprehensive description of their service it will steer them through the practical considerations and specific factors that are relevant to their situation.

Acknowledgements

The work described here is being funded as part of the Fest project (Framework for European Services in Telemedicine) by the EC Telematics program and the authors wish to thank all those involved with this task.

References

1.) RADA R. (1991) *Hypertext: From Text to Expertext* McGraw-Hill

86. M.H.WILLIAMS and G.MUIRHEAD, Database query interface for medical information systems, Proceedings of IDS 92, Springer Verlag, to appear 1993.

Database Query Interface for Medical Information Systems

M.H. Williams
Department of Computing and Electrical Engineering,
Heriot-Watt University, Edinburgh.

G. Muirhead
Department of Computing and Electrical Engineering,
Heriot-Watt University, Edinburgh.

Abstract

With the development of multimedia technology, there is considerable interest from the medical profession in developing multimedia information systems which can be used to store patient records which include the full set of patient information. In order to use such systems one needs powerful but simple user-friendly interfaces. MIDWIFE (Multi-media Information Database With Intelligent Front End) has been designed to meet these requirements. The basic concepts and model for interaction are described. The use of a hypertext-like approach is an important element in this. The use of browse mode for inspecting results fits well with this notion and is important for a medical information system.

1 Introduction

The need for effective information systems in health care is well understood [Oper90]. However, until recently database systems could handle only a part of the patient record. With the development of multimedia technology, the opportunity is created for handling the complete patient record. For the future a medical information system may include a variety of different forms of data and certainly should encompass:

1. Relational data. This includes conventional numeric items, string data, dates, etc., structured in tuples and stored in relations.

2. Image data. A range of different types of images such as X-ray, ultrasound scans, tomography scans, endoscope images, etc., will need to be catered for.

3. Video data. In some applications a sequence of images is required, e.g. in the case of an ultrasound scan of a pregnant woman.

4. Signal data. If a patient has been monitored for any reason, one may have various sets of signal data stored in the patient record. These include ECGs, EEGs, blood pressure recordings, foetal heart rate, foetal breathing movement, blood sugar, etc.

5. Documents. A large part of the hard copy form of a patient record is made up of written documents relating to the patient. These will be stored in electronic form in the future.

The Esprit II project EDS (European Declarative Systems) [AnBB90],[HHLR90] is aimed at the development of a high performance parallel architecture which is capable of supporting database applications, Prolog and LISP. The database system component of the project is an extended relational database with object-oriented and deductive capabilities. As one of the applications of EDS it was decided to design and implement a multimedia medical information system on top of the database system.

The prime objective of this application was to combine the conventional relational data of the patient record with both documents and images to provide a system which would fully meet the needs of health care users. Since the query language of the database system is ESQL [GaVa90],[GaVa92] (an extension of SQL) which is totally unsuitable for the average health care user, an interface was required which would provide the power of the underlying system in a style which is designed for use by health care users.

This paper describes the interface which was designed for this purpose. MIDWIFE (Medical Information Database With Intelligent Front End) provides a graphical interface which combines relational, document and image data in a manner which is easy to learn and to use. One of the important aspects of this interface which contributes to its ease of use is the deployment of a hypertext-like approach as part of the model of interaction with the user. This facilitates the realisation of two different modes for inspecting the results of a query - view mode (which is the conventional approach) and browse mode (which is more suited to the needs of health care users).

2 Requirements

2.1 System Requirements

The aim of Midwife was to provide a front-end to the overall EDS database system which would enable it to be used as a multimedia medical information system. This interface would communicate with three individual information-handling components dealing with the three main classes of information required.

The first of these components is the EDS database system containing conventional data records referring to the target domain, e.g. patient files, clinic visit files, etc. The EDS database system is an extended relational database with object-oriented and deductive capabilities. As such it is well-suited to handle such data.

The second component is the Image Handler. In order to satisfy the requirements of a multimedia information system in the medical domain, Midwife needs to be able to retrieve, display and manipulate a range of different types of images, e.g. X-rays, ultra-sound scans, endoscope images, etc. A set of one or more images may be associated with one existing record of a particular relation. For example, the set of X-ray images associated with a patient on a particular visit to a clinic may be represented by a link from the patient or clinic visit record to the image store.

The third component is concerned with the handling of documents and allows the user to create, update and retrieve documents which are relevant to a patient's medical history - for example, letters from doctors to hospitals, consultants to health visitors, etc. Again a set of documents may be associated with a particular record of a relation.

One important factor in the design of Midwife has been to combine these three information components in a clean way which is meaningful to the user, and therefore understandable. To achieve this a metaphor is required which is consistent with user expectations and capabilities.

2.2 User Requirements

At an early stage in the design, an analysis of the requirements of the health care domain [BMDA90] and the target users was conducted. This involved a significant amount of interaction with health-care professionals engaged at a large hospital in Madrid.

One of the most important outcomes of this analysis was the fact that the target users' computer literacy varied greatly and that any interface (or interfaces) designed would need to cope with this. In general, however, they all had high expectations of information technology implementations and recognised the usefulness of the target

system in a health care environment. From this document and subsequent meetings with other target users, evaluation criteria for the interface dialogue were developed.

This variation in computer literacy makes it important to find the optimum balance between flexibility and restriction in order to attain maximum usability for a wide range of users. If a system is too flexible, then user confusion can often occur; on the other hand, if it is too restrictive, users will not be able to complete the tasks they require. The problem is exacerbated by the great variety in the abilities of the target users.

This problem can best be overcome by finding an effective metaphor which all users understand [Eric90]. If the system can effectively represent this metaphor, then they may enter into various levels of complexity within the dialogue as their knowledge increases. It is desirable for the system to provide an interface which will support a sharp learning curve from new to expert user.

3 Conceptual Design

If one considers a hospital as the environment for the target system, then the natural way of grouping information would be based on **Folders**, e.g. patient folders, clinic folders, ward folders, laboratory folders, etc. A folder contains all the information which is relevant to the patient, clinic, etc. For the target application the patient folder has been the main focus of development, although in a total hospital information system there would be many other types of folders too.

There is one folder for each patient who has been or is being cared for by the medical establishment concerned. Such a folder will generally contain three further types of objects - **Forms, Documents** and **Images.** Forms are filled out at various times - for example, when a patient is admitted to hospital, visits a clinic, etc.

If an image is captured, this is usually associated with one of the forms in the system, and the metaphor used is that of **stapling.** Thus if a patient has an X-ray during a clinic visit, this image will be stapled to the clinic visit form. It is also possible to 'clip' an image to a form using a paper clip. It is important to be able to attach the same image or document to more than one form in the same or a different folder - this is done by photocopying and stapling.

In order to represent such a system in a computer application, the notion of forms maps into sets of relations in the database. The extra documents and images which are associated with a particular tuple of a relation represent the pieces of paper which are stapled to a form.

In order to provide a method by which users may store their own personal information, three separate user stores are introduced - the **Filing Cabinet**, the **In-Tray** and the **Notepad.**

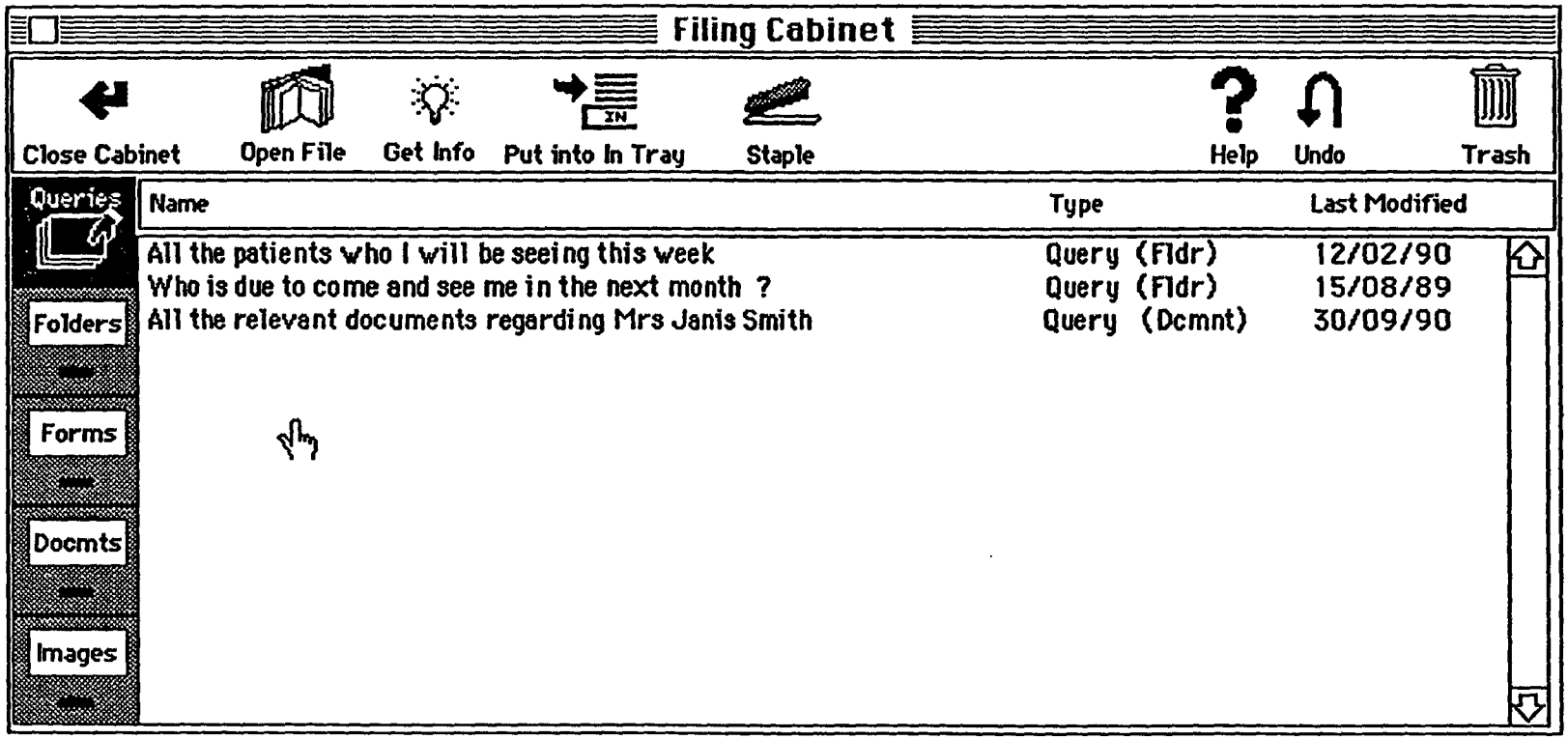


Figure 1 : The filing cabinet

The Filing Cabinet is the main store for the user's personal files. To conform with the metaphor of a cabinet, upon opening the cabinet a window appears showing a set of drawers. Each of the drawers can contain different objects - Folders, Images, Documents and Queries. By comparison, the In-Tray provides the means through which the user may communicate with other users on the network. This may be via mail or via orphans. The latter are a conceptual representation of documents or images created somewhere on the network but not yet stapled to a form. For example, a patient may have had an X-ray taken, and the resultant image file is left unstapled awaiting attention. This will be kept in the appropriate user's In-Tray as an orphan until it is filed.

The Notepad is provided to allow the user the ability to Cut, Copy and Paste almost any object of the system. When, for example, a document or part of the text of a document is Cut or Copied, the selected item is placed in the Notepad. From here the user may Paste it into an appropriate place in the system. Unlike the Clipboard provided by the Apple Macintosh, the user may Cut or Copy more than one object into the Notepad.

4 A Model for Interaction

Considering the manual system for a moment, one could imagine a situation where the hospital has hired a 'Query Administrator' who receives all queries from the various users within the hospital and processes them, returning the resultant records.

In order to register a query with the 'Query Administrator', the person requiring the information must first select the required forms from the set of different form types which exist in the hospital, and indicate on these forms the conditions for the individual fields which are required for the query. For example, if a doctor wished to find all patients who were admitted to the hospital since 1st February 1992 for an appendectomy, then s/he would first select the Hospital Admission form and insert the condition 'Date of Admission > 1/2/92'. Then selecting the Operations form, s/he would enter 'Operation Type = Appendectomy'. The doctor would then hand the forms representing this query to the Query Administrator who would search the database for the relevant records and return them to the doctor.

Now in order to implement this in the database system it is possible to realise the Folder/Forms metaphor through the use of a Hypertext approach. If, corresponding to each form in the database, a screen is created (called the Form Layout) which contains links to the individual attributes of relations held in the database, then one has essentially created a physical representation of the entire relational database.

The link between a form layout and the attribute of a relation is set up in the form of an empty 'Slot' in the form layout (as in a real form) and the person creating such a

form layout (generally the database administrator) is able to arrange any text and graphics about these blank slots to create an easily understandable representation of the information for the end-user. Thus each slot represents a window into part of a relation.

In addition to these linkages between the slots in a form layout and the attributes of relations, each form layout is linked to some subset of other layouts on the basis of 'natural' join operations between forms. Thus each form layout has associated with it a list of forms which may be reached from the current form via a simple join.

Due to the complex nature of the database network in a full medical information system, it will be important to provide a comprehensive help system. This will need to be able to give information on the composition of relations as well as on each field within a relation. Since the inexperienced user may have problems in navigating through the network, the functionality was developed to allow the user to find a specific field from anywhere within the database network.

Besides enabling the user to create queries to the relational database, it is intended that Midwife provide the facility to create a query to the document or image handlers and to either run it independently or attach it to a relational query. For example, a doctor may wish to find all the patients who conform to some particular query condition and then extract the documents associated with these patients which were written by Dr. Venters. In this instance, the user may attach a document query to a relational database query and the relational results will be used as a basis for the document query, i.e. the relational database results will restrict the number of documents which have to be searched.

The facility by which the user may query the document and image handlers operates under the same model as that for relational database queries. This is achieved by associating with documents and images a set of user-defined fields known as Tags. These represent user-specified values to certain pre-defined database variables. Examples for a document could be Name of Sender, Date Sent, Type of document, etc. When the user creates a document, s/he may insert values into some or all of the tags associated with the document.

Since the set of tags is pre-defined by the Database Administrator, and hence relatively static, it is quite straightforward to create queries to the document and image handlers by creating conditions based on the values entered as tags. The dialogue used to create queries to the document and image handlers has been developed in a similar way to that used to create conditions for forms.



Pregnancy Form

Mother ID Number
Pregnancy ID Number

Note : Both these numbers are unique and are unrelated

Date Of GP Visit
Consultant who was used during this pregnancy

Mother Info :

Date Of Leaving Work
Has she been resident in the Uk in the last 12 months ?
If not, does she intend to stay here permanently ?

Other Info :

Date Of Booking for place of birth
Type of Ante-Natal Care
GP assigned to Ante-Natal Care ?
Early Discharge ?

Social Security Help ?
Home Help ?

Operations :

1
2
3

Figure 2: A Form Representation Example

5 View and Browse Modes

In general there are two basic modes of interrogation of a database. The first mode is that conventionally used by database management systems, in which the user specifies the details of a search or update operation using a standard query language and gives this to the database management system to execute. The result is usually returned as a table of values or an appropriate system message. This mode of operation will be referred to as **query mode.**

An alternative mode is useful for stepping around a part of the database, one record at a time. This is particularly useful in a medical database where one may wish to inspect a number of different forms one by one. For example, various items of information relating to a patient will be stored in a number of different forms, documents and images in the database, linked together by attributes on which join operations may be performed. A user may start with one particular form relating to the patient and move successively to other forms, documents or images in a sequence of steps which are determined interactively by the user as required. Such a mode of operation is referred to as **browse mode.**

Both modes of operation can be accommodated in a system in a simple and consistent fashion. In the case of Midwife, a dialogue has been established to enable a user to create a query to retrieve objects from the database. Thus, using the query manager and the various condition creation dialogues, the user may move around the database representation, image and document handlers as s/he wishes, creating a query for the database. Once the query is complete, the user may start the search of the database looking for records which conform to the search specification, by clicking on the Start Search button. Alternatively, at any point in the creation of the query, the user may save the query in its current state and continue with something else.

Once the system has completed a search of the database, the number of objects which have conformed to the search specification and make up the current results is displayed to allow the user to decide how to proceed. The objects are usually taken to be folders, and under our metaphor the system has found a set of folders containing information on the patients who conform to the query. Thus, for example, the user may find that 1,000 folders conform to the search specification, in which case s/he may wish to refine the search requirements to reduce the number of folders which match the query. In this case the user will choose the "Edit query" option and return to the Query Manager.

If the size of the resultant set is acceptable, the user has a choice as to how to view the results. S/he may either view the results as a table in the standard way (for example, listing the names and addresses of the patients) or enter browse mode. This

latter option gives the user the opportunity to view the folders in full. This means that the user may view the values stored in each form in a particular folder. The system starts at the first folder in the results and allows the user to move through them one by one. For each folder the user may view any of the completed forms and associated images and documents.

Upon selection of the browse option, the system takes the first folder of the results and fills out one of the forms in the folder. There are several navigation options which the user may use to view the information required. These are represented in the palette window shown.

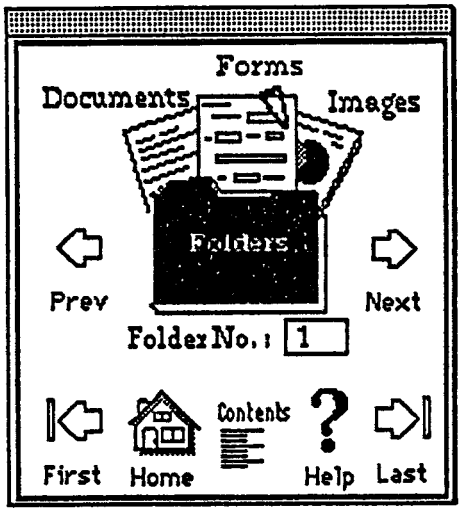


Figure 3: The Browse Buttons

The icon at the top of the window indicates which of the objects of the browse facility is currently active. The buttons surrounding the icon (excluding Help and Home) all refer to this object. For example, when the Folder part of the icon is highlighted, if the user clicks on the Next button, the system will extract the next folder for the user to browse. Similarly when the image part of the icon is highlighted, if the user clicks on the Contents button s/he will be shown a list of all the images which are attached to the present form, and so on.

The user may select the object which s/he wishes to deal with by clicking on the appropriate part of the icon. Upon this selection, the system will open/select the appropriate window, and change the menus accordingly. For example, if only the Forms window is open, then if the user selected the images object part of the icon, the system would open up the images window at the first image in the contents list and change the menus to suit the image application.

It is also possible to switch between each of these by clicking somewhere within the required window - this then becomes the active window, and the menu bar will change accordingly. Finally, the user may drag a particular image, document, form, or folder to the Notepad or the Filing Cabinet using the mouse in the usual way. This functionality is particularly useful when moving between forms and folders. For example, the user may be viewing a document and an image whilst looking at one form, but when s/he moves to the next form, these windows will automatically close. Thus by dragging (copying) the document and image into the Notepad, s/he may open a non-functional window showing the document and image at any time from then on.

When the Folder part of the browse icon is currently selected, the user can use the arrow keys to move between the folders in the results. It is also possible to type the number of the folder required into the Folder No. field under the icon.

Within a folder the user may select the form to be viewed from the Form menu which is available when the Forms part of the icon is highlighted. If there are more than one instances of a particular form in a folder, the user may navigate through these using the arrow buttons.

When a form is displayed the values in the particular database entry are filled into the corresponding slots of the Form Layout and the result is displayed. In setting up the Form Layout the Database Administrator may specify that a field is displayed in different ways according to its value.

When the image option is selected, the image window is opened showing the first image in the list of those stapled to the current form. The user may use the arrow keys to navigate through the images, and clicking the contents button shows the user this list.

6 Implementation

In order to allow an effective method for the development of the dialogue for Midwife, an iterative rapid prototyping approach was adopted. For this purpose an Apple Macintosh with the rapid prototyping package Supercard 1.5 was used. In the initial stages a conventional small-screen Macintosh was used - this was adequate for planning the basic dialogue. However, once the basic ideas had been mapped out, development of the prototype was switched to a Macintosh with a large screen in order

to provide a reasonable working environment for viewing and manipulating forms and images.

A number of different iterations of the dialogue have been produced and some of these exposed to potential users for comment until the design stabilised in its current form. In particular several iterations of the design were demonstrated to medical and technical personnel in Madrid and they were encouraged to use the system at each stage. Their comments were fed back, and observations of their use of the interface revealed ways in which the interface could be improved. The system was also demonstrated to a number of people in Edinburgh, including medical and technical personnel, and the feedback has been used to refine the interface.

The final phase involves implementing the interface in C to run on a Sun computer system. The first step in this process has been the creation of a set of tools for use by a database administrator to set up the form definitions for any arbitrary schema definition. This set of tools enables the user to lay out a screen with text, graphics and slots and to associate each slot with a field in the schema definition. Links can be created between records to enable automatic joins.

This set of tools is now complete and attention is currently focussed on the interface itself, which should be completed shortly. Unfortunately, problems within Telefonica Sistemas have meant that the system will no longer be incorporated into the EDS system as planned, but it is hoped that it will be able to be used to produce an effective interface for multimedia medical information systems for the future.

7 Conclusion

This paper describes an interface to a multimedia medical information system which was designed as part of the Esprit II project "European Declarative Systems" (or EDS).

The conceptual design established a set of metaphors which are readily understood by medical personnel [Rubi88]. With these a simple model of interaction has been developed (based on a hypermedia approach), for which the user need only have some knowledge of what constitutes a folder (the makeup of the forms which are used by the hospital) and understand the concept of related forms (by 'natural' common field joins). Most of the system uses these metaphors as a basis for its interaction with the user and hence the user can very quickly understand its functionality.

An important objective of the Midwife system is to cater for different levels of user expertise, ranging from inexperienced computer naive users to experienced regular users. The first of these levels is aimed at those users who have a set of queries prepared for them in advance. At this level the user will initiate simple commands in order to view the results of these queries. As such a user gains experience, s/he may edit these prepared queries and create new queries. There are several layers of query

complexity which are open to the user as his/her demands for information increase. Furthermore, as the user's expertise increases, s/he will wish to be able to create queries more quickly and thus facilities must be provided to allow the expert user to create queries using a variety of shorter dialogues which provide less assistance. Thus, one of the primary concerns in the design of the interface has been to allow the user to migrate between levels with ease.

Acknowledgements

The authors acknowledge the support received from the European Community under the Esprit II programme for this work which was conducted as part of project EP2025, European Declarative Systems. They also wish to thank Dr. J.M.P. Quinn (ICL), Dr. A. Bujosa (SIE) and Sr. I. Alcalde (TS) for their assistance and support.

References

[AnBB90] Andres F, Bergsten B, Borla-Samet P, et al. EDS - collaborating for a high performance parallel relational database, Proceedings of ESPRIT Conference, Brussels, November 1990.

[BMDA90] Boddy K, Marsh J, Dripps JH, Arredondo MT, Fernandex G, Vaz F. Telemedicine Requirements, Standards and Applicability to Remote Care Scenarios in Europe, Report D6, Dept. of Obstetrics and Gynaecology, Edinburgh University, July 1990.

[Eric90] Erickson TD. Working with Interface Metaphors, in the Art of Human-Computer Interface Design, Ed. B. Laurel, Addison-Wesley, 1990, pp.65-73.

[GaVa90] Gardarin G, Valduriez P. ESQL: An Extended SQL with Objects and Deductive Capabilities, International Conference on Database and Expert Systems Applications, Vienna, Austria, August 1990.

[GaVa92] Gardarin G, Valduriez P. ESQL2: An Object-Oriented SQL with F-logic Semantics, IEEE Int. Conf. on Data Engineering, Phoenix, 1992.

[HHLR90] Hammer C, Haworth G, Leunig S, Reeve M. The European Declarative System, Database and Languages, IEEE Micro, December 1990.

[Oper90] Operation 1992 - Investigation of requirements and options in the field of Advanced Informatics in Medicine: Consolidation of Results, Commission of the European Communities, XIII/F/GE0392, May 1990.

[Rubi88] Rubin T. User Interface Design for Computer Systems, Ellis Horwood Ltd., Chichester, 1988.

87. M.H.WILLIAMS and J.HU, Sharing patient data among heterogeneous medical databases, Proceedings of Medical Informatics Europe, to appear in April 1993.

SHARING PATIENT DATA AMONG HETEROGENEOUS MEDICAL DATABASES

Williams M H Hu J
Department of Computing and Electrical Engineering
Heriot-Watt University, Edinburgh, UK

Abstract

The increasing use of computer systems for storing patient information in different hospitals, in different departments within the same hospital, in clinics and even in doctors' surgeries, is creating a problem in terms of the fragmentation of the patient record. The lack of any coordination or standardization in the way in which data are represented and stored in these different systems, makes the task of finding and retrieving data that much more difficult. This paper considers two aspects of this problem and describes an architecture which can be used to overcome them.

1. Introduction

The usefulness of database technology within different branches of health care is widely recognized today, and many medical databases are being built up to store patient data for use in decision support for diagnosis and therapy [1]. However, as the number of different databases used to store patient information grows, so too does the problem of fragmentation of the patient record. For example, a large health organization may have many different medical databases stored on a number of different computers located at different sites - in this case, parts of the clinical record of a particular patient may be stored in several different medical databases, possibly in different hospitals. This fragmentation of the patient's medical record means that only a subset of the information relating to the patient's medical history is available at any point. Furthermore, inconsistencies will develop between different databases, a problem which databases were originally developed to solve.

To overcome these problems there is a need to interconnect relevant medical databases to create an information management system in which patient data are shared. A number of approaches have been put forward during the last decade which tackle the issue of cooperation between heterogeneous database systems [3, 5]. These approaches differ in their basic assumptions. In this paper the following assumptions will be made. Firstly, it will be assumed that the bulk of the queries to be dealt with are patient-centred. This means that the system should aim to handle efficiently those queries where the patient identification is known. Secondly, it must be anticipated that such an information sharing system will spread to cover the health services in a city or region,or even in a country or set of countries. This means that although only a few databases may be interconnected at the outset, the number could grow to thousands or even tens of thousands. Finally, it is essential that the autonomy of each participating database is maintained, i.e. the introduction of the information sharing system does not change the status of current use of the existing database systems.

The essential tasks in creating a system with the above features include:

(1) *Navigation* among databases to find which contain the relevant information on a given patient, and

(2) *Communication* of queries and data between heterogeneous databases.

2. Navigation

A number of different approaches have been proposed to handle the problem of locating the database where information on a patient is stored. The simplest of these is to broadcast a request to every database involved. However, this is impractical if the number of databases involved is large and only a very small subset is required for each query. Other approaches involve the maintenance and use of indexes which map patient_id to the subset of databases containing information on this patient. Three possible index-based strategies are:

(1) A single global main index is kept for the whole system. Every search request arising from any participating database system must refer to this index. With a large number of databases (and very large number of patients), access to this index becomes a bottleneck, greatly lowering the search efficiency of the overall system.

(2) Each site maintains its own copy of the global index. In this case, the search efficiency is greatly enhanced although update is now a problem. Each time that one copy of the global index is updated, all copies need to be updated, with accompanying synchronization of locks (requiring a significant amount of processor time overall).

(3) An intermediate solution is to keep a single global main index for the whole system and to build up local indexes at each site, each of which is a subset of the global index. A local index should include the set of patients associated with that site and is used for normal searches for the vast majority of queries. The global main index is referred to only occasionally when a patient index entry is not available locally, and for index maintenance. This solution takes advantage of both the high search efficiency of solution (2) and an efficient maintenance strategy as in solution (1). Further details on navigation strategies are given in [2].

3. Communication

The second problem relates to the heterogeneity of the databases. As there is a vast investment in existing databases, any system which attempts to connect together medical databases in the future must cater for a variety of different database sources. This heterogeneity applies not only to the hardware and software at the different nodes of the network, but also to the underlying data models, to the different ways of structuring the data at the different sites and even to different ways of representing the same data item. Thus the data inconsistencies that may exist in different medical data sources include:

(1) *Naming conflicts* in which semantically different data items have the same attribute name or semantically identical data items have different attribute names.

(2) *Data value differences* in which semantically identical data items have different value representations in different databases (e.g. string vs real, different units of measurement, etc).

(3) *Data accuracy conflicts* in which the same data item is represented with different degrees of accuracy (e.g. qualitative vs quantitative).

(4) *Data structure differences* in which different data structures are used to represent the same data (e.g. dates represented as a triple of integers, a single integer or a character string).

(5) *Intranslatability between semantically related data items*, e.g. a diagnosis code in one system may not be able to be translated to an appropriate code in another system if different classification systems are employed.

In order to handle these heterogeneities the notion of a *global standard medical attribute set* is introduced through which the data communication between databases can be mediated [7]. In Figure 1, a query presented in local database 1 is imposed on local database 2 after transforming it by applying the map $f_1 \circ f_2$, and the results of the query can be transformed back by applying the map $f_2^{-1} \circ f_1^{-1}$.

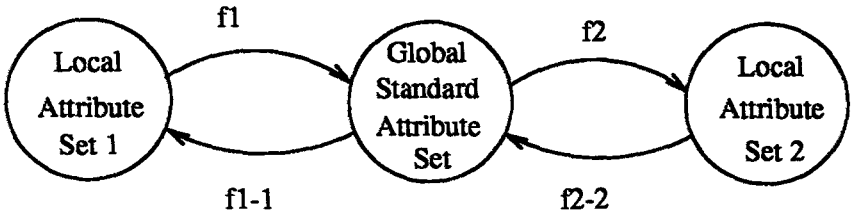


Figure 1. Mappings between attribute sets

4. System architecture

In this section an architecture for a general medical information system is described briefly. The proposed architecture for each of the nodes of such a system is shown in Figure 2.

The **Navigation Module** is responsible for finding where the relevant information on a particular patient is stored. More specifically, it accepts as input a query referring to a patient and returns information about the databases in which medical records might be found. It uses the third of the index-based strategies described in Section 2, i.e. a single global main index together with local indexes which are subsets of the global main index.

The system has two **Query Translator Modules.** The first accepts queries expressed in the local query language and local attributes, and translates them to a standard query language which can be used for communication throughout the system. The second accepts queries in the standard query language (from other database systems) and translates them into the local query language and local attributes. These translations are performed with the help of a set of **metadata.**

The query translator modules will only be invoked when data from an external database are required. It is assumed that the majority of queries will be local queries which can be satisfied from the local database without the need for external searches. Since the nature of the queries is assumed to be patient-centred, the standard query language used internally could be a very simple SQL-like language, or the ISO standard language proposed for information retrieval.

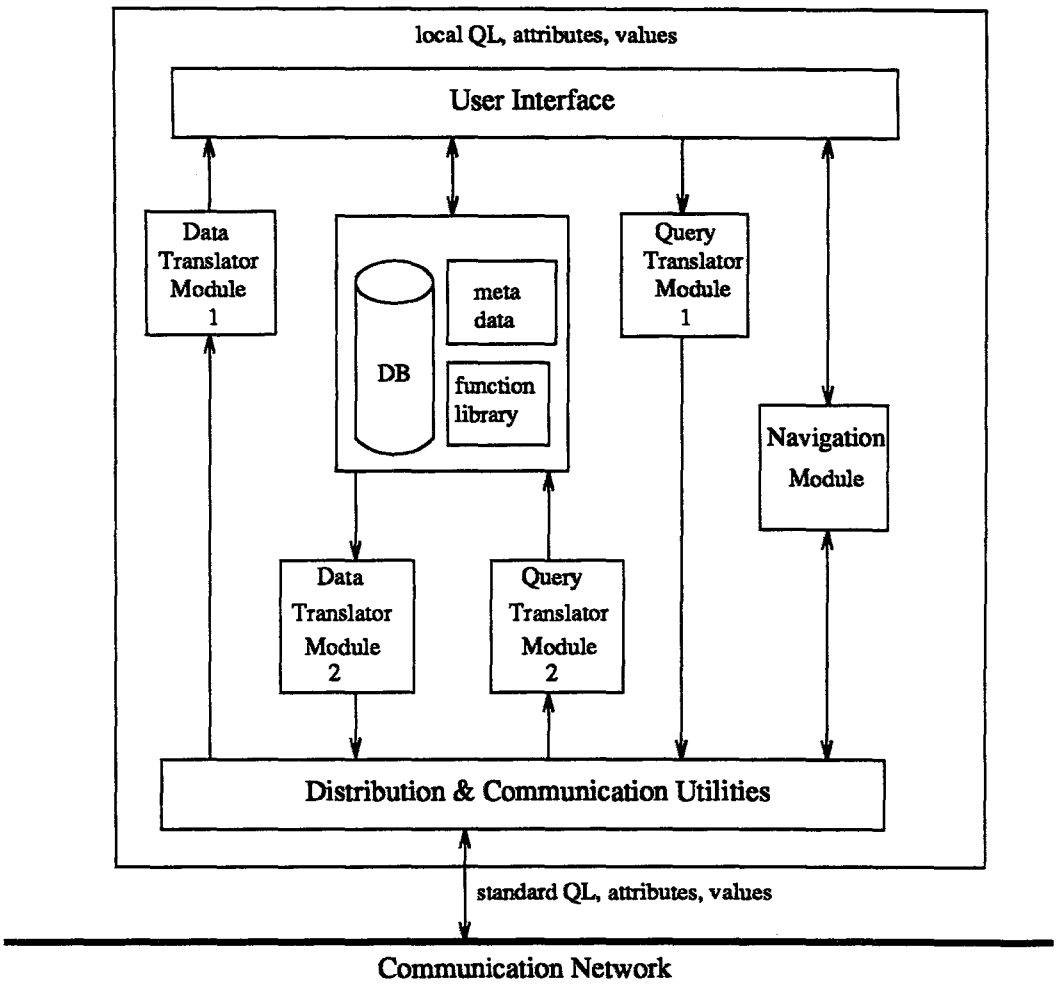


Figure 2. A proposed architecture

Two **Data Translator Modules** also form part of the system. The second module accepts results expressed in local attributes and local data values, and translates the values into the standard form, and tags and organizes them into the standard data transfer format. The first one performs the reverse process, i.e. accepts data in the standard transfer format, unpacks them and converts them into local data values.

The first module is used when data are received from another database in response to a query initiated from the local database, the second is used when responding to a query from another database. The standard data transfer format is a tagged data structure which provides a semantically rich representation capable of dealing with a variety of different kinds of data. Once again the translations are performed with the help of the set of metadata.

A prototype of the proposed architecture has been created and used to connect together three relational databases of different types - Ingres RDBMS (using the language QUEL [8], Oracle RDBMS (using SQL) [4], and a relational database simulated in Sicstus Prolog [6]. Currently the object-oriented database Postgres is being connected to the system using the same architecture. These databases are located on different computers which are connected together via the communication network. At each site a process referred to as the *server* runs continually and handles

the remote query requests from other sites. The indexes used for navigation, including the local indexes and the global main index, are themselves Ingres or Oracle databases so that the manipulation of indexes could be handled uniformly and the concurrent control mechanism used to ensure integrity of indexes.

5. Conclusion

As the number of databases storing patient information increases, so the problem of fragmentation of the medical record becomes more and more serious. One important consequence of this is that only a partial picture of a patient's medical history is available at any point. One solution to this problem is to interconnect existing heterogeneous medical databases to create patient data sharing systems. Under the assumption of a patient-centred query policy and the circumstance of a large set of existing heterogeneous databases, two fundamental issues for constructing such systems have been discussed. These are the problems of navigation among the existing medical databases to find the relevant databases,and that of communication between heterogeneous databases in order to obtain the relevant data from these databases in the format required. An architecture has been proposed for solving these problems and a prototype has been constructed which demonstrates the architecture.

Acknowledgments

The authors would like to thank the British Council for providing funding for Mr. Hu to conduct this research.

References

1. Ellis, D., *Medical Computing and Applications*, Ellis Horwood Limited, West Sussex, (1987).

2. Hu, J. and Williams, M.H., "Locating patient data among multiple heterogeneous medical databases," *submitted for publication.*

3. Litwin, W., Mark, L., and Roussopoulos, N., "Interoperability of multiple autonomous databases," *ACM Computing Survey*, **Vol. 22, No. 3**, pp. 267 - 293, (September 1990).

4. *Oracle Reference Manual Series*, Oracle Corporation, California, (1985,1987,1989).

5. Sheth, A.P. and Larson, J.A., "Federated database systems for managing distributed, heterogeneous, and autonomous databases," *ACM Computing Survey*, **Vol. 22, No. 3**, pp. 183 - 236, (September 1990).

6. Carlsson, M., *Sicstus Prolog User's Manual*, Swedish Institute of Computer Science, (1988).

7. Williams, M.H. and Hu, J., "Communication between heterogeneous medical databases," *Journal of Information Science and Technology*, **Vol. 2, No.1**, pp. 12 - 30, India, (October 1992).

8. Woodfill, J., Siegal, P., Ranstrom, J., Meyer, M., and Allman, E., *INGRES Reference Manual*, Electron. Res. Lab., Univ. of California, Berkeley, (1988).

88. M.H.WILLIAMS and J.HU, Communication between heterogeneous medical data bases, Journal of Information Science and Technology, vol.2, no. 1, 12-30 (1992).

Communication between Heterogeneous Medical Data Bases

M.H. WILLIAMS*
J. HU*

Abstract

As the number of data bases storing patient information increases, so the problem of fragmentation of the medical record becomes more and more serious. One solution is to interconnect medical data bases to create general medical information systems. This paper focusses on the problems of dealig with heterogeneities between existing data bases under the circumstance of a patient-centred query. The method of medical attribute extraction and mapping between attribute sets is discussed and a proposed architecture and its implementation are described in detail.

1 Introduction

The number of different data bases in use in the health care area for storing patient information has been growing steadily in recent years. At first sight this is a very positive statement, with clear advantages to medical practice. However, there is also a negative side to it. As the number of separate data bases storing patient information increases, so also does the fragmentation of the patient's medical record. As more and more information is stored electronically, this problem of fragmentation becomes increasingly serious. One important consequence of this is that only a partial picture of a patient's medical history is available at any point. This is undesirable, and potentially dangerous, for those responsible for diagnosis and treatment. Thus in general it is not sufficient to have access to only one medical data base; one may need access to a number of different data bases. The developments in distributed computing and in networking now make this possible.

**Computer Science Department, Heriot-Watt University, Riccarton, Edinburgh, EH 14 4AS, UK.*

Journal of Information Science and Technology, Vol. 2, No. 1, Oct. 1992.
ISSN 0971-1988 © 1992, ICCPI. Published by Tata McGraw-Hill.

One may begin by connecting together the different data bases within a hospital or even within a town or city but this could soon spread to all hospitals within a region, a country or even a collection of countries. Thus the situation may be characterized by a large number (potentially thousands or even tens of thousands) of *existing* data base sources located in *heterogeneous* hardware and software environments distributed among nodes of the computer network. The term *existing* refers to the fact that the data bases were created independently, in an uncoordinated way without consideration of the possibility that in due course they may need to cooperate with each other. The term *heterogeneous* indicates that the data bases may use different underlying data models, different ways of structuring the data, and different ways of representing the same data item, and even operate in different operating environments. The following gives a description of data inconsistencies that may exist in various medical data sources in the heterogeneous environment. These include:

1. *Naming conventions and naming conflicts*, in which semantically identical data items are named differently or semantically different data items are named identically.
2. *Data representation conflicts*, in which semantically identical data items are represented differently in different data sources. For example, data represented as a character string in one data base may be represented as a real number in another. It is also possible that semantically identical data items stored in different data bases may use different units of measure; for instance, for body temperature, one data base may use scale Centigrade, and another may use Fahrenheit.
3. *Data value conflicts*, a typical and difficult case in this category is the representation of fuzzy data values. For instance, body temperature can be represented by a specific real number; however, it may also be represented using 'High', 'Normal' or 'Low' to indicate a temperature range in another data base.
4. *Data structure differences*, in which semantically identical data items may use different data structures. For example, date_of_birth could be a single attribute in one data base, but be three attributes (dob_year, dob_month, dob_day) in another.
5. *Intranslatability between semantically related data items*; for instance, a diagnosis code for a patient in one data base cannot be translated directly to an appropriate code in another data source if they use different classification systems.

The problem of data inconsistencies is the main obstacle in data base communication. In practice, it is quite possible that more than one kind of heterogeneity is interwoven together so that the problem is made more complicated. Various researchers have worked on this problem and a number of methods have been proposed [1, 4, 6, 9, 11]. This paper presents a method to solve this problem under the assumption that the search policy in the medical information system is patient-centred (i.e. the aim is to provide information about individual patients) and that the patient identification is known. The rest of this paper is organized as

follows. First, the concept of mappings between medical attribute sets is presented. Then a general system architecture is depicted and explained by examples. Finally, the implementation of a prototype and some relevant aspects are described in detail.

2 Mappings between Attribute Sets

In a medical information system, the medical information required for a given patient can be thought of as a group of 'medical attribute values', e.g. body temperature, diagnosis, etc. For a local data base, all attributes concerned can be gathered to form a medical attribute set for such a data base. Therefore, in the environment of heterogeneous medical data bases, in order to enable data bases to communicate with each other, an important problem to be solved is how to overcome the heterogeneities of medical attributes in different data bases. A practical solution is through mappings between two local attribute sets. However, due to the potentially large number of data bases involved, a *global standard attribute set* is introduced.

The global standard attributes are agreed within the overall system, which mediates the communication between data bases. Therefore, a map from Local Attribute Set 1 to Local Attribute Set 2 is actually a compound map $f_1 \cdot f_2$, shown in Fig. 1. Furthermore, the map $f_1 \cdot f_2$ could be thought of as a query presented in Site 1 imposing on Site 2, and the map $f_2^{-1} \cdot f_1^{-1}$ as the results of the query returning to Site 1.

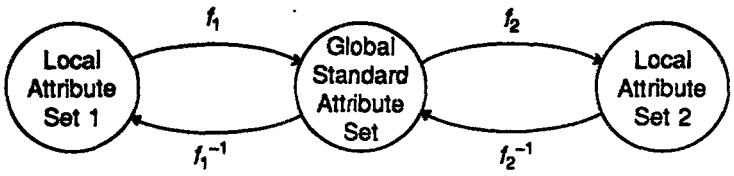


Fig. 1 *Mappings between attribute sets*

If this method is employed, a prime topic to be handled is how to construct the local attribute set in any existing data base without changing the structure and environment of the letter. The extended rlational model RM/T [3], first defined by Codd, is then investigated for this purpose. In the RM/T system, all entities are divided into three categories:

1. *Kernel entities* Kernel entities are those that have independent existence; they are "what the data base is really all about."
2. *Characteristic entities* A characteristic entity is one whose sole function is to describe or 'characterize' some other entity. Characteristics are existence-dependent on the entity they describe.
3. *Associative entities* An associative entity is an entity whose function is to represent a many-to-many relationship among two or more other entities. The entities associated may each be kernel, characteristic, or associative.

Entities can have *properties*, which fall into two categories: single-valued and multivalued. In any RM/T system, the single-valued (or immediate) properties of

an entity type are represented as distinctly named attributes of one or more property-defining relations, called *P-relations*. Nevertheless, the decompositions of immediate properties should be performed carefully. The general principle is to decompose properties no further than *minimal meaningful units* [3]. This raises questions such as: Should the day, month and year components of a date_of_birth be represented in separate binary P-relations?

Since RM/T is based on a well established and rigorous formalism and is known to be capable of simulating the semantics of widely different models besides the relational model, e.g. the functional and the object-oriented models [8], some of its methodology is absorbed to construct each local attribute set. The following describes how to capture the atomic semantic units to construct local attribute sets in relational data bases by using the notations in the RM/T model.

An atomic semantic unit, an atomic attribute, or an atom, is defined to be composed of one or more original attributes in a data base, which has a specific meaning to some extent but cannot exist alone. Each atom is associated with a type, and each type has a function which carries out data mappings. A P-relation in the RM/T system could be thought of as an atomic description of the entity. Rather than breaking up the original relations to a set of P-relations, a relation is introduced, called *atom_spec*, to describe every defined atom in all relations of a data base. This has the form

atom_spec (atom_id, relation, attribute, order)

When an atom is represented by more than one attribute, there is more than one tuple in the relation *atom_spec* to describe it, e.g. dob_m in Fig. 2(a). In this case the attribute *order* designates the relative position of a *relation attribute* in the atom.

Another problem which must be dealt with in any relational data base is that of multivalued attributes. Usually two methods could be adopted to handle it in an existing relational data base. In a traditional relational data base, a multivalued attribute may be divided into several single-valued attributes. For example, suppose that in an existing data base there is a relation *inpat* which describes a two-valued attribute *operation*.

inpat(crn, ... , dob_day, dob_month, dob_year, ... ,
opr_1_day, opr_1_month, opr_1_year, operation_1_text, operation_1_code,
opr_2_day, opr_2_month, opr_2_year, operation_2_text, operation_2_code,
...)

An illustration of atom capture and mapping from the standard attribute set to the local attribute set for this case is shown in Fig. 2(a), where two standard attributes, *date_of_birth* and *operation* and their mapping atoms are shown. The relation *multi_spec* describes all standard attributes which are multivalued. In this particular case, two intermediate single-valued pseudostandard attributes *operat_1* and *operat_2* are introduced, appearing in turn in the relation *att_map* which specifies the composition of single-valued standard attributes. Finally in the relation *atom_spec* they are related with two separate attributes representing information on an operation either as a text string or as a coded value.

Another method may be adopted in an existing data base by following the

multi_spec

multi_std	std_att
. . .	. . .
operation	operat_1
operation	operat_2
. . .	. . .

att_map

std_att	atom
. . .	. . .
date_of_birth	dob_m
. . .	. . .
operat_1	oper_m_1
operat_2	oper_m_2
. . .	. . .

atom_spec

atom	relation	attribute	order
. . .	. . .	. . .	. . .
dob_m	inpat	dob_day	2
dob_m	inpat	dob_month	1
dob_m	inpat	dob_year	0
oper_m_1	inpat	operation_1_text	0
oper_m_1	inpat	operation_1_code	1
oper_m_2	inpat	operation_2_text	0
oper_m_2	inpat	operation_2_code	1
. . .	. . .	. . .	. . .

Fig. 2(a) *Atom capture and attributes mapping (a)*

notation in the RM/T system, where entity types are so defined that each multi-valued property of an entity p is cast in the form of a characteristic entity q together with immediate properties for q. For example, the above relation *inpat* could be modelled as

inpat(crn, ... , dob_day, dob_month, dob_year, ... , operation, ...)
...
operation(crn, opr_day, opr_month, opr_year, operation_text, operation_code)

where, the attribute *operation* stores the *characteristic* relation operation which has three attributes in turn—*crn* (case reference number) as the internal patient identifier, *operation_text* and *operation_code*.

The atom description of atom *dob_m* (for date_of_birth) and atom *oper_m* (for operation) as well as the mapping from the standard attribute set to the local attribute set is shown in Fig. 2(b). There is no explicit indication as to whether or not the std_att *operation* multivalued, but this would be clarified during querying.

In addition to the atomic attribute capture, another important problem is the data value mapping between local attributes and standard attributes. This could be done by associating each atomic attribute with a 'type', each 'type' having a function

att_map

std_att	atom
...	...
date_of_birth	dob_m
...	...
operation	opr_date_m
operation	oper_m
...	...

atom_spec

atom	relation	attribute	order
...	...	...	...
dob_m	inpat	dob_day	2
dob_m	inpat	dob_month	1
dob_m	inpat	dob_year	0
opr_date_m	operation	opr_day	2
opr_date_m	operation	opr_month	1
opr_date_m	operation	opr_year	0
oper_m	operation	oper_text	0
oper_m	operation	oper_code	1
...	...	...	...

Fig. 2(b) *Atom capture and attributes mapping (b)*

which carries out data value transformations from local to standard. All mapping functions are gathered to form a function library.

Figure 3 gives part of the metadata supporting atomic attribute extraction and data value transformation, where:

(1)	multi_spec	multi_std		std_att	
(2)	att_map	std_att		atom	
(3)	atom_spec	atom	relation	attribute	order
(4)	atom_type	atom		type	
(5)	type_func	type		function	

Fig. 3 *Part of supporting metadata*

1. *multi_spec* specifies those standard attributes which are multivalued (only for the first case discussed above);
2. *att_map* specifies the mapping between global standard attributes and local atomic attributes. The value of a global standard attribute may be derived from the values of several local atomic attributes;
3. *atom_spec* indicates which original attribute(s) in a relation will comprise an atom;
4. *atom_type* indicates the corresponding type of an atom; and
5. *type_spec* specifies the function which transforms the local data values to standard ones for a type.

Using this structure each data base can be thought of as being encapsulated, i.e. a data base provides a subset of the global standard attribute set as an export interface through which the data related to exporting standard attributes in this data base can be shared by others whereas the details of attribute mapping are left hidden.

3 System Architecture

Based on the above discussion, a proposed architecture, shown in Fig. 4, is presented in the section for communication between existing heterogeneous medical data bases. A description of functionality modules is first given, and then two examples are provided for further clarification.

3.1 A Description of Functionality Modules

(a) Navigation Module
The *navigation module* accepts a query referring to a patient presented by a user through the User Interface. In response it gives information about where (in which data bases) the specified medical records on a particular patient might be found.

For a system characterized by a large number of existing data bases and considering the fact that medical information systems are usually patient-centred, the best way to implement this navigation facility is to build up a series of distributed index and create appropriate supporting software utilities. An essential part of the indexes has the form (patient_id, data base) indicating which data bases contain the information relating to a given patient. Another basic part of the indexes is of the form (data base, std_att) designating which standard attributes are stored in a data base. A detailed discussion of navigation among a large number of heterogeneous medical data bases is given in [5].

(b) Query Language (QL) Module 1
This module accepts queries expressed in the local query language and local attributes, and translates them to a standard query language which can be used throughout the medical information system. Since the system is only concerned with medical information about individual patients, the standard query language is actually defined as an SQL-like 'select' statement:

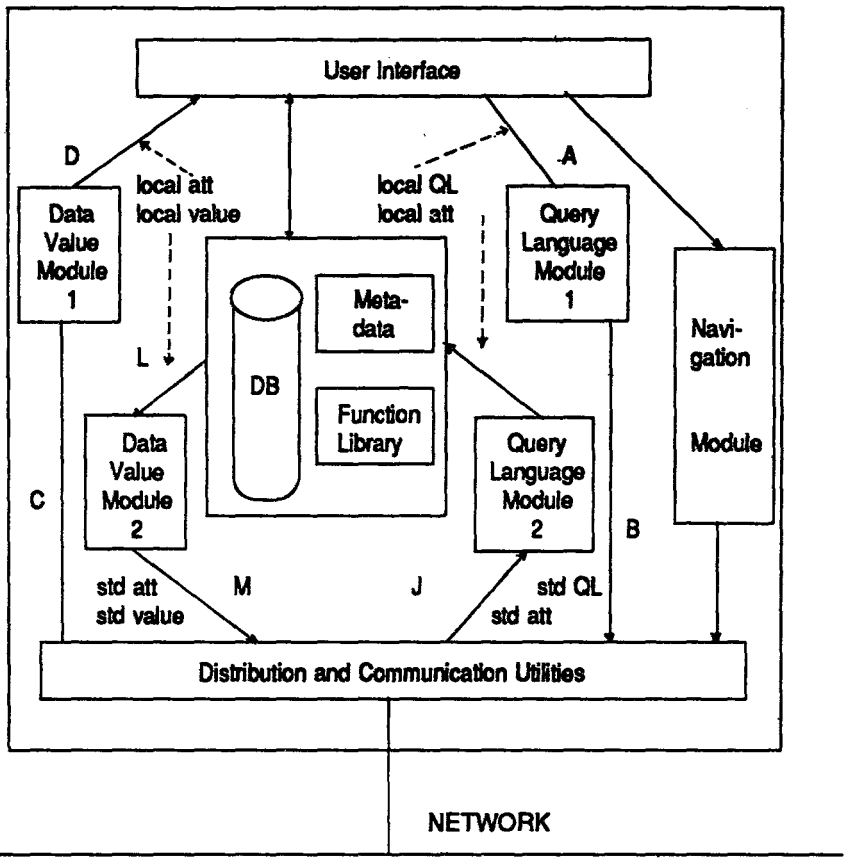


Fig. 4 *A proposed architecture*

select <std_att list>
from <data base list>
where pid=<patient id> and <condition>

where *from <data base list>* is dependent on the contribution of the navigation module, and *<condition>* specifies additional restrictions on the information, e.g. date between July 1989 and June 1990.

(c) Query Language (QL) Module 2
This module receives queries which have arisen from some other data base systems and which are expressed in the standard query language and standard attributes, such as *select . . . from . . . where . . .* . The standard QL is translated into the local one and the standard attributes are mapped into local attributes based on metadata (in a relational data base, the local attributes are specified by relations and attributes). This query is then applied to the local data base to obtain the relevant information.

(d) Data Values (DV) Module 2
This module takes in from QL Module 2 the results expressed in local attributes and local data values, changes the data values into the standard form, and then tags

and organizes them into the standard data transfer format. Some form of tagged structure may be adopted as the standard data transfer format. Nevertheless, the exact format is a matter for investigation. Usually, a criterion in choosing a representation should be of semantic richness so that the semantic meanings of a various kinds of data can be expressed.

In this prototype, the notation of *structures* in Prolog has been used. In Prolog, a structure is thought of as a single object which consists of a collection of other objects, called components. The components are grouped together into a single structure for convenience of handling. Structures help organise data because they permit a group of related information to be treated as a single object rather than as separate entities. Each component could be a structure (possibly recursive) so that complex objects can be expressed [2]. Due to such advantages, the concept of structures is adopted for the standard data transfer format. In addition, since the concept of *lists* in Prolog is very useful for handling objects with similar types, it is naturally employed to deal with the case of multivalued data. A description of the BNF grammar of the structure for the tagging is as follows:

<structure>	::=	<tag> (<term_list>).
<tag>	::=	<identifier>
<term_list>	::=	<term> [, <term_list>]
<term>	::=	<identifier>
	|	<char string>
	|	<number>
	|	<structure>
	|	<list>
<list>	::=	' [' ']'
	|	'[' <term_list>']'
<identifier>	::=	letters, digits and underscore, starting with a small letter
<char string>	::=	'any characters ... '
<number>	::=	signed or unsigned, real or integer number

The definitions of tags are agreed through the whole information system, while the explanations are various in different local data bases. The provinces of local explanations are confined within the respective local data bases. For instances, Fig. 5 shows an example of a local explanation for some tags. The *function* attribute specifies the name of the function which performs the data value transformation for the tagged data type.

(e) Data Values (DV) Module 1

The response to any previously presented query is expressed in the standard data transfer format in which standard data values of relevant medical attributes are tagged. For any local data base, each tagged data type (a structure, simple or nested) has associated with it a function which analyses the data according to a certain format and translates the data values into local representations. This module performs the detagging of results by calling the corresponding functions to transform them into the local format and local values, and then presents the information for display. This process is supported by table *tag_spec* which relates a tag to a local name and a function. Figure 5 shows an example.

tag_spec

tag	local_name	function
b1	blood_group	t_bg_func
b9	blood_pressure	t_bp_func
d	date	t_date_func
b	bld_p_rslt	t_bpr_func
t5	body_temper	t_temp_func
t	temp_rslt	t_tempr_func
sm	smoking	t_smoke_func
o7	occupation	t_occup_func
d4	diagnosis	t_diag_func
dx	diag_text	t_diagt_func
dc	diag_code	t_diagc_func
ds	classifying_system	t_diags_func
st	disease_set	t_dis_func
u3	urine_test	t_utst_func
u100	urine_protein	t_up_func
u110	urine_sugar	t_us_func
a34	alcohol	t_alc_func

Fig. 5 *Tags explanation*

Besides the case described above, another situation must be considered in which the user just wants some information beyond the scope specified by his local data base. The best way is to consult the navigation module which will tell what information is stored in which data bases. Therefore, there is a need for a user to present his query against standard attributes. And when the results are returned the display could only be in terms of the general data format.

As a summary, Fig. 6 gives all supporting metadata for the attribute mapping, where

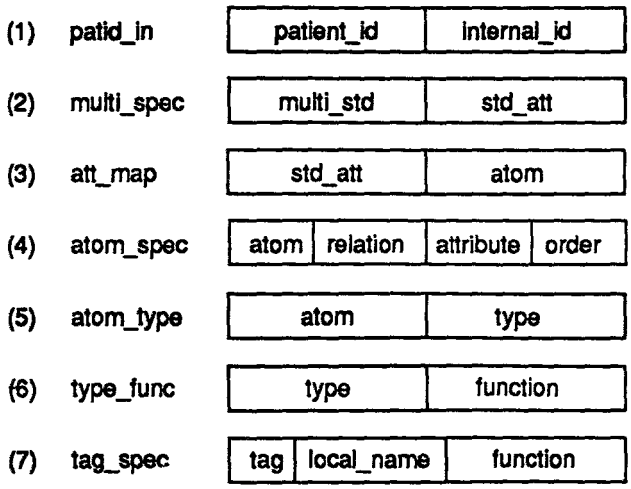


Fig. 6 *Metadata*

(1) *patid_in* converts the global patient identifier to the internal,
(2) –(6) are the same as those in Fig. 3, and
(7) has been explained in the description of DV Module 1.

3.2 Examples

Suppose there are three relational data bases (illustrated in Fig. 7) which contain some medical information on a patient Mrs X. *clinic_db* is an Ingres data base whose query language is QUEL, *infirmary_db* is an Oracle data base whose query language

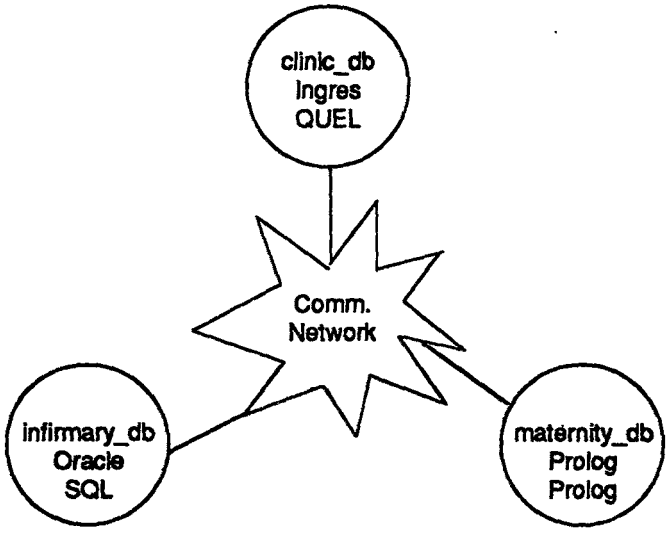


Fig. 7 *The prototype*

is SQL and *maternity_db* is an Prolog_implementing relational data base whose query language is PROLOG. The descriptions of these three data bases are given in the appendix. The following shows the process of querying several medical attributes: blood_group, blood_pressure, body_temper, smoking, occupation, diagnosis.

Suppose that in the model data base clinic_db (Ingres with QUEL) a user presents the following statements:

```
range of p is personal
range of c is normal_chk
range of r is record
retrieve (c. bgrp, c. bp_1, c.-temp, r. smoking, p. occupation, r. diag_text)
where pid="XId"
```

which involve the standard attributes: blood_group, blood_pressure, body_temper, smoking, occupation and diagnosis.

According to the metadata, the QL module 1 will produce a standard query

```
select blood_group, blood_pressure, body_temper,
   smoking, occupation, diagnosis
where pid="XId"
```

The navigation module will work out which data bases are relevant to this query. For this particular case, two data bases *infirmary_db* and *maternity_db* are involved. Then two queries derived from the 'original standard query' are sent to the two data base sites respectively.

```
select blood_group, occupation, diagnosis
from infirmary_db
where pid="XId"
select blood_group, blood_pressure, body_temp,
   smoking, occupation, diagnosis
from maternity
where pid="XId"
```

When a search request reaches the destination data base, the QL module 2 in that data base transforms it into the local query for the relevant attributes and applies it to the data base. The data values of the relevant attributes are first changed to the standard form, and then tagged and organized into the standard data transfer format (shown in Fig. 8) by DV module 2. These tagged and organized data are then sent back to the sender in response to the search request. In Fig. 8, the argument of *sm* (smoking) is the number of cigarettes smoked per day; the arguments of *o7* (occupation) comprise a list containing one or more standard occupation codes.

```
from_infirmary_db
b1(2).
o7([100]).
d4([d4(d(891024),dx('disease_a'), dc(37), ds(ICD9'), st([]),
   d4(d(891024),dx('disease_b'),dc(8),ds('ICD9'),st([]))]).
end_of_infirmary_db
```

Fig. 8(a) *Tagged data (a)*

```
from_maternity_db
b1(2).
b9(d(900109), b(140,90)).
t5(d(900109), t(100.4)).
sm(40,100).
o7([39, 100, 278]).
d4(d(null),dx('disease_x'),dc(123),ds('Local'),st([a_dis_1,a_dis_2,a_dis_3]))
end_of_maternity_db
```

Fig. 8(b) *Tagged data (b)*

After the results (shown in Fig. 8) are returned, they are analysed and detagged by the DV Module 1 (see Fig. 5) and the final results might be displayed as shown in Fig. 9. Since the diag_code from infirmary_db is locally(infirmary_db)-based,

```
Data from infirmary_db
blood_group : AB
occupation : operator
diagnosis :
    date: 24-Oct-89,
    diag_text: disease_a,
    diag_code: 37

    date: 24-Oct-89
    diag_text: disease_b,
    diag_code: 8

Data from maternity_db
blood_group: AB
blood_pressure :
    date: 9-Jan-90,
    bp_h: 140,
    bp_l: 90
body_temper :
    date: 9-Jan-90,
    temp: 38
smoking: A
occupation : [ clerk, operator, secretary ]
diagnosis :
    date: null
    diag_text: disease_x,
diag_code: (4 : 70%, 9 : 40%, 5 : 10%)
```

Fig. 9 *The display form*

the values of *dx* (diag_text) and *st* (disease_set) are analysed and the resulting diag_code might be: 4 of 70% probability, 9 of 40% and 5 of 10%.

Another example is more straightforward. Suppose a user wants to find information other than that already retrieved from his local data base. The navigation module needs to be called first to consult the indexes and determine which data bases store what information in terms of standard attributes. The query is then presented using the standard attributes. After the results in terms of standard format are returned, DV Module 1 processes them for submitting to display. Suppose, for instance, that Ms Y's records about alcohol and urine tests are required. The consulting of indexes gives the information that *urine_test* is in infirmary_db and *alcohol* is in maternity_db. Therefore, the following queries

retrieve std_att (urine_test) from infirmary_db where pid="YId"
retrieve std_att (alcohol) from maternity_db where pid="YId"

are presented and sent to *infirmary_db* and *maternity_db* respectively. The results in intermediate format and in display might be as shown in Fig. 10.

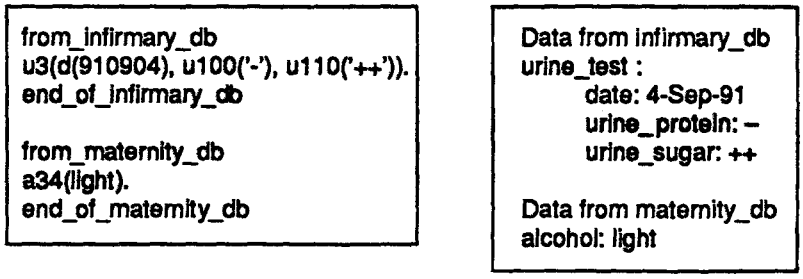


Fig. 10 *The tagged data and the display form*

4 Implementation

A prototype has been created based on the proposed architecture, where three relational data bases of different types, Ingres RDBMS with the language QUEL, Oracle RDBMS with the language SQL and a relational data base simulated in Prolog, are involved. These data bases are situated at different sites (machines) which are connected to each other through the communication network, as shown in Fig. 7. There is a process at each site referred to as the *server* which runs continually and handles the remote query requests from other sites. The working flowchart of the server is shown in Fig. 11, where the main task is fulfilled in the child process by calling utilities of QL module 2 and DV module 2. The remainder of this section gives a detailed description of the issues in the implementation.

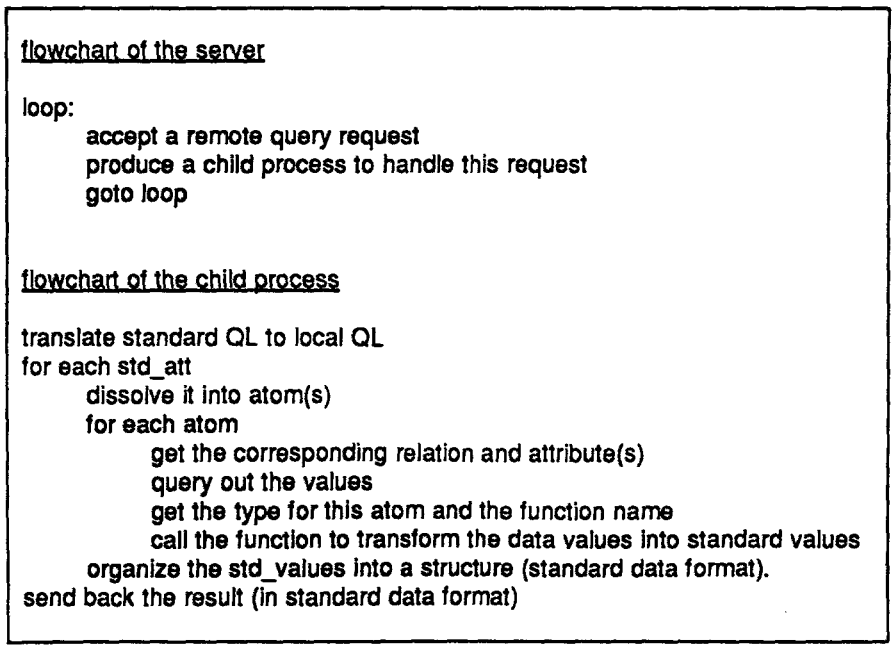


Fig. 11 *Working flowchart of the server*

4.1 Ingres and Oracle

(a) QL Module 1: an Interpreter/Translator
Most data base systems provide an interactive interface for users to present their queries expressed in a specific query language. Ingres and Oracle are typical examples of this. In order to let users operate in accordance with the environment to which they are accustomed, an interactive interpreter/translator simulating the original interactive environment is constructed. In the case of Unix, this interactive interpreter/translator is implemented by using the development tools *lex* and *yacc*. Consider the case of Ingres with QUEL as an example. Two QUEL statements—range and retrieve are supported in the multidata base query interpreter. The local attributes appearing in the QUEL statements, in the form of *tuple_variable attribute*, are translated into appropriate standard attributes. The interpreter/translator for the Oracle data base is implemented in a similar way.

(b) QL Module 2: Embedded Query Interface
The data base search for an external query is carried out by means of an embedded query facility which allows queries to be embedded in a host programming language, e.g. C. As the query commands are varied in each search request, i.e. the types of query statements can only be known at runtime, the embedded dynamic query facility is needed. The Oracle RDBMS provides a dynamic SQL facility which can be utilized directly for this purpose [7]. However, since Ingres does not provide such a dynamic facility explicitly, a 'dynamic' tool for this application was constructed by using the *param* function and the system catalogue [12].

(c) DV Modules 1 and 2:: Function Library
All functions for data value transformation are implemented in C. There are two kinds of functions. The first takes the value of an atom (which may include several original attributes in a relation) as input argument and translates it into standard data values. The second transforms the standard data values into local ones. The collection of all such functions forms a function library.

4.2 Relational Model Implemented in Prolog

Before the issues of implementing the four modules are discussed, two important aspects in this case must be clarified first: the *name-position conversion* and the *interface to foreign language functions.*

In Prolog, arguments are mapped by their position in the structure, whereas attribute names are not supported at all. In the relational model, the attributes of relations are referred to by names, which proves to be even more useful in a heterogeneous environment. To solve this problem, a data dictionary, which maintains information about each of the tables in the data base, is introduced. Each entry in the data dictionary is a template of the table being defined, giving the name of the table and the names of the columns of the table. An example is shown as follows:

```
table (mother(mid, surname, forename, ...)).
table(pregn(mid, ...)).
...
```

Based on such a data dictionary some predicates are constructed which convert the attribute names specified to the corresponding position list.

In Sicstus Prolog an interface for foreign language functions is provided through three built-in predicates [10].

```
foreign_file(+ObjectFile, +Functions)
foreign(+CFunctionName, +Language, +Predicate)
load_foreign_files(+ObjectFiles, +Libraries)
```

The third argument of the predicate foreign/3 + *Predicate* specifies the name of the Prolog predicate that will be used to call *CFunction( )*. It also specifies how to translate between Prolog arguments and C arguments. The permitted data types in Sicstus include: integer, float, atom, string, and address. By taking advantage of this feature, all data value transformation functions are implemented in C and each of them might be called as a predicate in Prolog.

Having discussed the above two basic aspects, we shall now give a brief description of the implementation of the four modules.

(a) QL Module 1: Prolog Interpreter

A Prolog interpreter provides the user with an interactive environment in which he can present his queries in the form of Prolog predicates. For the case of multidatabases, a new predicate is designed for the multidatabase query. Whenever this predicate is invoked, it is translated into the standard query form which is then passed on to the navigation module and transmitted.

(b) QL Module 2: 'Embedded' Query Facility

Unlike Oracle or Ingres, there is no 'embedded' facility which allows Prolog queries to be embedded into a C program. Therefore, the search task is actually performed by Prolog predicates. The main search predicate *select* provides the entry through which a search request could be presented to the Prolog data base. This predicate can be called in the server by taking advantage of the Unix pipe facility, '<<' function. Suppose the data base is saved as *maternity_db*, and can be called using this name whenever needed. Therefore if the server issues the command

```
maternity_db << !
select ([blood_pressure, diagnosis], pid, Pid).
!
```

the data base will be opened and the predicate *select* invoked to conduct a search of the specified attributes for a given patient and data value transformation by calling additional predicates including name-position conversion predicates and those which are implemented as C-functions.

(c) DV Modules 1 and 2: Functions Library

The data value transformation functions are implemented in C. The only difference between these and the ones for Ingres and Oracle is that all functions in this circumstance are defined as Prolog predicates.

5 Conclusion

As the number of data bases storing patient information increases, the problem of fragmentation of the medical record becomes more and more serious. One important consequence of this is that only a partial picture of a patient's medical history is available at any point. This is undesirable, and potentially dangerous, for those responsible for diagnosis and treatment.

One solution to this problem is to interconnect existing heterogeneous medical data bases to create general medical information systems. One obstacle that must be overcome is the heterogeneity between different data bases. This includes data conflicts in naming, representation, precision, and structure, as well as data intranslatability. Due to the fact that medical information systems are usually patient-centred, one solution to this problem involves:

1. constructing local attributes under the principle of atomic semantics, and
2. carrying out mappings between the sets of local medical attributes.

Under the circumstance of a large number of existing data bases a further improvement is made by introducing a set of global standard medical attributes which serves as a mediator between data bases. The system fulfils the mappings based upon metadata and the function library so that a user of one data base can obtain relevant information from other data sources without any knowledge of their scheme details.

A prototype has been implemented which links together three different types of data base management systems—Ingres RDBMS and Oracle RDBMS with their own query languages, and a relational data model implemented in Sicstus Prolog.

Appendix

A description of three model data bases is given here.

(1) **clinic_db** (Ingres DBMS with the language QUEL)
(a) personal relation

crn	surname	forename	sex	dob	address	occupation	phone
2469	X	M	female	12-Jan-60	1 Any Street	clerk	4564321
2468	Y	N	male	14-Feb-65	2 Any Terrace	teacher	4441111

(b) normal_chk relation

crn	surname	date	bp_h	bp_l	bgrp	temperature
2468	X	23-Sep-90	120	70	AB	37.200
2469	Y	2-Jul-89	170	90	B	35.800

(c) record relation

crn	date	smoking	symptom	diag_text	diag_code
2468	30-Nov-90	A	headache	any disease	2
2469	3-Jan-90	L	cough	one disease	29

The attribute crn (case reference number) is the key in each relation. The domain of smoking is: A(Addict), H(heavy), S(Smoker), L(light), N(Non-smoker).

(2) Inflrmary_db (Oracle DBMS with the language SQL)

(a) impatient relation

crn	date	surname	sex	birthdate	occupation
123	9/7/88	X	F	30/11/50	operator
789	8/12/90	Y	F	3/1/55	teacher

diagnosis 1	diagnosis 2	disch_date	disch_code
disease_a	disease_b	24/10/89	49
disease_p	disease_q	24/2/92	3

(b) labtest relation

crn	urine_date	urine_p	urine_s	grp_date	bld_grp	rh_date	rh_shift
123	14/5/89	+	–	14/5/89	AB	29/5/89	+
789	4/9/91	–	++	4/9/91	B		

The attribute crn (case reference number) is the key in each relation.

(3) **maternity_db** (Prolog implementing relation model with the language PROLOG

The data dictionary of each table is as follows:

table (mother (mid, surname, forename, dob, marital, occup_code, bld_group, main_dis)).
table (pregn (mid, smokes, alcohol, chk_date, temperature, bp_syst,bp_diast,
hb_date,hb_result)).

The key in each relation is mid (mother identifier). The attribute occup_code in relation mother gives an occupation code which indicates a type of occupations, and the classification criterion is local. The domains of both smokes and alcohol are: heavy, mod, light, non.

(a) mother relation

mother (m9203, 'X', 'M', '20-Sep-60', m, 38, ab, disease_x).
mother (m7234, 'Y', 'N', '20-Sep-63', m, 74, b, null).

(b) pregn relation

pregn (m9203, heavy, mod, '9-Jan-90', 38, 140, 90, null, null).
pregn (m7234, non, light, '18-Jun-91', 37, 130, 80, null, null).

REFERENCES

1. Breitbart Y., Multidatabase interoperability, *SIGMOD Record* vol. 19, no. 3, pp 53–60, Sep. 1990.
2. Clocksin W.F. and Mellish C.S., *Programming in Prolog*, Springer-Verlag, 1984.
3. Codd E.F., Extending the data base relational model to capture more meaning, *ACM Trans. Data base Systems*, vol. 4, no. 4, pp 397–434, Dec. 1979.
4. Heimbigner D. and McLeod D., A federated architecture for information management, *ACM Trans. Office Information Systems*, vol. 3, no. 3, pp 253–278, July 1985.
5. Hu J. and Williams M.H., Navigation among multiple heterogeneous medical data bases, *submitted for publication*.
6. Litwin W., Mark L., and Roussopoulos N., Interoperability of multiple autonomous data bases, *ACM Computing Survey*, vol. 22, no. 3, pp 267–293, Sep. 1990.
7. *Oracle Reference Manual Series*, Oracle Corporation, California, 1985, 1987, 1989.
8. Papazoglou M.P., Marinos L., and Bourbakis N.G., Distributed heterogeneous information systems & schema transformation, *Proc. Parbase-90 Int. Conf. on Data bases, Parallel Architectures and their Applications*, pp 388–397, 1990.
9. Sheth A.P. and Larson J.A., Federated data base systems for managing distributed, heterogeneous, and autonomous data bases, *ACM Computing Survey*, vol. 22, no. 3, pp 183–236, Sep. 1990.
10. Carlsson M., *Sicstus Prolog User's Manual*, Swedish Institute of Computer Science, 1988.
11. Thomas G., Thompson G.R., Chung C., Barkmeyer E., Carter F., Templeton M., Fox S., and Hartman B., Heterogeneous distributed data bases systems for production use, *ACM Computing Survey*, vol. 22, no. 3, pp 237–266, Sep. 1990.
12. Woodfill J., Siegal P., Ranstrom J., Meyer M., and Allman E, *INGRES Reference Manual*, Electron. Res. Lab. Univ. of California, Berkeley, 1988.

89. M.H.WILLIAMS, Batch sizes for the batching method of colouring planar maps, Information Processing Letters, vol. 11, nos. 4-5, 186-189 (1980).

BATCH SIZES FOR THE BATCHING METHOD OF COLOURING PLANAR MAPS

M.H. WILLIAMS
Department of Computer Science, Rhodes University, Grahamstown, South Africa

Received 11 April 1980

Graph theory, map colouring

1. Introduction

In a recent paper Lipton and Miller [1] pointed out that by collecting the regions of a planar map into batches and then removing these by reduction, one can colour any map with at most five colours in $O(n \log n)$ rather than in n^2 time [2].

This paper reviews their results and attempts to obtain, for the case of cubic maps, an improved estimate for the lower bound for the number of regions in a batch. Since the number of regions in a batch affects the efficiency of their method, the improved estimate shows that their method is likely to be more efficient than their paper suggests.

Before discussing these ideas a brief explanation of terminology is necessary. Whereas Lipton and Miller dealt with the planar graph associated with a map, in which each vertex of the graph represents a region of the map, this paper deals directly with the map itself.

A *map* consists of a connected planar graph G embedded in a plane so that it divides the plane into regions. Two regions in a map are said to be *neighbours* if their boundaries have an edge in common, otherwise they are *nonadjacent*.

The following assumptions will be made with respect to maps:

(a) Maps will be assumed to be cubic, i.e. each vertex in the graph G has exactly three edges connected to it.

(b) Multiple adjacencies do not occur, i.e. no two regions may have more than one edge in common.

(c) No edge may have the same region on either side of it.

From the point of view of colouring a map will be assumed to consist of at least five regions, since one can terminate the reduction process if one reaches a map consisting of five or fewer regions. The *degree* of a region is the number of neighbours which it has. Each region in a cubic map has a degree of at least three.

2. Size of a batch

For a cubic map with n_3 regions of degree three, n_4 regions of degree four, etc., it has been shown that [3]

$$\frac{3n_3}{2} + n_4 + \frac{n_5}{2} - \frac{n_7}{2} - n_8 - \frac{3n_9}{2} - \cdots = 6. \qquad (1)$$

The number of regions with degree $\leqslant 6 = N = n_3 + n_4 + n_5 + n_6$. The number of regions with degree $>6 = M = n_7 + n_8 + \cdots$.

Lipton and Miller showed that more than one quarter of the total number of regions in any map must have a degree less than seven (i.e. $3N > M$). Since in the worst case N will consist only of regions of degree three and M only of regions of degree seven, the result follows immediately from (1).

They argue further that if all of these regions are of degree *six*, and if all are adjacent to one another, then on eliminating neighbours from the set to produce a set of nonadjacent regions, the set will be reduced to one seventh its original size. Hence for a map with n regions the minimum number of regions in a batch must be $>\frac{1}{28}n$.

However, if most of the regions are of degree six

(they could not all have degree six) then the number of regions of degree less than seven in a map would be considerably more than one quarter of the total number of regions. Likewise, if the regions with degree less than seven are all regions of degree three then on eliminating neighbours from the set, the set will be reduced to one quarter of its original size. Thus it would seem that $\frac{1}{28}n$ may be too drastic a lower bound.

Before looking at a few examples, consider the reductions which can be performed on the map [3,4]:

(a) A region of degree three can be replaced by a single vertex. Because of the initial assumption about multiple adjacencies, a region of degree three cannot have as neighbour another region of degree three nor can it have as neighbours more than one region of degree four. Thus at worst it can have one region of degree four and two of degree five. In this case removing n_3 regions of degree three will eliminate a further n_3 regions of degree four and $2n_3$ regions of degree five.

(b) A *cluster of degree four* is a set of three regions of degree four which share a common vertex. A cluster of degree four can be replaced by a single vertex. The three regions which neighbour on the cluster must each have a degree of at least five, and no more than two of these regions may have degree five. Thus at worst the cluster can be surrounded by two regions of degree five and one of degree six.

(c) A region of degree four which is not part of a cluster, can be removed by merging it with two of its neighbours. Such a region can have at most two other regions of degree four as neighbours.

(d) Each region of degree five or six can be removed by merging it with two of its neighbours.

In the process of reduction the regions of degree three will be removed first followed by any clusters of degree four, and then any regions of degree four which are not part of a cluster, and so on. In the worst case there will be no clusters of degree four – so these will be ignored for the remainder of the discussion.

Consider a few special cases:

(a) A map containing only regions of degree three and seven

In this case

$$3n_3 = n_7 + 12,$$

and since each region of degree three has no neighbours of degree three, the number of regions in a batch is greater than $\frac{1}{4}n$.

(b) A map containing only regions of degree four and seven

In this case

$$2n_4 = n_7 + 12.$$

On reducing the set to obtain a set of nonadjacent regions the set of regions of degree four could at worst be reduced to one third of its size. Thus the number of regions in a batch exceeds $\frac{1}{9}n$.

(c) A map containing only regions of degree five and seven

In this case

$$n_5 = n_7 + 12.$$

On reducing the set to obtain a set of nonadjacent regions the set of regions of degree five could be reduced to one sixth of the size. Thus the number of regions in a batch is greater than $\frac{1}{12}n$.

3. General case

(a) A map containing only regions of degree three, four, five, six and seven

In such a map

$$3n_3 + 2n_4 + n_5 = n_7 + 12,$$

and there can be any number of regions of degree six. The two extreme cases are:

(i) Every region of degree three, four and five is surrounded by regions of degree six. Assume that

$$n_6 \geqslant 3n_3 + 4n_4 + 5n_5.$$

Now removal of regions of degree less than seven yields a set of at least

$$n_3 + n_4 + n_5 + \frac{n_6 - 3n_3 - 4n_4 - 5n_5}{7}$$

nonadjacent regions. The total number of regions in the map is

$$n_3 + n_4 + n_5 + n_6 + n_7.$$

Thus the proportion of regions in a batch to the total

number of regions is

$$\frac{7n_3 + 7n_4 + 7n_5 + n_6 - 3n_3 - 4n_4 - 5n_5}{7(n_3 + n_4 + n_5 + n_6 + (3n_3 + 2n_4 + n_5 - 12))}$$

which is approximately equal to

$$\frac{4n_3 + 3n_4 + 2n_5 + n_6}{7(4n_3 + 3n_4 + 2n_5 + n_6)} = \frac{1}{7}.$$

(ii) Every region of degree three is surrounded by one region of degree four and two regions of degree five, every other region of degree four is surrounded by two regions of degree four and two of degree five, every remaining region of degree five is surrounded by five other regions of degree five and there are no regions of degree six.

Now extract n_3 regions of degree three. These will all be nonadjacent and one will be left with

$n_4 - n_3$ regions of degree four,

and

$n_5 - 2n_3$ regions of degree five.

Next extract the regions of degree four. This yields $(n_4 - n_3)/3$ nonadjacent regions of degree four and leaves

$$\frac{3n_5 - 2n_4 - 4n_3}{3}$$ regions of degree five.

Finally, extract the regions of degree five. This gives $(3n_5 - 2n_4 - 4n_3)/18$ nonadjacent regions of degree five. It is assumed that $n_4 \geqslant n_3$, $n_5 \geqslant \frac{2}{3}n_4 + \frac{4}{3}n_3$.

Hence, the total number of regions in a batch is

$$n_3 + \frac{n_4 - n_3}{3} + \frac{3n_5 - 2n_4 - 4n_3}{18} = \frac{8n_3 + 4n_4 + 3n_5}{18}.$$

The total number of regions in the map is

$n_3 + n_4 + n_5 + n_7$.

Thus, the ratio of the number of regions in a batch to the total number of regions is

$$\frac{8n_3 + 4n_4 + 3n_5}{18(n_3 + n_4 + n_5 + 3n_3 + 2n_4 + n_5 - 12)}$$

which is approximately equal to

$$\frac{8n_3 + 4n_4 + 3n_5}{18(4n_3 + 3n_4 + 2n_5)}.$$

The extreme cases are:

(1) $n_3 = n_4$, $n_5 = \frac{2n_4}{3} + \frac{4n_3}{3}$ $(= 2n_3)$.

If $n_3 = n_4$ the proportion reduces to

$$\frac{12n_3 + 3n_5}{18(7n_3 + 2n_5)}.$$

If $n_5 = 2n_3$ it reduces further to

$$\frac{18n_3}{18(11n_3)} = \frac{1}{11}.$$

(2) $n_3 = n_4$, $n_5 \gg n_3$.

The larger n_5 becomes, the more the value tends to

$$\frac{3n_5}{36n_5} = \frac{1}{12}$$

(i.e. the value obtained in special case (c) in Section 2).

(3) $n_5 = \frac{2n_4}{3} + \frac{4n_3}{3}$, $n_4 \gg n_3$.

If $n_5 = (2n_4/3) + (4n_3/3)$ the proportion reduces to

$$\frac{(12n_3 + 6n_4)}{6(20n_3 + 13n_4)}.$$

The larger n_4 becomes the more the value tends to

$$\frac{6n_4}{6 \times 13n_4} = \frac{1}{13}.$$

(b) A map containing only regions of degree four, five, six and seven

In this case

$2n_4 + n_5 = n_7 + 12$,

and there can be any number of regions of degree six in the map. The extreme case in which each region of degree four or five is surrounded by regions of degree six can be obtained directly from case a(i) above by setting $n_3 = 0$. The ratio remains $\frac{1}{7}$.

In the other extreme every region of degree four is surrounded by two other regions of degree four and two regions of degree five, every remaining region of degree five is surrounded by five other regions of degree five and there are no regions of degree six.

First extract the regions of degree four. This gives $n_4/3$ nonadjacent regions of degree four and leaves

$n_5 - (2n_4/3)$ regions of degree five. Then extract regions of degree give. This yields $(3n_5 - 2n_4)/18$ non-adjacent regions of degree five. It is assumed that $n_5 \geqslant 2n_4/3$.

Hence, the total number of regions in a batch is at least

$$\frac{n_4}{3} + \frac{3n_5 - 2n_4}{18} = \frac{4n_4 + 3n_5}{18}.$$

The total number of regions in the map is

$$n_4 + n_5 + n_7 = n_4 + n_5 + 2n_4 + n_5 - 12.$$

Thus, the proportion of regions in a batch to the total number of regions is approximately

$$\frac{4n_4 + 3n_5}{18(3n_4 + 2n_5)}.$$

There are two cases to consider:

(1) $n_5 >> n_4$.

In this case the formula reduces to

$$\frac{3n_5}{36n_5} = \frac{1}{12}$$

which corresponds to special case c in Section 2.

(2) $n_5 = 2n_4/3$.

The formula now reduces to

$$\frac{4n_4 + 2n_4}{6(9n_4 + 4n_4)} = \frac{1}{13}$$

which is the same result as was obtained in special case 3 of a(ii) above.

Although this does not take account of every case it does cover the worst possible situations. Thus, the size of a batch must always be at least n/13 where n is the total number of regions in the map.

In practice, it is exceedingly unlikely that a large map containing regions of degree seven will have all the regions of degree seven clustered together in one area and all regions with degree less than seven clustered together in another. Furthermore, it is likely that such a map will also contain regions of degree eight or more. Thus, the size of a batch will almost certainly be considerably greater than n/13.

Finally, a word of caution about applying the method of reduction to planar maps. If one is careful about applying the reductions and follows the steps suggested in [4] (or something similar) the map produced at each stage will satisfy the assumptions made in the first section. However, if one does not follow these steps, one may have a map such as that in Fig. 1a which after reduction yields the configuration shown in Fig. 1b. This is a map with multiple adjacencies. Applying further reductions to this map could even lead to a disconnected map, i.e. a map with islands.

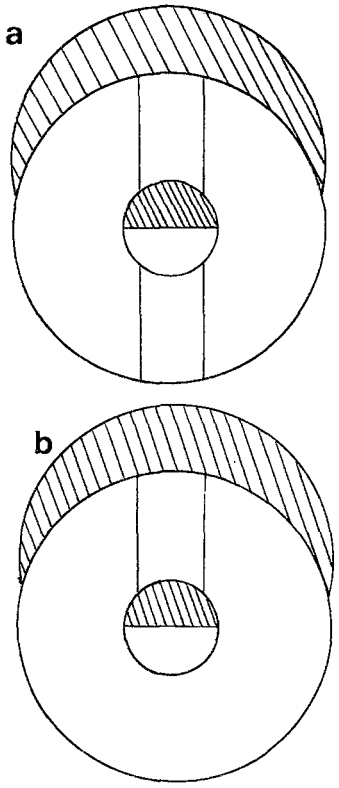


Fig. 1. (a) Map before reduction. (b) Map after reduction. Shaded area indicates two or more regions.

References

[1] R.J. Lipton and R.E. Miller, A batching method for coloring planar graphs, Information Processing Lett. 7(4) (1978) 185–188.

[2] D.S. Johnson, Worst case behaviour of graph coloring algorithms, in: Proc. Fifth Southeastern Conference on Combinatorics, Graph Theory and Computing (Utilitas Mathematica, Winnipeg, Canada, 1974) 513–528.

[3] M.H. Williams, Cubic map configurations, Information Processing Lett. 11 (4, 5) (1980) 180–185.

[4] M.H. Williams, Generating random planar maps, Comput. J., submitted.

90. M.H.WILLIAMS, A linear algorithm for colouring planar graphs with five colours, The Computer Journal, vol. 28, no. 1, 78-81 (1985).

A Linear Algorithm for Colouring Planar Graphs with Five Colours

M. H. WILLIAMS
Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ

A linear algorithm for colouring planar graphs with at most five colours has recently been published. However, this algorithm, which operates by recursive reduction of the graph, is unnecessarily complicated. An alternative method which is much simpler is presented in this paper.

1. INTRODUCTION

Chiba, Nishizeki and Saito[1] recently put forward a linear algorithm for colouring a planar graph which employs at most five colours. However, their method is unnecessarily complicated and a much simpler algorithm is presented in this paper.

The nomenclature which is used here is the same as that followed in Refs. 1 and 2. All graphs considered here are assumed to be connected and have no multiple edge. The number of vertices in a graph, G, is denoted by n. For any vertex v, the set of all vertices adjacent to v is termed the *neighbourhood of* v,$N(v)$, and the number of vertices in this set is termed the *degree of* v,$d(v)$. The basic operations used in the algorithm are *deletion of a vertex* v which removes v from G, and *identification or merging of vertices* u *and* v which removes u and adds to $N(v)$ any vertices from $N(u)$ which are not already in $N(v)$. When a vertex is removed all edges incident upon that vertex are discarded.

2. THE ALGORITHM

This algorithm is based on successively removing vertices of degree less than six from the graph until at most 5 vertices remain. These vertices can then be coloured with at most five colours. The vertices which had been removed are then reinstated one by one and coloured appropriately.

Two basic transformations are used in reducing the graph, viz.

(1) if $d(v) \leqslant 4$ then remove v (type 1 reduction)
(2) if $d(v) = 5$ then remove v and identify a pair of non-adjacent vertices in $N(v)$ (type 2 reduction).

In order to implement this, the graph G is represented by an adjacency list $L[v]$ for each $v \varepsilon V$ while two queues, Q_4 and Q_5, defined as

$$Q_4 = \{v | d(v) \leqq 4\}$$

$$Q_5 = \{v | d(v) = 5\}$$

are used to contain all vertices available for reduction. Both the adjacency lists and the queues are doubly linked, as are the two copies of each edge (u, v) in $L[u]$ and $L[v]$.

In addition a stack S is used to contain vertices removed from the graph. Each element of S consists of two fields:

(*a*) the vertex v removed, and
(*b*) either a pointer to the adjacency list for v at the time v was removed, or a vertex u with which v was identified.

Before reducing a vertex of degree 5, its neighbours are checked to find two neighbouring vertices with sufficiently low degree which can be identified. For the purposes of the algorithm the vertices selected must have degree less than some constant k. The value of k will be defined later.

The process of identifying two vertices involves the merging of their two neighbour sets, while taking care to avoid the creation of multiple edges. To this end a Boolean array MARK[v] is used, the elements of which are all set to false initially. Each member of the neighbour set of vertex 1 is 'marked' by setting the corresponding entry of MARK to true. Then each 'unmarked' member of the neighbour set of vertex 2 is added to the neighbour set of vertex 1 while each 'marked' member is simply ignored.

Two additional arrays used are: DEG[v] stores the degree of each vertex v, while DP[v] contains a pointer to an element v in Q_i ($4 \leqq i \leqq 5$) if v is present in Q_i.

The algorithm is as follows:

```
procedure COLOUR5;
  procedure CHECK (w);
  if DEG[w] = 5 then add w to Q5
  else if DEG[w] = 4 then move w from Q5 to Q4;
  procedure DELETE(v);
    begin
    for wεL[v] do
      begin
      delete v from L[w];
      DEG[w]: = DEG [w]−1;
      CHECK (w)
      end;
    push (v, pointer to L[v]) on to S;
    delete from Qi the node pointed to by DP [v];
    noofvert: = noofvert−1
    end;
  procedure IDENTIFY (u, v);
    begin
    for wεL[v] do MARK [w]: = true;
    for wεL[u] do
      begin
      delete u from L[w];
      if not MARK[w]
      then begin (*w is adjacent to u but not v*)
        add w to L[v]; add v to L[w];
        DEG[v]: = DEG[v]+1;
        if DEG[v] = 6 then delete v from Q5
        else if DEG[v] = 5 then move v from Q4 to Q5
        end
      else begin
        DEG[w]: = DEG[w]−1;
        CHECK (w)
        end
      end;
```

```
    for wεL[v] do MARK[w]: = false;
    delete u from Q_i;
    push (u, v) on to S;
    noofvert: = noofvert − 1
    end;
procedure REDUCE;
while noofvert > 5 do
if Q_4 ≠ Φ then DELETE (top entry from Q_4)
else begin
    take top entry v from Q_5;
    if two non-adjacent vertices x,yεN(v) can be found such
    that DEG[x] < k and DEG[y] < k
    then begin
        DELETE(v);
        IDENTIFY (x, y)
        end
    else return v to rear of Q_5
    end;
procedure COLOUR;
    begin
    colour remaining vertices (i.e. those on Q_4);
    while S not empty do
        begin
        pop top element (x, y) from S;
        if y is pointer to adjacency list L then colour x with colour
        different from vertices on L
        else colour x with same colour as vertex y
        end
    end;
begin
scan graph and set up arrays MARK, DEG, Q_4, Q_5 and DP;
noofvert: = n;
REDUCE;
COLOUR
end (*OF COLOUR5*)
```

3. TIME COMPLEXITY

Initially Q_4 is set up to contain those vertices with degree less than or equal to four, Q_5 to contain those with degree five. That both queues cannot be empty follows from the fundamental property of planar graphs that the average degree of vertices in a planar graph is less than 6. We now establish the following theorem concerning the queues Q_i.

Theorem 1

At the beginning of each iteration of the loop within the procedure REDUCE the queues Q_4 and Q_5 will contain all vertices v with $d(v) < 6$ in the graph.

Proof. Proof is by induction. Assume that the theorem holds at some point in the execution of REDUCE. The only operations performed by REDUCE which can alter the state of the graph or the queues (other than their order) are as follows

(*a*) DELETE (top entry from Q_4) or DELETE(v). In this case a vertex is deleted from the graph and from Q_i. It is deleted from the adjacency lists of each of its neighbours and the degree of each neighbour is decreased by one. After doing so the degree of each neighbour is checked and if it is less than 6, the neighbour is added to Q_5 or moved from Q_5 to Q_4 if necessary. No other vertex of the graph will be affected by the deletion. Thus after the deletion the Q_i will contain all vertices v with $d(v) < 6$.

(*b*) IDENTIFY (x, y). In this case vertex x is deleted from the graph and from Q_i, and each of the neighbours of x which is not a neighbour of y will be connected to y. As in (*a*), the degree of each affected vertex is checked so that after identification of x and y the Q_i will contain all vertices v with $d(v) < 6$.

Since Q_4 and Q_5 are set up initially to contain all vertices v with d(v) < 6, the theorem is proved. ∎

Since the average degree of vertices in a planar graph is less than six, these queues can never both be empty as long as there are vertices in the graph.

Noofvert is a variable which represents the number of vertices in the graph. It is initially set to have value n and is reduced by 1 in each of the procedures DELETE and IDENTIFY when a vertex is removed from the graph.

Theorem 2

The procedure COLOUR5 will colour any planar graph G with at most five colours.

Proof. Each reduction step removes either one or two vertices from the graph until it has no more than five vertices. These can be coloured with at most five colours.

In the final phase each of the vertices removed from the graph is unstacked from the stack S and assigned a colour. If at the time that it was removed from the graph a vertex v had:

(*a*) fewer than five neighbours

then v will be connected to vertices coloured with at most four colours;

(*b*) five neighbours

then S will contain two entries: an entry E_1 identifying one of the neighbours of v, x, with another, y, and below it an entry E_2 containing a pointer to an adjacency list with five vertices (including x and y). At this point a colour will have been assigned to vertex y, and entry E_1 can be unstacked and vertex x assigned the same colour. Then E_2 can be unstacked, and v will be connected to vertices coloured with at most four colours.

Thus each vertex can be coloured with one of at most 5 colours. ∎

Now consider in more detail the way in which Q_5 is handled and vertices are selected for identification. Initially Q_5 is set up to contain all vertices with degree 5. In the procedure REDUCE an entry may be removed from the front end of Q_5. If the vertex corresponding to this entry has two non-adjacent neighbours of sufficiently small degree, it will be deleted; otherwise the vertex will be returned to the rear end of the queue (such a vertex will be termed a *blocked vertex*). During the process of a deletion or identification vertices may be removed from any point within the queue or added to the rear end of the queue.

Consider a particular instance of Q_5 at a point where Q_4 is empty. The term *stage* will be used to refer to the period from this point until Q_4 is (once again) empty and all the original entries on this instance of Q_5 have been removed, so that Q_5 consists entirely of new entries and entries which have been returned to the rear of the queue (blocked vertices) or the graph has been collapsed to a point that it contains less than six vertices. The proof of linearity which follows later relies on the fact that the total number of blocked vertices encountered (or, if you prefer, the total number of times a vertex is returned to the end of Q_5) depends linearly on n.

The first result which is required concerns the maximum number of blocked vertices in a stage.

Lemma (Upper bound on number of blocked vertices in a stage.)
If at the beginning of a stage a graph contains n_i vertices of degree $i (i \geqq 5)$, and during the stage d type 2 reductions and e type 1 reductions are performed, the maximum number of blocked vertices encountered during the stage (B) cannot exceed

$$\sum_{i=k} in_i/2 + (2k-4)\,d/2.$$

Proof. Consider a vertex v of degree 5 in the graph. For v to be a blocked vertex it must have at least two neighbours of degree at least k (since otherwise it is always possible to find two non-adjacent neighbours each with degree less than k).

If each vertex of degree i ($i \geqq k$) has edges in common with i blocked vertices then since each blocked vertex must have edges in common with at least two vertices of degree greater than or equal to k, it follows that, ignoring any new vertices of degree at least k created during the stage, the number of blocked vertices cannot exceed

$$\sum_{i=k} in_i/2.$$

Since for each type 2 reduction performed during the stage, a pair of vertices will be identified, this could create up to d new vertices of degree greater than k. As the degree of each of the vertices identified must be at most $k-1$ before deletion and identification, the largest degree which the resulting vertex may have is $2k-4$. Each type 1 reduction only decreases the degrees of vertices and thus cannot create additional blocked vertices.

Hence the total number of blocked vertices encountered during the stage may not exceed

$$\sum_{i=k} in_i/2 + (2k-4)\,d/2. \quad \blacksquare$$

For the current purposes k will be taken to be the value 13.

Theorem 3

If at the beginning of a stage a graph contains n_i vertices of degree i ($i \geqq 5$) and during the stage R reductions are performed and B blocked vertices are encountered then

$$\frac{B}{R} < 237.$$

Proof. Assume that during the stage d type 2 reductions and e type 1 reductions are performed. For each type 2 reduction one vertex of degree 5 will be deleted and the degree of each of its neighbours decremented (removing the neighbour from Q_5 if necessary). For each type 1 reduction a vertex of degree less than 5 is deleted and the degree of each neighbour decremented (removing it from Q_5 if necessary). Thus altogether $fd+ge$ vertices will be removed from Q_5 during the stage, where $1 \leqq f \leqq 6$ and $0 \leqq g \leqq 4$. All other vertices originally on Q_5 must be blocked vertices, i.e.

$$\text{no. of blocked vertices} = B = n_5 - fd - ge.$$

From ref. 2, for a maximal planar graph (i.e. one in which each face is bounded by three edges)[3] the number of vertices of degree 5 initially on Q_5 is given by

$$n_5 = 12 + \sum_{i=1} in_{i+6}.$$

If the graph at the beginning of the stage is not maximal planar and requires the addition of m edges to make it maximal planar, it can easily be shown that

$$n_5 = 12 + 2m + \sum_{i=1} in_{i+6}, \qquad (1)$$

from which it follows that

$$B = 12 + 2m + \sum_{i=1} in_{i+6} - fd - ge. \qquad (2)$$

From the previous lemma

$$12 + 2m + \sum_{i=1} in_{i+6} - fd - ge \leqslant \sum_{i=k} in_i/2 + (2k-4)\,d/2.$$

For $k = 13$

$$11d + fd + ge \geqq 12 + 2m + \sum_{i=1} in_{i+6} - \sum_{i=13} in_i/2$$

$$= 12 + 2m + \sum_{i=1}^{6} in_{i+6} + \sum_{i=1} in_{i+12}/2. \qquad (3)$$

Now since $(11+f)\,d + ge \leqslant (11+f)\,(d+e)$

$$\leqslant 17\,(d+e)$$
$$= 17R,$$

equation (3) becomes

$$17R \geqslant 12 + 2m + \sum_{i=1}^{6} in_{i+6} + \sum_{i=1} in_{i+12}/2. \qquad (4)$$

Since $f \geqq 1$,

$$\frac{11d}{12} \leqslant \frac{11fd}{12}.$$

Adding $fd/12$ to each side gives

$$fd \geqslant (11+f)\,d/12$$

and hence

$$(fd+ge) \geqslant ((11+f)\,d+ge)/12$$

Combining this with equations (2) and (3) gives

$$B \leqslant 12 + 2m + \sum_{i=1} in_{i+6} - \frac{1}{12}\left(12 + 2m + \sum_{i=1}^{6} in_{i+6} + \sum_{i=1} in_{i+12}/2\right)$$

$$= \frac{11}{12}\left(12 + 2m + \sum_{i=1}^{6} in_{i+6}\right) + \sum_{i=1}\left(\frac{23i}{24} + 6\right) n_{i+12}. \qquad (5)$$

Comparing each term of (5) with the corresponding term of (4) the largest ratio occurs when $i = 13$, namely

$$\frac{167}{24}\Big/\frac{1}{2} = \frac{167}{12},$$

from which it follows that

$$\frac{B}{17R} \leqq \frac{167}{12}$$

or

$$B < 237R. \quad \blacksquare$$

Theorem 4

The procedure COLOUR5 will colour any planar graph G containing n vertices in $O(n)$ time.

Proof. It can easily be verified that the procedure DELETE deletes a vertex v from the graph in $O(d(v))$ time

where $d(v) \leqslant 5$. Since this procedure can be called at most $n-5$ times the contribution to the total running time is at most $O(n)$.

Likewise the procedure IDENTIFY (x, y) can be seen to operate in time of order $O(d(x)+d(y))$. Since for each call $d(x) < k$ and $d(y) < k$, and since the number of calls can be at most $(n-5)/2$, the total time spent by the procedure IDENTIFY is at most $O(n)$.

On each iteration of the loop in the procedure REDUCE either at least one vertex will be removed from the graph or a blocked vertex will be moved from the front to the rear of the queue. From the previous theorem the total number of blocked vertices encountered in all the stages must be less than 237 times the total number of reductions performed, i.e.

$$\Sigma B < 237 \Sigma R.$$

Since the total number of reductions performed is at most $n-5$, the total number of blocked vertices must be less than 237 $(n-5)$ and hence the number of iterations of the loop in the procedure REDUCE is $O(n)$. Each operation can be shown to operate in time less than or equal to some fixed constant (since even the test for two non-adjacent vertices $x, y \varepsilon N(v)$ such that DEG$[x] < k$ and DEG$[y] < k$ depends only on k).

In the procedure COLOUR the stack S is unstacked element by element and either

(a) an adjacency list of up to 5 elements is scanned and a set S′ of colours used is constructed; then a colour not contained in this list is selected (this is always possible otherwise Theorem 2 would not hold), or

(b) the colour of vertex y is assigned to vertex x.

Since the stack contains just less than n elements, the time taken for this operation is $O(n)$.

Hence the procedure COLOUR5 operates in $O(n)$ time. ∎

This algorithm has been tested using a set of 3000 planar graphs, each with between 50 and 200 vertices, and an analysis of its performance is given in ref. 4.

REFERENCES

1. N. Chiba, T. Nishizeki and N. Saito, A linear 5-coloring algorithm of graphs, *Journal of Algorithms* **2**, 317–327 (1981).
2. M. H. Williams, Cubic map configurations, *Information Processing Letters* **11**, 180–185 (1980).
3. O. Ore, *The Four-color Problem*. Academic Press, New York (1967).
4. M. H. Williams and K. T. Milne, The performance of algorithms for colouring planar graphs. *The Computer Journal* **27**, 165–170 (1984).

91. M.H.WILLIAMS, Cubic map configurations, Information Processing Letters, vol. 11, nos. 4-5, 180-185 (1980).

CUBIC MAP CONFIGURATIONS

M.H. WILLIAMS
Department of Computer Science, Rhodes University, Grahamstown, South Africa

Received 15 April 1980; revised version received 27 May 1980

Map colouring, cubic map

1. Introduction

A planar map consists of a connected planar graph embedded in a plane so that it divides the plane into regions. In addition the following two assumptions are generally made with regard to planar maps [1]:

(a) No two regions of a map have more than one edge in common (i.e. no multiple adjacencies).

(b) No edge can have the same region on either side of it.

If each vertex of a planar map has three edges incident to it, the map is called a cubic map.

Each region in a map is bounded by the edges of the graph and two regions are said to be neighbours if they share a common edge. The degree of a region is the number of edges in its boundary (or the number of neighbours which it has). The order of a map is the degree of the region with smallest degree in the map.

A necessary condition for planarity is given by Euler's formula which states simply that for a map with v vertices, e edges and r regions

$$v - e + r = 2 .$$

Applying this formula to cubic map configurations reveals that not all the configurations predicted by the formula are realizable.

2. Possible configurations

Since in a cubic map each vertex has three edges incident to it, and each edge will be incident to two vertices

$$3v = 2e .$$

Euler's formula states that

$$2 = v - e + r .$$

Substituting the first equation into the second gives

$$e = 3r - 6 . \tag{1}$$

For a map with n_3 regions of degree 3, n_4 regions of degree 4, n_5 regions of degree 5, etc., the number of edges is given by

$$e = \frac{3n_3 + 4n_4 + 5n_5 + 6n_6 + 7n_7 + \cdots}{2} ,$$

and the total number of regions is

$$r = n_3 + n_4 + n_5 + n_6 + n_7 + \cdots .$$

Substituting these into (1) yields

$$\frac{3n_3}{2} + n_4 + \frac{n_5}{2} - \frac{n_7}{2} - n_8 - \frac{3n_9}{2} - \cdots = 6 . \tag{2}$$

Consider a few of the configurations which are possible:

(*a*) *Cubic map in which each region is of degree 3*

In this case, (2) reduces to

$$3n_3/2 = 6 .$$

Hence, there is only one configuration in which each region is of degree 3, and this consists of exactly four regions.

(*b*) *Cubic map in which each region is of degree 4*

There is only one configuration consisting of exactly six regions.

(*c*) *Cubic map in which each region is of degree 5*

Once again there is only one configuration consisting of exactly twelve regions.

(*d*) *Cubic map in which each region is either of degree 3 or of degree 5*

In this case, (2) becomes

$$\frac{3n_3}{2} + \frac{n_5}{2} = 6 .$$

There are five possible solutions to this:

$n_5 = 0, n_3 = 4$ – each region is of degree 3 (case a above)
$n_5 = 3, n_3 = 3$
$n_5 = 6, n_3 = 2$
$n_5 = 9, n_3 = 1$
$n_5 = 12, n_3 = 0$ – each region is of degree 5 (case c above)

(*e*) *Cubic map in which each region is either of degree 5 or of degree 6*

Equation (2) now becomes

$$n_5/2 = 6 .$$

In other words a map consisting only of regions of degree 5 and regions of degree 6 must have exactly twelve regions of degree 5 but may have any number of regions of degree 6.

(*f*) *Cubic map in which each region is either of degree 5 or of degree 7*

In this case

$$\frac{n_5}{2} - \frac{n_7}{2} = 6 .$$

Again there are an infinite number of possible configurations – each with twelve more regions of degree 5 than there are regions of degree 7.

3. Nonrealizable configurations

If one attempts to construct these various maps, it will be found that not all the configurations predicted by the formula are realizable. To illustrate this, Fig. 1 shows the possible configurations of cubic maps in

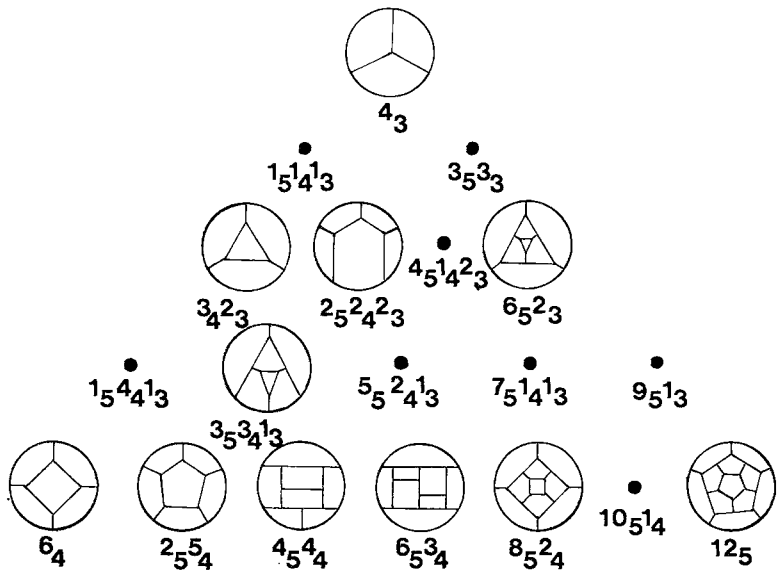


Fig. 1. Cubic map configurations containing only regions of degree 3, 4 and 5, predicted by Euler's formula. Non-realizable configurations are indicated by a black dot.

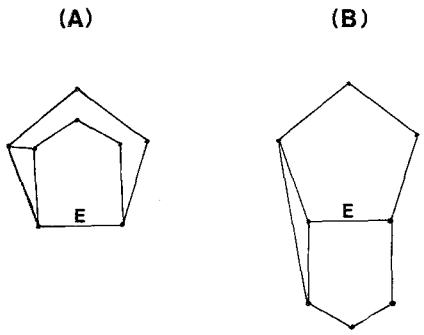


Fig. 2. Partial construction of the map $3_5 3_3$ with two edges and three regions (one of degree 5, two of degree 3) outstanding.

which the degree of each region is between three and five. Those configurations predicted by the formula which are not realizable are indicated by means of a black dot. The notation $i_j k_\ell \cdots$ is used to denote a map with i regions of degree j, k regions of degree ℓ, etc.

Note that in some cases there are two or more different ways of realizing a particular configuration. These alternative representations are not given here.

To verify that some of these configurations are not realizable, consider the case of a configuration with three regions of degree 5 and three of degree 3 ($3_5 3_3$). From (1) this map has twelve edges. Now since there are only three regions which are not of degree 5, two of the regions of degree 5 must have an edge in common (call this edge E). This leaves one region of degree 5 and three of degree 3.

Consider the two regions at either end of E. Since there is only one region left which does not have degree 3, at least one of the two regions at either end of E must have degree 3. At this stage the partial construction of the map must look like one of the two configurations given in Fig. 2, with two edges outstanding and 3 regions (one of degree 5, two of degree 3) to be created. It is obviously impossible to construct this configuration.

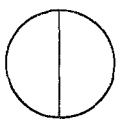


Fig. 3. The map 3_2.

4. Method of generating cubic maps

In another paper [2] a method is described for generating cubic maps, and a proof is given that the method will cater for any map. Briefly the method is as follows:

Start with the initial map $n_2 = 3$ as shown in Fig. 3. Repeatedly select one of the following steps and execute it until a map of the desired size is obtained:

(*a*) *Create region of degree 3.* Select an arbitrary vertex and replace it by a region of degree three (Fig. 4A).

(*b*) *Create region of degree 4.* Select an arbitrary region X and choose two of the edges (e_1 and e_2) on its boundary. Join e_1 and e_2 with two edges thereby

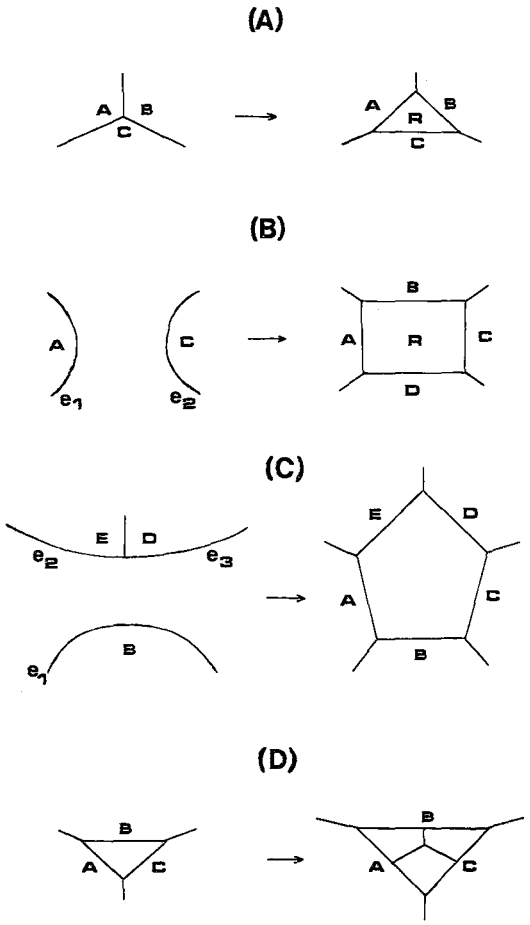


Fig. 4. The four substeps which may be used to create any cubic map.

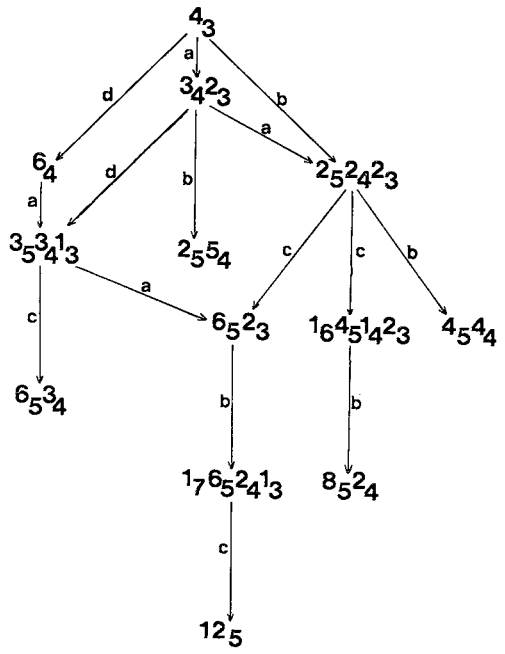


Fig. 5. Illustration of how the realizable configurations of Fig. 1 can be derived using the method of map generation.

replacing X by one region of degree four and two other regions flanking it (Fig. 4B).

(*c*) *Create region of degree 5.* Select an arbitrary region X and choose one edge (e_1) and two other edges different from e_1 and meeting in a common vertex (call them e_2 and e_3). Join each of e_2 and e_3 to e_1 with an edge, thereby replacing X with three regions (Fig. 4C).

(*d*) *Create three regions of degree 4.* Select a region of degree three and replace it with three regions of degree four (Fig. 4D).

Fig. 5 shows the way in which the configurations of Fig. 1 may be derived by this method. The letters labelling the edges in the diagram represent the steps (as given above) which must be applied in order to move from one configuration to another. From this the nonrealizable configurations can be deduced. It would be very useful if a method could be found for predicting directly whether or not a particular configuration is realizable. However, such a method has not been found and this indirect approach appears to offer a systematic, though crude, alternative.

5. Maps of order five

Cubic map configurations involving only regions of degree five or more are of particular interest from the point of view of colouring planar maps. This section looks at some properties of maps of order five.

First, a few more terms will be introduced. Three

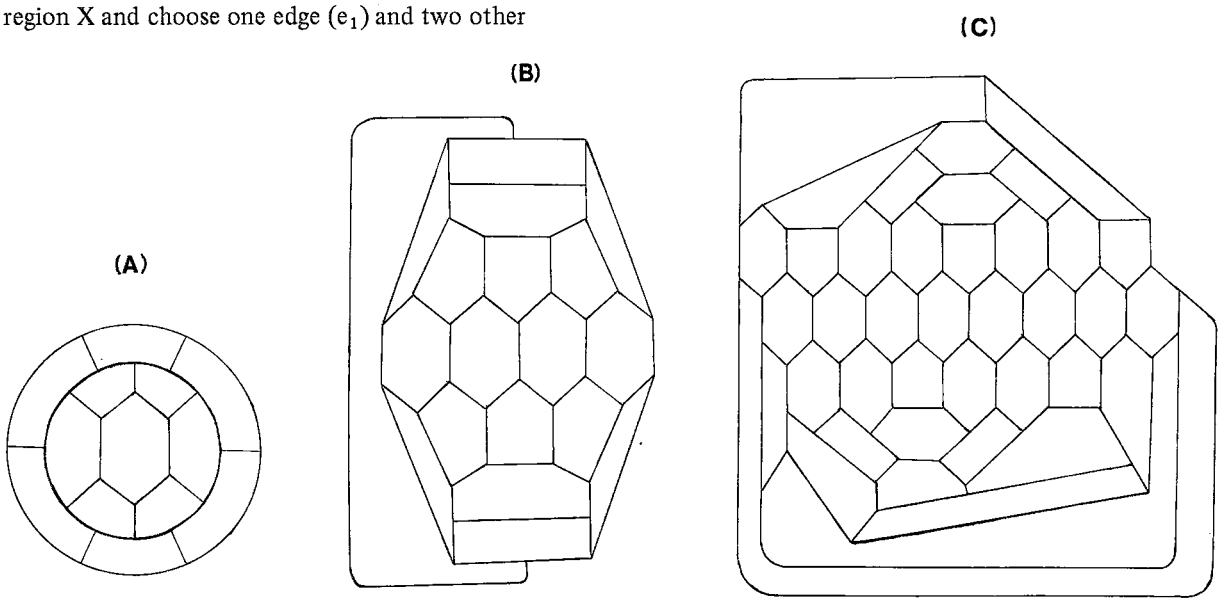


Fig. 6. Some configuration involving only regions of degree 5 and 6. (A) $2_6 12_5$, (B) $8_6 12_5$, (C) $20_6 12_5$.

regions of degree n which share a common vertex are referred to as a *cluster of degree n.* Three or more regions of degree n which share common edges (so that any two regions of the set are joined by a path of zero or more regions from the set) but which do not form a cluster, are said to constitute a *chain of degree n.* Two regions of degree n which share a common edge but have no other neighbours of degree n are referred to as a *neighbouring pair of degree n* (or simply a *pair of degree n*). A region of degree n which has no neighbours of degree n is called an *isolated region of degree n.*

For the purposes of this section we will regard a vertex as being made up of three *angles*, one for each region adjacent to the vertex. The angles which form the border of a region of degree n will be called *angles of degree n.*

Some configurations involving only regions of degree five and six are shown in Fig. 6. From Section 2 such configurations must consist of exactly twelve regions of degree five and an arbitrary number of regions of degree six. It turns out that the configuration consisting of twelve regions of degree five and one region of degree six is not realizable. Thus, the configuration shown in Fig. 6A ($2_6 12_5$) is the minimum configuration containing both regions of degree five and regions of degree six.

The configuration $2_6 12_5$ contains a number of clusters of degree five. Suppose that one wants the minimum configuration consisting only of regions of degree five and six which contains no clusters of degree five. Such a configuration must contain twelve regions of degree five and hence 60 angles of degree five. Since clusters of degree five are not permitted, each region of degree five must be adjacent to at most two other regions of degree five, and hence at least twelve of the angles of degree five will belong to vertices whose other angles are of degree six (i.e. for which there must be 24 angles of degree 6) while at most 48 of the angles of degree five belong to vertices having two angles of degree five and one of degree six (i.e. another 24 angles of degree six). The resulting configuration must possess at least 48 angles of degree six. This means that the map must contain at least 8 regions of degree six. Fig. 6b shows the configuration $8_6 12_5$ which is the minimum configuration satisfying these conditions.

The configuration given in Fig. 6B contains chains of degree five. Suppose that one wants the minimum configuration which contains only regions of degree six and isolated regions of degree five. In this case each region of degree five is surrounded by regions of degree six. Hence the map must contain at least 60 vertices.

Since there are twelve regions of degree five, there must be 60 angles of degree five, which means that there must be at least 120 angles of degree six.

Since each region of degree six has six angles of degree six, the minimum number of regions of degree six which is required, is 20. Fig. 6C shows the map $20_6 12_5$ which satisfies the condition of having only isolated regions of degree five.

In the case of configurations involving only regions of degree five and seven, the configuration $1_7 13_5$ is not realizable and the minimum realizable configuration is $2_7 14_5$.

A map consisting only of regions of degree five and seven and in which regions of degree five appear only in pairs or as isolated regions, will contain n regions of degree 7 and n + 12 regions of degree 5. Since each pair of degree 5 requires 8 vertices, there must be at least ((n + 12)/2) * 8 vertices in the map. Hence, there must be at least 12 * (n + 12) angles in the map.

Since there are n + 12 regions of degree 5, these will contribute 60 + 5n angles; the n regions of degree 7 will contribute 7n angles. Hence the total number of angles in the map must be 60 + 12n. However, this is less than the minimum required if the regions of degree five are to occur only in pairs or as isolated regions.

Thus, a map consisting only of regions of degree 5 and regions of degree 7 must contain at least one chain of degree 5.

Similarly, it can be shown that the minimum configuration consisting only of regions of degree five and seven and containing no clusters of degree five must consist of at least forty-four regions ($16_7 28_5$), while any configuration consisting only of regions of degree five and eight or more must contain at least one cluster of degree five.

Finally, the average degree of a region in a map is given by

$$a = \frac{\sum_{i=1}^{r} \text{no. of edges of region } i}{r} = \frac{2e}{r}.$$

Substituting, (1) gives

$$a = 6 - (12/r)$$

from which it follows that the average degree of a region in a cubic map depends only on the total number of regions in the map.

References

[1] T.L. Saaty and P.C. Kainen, The Four Color Problem (McGraw-Hill, New York, 1977).
[2] M.H. Williams, Generating random planar maps, Comput. J., submitted.

92. M.H.WILLIAMS and K.T.MILNE, The performance of algorithms for colouring planar graphs, The Computer Journal, vol. 27, no. 2, 165-170 (1984).

The Performance of Algorithms for Colouring Planar Graphs

M. H. Williams and K. T. Milne
Computer Science Department, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ, UK

As the number of different methods for colouring graphs increases so also does the need to evaluate and compare their relative performances. Of particular interest is the case of planar graphs. In the study described in this paper a set of 6000 planar graphs was generated using several different techniques. Each graph has between 20 and 200 vertices. These graphs were then used to compare the performance of ten different colouring algorithms. Results show clearly the advantage of the reduction algorithms for colouring planar graphs.

INTRODUCTION

The problem of colouring graphs or maps and the associated problems of scheduling have attracted considerable interest for many years, and numerous methods have been proposed for performing colourings. As the number of such methods increases it is essential to have some quantitative comparison of their performance. For this purpose one may concentrate either on *planar* graphs (i.e. graphs which can be drawn in a plane without any edge crossing over another edge or vertex) or *non-planar* graphs (i.e. graphs which cannot)—or equivalently planar or non-planar maps.

Brélaz[1] provided such a comparison using eleven of these methods to colour a set of general non-planar graphs. In his study he used a total of 80 graphs. These were divided into groups according to the number of vertices in the graph (each group contained between two and nine graphs, each with between 10 and 100 vertices) and according to the density of the graph (four groups, each with either 15 or 25 graphs). However, the small sample size used makes it difficult to know what credence to place on the results. This is underlined by the wide variations in the results tabulated in his Table 4.

This paper describes a similar comparison performed on a different set of algorithms using a set of planar graphs. In this case a much larger set of graphs has been used in order to obtain more reliable estimates of the performance of the various methods under investigation.

In a graph the number of edges connected to some vertex v is referred to as the *degree of* v; the average number of edges per vertex for the graph as a whole is referred to as the *edge density*. The process of generating non-planar graphs is trivial and different edge densities may be obtained simply by stopping the process at different points. However, in the case of planar graphs, the edge density is determined solely by the number of vertices in the graph. The only variation which can be achieved is in the distribution of the degrees of the individual vertices.

For this reason three different methods were used to generate the planar graphs. Each starts with some initial graph and repeatedly modifies it to produce progressively larger graphs until one with the required number of vertices is reached. The modification procedures employed were:

(a) Randomly select a vertex v with edges connecting to some set of neighbours N and replace it with a set of three interconnected vertices which share the original neighbours of v.
(b) Randomly select any vertex v and replace it by a set of interconnected vertices, the degree of each of which is at least five.
(c) Randomly select any vertex v in the graph with degree at least five and replace it by a pair of vertices.

In each case the set of neighbours of each vertex in the graph must be maintained in some order (clockwise or anticlockwise) so that the planarity of the graph is preserved through each step of the modification process.

Each of the three methods was used to produce 10 sets of graphs, with about 20, 25, 30, 35, 40, 50, 75, 100, 150 and 200 vertices, respectively. Each set comprises 200 graphs, making a total of 6000 graphs in all. The different distributions produced by each of the three methods are shown in Fig. 1. A fourth method was also tried but it yielded a distribution very similar to that produced by (a) above, and has therefore been omitted from this study.

THE GRAPH COLOURING ALGORITHMS

In order to compare the performance of different graph colouring algorithms a Pascal program was implemented for each method and executed with five or six different sets of graphs (corresponding to different graph sizes) for each of the three types of graph (a total of either 3000 or 3600 graphs in each case). Whenever an algorithm was applied to a set of 200 graphs, a running average of the time taken to colour each graph was computed and used to determine the average colouring time. The programs were run on a Burroughs B6930 and the program timings obtained are in units of 1/60th of a second.

Two of the methods used are exact methods, i.e. methods which colour the vertices of a planar graph using exactly four colours. The remainder are heuristic methods which require less time than the exact methods but in general use more than four colours. In each case the average colouring times and the numbers of colours required are given in Tables 1 and 2 respectively. Colouring times for individual methods have also been illustrated graphically in some cases.

CCC-0010-4620/84/0027-0165$03.00

© Wiley Heyden Ltd, 1984

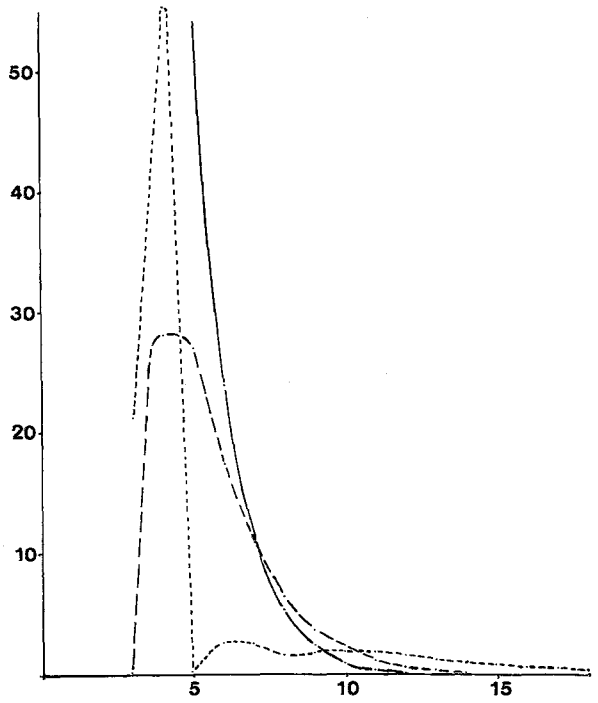


Figure 1. Distribution of degrees of vertices in graphs produced by the three different methods of generation (degree of vertex is plotted in x-direction, percentage occurrence of vertices of that degree in the y-direction).

Welsh and Powell

The method proposed by Welsh and Powell[2] was the first of the algorithms which operate by assigning the least possible colour to vertices in a given order, and was included in Brélaz's comparison. The algorithm assigns a colour c to the first uncoloured vertex of the graph which can be found, and thereafter to each uncoloured vertex which is not adjacent to a vertex already coloured with c. This is repeated with each colour in turn until no uncoloured vertices remain.

Assuming that the vertices of the graph are presented to the program in order of descending degree, the results for the times taken to colour graphs are presented in Fig. 2(a). From this Figure the time taken to colour a graph varies linearly with the number of vertices in the graph with slight variations in slope for the different types of graph. Expressed as a formula the average time taken to colour a graph of n vertices is:

$$t = an$$

where $a = 0.028\ (\pm 0.001)$.

If the vertices of the graph are sorted into descending order by the program, this increases the colouring time considerably. This case is not considered here.

Extended Welsh and Powell

The extended method described by Saaty and Kainen[3] sorts the vertices on the basis of density rather than simply by degree. In this case the time taken to sort and colour a graph is considerably greater than that of the previous method, owing to the density calculations. Colouring times for the different types of graph are illustrated graphically in Fig. 2(b).

Dsatur

The Dsatur method described by Brélaz is based on the notion of the *saturation degree* of a vertex, i.e. the number

Table 1. Average time required to colour graphs using various methods

Colouring algorithm	Number of vertices in graph 20	25	30	35	40	50	75	100	150	200
1. Welsh and Powell						1.4	2.1	2.8	4.2	5.6
2. Extended Welsh and Powell						47.7	105	216	480	987
3. Dsatur						17.5	35.7	59.7	128	220
4. DSI						35.5	66.3	104	205	334
5. Dutton and Brigham	157	322	618	1093	1762	3825				
6. Tree	7	51	173	907						
7. Randall-Brown modified	20	28	40	54	145	311	~800	~10000		
8. Simple reduction						7.8	11.5	15.4	23.4	31.3
9. Improved reduction						14.2	22.1	30.5	49.1	67.3
10. Extended reduction						61	95	153	258	475

Table 2. Average number of colours required to colour graphs using various methods

Colouring algorithm	Number of vertices in graph 20	25	30	35	40	50	75	100	150	200
1. Welsh and Powell						5.02	5.09	5.09	5.19	5.20
2. Extended Welsh and Powell						4.92	4.91	5.06	5.11	5.24
3. Dsatur						4.69	4.74	4.85	4.82	4.93
4. DSI						4.65	4.73	4.77	4.80	4.80
5. Dutton and Brigham	4.01	4.03	4.04	4.05	4.06	4.08				
6. Tree	4.00	4.00	4.00	4.00						
7. Randall-Brown modified	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00		
8. Simple reduction						4.99	5.00	5.00	5.00	5.00
9. Improved reduction						4.34	4.39	4.37	4.47	4.50
10. Extended reduction						4.03	4.00	4.03	4.03	4.05

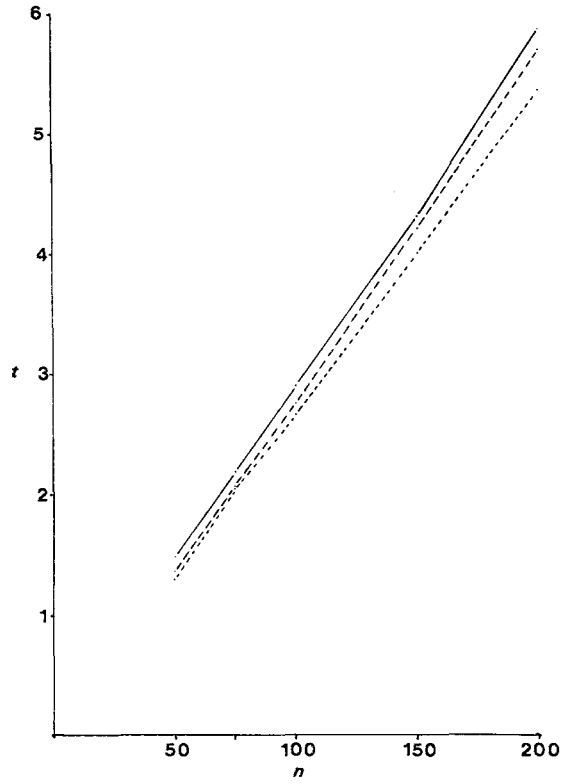


Figure 2(a). Average time taken to colour graph using Welsh and Powell method with pre-sorted data.

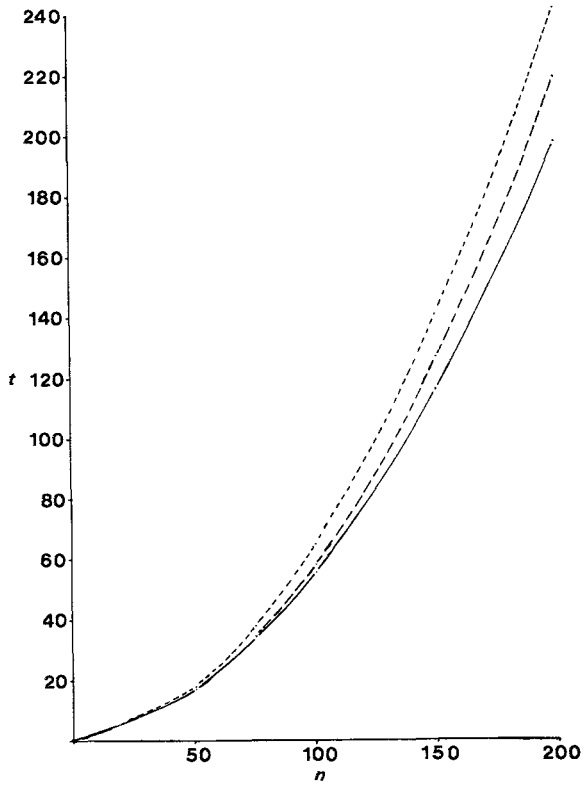


Figure 2(c). Dsatur method.

of different colours to which that vertex is adjacent at any point in the colouring process. The algorithm operates by repeatedly choosing a vertex with maximal saturation degree and assigning it the least possible colour. The colouring times obtained for the three types of graph are shown in Fig. 2(c).

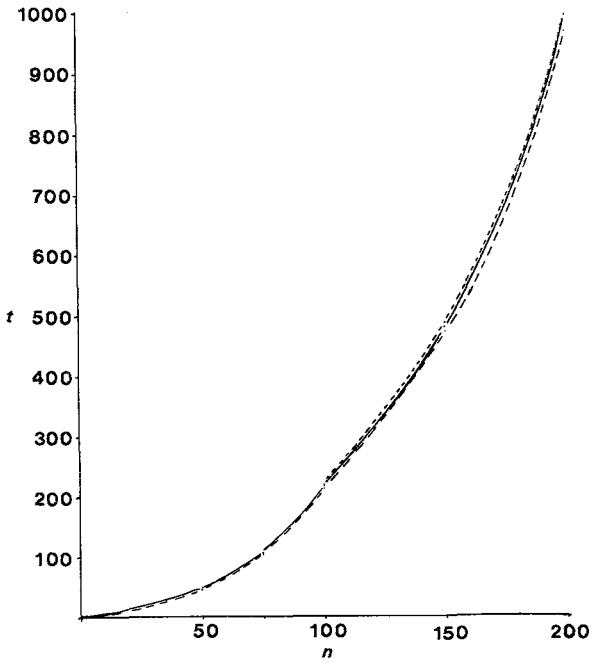


Figure 2(b). Extended Welsh and Powell method.

DSI

The DSI method is an extension of the Dsatur method in which an interchange of colours is attempted whenever it is required to introduce a new colour. The DSI method (Dsatur with Interchange) was originally suggested by Matula[4] and also described by Mitchem[5] and was one of the methods included in Brélaz's study.

Dutton and Brigham's algorithm

This heuristic method is considerably slower than the other heuristic methods described thus far but requires fewer colours. It attempts to merge non-adjacent vertices of a graph which can be coloured with the same colour until all the vertices are neighbours of each other. The graph is then coloured and 'unfolded'. The method is described in detail by Dutton and Brigham.[6] Colouring times for the different types of graph are displayed graphically in Fig. 2(d).

Tree method

The simplest exact method is a tree method which recursively tests partial solutions until an exact solution is found. This is effectively the same as the tree method considered by Brélaz. Colouring times for individual graphs showed a wide variation and results for sets of graphs with more than 35 vertices each were considered to be not very reliable and have not been included in Tables 1 and 2.

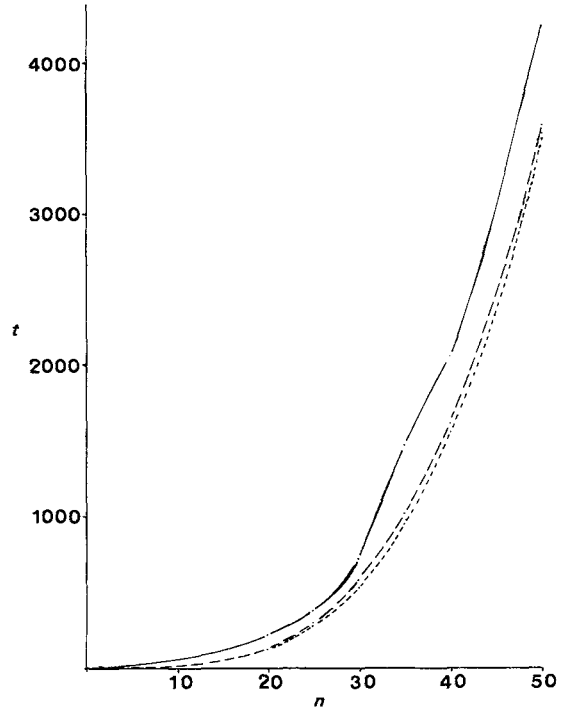


Figure 2(d). Dutton and Brigham algorithm.

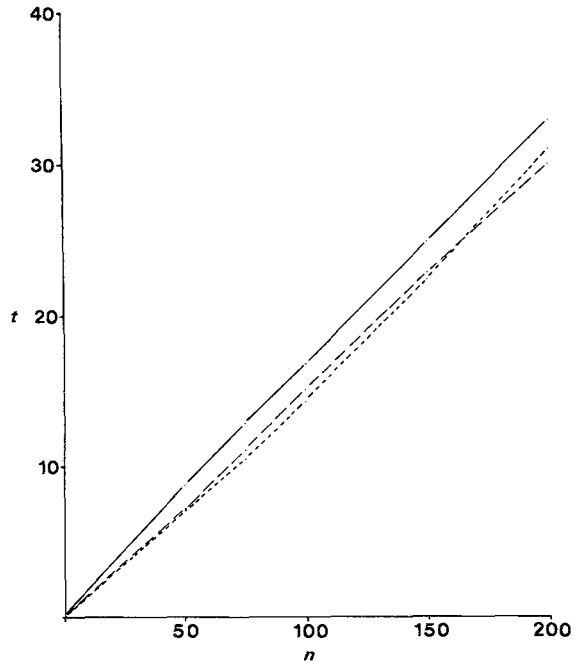


Figure 2(e). Simple reduction method.

Randall-Brown modified algorithm

The second exact method to be considered is a modification (due to Brélaz) of an algorithm suggested by Randall-Brown.[7] The algorithm starts by using the DSI method to produce an initial colouration of the map and then by the use of cliques seeks to reduce the number of colours needed. The complete algorithm is described by Brélaz.

The colouring times obtained for graphs with 75 and 100 vertices are less accurate than other measurements given in Table 1 because of the wide variation in values for individual graphs. Nevertheless the values recorded are useful as an indication of the order of magnitude of the values concerned.

Simple reduction

The reduction methods operate by repeatedly selecting and removing one or more vertices from the graph until it is reduced to a size that can easily be coloured with only four colours. The vertices removed are then reinserted in reverse order and coloured appropriately.

The first of the reduction methods is the simple method described in Ref. 8. This algorithm removes vertices of degree 3, 4 or 5 but treats vertices of degree 4 and 5 very simply, colouring them with any one of up to five colours. Results for colouring times are illustrated in Fig. 2(e). As expected the colouring time varies linearly with the number of vertices n and may be described by the formula

$$t = 0.16n$$

Improved reduction

An improved version of the reduction method is described in Ref. 9. This version colours vertices of degree 3 and 4 using only four colours whereas vertices of degree 5 are treated in the same manner as the simple reduction method.

Colouring times using this method also vary linearly with the number of vertices, although there is a great variation in the constant of proportionality for different types of graphs. This is related to the relative proportion of vertices of degree four encountered in the reduction process.

Extended reduction

The improved reduction method can be extended still further to handle vertices of degree five with fewer colours. Various transformations can be incorporated for reducing groups of vertices of degree five and six so that they can be coloured using only four colours. Four transformations have been included here, viz.

(a) Three vertices of degree five adjacent to two of degree six as shown in Fig. 3(a) can be reduced by removing the five vertices and merging neighbours B and H, and C, E and G.
(b) Three vertices of degree five adjacent to one of degree six as shown in Fig. 3(b) can be reduced by removing the four vertices and merging neighbours C with G and D with F.
(c) Two vertices of degree five surrounded by six vertices of degree six as shown in Fig. 3(c) are reduced by removing all eight vertices and merging neighbours A, C and I, and D, F and H.
(d) A single vertex of degree five surrounded by five vertices of degree six (as illustrated in Fig. 3(d)) are reduced by removing the six vertices and merging neighbours A, C, E, G and I into a single vertex.

The transformations are similar to those described by Ore.[10]

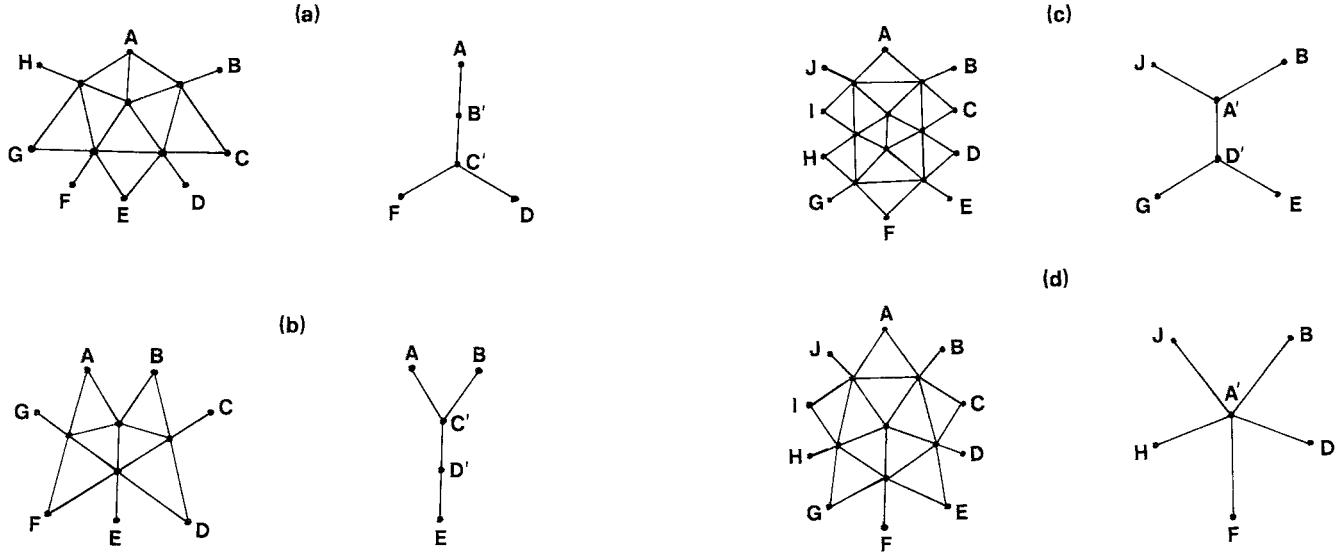


Figure 3. Transformations used in the extended reduction method.

If none of these transformations can be applied then a vertex of degree five is removed as in the previous two reduction methods. On reinstating the original vertices, each of the above configurations can be coloured with four colours, although some recolouring may be required in order to achieve this. When a single vertex of degree five is reinstated, an attempt is made to colour this with one of the four basic graph colours (using recolouring if necessary), and a fifth colour is used only where this cannot be avoided.

CONCLUSION

Table 1 compares the average times required to colour graphs of various sizes using the different methods, whereas Table 2 compares the average number of colours per graph used by the different algorithms. Values have been omitted from the lower ends of these tables (20–40 vertices/graph) where these are of no particular interest. Values omitted from the top ends of the tables (methods 5–7) are due to the fact that average colouring times are very large (in some cases out of range) for graphs with these sizes, and no reasonable estimates could be obtained.

From these two tables it is clear that there is no ideal algorithm. The exact methods will colour a graph with exactly four colours but will require a considerable amount of processor time (for a large graph the amount of time required may well be intolerably large). On the other hand methods which require very little processor time will use more colours.

One way of comparing the various methods which highlights this trade-off is to find the method with the lowest average colouring time for colouring graphs of a given size with an average number of colours which is less than or equal to some desired limit M. The best colouring times (with the number of the corresponding algorithm in parentheses in each case) for various graph sizes and average numbers of colours are given in Tables 3 and 4. Table 3 compares the performance of the first

Table 3. Best colouring times (algorithms) among first seven algorithms considered

Average number of colours	Number of vertices in graph 50	75	100	150	200
4.00	311(7)	~800(7)	~10000(7)		
4.05	311(7)	~800(7)	~10000(7)		
4.1	311(7)	~800(7)	~10000(7)		
4.2	311(7)	~800(7)	~10000(7)		
4.3	311(7)	~800(7)	~10000(7)		
4.4	311(7)	~800(7)	~10000(7)		
4.5	311(7)	~800(7)	~10000(7)		
4.6	311(7)	~800(7)	~10000(7)		
4.7	18(3)	~800(7)	~10000(7)		
4.8	18(3)	36(3)	104(4)	205(4)	334(4)
4.9	18(3)	36(3)	60(3)	128(3)	334(4)
5.0	18(3)	36(3)	60(3)	128(3)	220(3)
5.1	1.4(1)	2.1(1)	2.8(1)	128(3)	220(3)
5.2	1.4(1)	2.1(1)	2.8(1)	4.2(1)	5.6(1)
5.3	1.4(1)	2.1(1)	2.8(1)	4.2(1)	5.6(1)

Table 4. Best colouring times (algorithms) among all algorithms

Average number of colours	Number of vertices in graph 50	75	100	150	200
4.00	311(7)	95(10)			
4.05	61(10)	95(10)	153(10)	258(10)	475(10)
4.1	61(10)	95(10)	153(10)	258(10)	475(10)
4.2	61(10)	95(10)	153(10)	258(10)	475(10)
4.3	61(10)	95(10)	153(10)	258(10)	475(10)
4.4	14(9)	22(9)	31(9)	258(10)	475(10)
4.5	14(9)	22(9)	31(9)	49(9)	67(9)
4.6	14(9)	22(9)	31(9)	49(9)	67(9)
4.7	14(9)	22(9)	31(9)	49(9)	67(9)
4.8	14(9)	22(9)	31(9)	49(9)	67(9)
4.9	14(9)	22(9)	31(9)	49(9)	67(9)
5.0	8(8)	12(8)	15(8)	23(8)	31(8)
5.1	1.4(1)	2.1(1)	2.8(1)	23(8)	31(8)
5.2	1.4(1)	2.1(1)	2.8(1)	4.2(1)	5.6(1)
5.3	1.4(1)	2.1(1)	2.8(1)	4.2(1)	5.6(1)

seven algorithms, Table 4 compares all ten algorithms. Thus, for example, if one were looking for an algorithm from among the first seven which is to be used to colour a graph with 50 vertices using at most 4.8 colours on average (or in fact any limit between 4.7 and 5.0), the best method to use would be method 3 (Dsatur) with an average colouring time of 18 units.

In Table 3 the only estimates for colouring graphs with 75 or 100 vertices with less than 4.8 colours on average are those obtained with method 7 (Randall-Brown modified) which are not very accurate because of wide variations in the colouring times. Nevertheless they do give an order of magnitude for these colouring times. The other two methods requiring fewer colours took considerably longer and it was not possible to obtain reasonable estimates for them. It was not possible to obtain any estimates of average colouring times for graphs with 150 or 200 vertices using methods which would require an average of less than 4.8 colours since colouring times were out of range of the time limit used.

These two tables demonstrate clearly the efficiency of the reduction methods over the other algorithms studied for colouring graphs with between 4 and 5 colours per graph. The fact that both the simple reduction algorithm (8) and the improved reduction algorithm (9) operate linearly with the number of vertices in the graph will make these methods even more attractive when handling larger graphs.

REFERENCES

1. D. Brélaz, New methods to color the vertices of a graph. *Communications of the ACM* **22**, 251–256 (1979).
2. D. J. A. Welsh and M. B. Powell, An upper bound for the chromatic number of a graph and its applications to timetabling problems. *The Computer Journal* **10**, 85–86 (1967).
3. T. L. Saaty and P. C. Kainen, *The Four Color Problem*, McGraw-Hill, New York (1977).
4. D. Matula, G. Marble and J. Isaacson, Graph coloring algorithms, in *Graph Theory and Computing*, pp. 109–122, Academic Press, New York (1972).
5. D. Mitchem, On various algorithms for estimating the chromatic number of a graph. *The Computer Journal* **19**, 182–183 (1976).
6. R. D. Dutton and R. C. Brigham, A new graph colouring algorithm. *The Computer Journal* **24**, 85–86 (1981).
7. J. Randall-Brown, Chromatic scheduling and the chromatic number problems. *Management Science* **19** (4), Part I, 456–463 (1972).
8. M. H. Williams, Improved linear 5-coloring algorithm for planar graphs, Internal Report, Computer Science Department, Heriot-Watt University.
9. M. H. Williams, Linear algorithms for colouring planar graphs, submitted to *Information Processing Letters*.
10. O. Ore, *The Four-color Problem*, pp. 198–209, Academic Press, New York (1967).

Received February 1983

93. A.A.A.FERNANDES, N.W.PATON, M.H.WILLIAMS and A.BOWLES, Approaches to deductive object-oriented databases, to appear in Information Software and Technology (1993).

Approaches to Deductive Object-Oriented Databases

Alvaro A A Fernandes, Norman W Paton, M Howard Williams and Andrew Bowles*

Department of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh EH14 4AS, Scotland

This paper is concerned with the problem of combining deductive and object-oriented features to produce a deductive object-oriented database system which is comparable to those currently available under the relational view of data modelling not only in its functionality but also in the techniques employed in its construction and use. Under this assumption, we highlight the kinds of issues that have to be tackled for a similar research strategy to produce comparable results. We motivate our terms of comparison, characterise three broad approaches to deductive object-oriented databases and introduce the notion of *language convergence* to help in the characterisation of some shortcomings that have been perceived in them. Three proposals that have come to light in the past three years are looked into in some detail, insofar as they exemplify some of the positions in the space of choices we have defined. The main contribution of the paper is towards a characterisation of the language convergence property of deductive database languages which has a key role in addressing critiques of the deductive and object-oriented database research enterprise. We assume a basic familiarity with notions from deductive databases and from object-oriented databases.

Keywords. Deductive databases; object-oriented databases; data modelling; Datalog; Prolog.

1 Introduction

Scope and Goals

Deductive object-oriented databases (DOODs) constitute the most recent research area falling within the intersection of logic and databases. They reflect a growing awareness of researchers to the needs of both database designers and users for a representation tool which is semantically richer than that provided by relational databases (RDBs), as well as to the widely acknowledged benefits of using logic as a tool for the formalisation of both the static and the dynamic aspects of databases. In the past few years the research activity centred on DOODs has grown at an impressive rate — not only has a biannual international conference taking them as their theme already been held twice [22, 36], the number of publications focusing on the more general goal of integrating ideas from logic programming and the object-oriented world is also mounting rapidly. It is clear to us that any attempt at surveying the area in-depth and comprehensively would be a major undertaking, and we wish to stress we have *not* attempted to provide any such survey in the limited space of this paper. The proposals we will come to discuss, and the viewpoint from which we will come to consider them, are subordinated to our main objective in this paper, which we proceed to describe.

The research strategy adopted in the development of deductive relational databases (DRDBs) could be characterised as that of conceiving a RDB as a store of facts within a theorem-proving framework. We assume that this research strategy is equally worth pursuing if the data modelling primitive used to represent facts is the notion of an object instead of that of a relation. One of our main objectives in this paper is to characterise the language convergence property of DRDBs that allowed them to be developed so successfully in such a comparatively short time and to argue that it is also crucial for the success of DOODs.

To meet this goal, we associate with each of the kinds of database under consideration, viz. relational and object-oriented, a language space defined by abstracting away from details in particular

*Now at the Department of Artificial Intelligence, University of Edinburgh

proposals. More specifically, we fix as the unique source of the space, the logic that underlies every language derived from it. As a result, in a given language space, points are languages, while what we call *approaches* are development paths in the same space. We then proceed to consider briefly some proposals on the basis of the approaches they adopt. Note, however, that such language spaces cannot contain, or be provided for, every existing proposal merely as a consequence of their being the product of a particular abstraction. In other words, some proposals would simply not be captured by the abstraction principle we have used.

The language spaces are instrumental in allowing us to characterise the importance of a property of DRDBs which we take to have been crucial to their success. Our claim is that if the research strategy used for DRDBs is to be emulated for DOODs then this property must be clearly acknowledged as a necessary one. Hence, the main contribution of this paper is to characterise this property of deductive database languages obeying the topology of a language space determined by the abstraction principle described above. We call this property *language convergence*. It is defined on nodes in the language space. Naturally, several possible development paths, exist. We claim that some of these paths lead to languages that converge to a lesser degree, or less strictly, than others, thus providing a means for comparing them. We are then able to point out that at least one line of criticism raised against the DOOD enterprise can be addressed by the separate formal definition of a data model taking objects as its modelling primitive. With clearly understood languages for the definition and the manipulation of objects, the degree of convergence which can be achieved is much greater, and so is the ability of the resulting DOOD to cater for crucial object-oriented notions.

It is not to be implied that the proposals we discuss are the only ones worth mentioning or that these, and others we only mention in passing or not at all, would not come to be judged as fully successful under different initial assumptions.

Antecedents, Problems and Perspectives

A *deductive database* allows users to deduce facts concerning data stored in the database using the logic-programming paradigm of computation. DRDBs, i.e. those built on the assumption that the underlying data model is relational [20], are well-understood by now and some implementations are reasonably well-known (for these, see [12], especially chapters 5 and 12). The transition from conventional RDBs to deductive ones was relatively smooth and natural because the relational data model has a simple, independent and well-understood expression in terms of first-order predicate logic (FOPL) [26, 48]. The idea of coupling a RDB to a relation-based logic-programming language (such as Prolog) can be motivated either by the enhanced expressiveness which deduction brings to RDBs, or by the efficiently managed, shared, persistent store which RDBs bring to deductive systems.

The scenario is altogether different if an *object-oriented database* (OODB) is required instead. Wegner [51] defines an *object* to be "a collection of operations that share a state". This definition gives precedence to the behavioural view of objects, viz. from the users' point of view, what matters about objects is the behaviour they display and not their underlying structure. In this paper we assume that OODBs must not undermine these expectations by their understandable concentration on representational issues. In other words, in order to benefit from data independence users have long been advised to shift the responsibility for managing their data requirements from application programs to enterprise-wide databases under some management system. We assume that if those application programs were written in an object-oriented programming language, the move to an object-oriented database should not prevent users from perceiving the objects stored in the OODB primarily by the behaviour they display. Note that this raises important data modelling questions concerning the dynamic aspects of data that in relational data modelling only arise in a comparatively much simpler form.

OODBs are perceived to be most desirable in non-business applications, of which the classical examples are those supporting design activities or graphic data manipulation [7, 24, 56]. The object-oriented view of information systems relies crucially on four features [6, 7]:

1. **object identity** - in OODBs two entities in the real world can be distinguished in spite of having identical values for all their properties, as long as the database came to be aware of them through distinguishable events, referred to as object creation;

2. **classes with encapsulation** - in OODBs a varied and extensible type system is available to organise knowledge about sets of real-world entities, this knowledge is of two kinds:
   (a) *about structure* - defining which properties comprise the state of instances of the class,
   (b) *about behaviour* - defining which operations (or methods, as they are usually called) are granted exclusive rights of altering the state of instances of the class, a notion known as *encapsulation*;
3. **inheritance** - in OODBs classes are organised in a hierarchy so that a subclass inherits the structure and behaviour of each of its superclasses;
4. **late binding** - in OODBs the same method can be applied to instances of different classes, and the interpretation of a method name used in a request can only be carried out when the system knows the class of the object to which the request is addressed, i.e. at run time.

Clearly, there is a conceptual mismatch between logic-programming languages such as Prolog and the notion of *object* we are assuming to be the relevant one. This raises a series of difficult problems that must be tackled for DOODs to be seen as a step forward. Any solution seems to involve compromise involving either or both the database and the deductive engines. In fact, addressing the question of whether either of them can subsume the other and stand alone is a main concern of this paper. Some of the problems in this respect were raised by Ullman in [50]. They can be described as follows:

1. **Identity of entities obtained by different proofs** - a naive (Ullman prefers to say 'serious') approach to object identity could lead to the creation of unnecessary objects. The example given by Ullman is based on the recursive definition of *path* from the *arcs* of a graph:

   ```
   path(X,Y) <- arc(X,Y).
   path(X,Y) <- path(X,Z) & arc(Z,Y).
   ```

   A ground instance of $path(X, Y)$ is taken to mean that there exists a path that starts in node X and ends in node Y in the graph determined by the *arc* relation. For instance $path(a, b)$ asserts that there exists a path from a to b in the relation *arc*. Given two nodes a and b the different paths that may exist between them are treated as a single one in a DRDB, because in a value-based interpretation all facts of the form $path(a, b)$ are identical. In an identity-based interpretation one might (Ullman prefers to say 'must') create for each different proof of the fact that there exists a path from a to b a different entity, i.e. a distinct identity with value $path(a, b)$.

   One flaw in this example is that it depends on an ambiguous use of what the predicate symbol 'path' is used for. As stated in Ullman's argument, the predicate symbol 'path' is assumed, in the DRDB case, to denote (i.e. to be interpreted by) a binary relation with both arguments being points. What we know if the sentence $path(a, b)$ evaluates to 'true' is that a and b are connected. In the DOOD case, however, Ullman assumes that the predicate symbol 'path' must denote a ternary relation, since it is clear that what we now know if the sentence $path(a, b)$ evaluates to 'true' is that a and b are connected by a sequence of intermediate points. This brings out the fact that, in Ullman's argument, *path* is denoting different things.

2. **Dynamic typing in ad-hoc queries** - an ad-hoc query declaratively phrased by a user in a high-level language may have as result entities of unforeseen type. The example given by Ullman is based on joining two relations, R and S with schemes (A, B) and (B, C) respectively, to produce a third relation T with scheme (A, B, C). In a DRDB this could be written as the rule:

   ```
   t(A,B,C) <- r(A,B) & s(B,C).
   ```

   Note that all the operations of relational algebra are well-defined on the new relation T and its elements. In an OODB it is not easy to see how a new type can be left undefined, or be defined on the fly if one takes the latter to mean defining which operations are to be allowed on instances of the newly created type as well.

3. **Optimising declarative queries** - since OODBs support progressively higher levels of abstraction by defining ever more general types accompanied by their admissible operations, it could be argued that to cater for the declarativeness and ad-hoc nature of a join such as $R \bowtie S$, an OODB could have a defined class `relation` on instances of which the expected methods, such as `join`, are admissible and interpreted in the expected way. This would allow one to write expressions such as `R join T`. Ullman's reply is that this does not suffice for optimising ad-hoc queries, insofar as the properties of the method `join` that would allow an optimiser to handle properly the expression `R join T` are not known in any equivalent manner to that in which the properties of the relational algebra operator $\bowtie$ are.

Finally, a point not mentioned by Ullman in [50] is related to **unification** [49], which lies at the heart of the logic-programming approach to computation. In DRDBs, unification is over terms whose functors, i.e. relation names, are free (i.e. uninterpreted) and whose arguments are free 0-ary function symbols, i.e. atomic values of attribute domains, and in this case linear unification algorithms exist. In OODBS, arguments can be set-valued, for instance, and set unification is known to be NP-complete, again barring naive approaches.

These issues must be tackled if the logic-programming and object-oriented paradigms can be combined in a deductive database framework under similar assumptions to those that were applied in the combination of logic-programming and relational modelling. In this paper we argue that the role of a data model in providing the basis for addressing most of the above issues has not been fully recognised. In particular, we argue that by leaving certain data manipulation aspects of the framework partially undefined leads some influential proposals without clear answers to the above issues, and we have in mind, e.g., the definition of the abstract operations supported by objects in general, i.e. seen as a general-purpose modelling primitive

The remainder of the paper is structured as follows. In section 2 a brief description of the strategies that have been used in recent years to design DRDBs is given. We then characterise Datalog as the converged language from Prolog and the relational sublanguages. This allows us to depict the relational language space within which the motivation and constraints influencing the design of Datalog are reflected. We point out that extensions to Datalog exist and that each raises important problems for the theorem-proving aspects of the framework. In section 3 we concentrate on the language convergence problem under DOODs. The lack of clear candidates as the source languages to converge is characterised. We propose that three approaches, among others, suggest themselves more prominently, and indicate how they configure different language spaces. In sections 3.1, 3.2 and 3.3 we look in some detail into proposals that we see as adopting each of these three approaches. We summarise our analysis in section 4 by comparing the approaches and indicating the shape that answers to the perceived problems in combining deductive database with OODBs might take. Finally, in section 5 we draw our conclusions by explaining the importance of an explicit data modelling formalism in addressing Ullman's criticisms.

2 Strategies in DRDBs

Two main strategies have been used to build DRDBs, viz. *coupling* and what we propose to call *language convergence*. Briefly, the coupling strategy assumes that an existing deductive system and an existing database management system (DBMS) will be made to interact by the mediation of two interposed software components, usually called the deductive interface and the database interface. The language-convergence strategy seeks instead to produce a more integrated deductive database system, by redesigning the interaction between the deductive engine and the database engine to the point in which they become indistinguishable for all practical purposes, insofar as they operate on expressions of a single, converged language.

The Coupling Strategy

This strategy consists of using a general, computationally-complete deductive language, usually Prolog, and providing its main-memory resident data store with a connection to a full-fledged DBMS in as transparent a way as possible. The connection can be thought of as comprising an escape valve and a pipeline. Thus, whenever the deductive component is short of facts in the main-memory data

store, the valve is activated and the pipeline pipes in from the mass-storage device a new supply of facts so that deduction can proceed.

The coupling strategy can be further refined by distinguishing between *loose* and *tight* coupling. In a loose-coupling approach, the logic-programming system accesses the database by generating calls via the latter's own query language, which is the normal route for application programs. This is safe (because preserving integrity remains entirely the responsibility of the DBMS) but prone to inefficiency. In a tight-coupling approach, the logic-programming system accesses the database by generating calls directly to the latter's access methods. This is more efficient but less safe (as, in a sense, the DBMS is being bypassed with respect to integrity checking). It should be remarked that others (e.g. [45]) use the above notions differently and call tightly-coupled what we prefer to call converged.

The attractiveness of coupling comes from its generality with respect to both the system that is used to perform deduction and the DBMS that is used to store the data involved. Although most Prolog-coupling systems have used RDBs, arguably most results would still hold for non-relational systems. Note that with coupling, performance hinges on adequate engineering, and the interest of major vendors is evidence for this. Nevertheless, some problems remain which are inherent in the coupling strategy itself. Firstly, the engines cannot be tuned by taking into account each other's characteristics; secondly, there is an overhead involved in managing the interaction between the components of such a multicomponent architecture (the impedance mismatch, for instance, plays a large part in determining this overhead); and, finally, there is a trade-off between safety and efficiency that emerges, respectively, in the loose and tight variants of coupling. It should be stressed that more recent integrated systems based upon Prolog, such as [10], have engineered solutions to overcome many of these problems in a RDB setting.

The Language Convergence Strategy

Language-convergence helps in overcoming the problems inherent in coupling and is attractive also from the point of view that it is less dependent on the actual engineering of a particular implementation. The price is paid in the form of a loss of generality. This comes about precisely because the point of making the languages converge is to produce a finely-tuned formalism in which storage issues are as adequately and transparently treated from the perspectives of both the deductive and the database engines. Intuitively, this process results in a formalism that is a specialisation of, and thus subsumed by, both the original engines.

In other words, attention is given to a triple of languages: the *logic-programming language* used in the deductive system, the *data definition language* (DDL) and the *data manipulation language* (DML) used in the database system. The language convergence strategy consists in fine-tuning these three languages to one another, so that, for instance, the benefit of being able to express in a single declarative language queries, integrity constraints and derived data, is available to users. Consider Datalog, arguably the best-known example of such a converged language.

Datalog as a Converged Language

Datalog [12] is a language for deductive databases which was designed to be converged with respect to extensional RDBs. This was done by constraining the expressiveness of Prolog within the limits imposed by relational-database semantics and by adjusting the computational paradigm of Prolog to that of the relational sublanguages.

It is important to make explicit the assumption that the deductive language and the database languages converge only if, from the point of view of logic, the *terms* of the languages involved (which relate to their model theory) and their respective *evaluation strategies* (which relate to their proof theory, or computation paradigm) become as similar as possible. Note that, from a somewhat different perspective, the failure to meet these two requirements is another way of characterising the *impedance mismatch*, in both its facets, viz. the typing-system mismatch and the computational-paradigm mismatch. The languages will be said to converge if there is a formal account of their respective interpretation taking the same set of entities as their interpretation domain, which is thus *shared* in a formal sense. Note that in ordinary Horn-clause logic programs a clause such as

```
employee(12345,john,15000,sales)
```

is part of the domain of interpretation of programs using a predicate **employee**. The clause is used as a fact to compute the truth value of closed formulae in which that predicate is used. Now, in a RDB, the tuple **(12345,john,15000,sales)** is part of the domain of interpretation of queries using a relation

employee(id,name,salary,dept).

In this sense, one can say that there is a formal account of how logic programs and relational queries are interpreted such that a shared domain of interpretation exists. This observation can be used to inform the design of an integrated language. In the integrated language, correspondences are expected to be so direct that the mediation role played by the interfaces in the coupling strategy is significantly reduced or even ceases to be necessary. This is how efficiency is maximised, the impedance mismatch is minimised and guaranteeing database safety becomes an achievable goal.

Now, if the source languages are Prolog and the relational sublanguages, the convergence that is being sought implies stripping out of Prolog the extralogical features that impair the declarativeness and the safeness of programs, minimising the data-structuring features available, and adjusting the clause-at-a-time computational mode of Prolog to the set-at-a-time mode used with relations. For this reason, Datalog, which is the targeted language for this triple, has *no function symbols*, *no order-sensitivity*, *no extralogical-predicates* (such as the infamous **cut** and updating predicates), and *no unsafe notions* (such as negation in rules and built-in comparison predicates). It is *set-oriented* and (usually) adopts a *breadth-first search strategy*. The language that results is referred to as pure Datalog, to distinguish it from several proposed extensions.

Pure Datalog is a safe, recursive query language for RDBs not only capable of ranging over the usual extensional database (i.e. ground facts about the predicate to which the relation corresponds), but also of defining an intensional database expressed by means of Horn clauses interpreted as rules. As such, it subsumes in an important way the relational sublanguages from which languages such as SQL and QUEL originated.

Of course, the ease with which Prolog and RDBs can be made to converge in a language such as Datalog depends not only on implementing the adaptations mentioned above but, much more crucially, on keeping and, as is in fact the case, even tightening the logical equivalence of concepts, so that the model theory for the language becomes a close account of the admissible instances of the database, and the proof theory for the language becomes a close account of what is involved in answering queries and retrieving data from the database. In the relational case this is possible because every element in the triple ⟨*Prolog, relational DDL, relational algebra*⟩ can be formally defined from a common origin, viz. FOPL.

The language space we have discussed in the preceding paragraphs is depicted in figure 1. Note that two branches emerge from the source in FOPL: a *lower* one which is governed by a theorem-proving approach to computation; and an *upper* one which determines a data model by its sublanguages for the definition and manipulation of the formalism's modelling primitive, and ends up embodied in particular database management systems. The arrows indicate that mappings can be formally given between the languages concerned. In particular, note that in the case of figure 1 such mappings bridge between the two branches in both directions, reflecting the strict convergence achieved by the languages in the space.

Datalog Extensions

The qualifier *pure* is applied to Datalog because that language can be seen as falling into one extreme position in the language-convergence approach, viz. the extreme in which a predefined database component is given the highest priority. But we can conceive of several languages falling in positions between this and the opposite extreme. As we move from pure Datalog towards the opposite extreme, and allow the deductive component also to have a greater influence, we find that a set of extended languages have already been proposed. These give rise to an expressiveness hierarchy, as pure Datalog is progressively enhanced with features such as built-in predicates for equality and comparison, negation, and so on.

These features do introduce problems of their own. Predicates for equality and comparison can lead to unsafe programs which return infinite answers unless safety conditions are placed on clauses

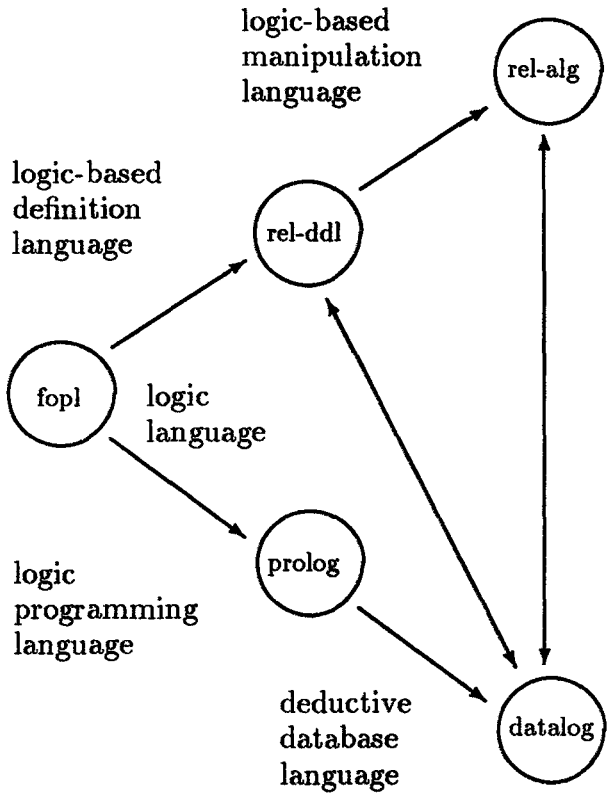


Figure 1: The Relational Language Space

containing that kind of predicates. Negated predicates can also be unsafe. In the case of negation, however, solutions tend to be much more complex (again, see [12] for an extensive treatment of these problems). However, note that these extensions introduce problems to the theorem-proving aspect of a deductive database, rather than to its data modelling aspect. Due to this fact they resurface in every deductive database built under the theory of logic programming, irrespective of whether an OODB is being assumed instead of a RDB. Further, note that the same observation applies to the unification problems introduced by the use of interpreted functors and of complex terms to represent the complex types found in OODBs. Since our main goal in this paper is to consider how greater attention to data modelling issues addresses problems such as the ones raised by Ullman (see section 1), a detailed account of the nature and proposed solutions to those theorem-proving problems, relevant as they are to DOODs, lies beyond the scope of this paper.

In what follows when we use the term 'Datalog' we mean an adequately extended version of pure Datalog, handling arithmetic, comparison predicates, negation and complex terms, and possibly, but not necessarily, declaratively expressed updates. A well-known exemplar of these extended versions of Datalog is $\mathcal{LDL}$, fully described in [45].

3 Approaches to DOODs

There is some agreement on the superiority of the language-convergence strategy for the design of a well-founded and easy-to-use DOOD. The problem lies on which configuration of languages to start from. Our claim in this paper is that the origins of the problem can be largely traced back to the lack of an object-oriented data model (OODM) which can be compared to Codd's relational model (RDM) in terms of the latter's logical foundations, simplicity, and intuitive appeal. The design of such a data model has proved an elusive goal, in the sense that despite some very important attempts (e.g., [9, 40], among others), no formal model of objects as a data modelling primitive has yet been granted widespread acceptance.

One immediate consequence of this is that proposals for DOODs must either specify their departing DDL and DML or leave them to be inferred or, in the extreme case, wholly ignore them. Bearing in mind the consensual qualities of the RDM, it can easily be seen that this whole initial-selection process becomes a much more complicated task in the case of OODBs. In other words, DRDB researchers could take for granted that in the triple ⟨*logic-programming language, DDL, DML*⟩ the second and third elements, far from being variables, were in fact ground to those of the RDM, and only the choice of the logic language needed be an issue at all. Having a predicate-based, hence relational-based, language such as Prolog at hand, even this last choice point was much constrained. DOOD researchers, however, must specify all three elements insofar as Prolog is no longer such an obvious choice and database sublanguages for OODBs are barely formalised in most cases. Roughly, the only element in the language space on which there is agreement is in taking some first-order logic as the source of the space, though this does not necessarily mean that this logic is to be based on predicates as they are classically construed.

For the purposes of the remaining exposition, we concentrate on three possible approaches that have been used to define a converged language for DOODs. These will be referred to as: the *OO-Datalog* approach, the *OO-Prolog* approach and the *OO-logic* approach. We stress once more that our choice of proposals to exemplify each of the approaches was guided only by their adequacy in illustrating our argument. Under this assumption, citing many other comparable proposals would just have been redundant.

3.1 The OO-Datalog Approach

We apply this label to proposals that depart from Datalog, i.e. which assume that there is no significant conceptual conflict which could prevent a further-extended Datalog from hosting an OODM. We will take as our example a family of three languages: COL (Complex Object Language) [2], IQL (Identity Query Language) [4], and the nameless language recently proposed in [1], which, for convenience, we shall call **KL** (**K**yoto **L**anguage, for the place where the language was disclosed). Together they subsume Datalog by extending it with certain object-oriented features and updates.

Main Features

This family of languages can be characterised by the adoption of Datalog as a platform from which a deductive object-oriented database language can be derived in a stepwise manner. The three languages mentioned above are steps that progressively try to incorporate more notions from the object-oriented paradigm, without compromising the goal of having the relational languages as particularisations of the extended one.

This evolution can be briefly summarised as follows. COL was proposed as an extension of (pure) Datalog with complex typed-terms. It also had other design objectives which are of less interest in our present context. COL uses a fixpoint semantics. It is, however, a value-based language and does not cater for the essential object-oriented notion of identity. IQL extended COL with object identity, thus providing the essential element for the modelling of inheritance and for a richer type system. IQL supports classes as sets of objects identifiers, as well as relations as sets of values. Object identity also plays a part in improving the declarative handling of updates and set terms. IQL was seen as the operational part of a model whose structural aspects were object-oriented. Note, however, that what is taken as the operational part of a model in this context is not nearly as complex as suggested by Wegner's definition (see above, section 1). Nevertheless, IQL represented an important step ahead in the direction of a robust and well-founded DOOD.

KL, the most recent member of the family takes previous achievements further by providing the language with methods, thus bringing the resulting language closer to the object-oriented paradigm. Recalling the four crucial features of OODBs mentioned in section 1, viz. object identity, classes with encapsulation, inheritance and late binding, KL is lacking only in fully catering for some aspects of object-oriented behaviour modelling which play a part in allowing users to view object-orientation as being tantamount to interaction by means of message-exchanging [51]. In particular, although KL has methods and method inheritance, the interplay of encapsulation and late binding is not addressed. The presentation that follows concentrates on the language we are calling KL [1], with reference being made to the other two languages [2, 4] where necessary for clarity of exposition.

Definitional Aspects

The notions of object-identity, classes and types are basic, with classes being names for sets of object-identifiers whose structure is defined by some type. Two constructors are used, for sets and for tuples. The distinction between schema- and instance-level concepts is retained. This is one important property of this group of languages which distinguishes them from the Stony Brook family which we will come to consider in section 3.3.

A schema is declared using a type language. Examples of statements are:

```
class person:
      equal student or employee
      and   [name:string];

class student:
      isa   person;

class employee:
      isa   person
      and   [supervisor:senioremployee];

class senioremployee:
      isa   employee;

class dept: [dname:string];
...
```

From statements such as these it is possible to catch a glimpse of the definitional expressiveness of the language. The keyword **equal** is used to define a class as a *generalisation* (a person is either a student or an employee, i.e. no objects are declared to be persons: the extension of the class **person** is derived by taking the union of the extensions of its subclasses **student** and **employee**). The keyword **isa** is used to define a class as a *specialisation* (a senior employee is an employee, i.e. some objects are declared to be employees, but the extension of class **employee** also contains those declared as senior employees). Expressions enclosed in square brackets define tuple-types, and colons associate attribute names to types. Structural inheritance proceeds from such declarations.

One aspect that is not evident from the above is that the notion of *relation*, a set of values (as opposed to object-identifiers), all of the same type, is also supported. This indicates the concern with keeping two versions of the model, viz. the main identity-based version, and a value-based one, which motivates several OO-Datalog proposals, and this family in particular.

Manipulative Aspects

Methods are modelled as functions obeying typing constraints, and in the spirit of COL can either be stored extensionally or derived from rules. Since the COL proposal, stress has been placed on the utility of the data-function notion in terms of allowing a switch between a relational and a functional view of the extensional and intensional database. In a predicate-based language, data functions can be characterised as relations constrained by a functional requirement, which the system is responsible for enforcing. Thus, just as atomic formulae can have their extension asserted as relations in the database, so do functional terms. It is claimed that this facility clears the way for a cleaner integration of functional and relational approaches to database systems [3].

Methods can be declared as follows:

```
method graduationdate:
       person -> <[university:string,
                   date:int]>;
```

This statement declares that **graduationdate** is a function from persons to lists of tuples having as first element a university of type **string** and as second element a date of type **int**. Methods can map into single- or set-values, the latter being taken to include lists as well.

Methods are defined on classes and inherited as expected. Overloading and overriding are possible by associating methods with more than one typing constraint in the method declaration and by propagating this under inheritance. However, it is not clear from [1] how the functionality associated with late binding is to be provided.

The language handles updates by extensions that are briefly described below. Beyond the notion of a program as a set of rules with a fixpoint semantics, which is said to constitute a *local level*, the language is further extended to what is called a *top level* by the use of explicit control on which rules are active and which are not, and then to an *external level* which provides extensibility by embedding it into a host language. Features that result from this extended language include:

Views and Updates - contrary to the norm in deductive databases, not only views (i.e. intensional predicates) but also database elements (i.e. extensional predicates) can appear in the heads of rules. The way this is handled is by explicitly declaring relation- and method-declarations in the schema to be views or not. This controls the interpretation of heads of rules, i.e. if a rule head is not a view, whenever its body is proven true for some instantiation this is interpreted as a request for inserting a new fact into the extensional database. Deletion is signaled by a negative literal in the head. Both facts and rules can be inserted and deleted.

Constructs for Explicit Control - programs can be structured with the aid of constructs such as **begin-end** blocks, **if** rules (execute once) and **while** rules (execute iteratively). The last two will help to define the notions of triggers and integrity constraints. Finally, procedures using local temporary relations and methods can be defined using a **let ... be** construct.

Persistence - schema elements must be declared to be **permanent**, as opposed to **temporary**, in order for them not to be discarded at the end of the session.

External Methods - methods can be written in host languages and referred to in rules.

The language also provides for the creation of object-identifiers (through variables that only appear in the head of a rule) and for their inclusion in a class. In particular, a new object need not be created if an existing one fulfills the specifications. As object-identifiers can be created without an initially assigned state, the language provides for the modification of the state of objects, as well as the modification of the value of methods (recall that methods can be extensionally stored as relations under a functional constraint). Finally, it is also possible to modify an object-identifier assignment, and thus remove an object-identifier from a class or transfer it from one class to another.

Deductive Aspects

The rule-based language uses constants, tuples, lists and sets as terms. As declared in the schema, class and relation names are terms, as are methods with properly typed arguments. Built-in operators provide for projection (denoting a field in a tuple), dereferencing (denoting the value of an object-identifier) and the classical *cons* operation on lists.

Literals are expressions built from properly typed terms using equality and membership predicates. Note that a predicate such as `worksfor(X,Y)` has the alternative notation `worksfor` $\ni$ `[X,Y]`, where the square brackets indicate a tuple term.

The database is thus a set of facts (i.e. ground literals) about class members, object values, relation tuples, and methods. Examples of these are, respectively:

```
employee(#1)

*(#1) = [john,phil]

worksfor(#1,#7)

graduationdate(#1) = <[yale,1965],[mit,1968]>
```

In these examples, object-identifiers have been prefixed by #, although this is not a requirement of the language. The dereferencing operator is *. List terms are enclosed in angle brackets, tuples in square brackets, sets in curly brackets.

Rules have a positive literal as head and a list of positive and negative literals in the body. As derived facts are obtained from rule heads the particular form these take is determined by the kinds of terms that are stored in the database (see examples above). Since all terms are typed, the logic is technically many-sorted.

The membership predicate is used in derived methods to express facts concerning set-valued objects, while the equality predicate achieves the same purpose for single-valued ones. Thus suppose that a method is declared as:

```
method friends: student -> {person};
```

then the rule `friends(X) ∋ Y <- teaches(Y,X)` allows the derivation of a fact about a set-valued object, viz. the set of all people Y who are friends of X because Y teaches X, i.e. because `teaches ∋ [Y,X]` holds. Correspondingly, suppose a method is declared as:

```
method managedby: employee -> string
```

then `managedby(X) = *(X.supervisor) <-` is an (unconditional) rule allowing the derivation of a fact about a single-valued object, viz. the name of the senior employee of an employee X. Note that if o is an object-identifier, $o.A$ uses the projection operator . to denote the attribute A of o.

The semantics for the language uses stratification to handle negation, the existence of set-valued terms, and also the non-monotonicity that results from the overriding of methods. Fixpoint computation on the resulting strata is used to compute the facts that can be derived from the program.

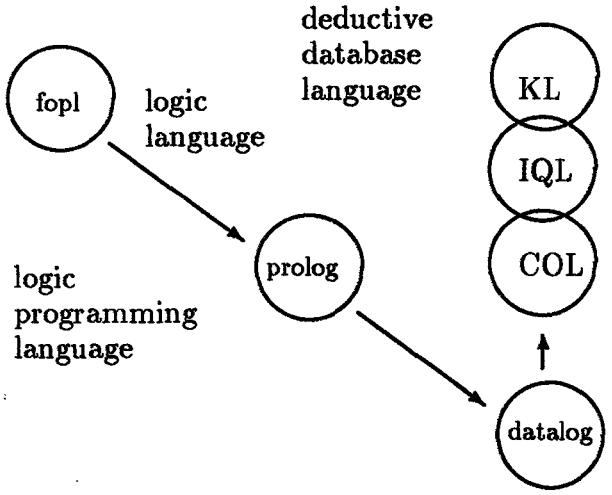


Figure 2: The COL/IQL/KL Language Space

Summary

The language space determined by the COL/IQL/KL family of languages we have discussed is depicted in figure 2. Contrast this with figure 1. Here there is no definition of a data model, as spelled out in its sublanguages, that is separate from the deductive database language. We acknowledge this fact by arguing that KL is not strictly converged, in the sense of being related to preexisting object-oriented database languages that are independently defined and independently used. This analysis helps in understanding that the underlying data model which Datalog inherits from the RDM via FOPL is deeemed to be extensible to an object-oriented one by features progressively added to Datalog in COL, IQL and KL. KL was designed to be a stand-alone language, with no provision for simple mappings to any existing OODB management system (OODBMS). Hence, it assumes the burden of catering for all perceived improvements that users expect from adopting the object-oriented paradigm. As an aside, note that a fully-fledged logic-programming language such as Prolog is dispensable, though one could argue that KL is the equivalent to Prolog in this scenario, in its concern with practicality.

3.2 OO-Prolog

We apply this label to proposals that depart from (full) Prolog, again under the assumption that no significant conceptual conflict arises from this. The OO-Prolog approach is represented by [25, 43, 46], and we will take [46] as our example.

Much work [21, 44, 54, 55] has also been done on extending Prolog with object-oriented notions. The motivations, strategies, techniques and goals are defined more from a programming-language point of view than from a database-language one. These proposals have in common their assumption that the syntactic machinery of Prolog suffices to express several object-oriented notions. They are of less interest in the context of this paper since their concern with data modelling issues is subsidiary. Because of this, important problems from a database perspective tend to be either disregarded as side issues or treated in a simplistic way.

Proposals here, from both the programming- and the database-language point of view, tend to be harder to classify on the basis of the language-convergence notion. In this approach it is often the case that the reliance on one of the elements of the language-triple at the expense of the others results in a heavily imbalanced language.

Main Features

Throughout the 1980s, in parallel with the largely theoretical research into DRDBs summarised in section 2, there was a separate implementation-oriented activity which looked to develop database systems based upon Prolog. This branch of research gave rise to a number of implementations of coupled or more fully integrated Prolog/Database systems [28]. Such systems could subsequently be used to implement database systems in which Prolog was used to implement such components as the data model, alternative query languages and optimisers [29, 30, 52]. At the same time, researchers from outwith the database community were proposing various object-oriented programming systems implemented in Prolog [25, 43, 54], in which the flat Prolog clause base was structured using object-oriented mechanisms, while Prolog was retained as a manipulation language. It is from this broad, experimental environment that OO-Prolog systems and OODBs implemented in Prolog have emerged.

For OO-Prolog systems, the triple ⟨*logic-programming language, DDL, DML*⟩ can be considered to be partially instantiated as ⟨*Prolog, x, Extended-Prolog*⟩. While the DML is ostensibly Prolog, there are usually a number of additional operators for performing such tasks as sending messages and updating structures. The data model varies considerably from proposal to proposal, with concepts such as encapsulation, classification and identity supported in a variety of different ways in different systems. The uninstantiated DDL in the triple above is a sign of this variety.

Definitional Aspects

This section uses the syntax of ADAM [46] to illustrate some typical data modelling features. The following definitions specify the schema of a simple parts database:

```
:- new([course,[
        slot(slot_tuple(code,global,single,
                        total,string)),
        slot(slot_tuple(level,global,single,
                        optional,integer))
    ]]) => class.

:- new([person,[
        slot(slot_tuple(cname,global,single,
                        total,string)),
slot(slot_tuple(sname,global,single,
      total,string))
    ]]) => class.

:- new([employee,[
```

```
        is_a([person]),
        slot(slot_tuple(phone,global,single,
                        optional,integer)),
        slot(slot_tuple(teaches,global,set,
                        optional,course))
    ]]) => class.

:- new([student,[
        is_a([person]),
        slot(slot_tuple(year,global,single,
                        total,integer)),
        slot(slot_tuple(takes,global,set,
                        optional,course))
    ]]) => class.
```

The message-sending operator => sends the message specified by the first operand to the object which is the second operand. Each slot in ADAM is described by a number of facets, which specify its name, visibility, cardinality, status and type [46]. The statements for class creation given above are message-sends, as in ADAM all classes are themselves objects [47]. Instances of the new classes are in turn created by sending messages to the newly created classes:

```
:- new([SiD,[
        cname(['Fred']),
        sname(['Flintstone'])
   ]]) => student.
```

In this example, the Prolog variable SiD is unified with the system-generated object-identifier of the new *primitive_part*. Features of this specific model include multiple inheritance, encapsulation and run-time type checking.

Manipulative Aspects

Behaviour can be associated with classes by defining methods which are attached to a class either when it is created using *new*, or by the method *put_slot*. The following program attaches the method *num_courses* to the class *student*, which can be used to find the number of courses taken by a *student*.

```
:- put_method([(num_courses(global,single,
                            [],integer,[Num]) :-
        findall(C,
            get_takes(C) => self,
            CList),
        length(CList,Num)
    )]) => student.
```

Queries can subsequently be written which retrieve information from the database. For example, the following program writes out the *cname* and number of sub-parts of all *students* in the system. The method *get* retrieves the instances of a class, while the method *get_cname* retrieves the value from the *cname* slot of the object to which it is sent:

```
:- ( get(S) => student,
     get_cname(Cname) => S,
     num_courses(Num) => S,
     write('Name,' has ',Num,' courses'),nl,
     fail
   ; true
   ).
```

Deductive Aspects

The examples given above are of a specific OO-Prolog system, and the variety of such systems in the literature [25, 43, 44, 54] makes it difficult to extrapolate from an example system towards general conclusions for the family. We note, however, that the code in the examples above leans very heavily upon a procedural rather than a declarative interpretation of Prolog, in which objects have an encapsulated state which changes over time as the result of message-sends. The Prolog resolution theorem prover is used to evaluate both top-level programs and methods, where messages are sent either to retrieve information from the object-oriented store, to deduce properties defined using methods, or to change the state of the database.

In ADAM the distinction between classes and instances is preserved by forcing every object to conform to the structure defined in the class of which it is an instance. In certain other OO-Prolog systems [44, 54], this distinction is removed, and all new objects are *is-a* linked to prototypes whose properties they inherit.

A further characteristic of this family of systems is that an individual OO-Prolog is defined as a program (written in Prolog) which has some specific functionality. Thus, for example, support for inheritance is not specified within the fixpoint semantics of a language (as in [1]), or as part of the underlying logic (as in [33]), but rather by using an algorithmic approach. Further, issues such as safety are rarely mentioned by the proponents of OO-Prolog systems, and such systems inherit their lack of safety from the language upon which they are built. Indeed, the notion of objects which have a modifiable state generally means that many of the extra-logical facilities of Prolog are used extensively within the implementation of OO-Prolog systems.

Finally, the distinction made within Datalog between an intensional database (i.e. a set of clauses in the logic language) and an extensional database (i.e. a set of positive ground unit clauses) has no place in this approach. Deduction becomes a less precise notion.

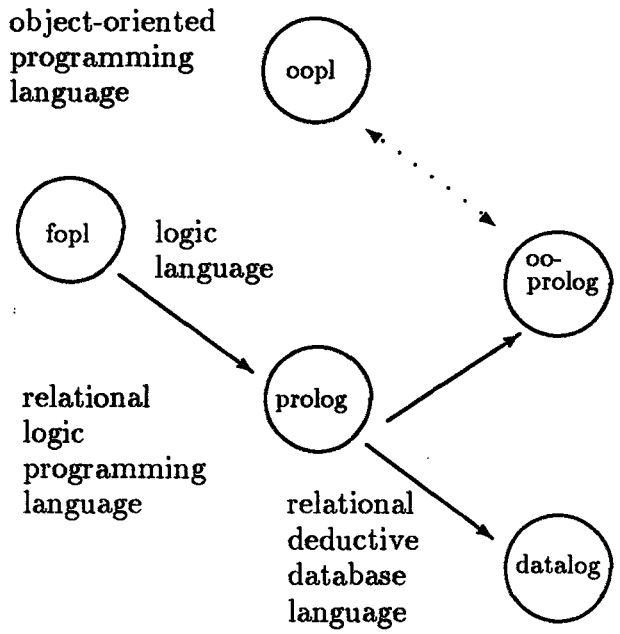


Figure 3: The OO-Prolog Language Space

Summary

The language space determined by proposals we label as OO-Prolog is much harder to depict. In figure 3 we offer no more than a general picture. As is the case of the COL/IQL/KL family, here also we find no separate definition of an OODM. However, while KL could be said to be converged, albeit in the less strict sense we have indicated when discussing the language space in figure 2, languages resulting from the OO-Prolog approach tend to be fully-fledged logic-programming languages. In terms of language convergence, it can be argued that OO-Prolog systems are a short-cut which

by-pass the language-convergence issue by using extra-logical features to introduce identity into a language which is essentially value-based. As such, the resulting systems retain both the virtues and vices of full Prolog. In fact, an OO-Prolog language is best seen, as is Prolog, as a programming language for applications which include, but are not limited to, those requiring a database, and in the case in hand, an object-oriented one.

In this sense of being unconstrainedly expressive, OO-Prolog proposals tend to be equivalent to object-oriented programming languages (OOPL) in general, hence it is easier for them to cater for the behavioural view of objects that we are assuming to be desirable for users. However, OO-Prolog languages also tend to be stand-alone, just as KL was seen to be. This is to say that OO-Prolog languages cannot in general be formally mapped into other languages, e.g. the OOPL of some existing OODBMS. (In figure 3 we indicate by dotted arrows the nonexistence of such mappings.) The difference to the OO-Datalog approach resides in the fact that while languages belonging to the latter are harnessed by logical strictness, OO-Prolog languages in general are not.

3.3 OO-Logic

We apply this label to proposals whose departure points are logics that alter in nontrivial ways the FOPL foundations which underlie Prolog and Datalog. This implies a degree of doubt as to whether these languages can serve as a platform for building DOODs. The rationale behind this approach is the belief that one must redesign the deductive language from a more fundamental point. We will take our examples from the a family of three languages developed at Stony Brook whose common origin is the now superseded Maier's O-logic [42]. As Maier himself acknowledged the existence of problems in his original proposal, this immediately prompted other researchers to improve the original O-logic. Thus, C-logic [19] is an improvement of the original O-logic. The proposal in [34] corrects and extends Maier's original proposal. This revised version of O-logic is such that C-logic is a proper subset of it. Finally, F-logic [32, 33] extends the revised version of O-logic.

Main Features

F-logic, the most recent member of this family of languages, is the outcome of a rethink on the kind of logic language one needs to deal with object-oriented features. One crucial question here is: will the language only refer to individuals or will it allow reference to be made to complex structures in which individuals can be seen to participate in the real world (e.g. sets)? Relational systems, for instance, choose the first alternative; OODBs choose the second. This means that OODBs are placed beyond the realm of classical first-order languages. This is relevant because a well-known result from mathematical logic states that it is not possible to provide a full-fledged higher-order logic system with a complete axiomatisation. To tackle the problems that result from that, the Stony Brook approach ultimately led to F-logic, a language whose expressions are not syntactically first-order but which admits an interpretation in first-order semantics. In other words, the model theory with which F-logic is equipped is such that all references are ultimately to individuals, rather than to structures they are perceived to be composing. Beeri [9] has argued that, while technically speaking a restriction on the expressiveness of the language, the technique used for this mixture of a higher-order syntax and a first-order semantics is probably adequate in the context of databases.

As we have discussed in section 2, Prolog and Datalog operate under assumptions about the static and dynamic aspects of the store of facts which the RDM amply realised. Under the Stony Brook view of the problem, switching from the RDM to an OODM means that new assumptions come into play, and that to cater for them as transparently one must have at hand at least a second-order syntax. In other words, an OODM brings with it a nontrivial reformulation of the static and dynamic aspects characterising the store of facts on top of which the deductive database language is now to operate. Thus, the latter must emerge from a more foundational look into the underlying logic. A problem that stands in the way of such strategy is the fact that no proposed OODM has a clear formulation in logic, hence there is no clear picture of what shape such a reformulation of the store of facts will end up having from a logical point of view.

Some of the technical novelties incorporated in the C-/O-/F-logic family are discussed in [17], in the more general context of logic programming. In what follows we will concentrate on F-logic [32, 33], as it is by far the most expressive member of this class of logics. The version published in [32] will

be the one used here as it is the most easily accessible. The revised version [33] contains changes which, though significant in several aspects, do not impact the discussion in this paper.

F-logic abolishes any significant distinction between schema and instance, or, to put it in other terms, between names and their denotations. By imposing a lattice structure on the universe of ground terms, this reification of intensional objects such as classes is used to derive a model-theoretic account of inheritance and a treatment of locally inconsistent information.

Another contribution of the Stony Brook approach is a better account of how new object identities are to be created as a result of rule evaluation. Skolemisation provides an existentially quantified interpretation to variables appearing in the head of a rule but not in its body. Then, when such a rule succeeds, this can be taken to be asserting the existence of an object whose identity is the denotation of the Skolem function. To illustrate this, consider the addition of a third argument to the `path` predicate used in section 1:

```
path(X,Y,i) <- arc(X,Y).
path(X,Y,o(I,Z)) <- path(X,Z,I) & arc(Z,Y).
```

Then, the third argument acts as a trace of the derivation of a `path(X,Y)` fact. As an example, suppose there are two paths from node `a` to node `c`, where the first is given by the tuple of nodes `<a,d>` and the second by `<a,b,c,d>`. Then the identity of the first path would be `i`, that of the second would be `o(o(i,b),c)`, where the second arguments at each level of nesting read from the deepest to the shallowest gives us the intermediate nodes in the path. Alongside with the proviso used in KL (see section 3.1) for creating new objects only if no existing object fulfills the specification, this Skolemisation technique for creating new object identities provides an appealing account for the problem of object creation in theorem-proving frameworks such as DOODs.

Definitional Aspects

The fundamental notion in F-logic is that of *id-term*. Id-terms denote individual objects in a classical way, thus a (complex) F-term can consist of attributes defined on sets of id-terms (a second-order syntactic construct) without jeopardising the first-orderness of the semantics. The notion of id-term can be seen as the F-logic account of object identity.

F-logic uses a domain that is organised in a lattice according to IS-A relationships. Classes (or more precisely, class-identifiers) and instances are equally placed as members of this domain. "This feature", it is claimed in [32], "is largely responsible for the fact that inheritance is naturally built into the semantics of F-logic, contrasting it to algorithmically defined inheritance in other approaches."

F-logic takes the general approach of giving full object status to all elements in the language. This leads to an unconstrained syntax in which symbols can appear in practically every syntactic role, and this, in turn, leads to variables being able to range over almost every conceptual notion.

Labels (the F-logic word for attribute-identifiers) are objects in their own right and, therefore, are typed as any other object. They can either be single-valued or set-valued. Complex terms can be built with indefinite depth of nesting.

A (complex) F-term can be exemplified by:

```
student:mary
  [name    -> 'Mary',
   age     -> 30,
   friends -> {bob,sally},
   works   -> dept:cs[dept-name -> 'CS']
  ]
```

where `student` is an id-term standing for a class-object, `mary` is an id-term standing for the object being defined, `name` and `age` are id-terms standing for single-valued labels, `friends` is an id-term standing for a set-valued label, `works` is an id-term standing for a relationship which the object being defined maintains with the object denoted by the F-term `dept:cs[dept-name -> 'CS']`. Note that `'Mary'`, `30`, `bob`, `sally`, and `'CS'` are also id-terms standing for objects in the domain.

The definitional power of F-logic is considerable, insofar as the model does not impose any important restrictions on what may be construed as an object. Work under this more foundational approach may provide a new basis for further developments because of its conceptual minimality, a property which proved to be a success factor for the RDM.

Manipulative Aspects

Inheritance in F-logic is built into the semantics of the language, in the sense that there exists a theorem stating that if a database satisfies a formula concerning properties of a given object then that formula is also satisfied for objects that, in the domain of interpretation, are hierarchically above (i.e. less general than, or more informative than) the former according to the lattice structure of the domain. This means that property inheritance is expressed directly in the structure used to assign meaning to formulae.

A powerful consequence of giving full object status to things like class-identifiers and attribute-identifiers is that objects that are traditionally considered to be schema-level elements, also exist at the data-level. They can be given attributes and be queried like any other objects. It is possible to construe a notion of the schema being dispersed in the statements of the language. In other words, interpreted in a database context, the statements of F-logic do not distinguish syntactically or semantically between schema-level and data-level information. It is up to users to do so when they wish. However, since updating the domain of interpretation can make the logic undecidable, a restriction is imposed whereby schema browsing is supported though schema evolution is not.

The notion of method is catered for in F-logic by taking advantage of variables ranging over id-terms. Consider an example from [32], where *graduation_date* is a method (i.e. a function from universities to years) defined by the following rule:

```
person:X[graduation_date(U) -> Y]  <=
  univ:U[alumni  ->
        {alumni_record:G[student -> person:X,
                         date -> year:Y]}]
```

This can be used to answer queries such as:

```
X[graduation_date(U) -> 1987]?
```

```
john[graduation_date(U) -> Y]?
```

The first query returns all graduates of all universities who have 1987 as a graduation date. The second query returns the universities and corresponding graduation dates for the individual `john`.

Note that this is possible because labels (a single-valued one in this case) are id-terms, and these do not have to be ground. Therefore, it is perfectly possible to construe a notion of methods as parameterised labels, or conversely, of attributes as parameterless methods. In [32, 33], a brief exemplification is given of how methods can be inherited and how overriding can be accounted for.

Deductive Aspects

One point worth noting is that predicate symbols are made available in F-logic only for the purposes of catering for a conventional-style syntax. However, predicate terms are notational variants of F-terms defined on class-identifiers, and these play the same role that predicate symbols play in a standard logic-language. Thus, for each predicate there is a corresponding F-term, obtained by mapping the predicate symbol into a class-identifier, supplying an id-term that is a function of both the class and the arguments, and devising labels for each argument. Using this construction procedure in reverse shows how F-logic can be reduced to a value-based framework. Notice that this is an example of the overall goals of the OO-logic approach, viz. a reworking of foundational aspects in such a way that a logic language can be blended with a language of objects in a more natural way, i.e. one based on a common core of conceptually equivalent notions.

In [33] a sound and complete proof procedure based on resolution is defined for the monotonic part of F-logic, extending the results published for the revised O-logic [34]. With respect to that proof

procedure two remarks are important. First, the problem of set unification still seems to require a solution like the one adopted in $\mathcal{LDL}$ [45], viz. a rule containing a set term can be compiled into a logic-programming procedure in which the set term is transformed into a first-order term while the idempotence and commutativity properties of sets is captured by the structure of the rules in the procedure. Second, in classical logic-programming the control problem is greatly simplified by the fact that a single inference-rule, i.e. *resolution*, suffices for any proof (though it is optionally complemented by *factorisation* so as to reduce the number of disjuncts in a clause). By contrast, F-logic needs several additional rules. This has the effect of expanding the search space as there is now a choice point as to which rule the proof procedure should apply at any given stage. The claim is made that the consequences in terms of complexity are acceptable but the evidence presented for this does not dispense with the need for further investigation.

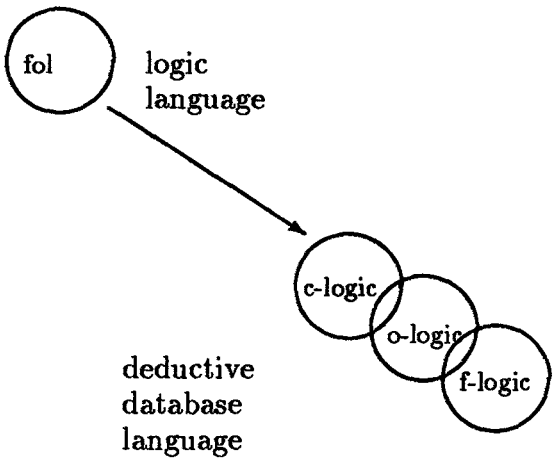


Figure 4: The C-/O-/F-logic Language Space

Summary

The language space determined by the C-/O-/F-logic family is depicted in figure 4. As one would have expected from a more foundational approach, this is a sparse space in comparison with those of the other approaches. More specifically, in comparison with the COL/IQL/KL language space in figure 2, it pushes back, so to speak, the point where the deductive language begins to be derived. Correspondingly, the C-/O-/F-logic family does not go as far in accounting for practical features of the resulting language, such as the updates and control structures KL provides.

The C-/O-/F-logic language space is, however, no different from the previous ones in not comprising the separate definition of an OODM. Again it is not possible to say unrestrictedly that the resulting deductive database language is converged. Hence, F-logic again assumes the burden of catering for all perceived improvements that users expect from adopting the object-oriented paradigm. It does go far in that: for instance, its account of overloading includes the case of methods defined in classes not related by inheritance, it provides schema browsing facilities, it manages to accommodate inheritance into the notion of semantic consequence. However, it is still not capable of modelling behaviour in the same abstracted way that users are accustomed to in object-oriented systems.

4 A General Summary

In this section we offer a brief evaluation of the proposals we have described. This is done in two stages: in the first we accept their own assumptions as reflected in the language spaces they generate; in the second we confront them with Ullman's criticisms [50] to argue that under his assumptions their responses may not be adequate. Finally, we indicate how adequate responses are possible.

We wish to remark that there is a wealth of research activity (e.g. [11, 41], among others) whose fruits would fit under one of the labels we used to discuss certain approaches to DOODs. The fact that we have not taken them as examples should not be taken as an implicit judgment of comparative

merit. Other proposals are more difficult to classify within a general framework of DOODs. Among these, proposals exist to integrate sets in logic programming [23, 37, 38, 39], to handle non-first-normal-form relations in a deductive language [13], and to extend (pure) Datalog to handle nested relations [14, 15]. Others are, strictly speaking, programming languages: the SQL extension towards deduction and abstract data types (ADTs) proposed in [27] and the object-oriented programming language incorporating rules proposed in [53].

The COL/IQL/KL Approach

For these languages the main issue is to what extent object-oriented notions can be accounted for in terms of an underlying predicate-based view of the deductive language, while providing users with a message-passing paradigm of computation. It could be argued that an underlying first-order view is inescapable, since first-order logic is the only suitable formalism to take as a departure point. However, this may not preclude a more expressive syntax being devised which would cater for an easier-to-use language with no loss in well-foundedness.

As an example, note that the effect of the distinction between schema- and instance-level concepts is that proposals under this framework become less expressive than, e.g., F-logic, in the sense that this additional constraint reduces the kinds of admissible statements. Such a distinction may have a positive impact on implementation efforts, since a trade-off between expressiveness and efficiency is clearly present in logic-based database-modelling efforts.

KL is certainly an attractive proposal and closer to what one would expect from a database point of view than F-logic. Since it was conceived as a stand-alone language it tackles the issue of controlling execution, upon which database technology is dependent for implementing the notions of transaction, concurrency, and recovery, for instance. If DOODs are to be an evolution of DRDB, then KL gives a much clearer picture of this evolution than F-logic. The lack of multiple inheritance and of support for late binding are points that make KL less attractive from an object-oriented point of view.

The OO-Prolog Approach

Strictly, the OO-Prolog approach can be seen from two perspectives. In the first, the attempt is to incorporate OO features into the Prolog language. There is no implied restriction to simple database applications of a language such as Datalog. This is consistent with the trend towards database- and persistent-programming languages.

In the second, which we concentrated on, Prolog is regarded as the platform from which OODBs can be implemented. The main attraction of the OO-Prolog approach is the ease with which it may be implemented. In terms of a practical database system, access to a non-resident clause store is required. The main disadvantage is the lack of theoretical soundness of the full Prolog language, giving rise to problems with optimisation, tuple-based evaluation, order-sensitive programs, etc.

The main issue here is that no language-convergence is apparent, so that it is difficult to see how the seamless integration achieved in Datalog with respect to the relational model would be possible here with respect to the object-oriented paradigm.

The Stony Brook Approach

The Stony Brook approach has the greatest appeal in terms of providing a major breakthrough. Their main claim is that returning to foundations is the only way of guaranteeing that the blend of logic and OODBs will be as fruitful as it has been in the case of RDBs. This seems quite appealing, but more evidence must be produced that the conceptual mismatch between a classical predicate-logic and notions brought into play by object-orientation have actual consequences that cannot be solved by evolutionary approaches. Certain technical aspects of the F-logic proposal appear to substantiate the Stony Brook claim to some extent. For instance, extending the range of constructs that can be assigned as values of a logical variable results in an important increase in expressiveness. This, in turn, can be used to account for object-oriented notions hitherto untreated by other proposals in an integrated way. Also, languages have, besides a syntactic and a semantic dimension, a pragmatic one as well. The ease of use of a language, its conciseness and expressiveness, all are important factors determining the appeal it will have for users.

The main issue is essentially implementational, and at this stage there is no certainty as to whether

the increase in expressiveness will not jeopardise the overall efficiency of systems. Overall the Stony Brook approach seems to be in a better position than the others to deliver a principled and pragmatically useful deductive database language, although practical systems are some way away.

5 Conclusions

The Independent Data Model Issue

We have noted that a common feature of the language spaces defined on each of the three approaches is the absence of an upper branch that is comparable to the one in figure 1. We claim that this upper branch, representing the independent definition of a data model from a common logical source, is indispensable if DOODs are to become useful in real-world situations. This could be argued for on pragmatic grounds alone, insofar as it seems much too ambitious to handle all the issues that arise in the context of enterprise-wide databases within a single linguistic framework, if the applications that are being catered for are complex enough to require the power of OODBs.

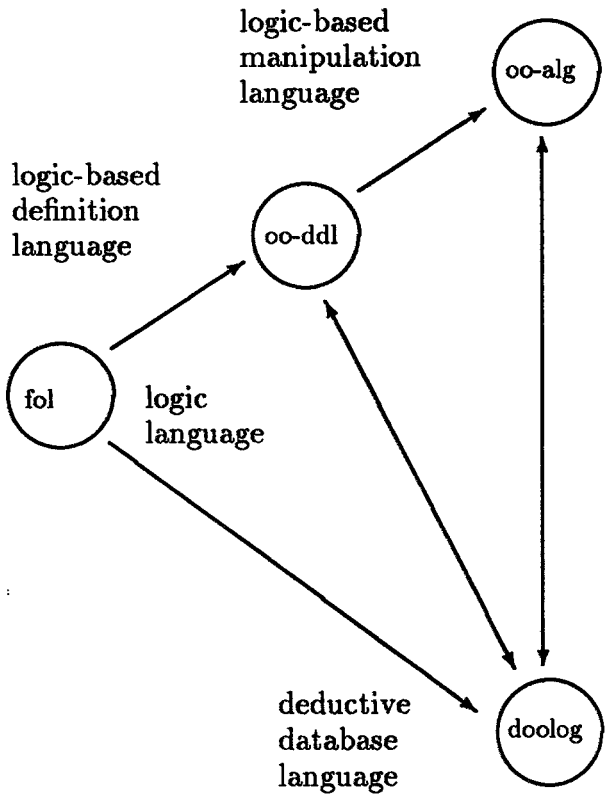


Figure 5: The Object-Oriented Language Space

Consider the object-oriented language space in figure 5. Apart from not containing an object-oriented equivalent to Prolog, it closely resembles the relational one in figure 1. It reflects the reliance of a deductive object-oriented language (which we are referring to as Doolog) on independent specifications of sublanguages for the definition and manipulation of objects as a modelling primitive.

Then, Doolog can be designed to be the converged language, strictly speaking, of a triple ⟨*FOL, OO-DDL, OO-Algebra*⟩. What we mean by *FOL* is some first-order language from which the two branches can spring. It could be classical FOPL, it could be some syntactically-adjusted version of it such as HiLog [17], or still, it could be some other, maybe new, logical framework or architecture. In any case, it *must* be capable of serving as the foundation of a Horn-clause resolution-based theorem-prover that is sound and complete. What we mean by *OO-DDL* is a language for defining objects, both structurally and behaviourally, with which application domains can be modelled. What we

mean by an *OO-algebra* is an algebraic specification as ADTs of the objects yielded by the OO-DDL, i.e. the equational definition of a minimal set of operations that are well-defined on objects independently of any additional application-dependent typing and, naturally, of representational choices.

Different Proofs Implying Different Identities

We now confront each of the approaches we have discussed with Ullman's criticisms (see section 1). These are strongly centred on a view of deductive databases as the ideal ground for declaratively expressing ad-hoc queries. Among other things, such queries must be amenable to having their safeness concluded, to producing results which are immediately available for further manipulation, and to being optimised automatically. One could argue that this is a very limited view of deductive database technology, but it is by no means an irrelevant one.

Ullman's first point, about assigning different identities to entities derived by different proofs, is probably the easiest to answer, insofar as there is simply no reason why deductive object-oriented databases are compelled to create object identities blindly for each proof of each fact.

In most OO-Prologs, objects are created explicitly as opposed to being derived by means of rules. The COL/IQL/KL family already accommodates the option of *not* creating a new object identity if an existing object meets the specification, i.e. if it already has the value that would end up being associated with the new identity. Ullman tries to preempt this response by arguing that this is not to take object identity seriously, as he has chosen to phrase it. However, in a seminal analysis of this notion [31], a **merge** operation has been described that destroys a superfluous identity because the value associated with it is equal to that of another existing object. KL simply allows one not to create that superfluous identity in the first place, and can therefore be seen to be addressing an issue about taking object identities seriously. The COL/IQL/KL answer to Ullman's first point can be adopted in the two other approaches. As noted above, the creation of object identities suggested by the Skolemisation technique used in F-logic can be similarly adopted in the other two approaches. Combined they adequately address Ullman's first point.

Dynamic Typing of Answers to Ad-hoc Queries

Ullman's second point, about the problem of manipulating objects (in Wegner's sense, see section 1) which do not conform to any previously declared type, is harder for the approaches we have discussed to answer, although, as noted above, in many situations objects need not be implicitly created. DRDBs are not affected by this problem because they inherit from the RDM the algebra which results from taking relations as an ADT, and no strong domain-specific typing constraints are declarable in the RDM. In fact, no application-specific behaviour is specifiable in the RDM: operations on relations are the same across applications no matter how different they are in reality.

If one attaches a strict interpretation to the role of typing in OODBs, the question could well be found to be indeed very hard to answer: there seems to be no obvious way of deriving a new entity by means of rules and simultaneously endowing it with a collection of operations which define it in the eyes of users. However, the lack of distinction between schema-level and instance-level elements in F-logic suggests that facts about the schema (hence including those about application-specific behaviour) can be the subject of deduction on a par with facts about its instantiation. The problem then becomes how to constrain this power within bounds that are determined by the users' shared expectation about how every object, no matter its application-defined type, should behave

It is arguable that neither KL nor F-logic takes a view of objects that conforms to Wegner's definition which is the one assumed by Ullman. Because they do not abide by this behaviourally-determined definition of objects they largely preempt the question. If conforming to a type has no effect on behaviour and only impacts the structure of objects, methods can be thought of simply as attributes, i.e. functions from objects to values. Note that this question is tied to the question of encapsulation, a feature in which the proposals we described are clearly inadequate, with the exception of ADAM which has strong encapsulation.

We argue that an *OO-algebra* as we defined it above offers a suitable compromise, i.e. it results in DOODs that offer the same kind of functionality that DRDBs provide to users. Objects can still be behaviourally-determined in general, but objects that are derived by rules need only display that behaviour which is defined on any object whatsoever simply by virtue of their being an instance of

a particular ADT. This is fully equivalent to the functionality offered by DRDBs.

Optimising Ad-hoc Queries

Ullman's third point, about the impossibility of optimising queries cast in terms of operations whose properties are not known to the optimiser program, can be addressed in similar fashion.

Again, by not facing up to the behaviourally-oriented definition of objects the proposals we have described can be said to not even acknowledge the problem. Again we argue that specifying objects as ADTs offers a suitable compromise. To use Ullman's example, if queries use methods such as `join`, the problem of assuring that its properties are known to a query optimiser could be reduced to the problem of giving it a semantics in terms of the basic operations defined on the object ADTs. This leads to a construal of methods as higher-level operations definable in terms of more basic ones. As the latter are algebraically defined by equations, a query optimiser has the means with which to conclude facts about equivalence and thus decide on the optimal formulation of a query.

Concluding Remarks

The DRDB research enterprise has been very successful. There is no reason not to try and use the same strategies used in it in the case of DOODs. However, criticisms have begun to appear to which the more prominent approaches to DOODs do not respond satisfactorily. In this paper, after describing the strategies used to build DRDBs, we noted that attempts of applying them to DOODS have overlooked the issue of an independently-defined but logically-compatible data model, mainly, we believe, because no such model has been proposed that is widely accepted as satisfactory. We argued that this definition, apart from being important in itself, is a necessary step in responding to a certain class of criticisms, because only then is it possible to characterise what we have described as a *converged language* for DOODs, of which Datalog is the equivalent in the DRDB case.

6 Acknowledgements

We would like to thank Dr. Keith G. Jeffery of Rutherford Appleton Laboratory for useful discussions on the subject of this paper. Our work is funded funded by the Science and Engineering Research Council through the IEATP programme, and their support is duly acknowledged. We would also like to thank Dr. J.M.P. Quinn of ICL and Mr. N. Smith of Ordnance Survey as the industrial partners in our present research project.

References

[1] Serge Abiteboul. Towards a Deductive Object-Oriented Database Language. *Data & Knowledge Engineering*, 5:263–287, 1990. Also in [36], pages 453-472.

[2] Serge Abiteboul and Stéphane Grumbach. COL: A Language for Complex Objects Based on Recursive Rules. In *Proceedings of the Workshop on Database Programming Languages*, pages 253–276. Roskoff, 1987. Complete version in [3].

[3] Serge Abiteboul and Stéphane Grumbach. COL: A Logic-Based Language for Complex Objects. In *[8]*, pages 347–374. 1990. Abstract in [2].

[4] Serge Abiteboul and Paris C. Kanellakis. Object Identity as a Query Language Primitive. In James Clifford, Bruce Lindsay, and David Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, pages 159–173, Portland,OR, June 1989. ACM Press. This is a preliminary version of [5].

[5] Serge Abiteboul and Paris C. Kanellakis. Object Identity as a Query Language Primitive. Technical Report 1022, INRIA, Le Chesnay, April 1989. A preliminary version appears as [4].

[6] Malcolm Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley B. Zdonik. The Object-Oriented Database System Manifesto. In *[36]*, pages 223–240. 1990.

[7] François Bancilhon. Object-Oriented Database Systems. In *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 152–162, Austin,TX, March 1988. ACM Press.

[8] François Bancilhon and Peter Buneman, editors. *Advances in Database Programming Languages*. Frontier Series. ACM Press/Addison-Wesley, New York, NY, 1990. ISBN 0-201-50257-7.

[9] Catriel Beeri. A Formal Approach to Object-Oriented Databases. *Data & Knowledge Engineering*, 5:353–382, 1990. Also in [36], pages 405-430.

[10] Jorge Bocca. Megalog: A Platform for Developing Knowledge-Base Management Systems. In *Proceedings of the International Symposium on Database Systems for Advanced Applications*, pages 374–380, Tokyo, 1991.

[11] F. Cacace, Stefano Ceri, S. Crespi-Reghizzi, Letizia Tanca, and R. Zicari. Integrating Object-Oriented Data Modeling with a Rule-based Programming Paradigm. In *Proceedings of the 1990 ACM SIGMOD International Conference on the Management of Data*, pages 225–236, Atlantic City, NJ, May 1990. ACM Press.

[12] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases*. Springer-Verlag, Berlin, 1990. ISBN 3-540-51728-6, ISBN 0-387-51728-6.

[13] Qiming Chen and Wesley W. Chu. HILOG: A High-Order Logic Programming Language for Non-1NF Deductive Databases. In *[36]*, pages 431–452. 1990.

[14] Qiming Chen and Georges Gardarin. An Implementation Model for Reasoning with Complex Objects. Technical Report 793, INRIA, Le Chesnay, February 1988.

[15] Qiming Chen and Georges Gardarin. Nested Datalog: A Rule Language for Complex Objects. In *Proceedings of the 1988 ACM SIGMOD International Conference on the Management of Data*, Chicago,IL, 1988. ACM Press.

[16] Weidong Chen, Michael Kifer, and David S. Warren. HiLog: A First-Order Semantics for Higher-Order Logic Programming Constructs. In *Proceedings of the North American Conference on Logic Programming*, pages 1090–1114, October 1989. ISBN/ISSN 0-262-62064-2. Expanded Version in [17].

[17] Weidong Chen, Michael Kifer, and David S. Warren. HiLog: A Foundation for Higher-Order Logic Programming. Technical report, Department of Computer Science. State University of New York at Stony Brook (SUNY), 1989. Expanded Version of [18, 16].

[18] Weidong Chen, Michael Kifer, and David S. Warren. HiLog as a Platform for Database Languages (or Why Predicate Calculus Is Not Enough). In Richard Hull, Ron Morrison, and David Stemple, editors, *Proceedings of the Second International Workshop on Database Programming Languages*, pages 315–329, Gleneden Beach, OR, June 1989. Morgan Kaufmann Publishers, Inc. ISBN 1-55860-072-8. Expanded Version in [17].

[19] Weidong Chen and David S. Warren. C-Logic of Complex Objects. In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 369–378, Philadelphia,PA, March 1989. ACM Press.

[20] Edgar F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[21] Mukesh Dalal and Dipayan Gangopadhyay. OOLP: A Translation Approach to Object-Oriented Logic Programming. In *[36]*, pages 593–606. 1990.

[22] Claude Delobel, Michael Kifer, and Yoshifumi Masunaga. *Deductive and Object-Oriented Databases (Second International Conference DOOD'91, Munich)*. Springer-Verlag, Berlin, 1991. ISBN 3-540-55015-1, ISBN 0-387-55015-1.

[23] Agostino Dovier, Eugenio G. Omodeo, Enrico Pontelli, and Gianfranco Rossi. {log}: A Logic Programming Language with Finite Sets. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 111–124, Paris, July 1991. The MIT Press (Logic Programming Series).

[24] Elizabeth Fong, William Kent, Ken Moore, and Craig Thompson. X3/ SPARC/ DBSSG/ OODBTG Final Report, September 1991.

[25] K. Fukunaga and S. Hirose. An Experience with a Prolog-Based Object-Oriented Language. In *OOPSLA '86 Conference Proceedings Object-Oriented Programming: Systems, Languages and Applications*, pages 224–231, Portland,OR, September 1986. ACM Press.

[26] Hervé Gallaire, Jack Minker, and Jean-Marie Nicolas. Logic and Databases: A Deductive Approach. *ACM Computing Surveys*, 16(2):153–185, June 1984.

[27] Georges Gardarin and Patrick Valduriez. ESQL2: An Extended SQL with Object and Deductive Capabilities. Technical Report 1185, INRIA, Le Chesnay, 1990.

[28] P.M.D. Gray and R.J. Lucas, editors. *Prolog and Databases: Implementations and New Directions.* Ellis-Horwood, Chichester, 1988.

[29] P.M.D. Gray, D.S. Moffat, and N.W. Paton. A Prolog Interface to a Functional Data Model Database. In *Advances in Database Technology: Proceedings of the EDBT '88 Conference*, pages 34–48, 1988.

[30] M.H. Kay and P.J. Rivett. An Overview of the Raleigh Object-Oriented Database System. *ICL Technical Journal*, November, 1991.

[31] Setrag N. Khoshafian and George P. Copeland. Object Identity. In *OOPSLA '86 Conference Proceedings Object-Oriented Programming: Systems, Languages and Applications*, pages 406–416, Portland,OR, September 1986. ACM Press. Also in [57], pages 37-46.

[32] Michael Kifer and Georg Lausen. F-logic: A Higher-Order Language for Reasoning about Objects, Inheritance and Scheme. In James Clifford, Bruce Lindsay, and David Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, pages 134–146, Portland,OR, June 1989. ACM Press. A revised and extended version exists as [33].

[33] Michael Kifer, Georg Lausen, and James Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. Technical Report 90/14, Department of Computer Science, State University of New York at Stony Brook (SUNY), June 1990. (2n Revision). A preliminary version appears as [32].

[34] Michael Kifer and James Wu. A Logic for Object-Oriented Logic Programming (Maier's O-logic: Revisited). In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 379–383, Philadelphia,PA, March 1989. ACM Press. Preliminary Report of [35].

[35] Michael Kifer and James Wu. A Logic for Programming with Complex Objects. Technical report, Department of Computer Science, State University of New York at Stony Brook (SUNY), 1989. Extended Version of [34].

[36] Won Kim, Jean-Marie Nicolas, and Shojiro Nishio, editors. *Deductive and Object-Oriented Databases (First International Conference DOOD'89, Kyoto).* Elsevier Science Press (North-Holland), Amsterdam, 1990. ISBN 0-444-88433-5.

[37] Gabriel M. Kuper. An Extension of LPS to Arbitrary Sets. Technical report, IBM, 1987.

[38] Gabriel M. Kuper. Logic Programming with Sets. In *Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 11–20, San Diego, CA, March 1987. ACM Press. Revised version in *Journal of Computer and System Sciences*, 41(1):44-64, August 1990.

[39] Gabriel M. Kuper. On the Expressive Power of Logic Programming with Sets. In *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 10–14, Austin,TX, March 1988. ACM Press.

[40] Christophe Lécluse, Philippe Richard, and Fernando Velez. O_2, an Object-Oriented Data Model. In *Proceedings of the 1988 ACM SIGMOD International Conference on the Management of Data*, Chicago,IL, 1988. ACM Press. Also in [57], pages 227-236 and in [8], pages 257-276.

[41] Yanjun Lou and Z. Meral Ozsoyoglu. LLO: An Object-Oriented Deductive Language with Methods and Method Inheritance. In James Clifford and Roger King, editors, *Proceedings of the 1991 ACM SIGMOD International Conference on the Management of Data*, pages 198–207, Denver, Co, May 1991. ACM Press.

[42] David Maier. A Logic for Objects. Technical Report CS/E-86-012, Oregon Graduate Center, Beaverton, OR, November 1986.

[43] J. Malenfant, G. Lapalme, and J. Vaucher. ObjVProlog: Metaclasses in Logic. In *Proceedings ECOOP*, pages 257–269, 1989.

[44] Frank G. McCabe. *Logic and Objects: Language, Application and Implementation.* PhD thesis, Imperial College of Science and Technology, London, November 1988.

[45] Shamim A. Naqvi and Shalom Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, Rockville, MD, 1989. ISBN 0-7167-8200-6.

[46] Norman W. Paton. ADAM: An Object-Oriented Database System Implemented in Prolog. In M. H. Williams, editor, *Proceedings of the Seventh British National Conference on Databases (BNCOD 7)*, pages 147–161, Edinburgh, July 1989. Cambridge University Press.

[47] Norman W. Paton and Oscar Díaz. Metaclasses in Object-Oriented Databases. In R.A. Meersman, W. Kent, and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design and Construction(DS-4)*, pages 331–347, Windermere, July 1990. North-Holland. ISBN 0-444-88929-9.

[48] Raymond Reiter. Towards a Logical Reconstruction of Relational Database Theory. In Michael L. Brodie, John Mylopoulos, and Joachim W. Schmidt, editors, *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, pages 191–233. Springer-Verlag (Topics in Information System Series), New York, NY, 1984.

[49] Jörg H. Siekmann. Unification Theory. *Journal of Symbolic Computation*, 7(3 & 4):207–274, March/April 1989.

[50] Jeffrey D. Ullman. A Comparison Between Deductive and Object-Oriented Database Systems. In *[22]*, pages 263–277. 1991.

[51] Peter Wegner. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger*, 1(1):7–87, August 1990. Expansion of Oct 4 OOPSLA-89 Keynote Talk.

[52] M.H. Williams, G. Chen, D. Ferbrache, P. Massey, S. Salvini, H. Taylor, and K.F. Wong. Prolog and Deductive Databases. *Knowledge-Based Systems*, 1(3):188–192, 1988.

[53] Limsoon Wong. Inference Rules in Object-Oriented Programming Systems. In *[36]*, pages 493–509. 1990.

[54] Carlo Zaniolo. Object-Oriented Programming in Prolog. In *IEEE International Symposium on Logic Programming*, pages 265–270, 1984.

[55] Carlo Zaniolo. Object Identity and Inheritance in Deductive Databases – An Evolutionary Approach. In *[36]*, pages 7–24. 1990.

[56] Stanley B. Zdonik and David Maier. Fundamentals of Object-Oriented Databases. In *[57]*, pages 1–32. 1990.

[57] Stanley B. Zdonik and David Maier, editors. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann, San Mateo, CA, 1990. ISSN 1046-1698, ISBN 1-55860-000-0.

RHODES UNIVERSITY LIBRARY