INTERRUPT-GENERATING ACTIVE DATA OBJECTS

submitted in fulfilment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

by

PETER GRAHAM CLAYTON

Department of Computer Science
Rhodes University

November 1989

ABSTRACT

An investigation is presented into an interrupt-generating object model which is designed to reduce the effort of programming distributed memory multicomputer networks. The object model is aimed at the natural modelling of problem domains in which a number of concurrent entities interrupt one another as they lay claim to shared resources. The proposed computational model provides for the safe encapsulation of shared data, and incorporates inherent arbitration for simultaneous access to the data. It supplies a predicate triggering mechanism for use in conditional synchronization and as an alternative mechanism to polling.

Linguistic support for the proposal requires a novel form of control structure which is able to interface sensibly with interrupt-generating active data objects. The thesis presents the proposal as an elemental language structure, with axiomatic guarantees which enforce safety properties and aid in program proving. The established theory of CSP is used to reason about the object model and its interface.

An overview is presented of a programming language called HUL, whose semantics reflect the proposed computational model. Using the syntax of HUL, the application of the interrupt-generating active data object is illustrated. A range of standard concurrent problems is presented to demonstrate the properties of the interrupt-generating computational model. Furthermore, the thesis discusses implementation considerations which enable the model to be mapped precisely onto multicomputer networks, and which sustain the abstract programming level provided by the interrupt-generating active data object in the wider programming structures of HUL.

ACKNOWLEDGEMENTS

Several post graduate students have worked under my supervision at the Honours and Masters Degree levels to produce software and hardware products which have made direct contributions to the research described in this thesis. In this respect, I am grateful to Kate Henderson, David Hill, and Wendy Sanger. The specific contributions of these students are acknowledged in footnotes in the relevant sections of the thesis.

I am also grateful for the contributions of Nic Cooke, Caroline Handler, Mike Hartman, William Hayes, Debbie Sole, Mike Ward, and Karen Wrench, who have worked under my supervision on projects which have been closely related to this one. Although they did not contribute directly to the research for this thesis, they shared my enthusiasm for parallel and distributed computing and helped to create a school of interest from which I and their fellow students were able to benefit.

I recognize the contributions of friends and colleagues who have patiently heard me out and have offered constructive criticism, in particular Denis Riordan and John Bradshaw of Rhodes University, Judy Bishop of the University of the Witwatersrand, and Peter Wentworth, formerly of the University of Port Elizabeth, now of Rhodes University. I also recognize the assistance of Paul Walker of Inmos Ltd., Bristol, in obtaining scarce items of literature, in putting me into contact with relevant colleagues of his, and in organizing the donation of an Occam system to Rhodes University at a time when funds were low.

I am thankful for the support of my wife, Lulu, who provided the emotional strength I needed and helped me to identify my priorities amongst research, teaching, family, and social commitments during the strained periods of preparing this thesis.

I owe a great debt of gratitude to Professor Patrick Terry, who has been a mentor and a friend, and whose Clang compiler was the starting point for developing a parser for the language described in chapter four.

This research was supported with capital equipment funding from the Rhodes University Council and Johnson & Johnson S.A.

PETER CLAYTON

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Trademark Notice

1. Introduction

1.1 Background

The emergence of VLSI technology has made it possible to construct a powerful microcomputer with memory, processor and communications on a single device. The ability to link such devices to each other in arbitrary topologies has encouraged the construction of high-speed parallel processing systems at relatively modest costs.

Having cheap parallel hardware to achieve high-performance computation represents only part of a solution to the problem of constructing powerful computing systems that were previously economically infeasible; concomitant software must also be provided.

In the past two decades, the field of concurrent programming has been a prolific area of research, spurred on by the realization that concurrent algorithms frequently provide a more naturally expressed solution to many problems. Concurrent programming principles are no longer solely the domain of the implementors of operating systems, but are being applied to an ever increasing range of applications. Several authors affirm that when a concurrent solution is formulated in a sequential programming language, the mapping of the solution onto the sequential program is unnatural, and therefore error prone, difficult to maintain, and unreliable [BUS 88] [BUR 88b] [JON 85]. Numerous programming languages have been developed for the expression of concurrent algorithms, of which Ada [ADA 83], Occam [MAY 87], and Modula-2 [WIR 85b] are currently prevalent. In addition to the software engineering benefits that modern concurrent programming languages afford applications in which concurrent behaviour is a fundamental aspect of the problem area, they provide for the direct representation of processes which will execute in parallel on multiprocessor hardware. This enables the software designer to obtain increased performance, in terms of speed [SIM 83], or reliability [CAR 88a], or both.

The memory of early multiprocessor computer systems was shared by the processors in the system [HWA 85], and early notations for expressing the interaction of concurrent processes were designed with this type of target hardware in mind. For instance, two powerful proposals for providing synchronization in shared variable systems were semaphores [DIJ 68] and path

expressions [CAM 74]. Other influential proposals, especially adapted to mutual exclusion in shared memory systems, were critical regions and monitors [HOA 74].

The sharing of either memory or a communications bus between processors places an upper limit on the number of processors which a multiprocessor computer can accommodate due to the finite bandwidth of the shared resource [QUI 87]. Since the appearance of VLSI computing devices which are affordable in large numbers, cost effective parallel processing hardware designs have tended towards systems constructed from processors which have local memory only, and which communicate with each other via high speed interconnection networks[1]. Instead of reading and writing shared variables, the concurrent processes of programs which execute on such systems communicate and synchronize with each other using message passing. The notations put forward in CSP [HOA 78] and DP [BRI 78] reflect this type of multiprocessor architecture, as do languages influenced by them such as Occam and *MOD[2] [COO 80][3].

It is readily accepted that the efficiency of solving a particular problem depends primarily on the degree to which the architecture of the computer supports the problem solving primitives. The term *semantic gap* [MYE 78] is used to describe the extent of the disparity between a computer's architecture and the programming languages which are used to solve problems on it. For example, the semantic gap between a computer's architecture and its assembly language is typically small, while the gap between its architecture and a general high level language is large, requiring more elaborate translation techniques to bridge the gap effectively. The semantic gap between concurrent programming languages based on shared memory principles and distributed processing networks which have no shared memory is particularly large. As yet, no conceptually innovative

---

[1]Notable commercial examples are Inmos's transputer [WAL 85], NCUBE's NCUBE/10 (and other similar N-cube machines) [HEA 87], and Intel's iPSC [INT 88].

[2]*MOD is pronounced STAR-MOD.

[3]Ada, also influenced by CSP and DP, does not completely reflect a distributed architecture and contains certain features (identified by Stammers [STA 85]) which rely on a single shared memory.

techniques have been put forward for narrowing this gap, and the languages which have been implemented successfully on these systems have been of the message passing class of concurrent programming languages.

A second semantic gap exists in programming environments, that which describes the extent to which the program suits the solution it represents and the ease with which the programmer is able to express the solution using the facilities of the programming language. In this respect, effective software development tools which are able to cope with distributed parallel processing networks (in which processors have no memory in common) have lagged behind innovative hardware developments. In their endeavour to narrow the semantic gap between the language and its host computer, and thereby increase the efficiency of the problem solving environment, the designers of languages like Occam and *MOD have increased the semantic gap between the programming language and its user by supplying only low level primitives for communication. This often results in references to passive data items in a solution having to be mapped by the programmer onto sets of message exchanges. If not done extremely carefully, this transformation can be the source of abstruse run-time errors. Whereas message-oriented languages are well suited for programming pipelined computations, shared variable languages are far better suited for programming the large class of client/server systems [AND 83]. In addition, the increased versatility of VLSI computing devices, which allows them to be linked to each other in arbitrary topologies, has prompted the designers of message-oriented languages to provide facilities within the language which require the programmer to specify (in the source code) on which processor of the host network each component of the program will execute. This significantly simplifies the implementation of the language and allows for efficient execution of the program in a distributed parallel processing environment, but it forces the software developer to solve problems at the level of the hardware, rather than at an abstract problem solving level.

Harland has noted the degree of complexity which low level, concurrent language features force upon the application programmer, and has written [HAR 86a]:

> *It is only by raising the means of expression's (sic) level of abstraction towards that of the conceptual model that the complexity of an application becomes manageable.*

The range of approaches to parallelism trades programmer efficiency against machine efficiency

and ease of implementation. At the lower end of the scale, parallel programming languages require explicit programmer control (such as in Occam), with a high degree of programming complexity and a high level of machine efficiency. Parallelizing compilers at the opposite end of the scale (such as for PARLOG [GRE 87]) provide completely automated parallelism which provides a low degree of programming complexity and a low level of machine efficiency[1]. The proposal of this thesis takes a middle-of-the-road approach. It attempts to provide a realistic programming environment by working from the lower end of the scale, the approach which facilitates a high level of machine efficiency, and by introducing an elemental programming construct which reduces the program complexity without unduly retarding machine efficiency.

The research described in this thesis arose out of the observation that concurrent entities interrupting one another as they lay claim to shared resources are a natural feature of human problem domains. The research sought to enhance the effectiveness of programmers of non-shared memory, message passing architectures by providing a tool to facilitate the natural modelling of physical interaction in their problem domains.

1.2 Aims

The primary aim of the research described in this thesis was to investigate the introduction of a novel, active object, incorporating an encapsulated, shared data item with a predicate triggering mechanism, as a means of reducing the effort of programming distributed memory multicomputer networks. To support the rapid development of reliable and maintainable distributed programs, the object was represented by a new class of elemental language structure, which would be simple and natural to use, which would facilitate a pertinent abstract programming level, which could be mapped precisely onto the host network, and which would guarantee that conditions needed to prove properties of programs were inherent in the axioms for the language feature.

---

[1] In developing a methodology for translating PARLOG programs to Occam 2, Scott and Trehan [SCO 89] discovered that the vastly contrasting mechanisms for specifying parallelism, process creation, and network placement in the two languages meant that their methodology could only be applied to the restricted class of PARLOG programs which created a deterministic process network structure. This enabled the Occam 2 constructs for specifying a static network structure at compile time to be used.

Naturally a new proposal for a programming construct could not be investigated without other language features to support the coding of test programs and examples. In designing an experimental programming language which presupposed the new data object, the research worked towards an ancillary aim, the investigation of wider support structures which made it unnecessary to compromise the benefits of the proposed language feature by breaching the abstract programming level which it facilitated.

1.3 Fundamental concepts

The use of interrupts is a well established principle of hardware interfacing and low level computing. Allowing an interface to inform the computer when it is ready to transfer data is an efficient and intuitive alternative to the CPU constantly monitoring a status register. In contrast, the regular monitoring of a status value is a common characteristic of the control structures of high level imperative programming languages. The designers of structured imperative languages have been unable to incorporate support for interrupts at the application program level because the conventional set of control structures for these languages has been designed uncompromisingly around environments for sequential execution[1]. Interrupts are only possible if two processes are active simultaneously, one to perform the task which might be interrupted, and the other to produce the interrupt signal. Concurrent programming languages afford program segments the potential for executing in parallel[2]; consequently, the support for interrupts between high level program segments is feasible.

A second feature, needed to reduce the programming effort for concurrent solutions which are naturally expressed using shared variables, is a mechanism for sharing items of data between

---

[1]Some multitasking operating systems (for example UNIX [BAC 86]) do allow system calls from high level languages which effect run-time interrupts in application programs.

[2]A *concurrent language* is a system which supports parallelism or pseudo-parallelism with constructs recognized by a compiler (not merely a library of system calls).

concurrent processes of the application program. In this thesis, a shared data mechanism is combined with the interrupt driven approach within a basic encapsulation model to produce the concept of an *interrupt-generating active data object*, an elemental language feature for combining a shared data facility with condition synchronization between concurrent processes. This concept is related to a number of previous language structures which have facilitated data encapsulation and sharing, and interrupt mechanisms[1], and the object model is deliberately intended to resemble a simple form of human *pop-up* memory[2]; this special type of object is expected to cause an interrupt to be generated when its stored data corresponds to a value of interest to a process of the application program.

The interrupt-generating active data object is something akin to a standard (multiple entry) function which has a *state* for remembering the effect of certain previous operations. Whereas function values are completely determined by their arguments; the value returned by an object may depend on its state as well as its arguments. In contrast to the objects of other object-based languages[3], the object proposed by this thesis is an elemental programming language feature, whose behaviour and interaction with other units in the program need not be explicitly defined by the programmer.

To support the proposal in a distributed parallel processing environment, the interrupt-generating active data object is not viewed as a passive resource; rather, it assumes the character of an active process which may communicate with other processes.

Figure 1 illustrates the functional components of a rudimentary interrupt-generating active data

---

[1]Related research is discussed in detail in chapter 2.

[2]Whereas existing computer processors are able to outstrip the human brain by orders of magnitude in the speed of performing calculations, no computer has matched the quite extraordinary speed with which information can be recalled in the human mind. Experimental psychologists have not yet been able adequately to explain the mechanism by which a piece of information relevant to a particular context is able to pop-up in the mind [EVA 82].

[3]These are discussed in section 2.3.

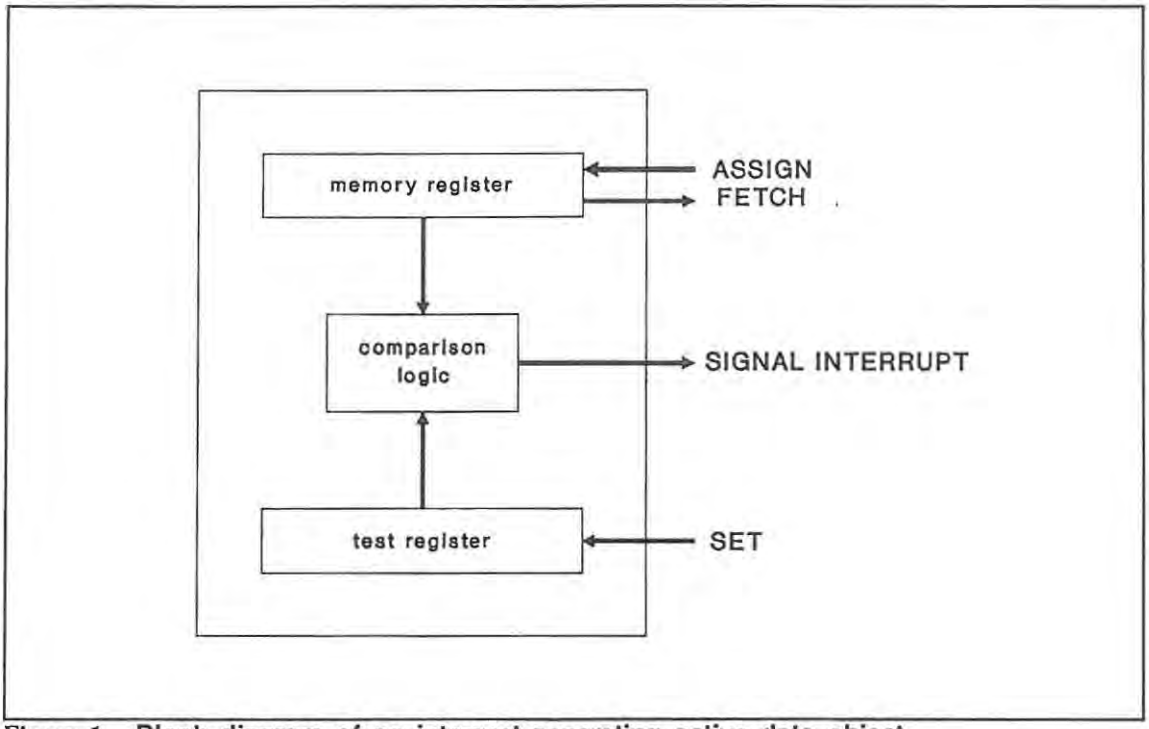


Figure 1. Block diagram of an interrupt-generating active data object.

object. This object incorporates a standard *memory* data register, a *test* condition register and a comparison mechanism. The comparison mechanism is able to perform a bit-wise comparison of the *memory* and *test* values and generate an interrupt signal if they are equal. The external signals depicted in figure 1 are events which represent both invocation signals and the movement of data, and are best viewed as synchronous messages. *ASSIGN* and *FETCH* represent the standard operations of sending values to - and fetching values from the object's *memory*. The *SET* message enables a *test* condition value to be recorded. The comparison mechanism is triggered automatically after every *ASSIGN* event (which alters the *memory* data value), to establish whether the object has assumed the particular *test* state of interest, and generate an interrupt signal if it has.

An important aspect of the interrupt-generating model of this thesis is that it provides guaranteed safety properties which safeguard against its injudicious use. These properties offer inherent protection to application programs, and provide guarantees which are useful in ensuring that an implementation meets its specification. The interrupt-generating active data object includes built-

in mutual exclusion to prevent interference caused by simultaneous attempts to access its data. In the model presented in chapter 3, this is achieved using synchronized communication within a CSP [HOA 85] form of alternative structure. This facility eliminates the need for the application programmer to write code to ensure exclusive access of the shared data (such code would normally be executed for every reference to the item even if attempts for simultaneous access were infrequent). As an additional precaution against interference, objects are limited to performing condition evaluation for only one client process. This safety property is formulated in terms of a restriction preventing several processes from assuming the same state simultaneously, a frequently used principle in concurrent programming [SCH 85]. The configuration of figure 1 illustrates this restriction at a practical level. The single *test* register is incapable of recording more than one distinct interrupt signalling value. The model is able to service *ASSIGN* and *FETCH* requests from any number of concurrent processes, but all interrupt signals will be sent to the process from which the *SET* message originated. In the text which follows, this restriction is referred to as the *ownership rule* of interrupt-generating active data objects[1].

This thesis not only proposes a new active object which behaves in a well defined manner, it also proposes that sequential processes have well defined properties which govern their interaction with the proposed object. Interrupt-generating active data objects are not simply autonomous encapsulation facilities, they require their client processes to be able to anticipate interrupt signals. Efficient and expressive linguistic support for this form of interaction requires the introduction of a particular class of programming construct, which avoids the use of polling or a blocking wait upon a conditional synchronization event.

Since a control structure is a technique that an application algorithm uses in determining the activity on some virtual machine, the design of a control structure must be concerned with the clear and axiomatic specification of what should happen in the computational process. Research into novel parallel programming structures has often taken the form of extensions to existing languages (as in the case of DISC (DIStributed C) [IAN 89] and some of the object-based

[1] In situations which require a broadcast interrupt, interrupt signals are fanned out by allowing the action sequence of one interrupt to trigger a number of interrupt-generating active data objects which in turn signal further interrupts.

languages discussed in section 2.3). This was not a viable development path for the proposal of this thesis, since the control structures which delimit code blocks in existing imperative languages do not possess the properties required to interact effectively with interrupt-generating objects[1]. The addition of the proposal to an existing programming language would require more than a simple language extension, it would require a fundamental change in the semantics of the language's control structures[2]. This thesis shows the formal properties needed for the proposed class of construct, and describes the semantics and use of an actual programming construct which falls into this class[3].

From the preceding discussion, it should be seen that a programming language incorporating the proposal will support concurrent programming by introducing two distinctly different conceptual entities: *objects* and *processes*. These entities represent two different levels of abstraction: processes are *actors* whose behaviour is defined by an application programmer, but whose interaction with objects is axiomatic, and objects are *servers* which, although they are also active entities, exhibit standard methods and constitute an elemental feature of the programming language.

---

[1]Chapter 2 describes the extremely cumbersome and non-intuitive approaches which are required to detect un-awaited interrupts in existing imperative control structures.

[2]Apart from this obstacle, the practice of adding low level parallel constructs onto conventional sequential languages is an insufficient approach to creating a new abstract programming environment, since it serves to stress the adaptation of the software tools to suit the hardware rather than the mapping of an abstract software development level onto the target hardware.

[3]The experimental programming language described in chapter 4 allows the programmer to make transparent use of interrupt-generating objects. Variables used within the context of interrupt driven constructs are implemented as active objects, while other variables are implemented as memory cells in the conventional way.

1.4 The scope of the research

The research described in this thesis was undertaken over a period of four years. It focused on creating a facility for constructing reliable and easily understood programs on MIMD[1], non-shared memory, message passing architectures. The research assumed reliable processing elements and communication mechanisms; it was not concerned with system fault-tolerance, and concentrated upon high level process interaction.

The bulk of the development effort concentrated on the design and implementation of an experimental concurrent programming language based on the proposed interrupt-generating active data object. This experimental language was given the name HUL[2]. Besides investigating the use of the proposed objects and their interaction with client processes, the design of HUL had the additional objective of facilitating host-independent concurrent programming, in the spirit of the ancillary aim of the research.

The first Occam language definition [MAY 83] was released at the time the research was conceived. Because it was a small, elegant and powerful concurrent language, Occam was chosen as the basis for HUL[3]. An existing recursive descent compiler for Clang [TER 86] was used as a springboard for implementing HUL[4]. The development of HUL was an evolutionary process, and the recursive-descent parser was soon replaced by a table-driven parser[5] which alleviated

---

[1]Multiple-instruction stream, multiple-data stream.

[2]HUL is an acronym for HUMAN-LIKE.

[3]Many of the language features of HUL are based on the original language features of Occam [INM 83], rather than Occam 2 [INM 87b], the more recent and larger version of the language.

[4]W.L. Sanger worked under the supervision of the author to produce the first parser for a subset of HUL, as a post graduate project [SAN 85].

[5]QPARSER, the LALR(1) parser generator by QCAD, was used [QCA 84].

several parsing problems, and provided a more robust implementation which could be altered easily for experimental purposes[1].

As part of the support environment for HUL, the model of the interrupt-generating active data object and its process interaction was tested using a set of pseudo-code interpreters on a time-sliced uniprocessor, on a microcomputer network using available IBM PC compatible microcomputers, and on a transputer-based multicomputer. In these environments, the proposed object was implemented as an active process in software, which executed in parallel with its client processes.

Much of the experimental work was done on the microcomputer network. When transputers became available, an Occam interpreter was developed to confirm the effectiveness of the distributed microcomputer network. The feasibility of migrating the support for interrupt-generating active data objects down to the hardware level was also tested, using a microprogrammable processor developed for this purpose [CLA 89c] and hosted in one of the microcomputers of the network.

Most of the software described in this thesis was written in Pascal. Early versions of the HUL development system were written in UCSD Pascal for execution on the Sage IV microcomputer. Later versions used Turbo Pascal and were developed under MS-DOS on the IBM PC range of microcomputers. The first Occam interpreter for HUL was developed using the evaluation kit available for the P-System on the Sage IV microcomputer [INM 84c]. This was later converted to Occam 2 to run on a transputer network.

The work described in this thesis is related to a number of previous research activities. To distinguish this work from other efforts, chapter 2 surveys a broad range of related research.

Although the work described in the thesis has a strong empirical component to it, it is supported

[1] K.A. Henderson developed the initial routines to perform the semantic actions of producing intermediate language codes for the table-driven HUL parser, as a post graduate project under the supervision of the author [HEN 86].

by the well established theory of communicating sequential processes (CSP) [HOA 85]. Chapter 3 of this thesis uses the theoretical foundation to present formal specifications for a safe interrupt-generating active data object and its proper interaction with communicating sequential processes.

Chapter 4 describes the use of the proposed interrupt-generating structure as part of the experimental programming language, HUL. The principal features of the language, and the endeavours at the syntactic level to conform to the central spirit of the research are discussed. Formal semantic descriptions and programming laws which relate to the proposed language feature are included. Chapter 5 presents a range of standard concurrency problems to illustrate the application and programming techniques of the interrupt-generating active data object.

Chapter 6 discusses the implementation considerations of the experimental programming language which are concerned with the proposed object, its interaction with client processes, and its incorporation into wider language structures. This chapter also evaluates the viability of the proposal in terms of the implementations' performance.

In conclusion, chapter 7 considers the potential of the proposed approach, and suggests possible avenues for future research.

This thesis makes frequent use of the theory of Petri nets as a modelling tool [PET 77], and the meta-language of CSP as a specification language [HOA 85]. A basic knowledge of Petri nets, the CSP notation, and the programming language Occam [INM 87b] is assumed of the reader. Appendix A contains a glossary of symbols used in this thesis, for reference by readers unfamiliar with these topics.

Appendix B defines the reserved identifiers and predeclared procedures available in HUL. The syntax of HUL is described in appendix C in Backus-Naur form, and as a set of syntax diagrams.

This thesis represents a revision of the original documentation describing the research. During the period of revision, several aspects of the implementation (covered in chapter 6) were published in book form as one of a collection of research essays [CLA 89a], and a paper

covering the central theme of the thesis was prepared and accepted for publication [CLA 89b]. Two aspects of the original document have been omitted from this thesis because they did not directly contribute to the central theme of this thesis. Both aspects, the microprogrammed emulation of the interrupt generation mechanism [CLA 89c] and an investigation into the use of interrupt driven data objects with sequential programming language constructs [CLA 89d], have now been reported in the Technical Document series of the Department of Computer Science at Rhodes University. The educational value of the microprogrammable hardware produced during the emulation exercise has been documented previously in a journal article [CLA 87a]. The investigation into sequential structures has been published in less detailed form as a conference paper [CLA 87c]. Two aspects of the design of HUL have been published previously in the Technical Document series of the Department of Computer Science at Rhodes University [CLA 86] [CLA 87b].

2. Related research

The distinguishing features of the interrupt-generating active data object proposed in this thesis are its data encapsulation and sharing facility, its interrupt generation mechanism, and its method of interaction with active processes at the level of an elemental language construct. This chapter surveys a broad range of related research which has made use of similar forms of one or more of these features, and compares each work with the proposal of this thesis to highlight common characteristics and differences. The discussion is confined to the concepts surrounding these distinguishing features, which are central to the proposal; further related research, which has influenced specific implementation details and the choice of other language features to support the coding of test programs and examples, will be discussed when these topics are introduced in subsequent chapters.

2.1 Interrupts and exceptions

Although the idea of avoiding the frequent execution of code for performing tests which will seldom evaluate to *true* is not a new one, the interrupt driven mechanism presented in this thesis is rather different from the paradigms of previous programming structures which have used this idea. Unlike interrupts which emanate from the hardware or the operating system, the interrupt proposed by this thesis is a structured message which signals an event in the application program.

A number of existing programming languages include high level constructs for handling *exceptional* events, notably PL/1, Ada, CLU, and Eiffel. The use of the term *exception* rather than *error* to describe a run-time failure for which recovery is possible was suggested by Goodenough [GOO 75]. The purpose of trapping exceptional conditions in existing programming languages is to make the writing of fault tolerant programs possible. The first language to incorporate such a feature was PL/1, by including structures called *ON-conditions* [CON 79]. A number of predefined exceptions exist in PL/1, which can be raised automatically when an error condition arises during the execution of a program. The *ON-conditions* are used to declare exception handlers in the source program. PL/1 does have a facility which allows programmers to specify the raising of

exceptions artificially, but no exceptions specific to the application program can be defined. The exception handling facilities of Ada are more general that those of PL/1. Ada provides each group of statements with an exception handling capability in a structure known as a *frame* [ADA 83]. Ada caters for both predefined and user declared exceptions, and allows the programmer to declare exception handlers for both types of exception. Predeclared exceptions are raised automatically, or artificially by the execution of a *raise* statement. A *raise* statement must be used to activate user declared exceptions. Ada's user declared exceptions allow *watch-dog* conditions to be specified which serve a similar purpose to the high level interrupts of this thesis. However, Ada exceptions do not facilitate any form of communication. Furthermore, exceptions are assigned the status of error reporting facilities and are intended as a means of writing fault-tolerant programs. The rationale for selecting Ada as an implementation language is most often its support for real-time embedded systems. Even so, for the interrupt handling semantics of the tasking mechanism of Ada to be effective, the implementation is required to provide support not specified by the Ada standard [RAS 86]. In particular, the tasking mechanism is unable to handle unexpected interrupts. This inability to handle an interrupt at an arbitrary point in the execution of the program is a major obstacle to the possible implementation of interrupt-generating active data objects in Ada[1] (although the semantics of a *select* statement within an Ada task body are similar to those of the mutual exclusion mechanism which the proposal of this thesis adopts). The recognition by the Ada Joint Program Office (of the US Department of Defence) of the deficiencies which still exist in Ada is manifested in the Ada IC contract announcement for a revision of the Ada standard, to be known as *Ada 9X* [2]. One of the primary areas of Ada which has been identified as requiring revision is the task model, which,

[1] Further features which make Ada unsuitable for implementing the data object of this thesis are described in section 2.5.

[2] Particulars of the *Ada 9X* revision request can be found in the INFO-ADA Digest, Volume 89, Issue 180, 13 August 1989.

if addressed suitably, would alleviate some of the limitations mentioned here[1].

Exceptions in Clu [LIS 77] are based on the termination model of exception handling [LIS 79]. A procedure can terminate under one of a number of conditions; one of these is considered to be the *normal* condition, while others are *exceptional* and are given user names. Exceptional conditions represent two categories of failure in Clu: expected but undesirable events (which have user defined names), and unexpected events, for which there is a standard *failure* condition. This model of exception handling has also been adopted by Argus [LIS 83]. The object-oriented language Eiffel [MEY 87] [MEY 88] provides source code *assertions* which are intended to increase the reliability of application programs by providing a debugging mechanism and a mechanism for fault tolerance and failure recovery. Used as an exception mechanism, the Eiffel assertion is more restrictive than the exception mechanisms of the other languages discussed; exceptions implemented in this way are raised automatically and cannot be raised artificially. Neither Clu nor Eiffel is an improvement upon Ada as an implementation language for modeling interrupt-generating active data objects.

In contrast to the primitive control structures of existing languages for exception handling, the interrupt-generating data object of this thesis provides a safer and more general facility which is not limited to the writing of fault tolerant systems.

2.2 Monitors

Concurrency brings with it the problem of synchronization. A popular way of addressing this problem in programming languages is the provision of a modular arbitration mechanism which controls the access of concurrently executing processes to a shared resource. Such mechanisms are required to receive messages from many processes simultaneously, but either block contending processes (as in the case of interrupt-generating active data objects), or queue up requests (as

[1] Van den Bos [VAN 80] provides a critical review of the concurrent programming facilities of Ada. Roberts *et al* [ROB 81] evaluate Ada in the context of real-time multiprocessor systems and conclude that Ada is somewhat lacking in this regard. The Ada tasking model generally has been the subject of much criticism [BAY 86] [BUR 85a] [BUR 85b] [GEH 84a] [RAS 86] [STA 85] [WEL 84] [WEL 86].

in the case of monitors) so that changes to the internal state of the shared resource are made serially, avoiding inconsistencies.

The interrupt-generating active data object is a close relative of the monitor, and owes a great intellectual debt to the pioneering work of Brinch Hansen [BRI 73] in conceiving this modular arbitration mechanism and Hoare [HOA 74] in further developing it. Because of its ease of use compared to previous synchronizing mechanisms, the monitor has become a widely applied concept, adopted by many concurrent languages, among them Concurrent Pascal [BRI 75], Modula [WIR 77], DP (Distributed Processes) [BRI 78], Pascal Plus [WEL 79] [BUS 88], Mesa [MIT 79] [LAM 80], Concurrent Euclid [HOL 83b], Emerald [BLA 86], Cedar [SWI 86] [ATK 89], and Joyce [BRI 87] [BRI 89a]. Monitor mechanisms are not actually part of the languages Edison [BRI 81] and Path Pascal [CAM 74][CAM 80], but the monitor concept is used extensively in these languages and its implementation by the *when* statement and *path expression* respectively is trivial, requiring no explicit semaphore variables. Monitors have also influenced the design of SR (Synchronizing Resources) [AND 82], which incorporates a variant of the monitor, and the object-oriented languages ABCL/1 [YON 86] and Orient 84/K [ISH 87], whose objects have a definite monitor-like behaviour.

The interrupt-generating object has been designed with built-in exclusive access. In this respect, its behaviour is similar in concept to the execution of procedures in a monitor, which is guaranteed to be mutually exclusive. When a process calls a procedure within a monitor, and there is no other procedure executing within that monitor, the call can be executed immediately. If the monitor is already executing, the calling process is placed on a monitor queue from which called procedures are executed in the order of call. References to an interrupt-generating active data object can be equated to calls to three procedures, *assign*, *fetch*, and *set*, declared within a monitor. The interrupt signalling mechanism proposed in this thesis is analogous to the *signal* operation on a condition variable proposed by Hoare for monitors [HOA 74], but is more general in that the process which receives the signal is not blocked while it awaits the signal.

This thesis proposes a more structured form of the monitor mechanism for standard access methods to a simple shared item, a mechanism which places arbitration and desirable abstract features of synchronization control into a lucid language construct. In doing so, the proposed

construct overcomes a number of problems associated with the use of the conventional monitor.

Interrupt-generating active data objects have better synchronization encapsulation than monitors in two respects. The first of these avoids the monitor's major source of programming errors. Monitors synchronize requests by providing a pair of operations for each request type. Such a pair of operations must be used in a particular order for synchronization to work properly, yet nothing in the monitor construct enforces this order. As a result, interleaving of operations due to *signal* and *wait* statements may invalidate the execution condition. The interrupt-generating active data object incorporates this structural aspect of synchronization into a unified message mechanism which relieves client processes of any responsibility for specifying event ordering, and which guarantees appropriate ordering for its inherent functions.

This problem has also been addressed in the development of the *serializer* [HEW 79a], another modular arbitration primitive, developed for use in the Actor languages[1] [AGH 86] [KAF 89]. Like the proposal of this thesis, the serializer is a more structured version of the monitor construct. The serializer solves the monitor's problem of interleaving operations by providing fewer entry points for external use. The serializer is able to provide a reasonably unified mechanism in shared memory systems, which caters for user defined methods within its structure by requiring additional *directions* of the client process during a single call. The call mechanism of the serializer is identical to that introduced for the *atomic object* of ConcurrentSmalltalk [YOK 86], and similar to that of the *atomic object* of Argus [LIS 83]. Because its set of operations is standard, the interface for client processes provided by the proposal of this thesis is much simpler than that of the serializer, and provides a natural interface for distributed environments. The original serializer model has been carried an important step further with the development of *primitive serializers* which can be implemented in distributed systems [HEW 79b].

The proposal of this thesis also improves upon the efficiency and modular encapsulation of the conventional monitor concept in the handling of conditional synchronization. In scheduling access to a protected resource, it is frequently necessary to require that a process remain waiting until

[1] Actor languages are discussed further in section 2.3.

a particular condition is satisfied before proceeding. To achieve this, monitors require a process to remain dormant until another process explicitly signals a dormant process that it should continue, inhibiting the degree of parallelism and requiring additional explicit programming. Path Pascal [CAM 80] addresses the problem of the degree of parallelism by requiring the user to implement a monitor explicitly using path expressions [CAM 74], thereby allowing the user to specify a high degree of parallelism, but with a loss of expressive power for specifying condition synchronization easily. The recent *coordinator* concept [KIM 89] adopts a similar approach to Path Pascal, but has improved the linguistic support for the expression of condition synchronization. *Serializers* improve upon the modularity of synchronization provided by conventional monitors, by providing that the condition for resuming execution must be explicitly stated when a process *waits*. This allows the serializer to determine the appropriate moment for further execution of the process. Interrupt-generating active data objects improve the modularity of synchronization still further, by providing a simple rendezvous interface with client processes; processes are simply blocked on a synchronous message exchange if their request cannot be dealt with immediately. Interrupt-generating active data objects also allow an unlimited degree of parallelism since there is only one condition for synchronization, and this condition does not affect participating processes unless they are explicitly waiting upon it, in which case they are notified automatically once the condition is met.

Because of the synchronous (blocking send and blocking receive) communication scheme between interrupt-generating active data objects and client processes, the proposed construct implicitly queues events which attempt to access the encapsulated data item concurrently. No queueing mechanism need be implemented as in the case of the conventional monitor or the more structured monitor extensions mentioned above. This provides for a very simple implementation model, but means that incoming communications of the same type from different processes need not be dealt with in the order in which they arrive. This does not affect the safety properties of the construct, or the fairness of selection provided that the implementation is correctly non-deterministic.

A further problem which plagues monitor implementations is the occurrence of deadlock which arises out of nested monitor calls. The linear relationship between interrupt-generating active data objects and their client processes sidesteps this problem.

With its improved intrinsic synchronization, the form of shared object put forward by this thesis can be seen as an extension of the monitor concept for a specific set of resources. The structures imposed by interrupt-generating active data objects provide important guarantees that aid in proving that the implementation meets its specifications. The specifications for an interrupt-generating active data object include integrity specifications relating to the type of access, to the order in which different types of access occur, and to the scheduling of condition synchronization activities.

2.3 Object-based programming structures

Object-oriented design is a natural consequence of the move towards abstraction in programming languages [BIS 86]. Objects are encapsulations of design modules based on data. Each object is protected from other objects by an encapsulation shell which restricts access to the object to some form of message passing mechanism. Each message is a request for an object to perform one of a number of enumerated actions, called *methods*.

Identity, encapsulation, and sharing, fundamental distinguishing characteristics of the existing broad class of object-based programming structures, are also central features of the active object model proposed in this thesis. With these properties in common, the superficial resemblance between the proposed interrupt-generating active data object and the multitude of existing object models might suggest that adequate facilities were already available for supporting the proposal. For the most part, however, existing object-based languages provide totally inadequate facilities for modelling the interrupt-generating active data object. In existing object-based languages, objects must be explicitly defined, placing the burden for ensuring that all necessary safety properties are present upon the application programmer. The object is not able to assume the level of a fundamental language component which has an implicit relationship with other language components. Indeed, the object-based languages completely ignore the possibility of unanticipated object interaction. In the area of exception handling, either a simple low level facility is provided, or exceptional conditions are ignored, resulting in the need for programmer level support to be provided for such conditions using language features which are unsuitable for the purpose.

From an object-oriented point of view, the proposal of this thesis does not provide the facilities offered by conventional object-based languages. The proposal focusses on high level interrupts and data sharing, not on the design of a new object-oriented language. The proposal provides a narrower form of data encapsulation than that which is associated with the object-orientated approach to programming. The encapsulation of data within an interrupt-generating active data object does not in itself provide support for data abstraction, although built-in support is provided to ensure that the constraints on multiple access and ownership cannot be violated. This means that the interrupt-generating active data object takes on much of the responsibility that application programmers would otherwise have to assume.

Object-oriented programming dates back to the design of the process-oriented language Simula [BIR 73] [KIR 89] over two decades ago[1], and has become a popular field of study in recent years. Simula pioneered the idea of *classes*, used to implement general objects that have knowledge to be shared by several instances. Subclasses could build on the behaviour of previously defined classes, introducing the idea of inheritance among objects. The class concept has been influential in the development of abstract data types and object-oriented programming, and is a feature (in various adapted forms) of many modern programming languages, such as the package concept in Ada [ADA 83], the object in Smalltalk [GOL 83], the module of Modula-2 [WIR 85b], Clu's clusters [LIS 77], the C++ class [STR 86], and Hoare's envelope [HOA 76], adopted by Pascal Plus [BUS 79] [BUS 88].

At roughly the same time (during the late 1970's), Kay at Xerox PARC and Hewitt at MIT began to use encapsulation principles as a fundamentally different way of looking at computation. They created the first programming languages designed specifically to support object-oriented design principles. Kay's Smalltalk [KAY 77] [ING 78] [GOL 83] was designed as a general information handling language for people other than computer scientists to use. Hewitt's Actor model [HEW 80] [AGH 86] was developed for applications in artificial intelligence, and as an attempt to provide a high level of inherent parallelism. Apart from these two specifically object-oriented

[1]Simula, developed as an extension of ALGOL 60 [NAU 63], was designed in the early 1960's at the Norwegian Computing Center in Oslo.

approaches, many object-based languages have taken existing languages with passive data and active procedures and have added language extensions to represent objects which can respond to messages. Examples of object-based languages which have been developed as extensions to the imperative programming mould are Traits [CUR 82] and Clu [LIS 77] based on Algol-like languages, Objective C [COX 86] and C++ [STR 86] based on C, Oberon [WIR 88a] based on Modula-2, and Apple's Object Pascal [APP 86], QuickPascal [BAR 89], and Turbo Pascal 5.5 [TUR 89] based on Pascal. On the declarative side, Flavors [MOO 84], CommonLoops [BOB 86], and CLOS (Common Lisp Object System) [BAR 89] have grown out of Lisp, and Intermission [KAH 82] and Vulcan [KAH 86] have grown out of Prolog[1].

In contrast to the emphasis which this thesis places upon concurrency and distributed processing, many object-based languages currently available do not support these features. Languages such as Simula [BIR 73], Smalltalk-80 [GOL 83], Objective-C [COX 86], C++ [STR 86], Trellis/Owl [SCH 86], Clu [LIS 77], CommonLoops [BOB 86], and CommonObjects [SNY 86b] aim at the high reusability of objects without emphasizing the need for concurrency. Some of these languages are quasi-concurrent (for example Simula and Smalltalk-80) because they allow objects to have independent threads of control, but only allow one thread to execute at a time. Examples of object-based languages which provide mechanisms for concurrency are Orient84/K [ISH 87], ABCL/1 [YON 86], ConcurrentSmalltalk [YOK 86], Emerald [BLA 86], Hybrid [NIE 87], CLIX [HUR 87], Pool-T [AME 87], and Vulcan [KAH 86]. The proposal of this thesis requires a language to have intra-process concurrency, as well as quasi-concurrency within its object structure. Monitor-like objects, such as are found in Orient 84/K, ABCL/1, and Emerald, have quasi-concurrent execution within objects, which allows threads of control to be suspended while waiting for a condition to be fulfilled and resumed when the condition is satisfied. Very few object models support unrestricted concurrency within objects. Those which do are Actors [AGH 86], guardians (Argus active objects) [LIS 82] [LIS 83], and SINA active objects [NIE 87]. Actors provide *serializers* [HEW 79a] and guardians provide *atomic objects* [LIS 83] so that all but one thread of control can be suspended in these models to allow safe access to shared data.

[1]Because of their diverse origins, the facilities provided by object-based languages vary considerably. Wolf [WOL 89] has made a comparison of two rather different object-based languages, C++ [STR 86] and Flavors [MOO 84]. Liskov [LIS 88b] has made a similar comparison of Smalltalk [GOL 83] and Clu [LIS 77].

Of existing object-based languages, the closest encapsulation model to the interrupt-generating active data object of this thesis is Hewitt's Actor [HEW 80] [AGH 86]. The Actor computational model is uniformly object-oriented, representing all computations in terms of objects and message passing. The actor model has evolved from research into parallel computation, especially that of massively parallel computer architectures. Actor languages support objects, abstraction, and concurrency, but not classes, inheritance, or strong typing. They have no explicit synchronization mechanism, lacking even the familiar blocking invocation (call) mechanism. The behaviour of an actor is defined by its actions in response to a communication. A pure actor can process a single communication before it *dies*. An actor may respond to a communication by sending messages, creating new actors, and, if necessary, creating a replacement actor with a specific replacement behaviour[1]. Actors are meant to model a primitive *natural concurrency* as one might expect to find in nature. This is reflected in the dynamic parallelism and limited life-span of the behaviour of an actor. The programming languages ACT 1 [LIE 87], ACT 3 [AGH 87], ACT++ [KAF 89], and Intermission [KAH 82] are based on the Actor model. ABCL/1 [YON 86] and Actra [BAR 87] are also derivatives of the Actor model, but, in contrast to pure Actors, they support more conventional, object-oriented programming constructs (such as method encapsulation, blocking invocations, and long lived processes).

The Actor computational model and the proposal of this thesis provide similar clear and simple encapsulation models for objects without classes. The serializer construct developed for Actors has addressed some of the modular arbitration issues addressed by this thesis[2]. Like Actors, interrupt-generating active data objects can be created and destroyed, but their life-span is determined by the lexical level of their definition in the source program rather than by the dynamic application-dependent evolution of the program, as is the case with Actors.

The Actor languages have some problems which do not exist in the proposal of this thesis.

---

[1] In its most recent form, the Actor persists and simply computes its replacement behaviour [AGH 89].

[2] This is discussed in section 2.2

The Actor model uses asynchronous queued messages, which rely on infinite buffering; secondly, due to the high degree of non-deterministic, dynamic parallelism and aliasing, it is difficult to control concurrency and reason about the behaviour and correctness of an Actor system. In contrast, the model proposed by this thesis uses synchronous messages with no queues, and depends on explicit parallelism, both of which can be more simply implemented and supported by theory.

Comparing Actors with the objects proposed in this thesis highlights an important difference between uniformly object-oriented programming and traditional multi-processing. An interrupt-generating active data object interacts with processes which are associated with a block of code, rather than with other similar objects, and so the proposal's interface is more akin to Smalltalk and C++ than to the Actor paradigm. A Smalltalk-80 process is associated with a block of code rather than an object. However, the execution of a Smalltalk-80 process is controlled by explicit messages (*resume* and *suspend*) rather than by being an automatic property of the object messaging mechanism. The active objects of this thesis fall somewhere between the Actor approach and the Smalltalk approach in this respect, by not only providing encapsulation (as in Smalltalk), but also providing a unified object activation and method invocation mechanism (as in Actor models).

The object model of this thesis has the advantage (not shared by any of the other object models) of constituting a rudimentary language structure, with built-in safety properties which need not be explicitly coded (or even necessarily thought of) by the programmer. The object needs no explicit definition or invocation, and its life-span is inferred from the lexical scope of its use in the source program. References to the object are mapped onto synchronous message exchanges. Nevertheless, by providing methods which receive and transmit values, and which send a further message when a predetermined condition arises, the basic behaviour of the object (but not necessarily its entire interface) could be coded in most object-based languages. This form of realizing the object is at a different conceptual level of programming to the approach taken by this thesis. The modelling of interrupt-generating active data objects in the current generation of object-based programming languages is something akin to the programming of structured loops

in FORTRAN 66[1]. Although it is generally possible to get the job done, the lack of linguistic support for the programming model requires additional effort of the application programmer to implement the desired behaviour, and places additional responsibility upon the application programmer to ensure that all possible safety properties have been included in the model. The result is generally an inferior product which is tedious to produce, difficult to maintain, and more complex, yet less reliable, than it need be.

Of more acute significance, the proposal of this thesis includes the specification its object's interaction with sequential processes which are able to acknowledge an unanticipated interrupt signal from the data object as an alternative occurrence to each of its constituent events at the most primitive language level[2]. Current object-based languages require an unduly large (and ungainly) programming effort to anticipate a high level interrupt communication explicitly at each point at which one should be trapped, or else require special modifications to the kernel of their operating environments to facilitate this form of programming. The following paragraphs indicate the extent to which unexpected events have been addressed in the current generation of object-based languages.

In the Smalltalk-80 interactive programming environment [GOL 84] exceptions have the status of error handling facilities. Should a program executing under the Smalltalk-80 environment produce one of the run-time error conditions which Smalltalk-80 is able to trap (such as a divide by zero operation), the system displays a notifier with override, debug, and abort options. The error handling system also provides protocols to enable a program developer to anticipate a possible exception condition and specify a suitable error handler to take the place of the default error notifier [CUN 88]. No facilities exist for programmer defined exceptions to be trapped at the

---

[1]FORTRAN 66 [ANS 66] had only two constructors for programming indefinite control loops, an unconditional *GOTO* statement and a simple *IF* statement (little more than a conditional *GOTO*).

[2]It should be noted that, within this context, an unanticipated signal is not unexpected. In the model presented later in this thesis, a process which receives an interrupt signal from an interrupt-generating active data object is required to set an interrupt condition for that object; therefore the interrupt is expected, but does not have to be explicitly anticipated at key points in the source program.

system level, and, although an object may be provided with a method which deliberately causes a system level exception, the detection of the exception at a random position in a large piece of code would require considerable programming effort because an explicit program structure is required to trap the exception at each point at which it might arise. Malcolm [MAL 88] describes a system called *Signals* that has been used to improve exception handling in Smalltalk-80 applications, but which nevertheless requires an explicit program structure to test for the possible presence of an exception at each point at which an exception might arise[1].

The facilities of C++ [STR 86] are similarly restricted in this respect. In their description of the use of the object-oriented facilities and class hierarchies of C++ to design the *Choices* multiprocessor operating system, Russo *et al* [RUS 88] report that exceptions must be implemented as primary objects and that interrupt exceptions must be awaited (if they are not to be missed) by the definition of an *await* method for each exception. Once a process invokes this method it is blocked until such time as the exception actually occurs[2].

The expressiveness of the object-based language Trellis/Owl [SCH 86] is significantly enhanced by the provision of linguistic support which allows the definition of a method to include a list of exceptions which the operation can signal. The syntax is very similar to the specification of Ada exceptions, and Trellis/Owl exceptions retain the status of error-reporting. They also have no inherent communication ability, and must be explicitly anticipated by the code which initiated the operation.

---

[1]Smalltalk has been criticized generally for inadequately addressing concurrency within the object-oriented paradigm, resulting in inadequate support for monitoring simulations and real-time systems (*cf.* Thomas's comments in the report by Power [POW 88]). The general concurrency deficiencies have been addressed to a limited extent in ConcurrentSmalltalk [YOK 86]. A distributed implementation of Smalltalk has also been attempted [BEN 87], but this implementation still requires the explicit anticipation of incoming communications, and identifies further semantic aspects of Smalltalk (in particular the mechanisms of class inheritance) which inhibit the building of distributed systems. More recently, the programming language CST [HOR 89] has extended ConcurrentSmalltalk to allow the programmer to describe distributed objects.

[2]Un-awaited interrupts do exist in *Choices*, but are outside the control of application processes.

Recent research by Thomas [THO 89] has produced some preliminary results which show that implementing objects as active processes on transputers is an approach which holds promise. Thomas has investigated the fusion of objects and Occam [MAY 87] processes in an implementation which includes full class inheritance for programmer defined objects. The implementation provides its extra-Occam capabilities in a run-time system called the object manager. The objects in Thomas's proposal suffer from the same drawbacks as those of other object-based languages for implementing interrupt-generating active data objects, and currently the object manager for each class object is mapped rather crudely onto existing Occam facilities[1]. However, Thomas's early results do confirm that the approach of implementing objects as active processes is worth pursuing.

2.4 Connectionist models

One of the more flourishing application areas of encapsulated processes with threshold methods has been the realization of artificially intelligent systems. This area provided the earliest influences for this style of programming in the form of demon and neuron models. Both models have now been absorbed into the prolific field of research called connectionist (neural) networks [DUR 89].

A demon is a separate, autonomous process which runs in parallel with other processes (demons) and may interact with them. It can be described simply as a predicate and a body. Whenever the predicate evaluates to true, the body is executed. The idea was introduced by Selfridge [SEL 59] in the late 1950's in a model called Pandemonium. Pandemonium was a model designed to perform automatic recognition of hand-sent Morse code by means of a large number of independent pattern-action modules called demons. The demons were essentially detectors for particular properties of the input, and the more evidence the demon accumulated for the unit it represented, the louder it *shouted* to other demons. Through the ensuing *pandemonium*, some demons became more strongly activated than others; at the top level of the system, a *decision demon* chose the most strongly activated demon as the final result.

[1]Thomas's research is still in its early stages, and more ambitious implementations are planned.

The term *demon* has caught on as a generic term[1] to describe a piece of program code which acts as a sentinel, watching for a specific situation to arise during the execution of a program and performing some action when it does. This type of demon is mainly used in artificial intelligence systems as a procedural attachment, usually just a piece of LISP code [WIN 79]. For example, the BORIS/DYPAR expectation-driven narrative understanding systems of Dyer [DYE 83] use this type of demon, and Hewitt's programming language PLANNER[2] [HEW 69] allows the programmer to define a set of procedures or demons which are executed when the pattern they represent matches the argument of a database operation. The term *demon* has also been used outside of the domain of artificial intelligence systems to refer to a background process which exists throughout the lifetime (or some portion of the lifetime) of a system, and which is automatically activated when needed. For example, the UNIX operating system makes provision for such processes, which are termed *daemon*[3] (*sic*) processes [BAC 86]. Daemon processes are distinct from user and kernel processes, and perform system wide functions (such as printer spooling).

In contrast to the collective representation of knowledge by the group of interconnected processes in Selfridge's original demon model, the general demon tends to be a solitary process representing an occasionally invoked function. In addition, the general demon process is an implementation level removed from Selfridge's simple processing device, and even further removed from the elemental interrupt-generating active data object described in this thesis. In contrast to the

---

[1] *Demon* has, in fact, been used as a generic term for over a century in the scientific community. It was used as early as 1871 by the Scottish physicist James Maxwell in his *Theory of Heat*, to describe a creature which, if small enough, might be exempt from the second law of thermodynamics, thereby possessing far reaching subversive effects on the natural order of things.

[2] PLANNER is a LISP-based programming language for inference control. It was designed in 1969 at the MIT AI Laboratory by Hewitt, and later extensively demonstrated in Winograd's SHRDLU project [HEW 72]. Only a subset of PLANNER, MICRO-PLANNER was ever implemented, but it provided a conceptual stepping stone to the Actor model of computing [GRE 75] (*cf.* section 2.3).

[3] This presumably follows the original Latin spelling.

standard operation and strict lexical scope of an interrupt-generating active data object, the influence of the general demon process is not normally restricted to a small portion of the program, and it performs whatever application the software designer wishes to specify.

One of the central motivations for the Pandemonium model was to distribute an information processing task to a large number of simple interconnected parallel processes, an approach which lives on today in a number of guises[1]. Recently there has been a strong resurgence of interest in the idea of distributing processing to large numbers of simple processing units of highly restricted complexity. These models are generally called connectionist [FEL 82] [RUM 86], and fall into the broader class of parallel distributed processing models. The basic computing elements in the connectionist class of models are called *cells* (or neurons). Individual computational cells are very simple processes with sophisticated interconnection schemes. Generally, each of these cells takes on an activation that is some monotonic function of its net input from other units, and sends out an output signal based on its activation. The input tokens are weighted by values associated with the interconnections[2], which bias the network towards an overall state or set of states.

The ideas of connectionism are older than Selfridge's model, dating back to the early work (*circa* 1943) of McCulloch and Pitts on neuron models [MCC 43] and the perceptron (in the early 1960's[3]) of Rosenblatt [ROS 62]. These early computational models of neural networks used computing elements that were simple threshold logic units. Rosenblatt's work consisted primarily of showing that systems which were composed of multiple layers of his cells exhibited learning behaviour in simple pattern association tasks. Unfortunately the behaviour of these networks did

---

[1] A modern computing model which is quite strongly related to Pandemonium is the Actor model, discussed in section 2.3, in which separate, autonomous Actors communicate via messages of arbitrary complexity and carry out arbitrary computations on these messages.

[2] The weights may be excitatory (positive) or inhibitory (negative).

[3] The groundwork of perceptron theory was laid in 1957 [ROS 57]. A number of variations on the original model were subsequently studied, culminating in a widely published book by Rosenblatt in 1962 [ROS 62].

not scale to interesting tasks, and Minsky and Papert [MIN 69] later showed that the perceptron convergence procedure of Rosenblat [ROS 62] was inadequate for networks containing more than one layer of modifiable connections, and that single layer networks were unable to handle some essential operations, such as the *exclusive-OR* operation needed for the superimposition of input sets. The perceptron turned out to be the linear case of a more general process.

The original neuron models were improved by a number of researchers as more appropriate parallel computing machines became available, and as more details of the cerebral structure were discovered. A notable contribution was made by Albus [ALB 71], who developed a modified model of Rosenblat's perceptron which he called a CMAC (Cerebellar[1] Model Automated Controller). Albus's model had a vastly reduced storage requirement compared to the classical perceptron, and at the same time overcame some of the original perceptron's limitations. The elements of modern connectionism have become more complex than the original models, providing multistate, continuous activation level cells. These units are similar in behaviour to the interrupt-generating active data objects proposed in this thesis in the sense that each device has a binary output which goes high if a particular function of the inputs corresponds to a threshold value, but, as is shown below, several other features make the two models incompatible.

Connectionist networks are *trained* rather than programmed. Virtually all learning rules for these networks can be considered variants of the Hebbian learning rule [HEB 49]. The simplest *learning* mechanisms merely preset the weights of the network to *remember* a number of input patterns. Several more sophisticated variants of the Hebbian learning procedure have been developed [RUM 86] [GRO 88] [HOP 89].

It is significant to note that vision was the area in which connectionism made its initial appearance. Low level vision computations appear to be particularly well suited to connectionist computations because they require the integration of large numbers of simple, interacting pieces of knowledge. Although connectionist models are now developed for a number of different purposes, their primary function is still centered about the representation of *IS.A* and *PART.OF*

[1]The cerebellum is involved in the control of movements of limbs, eyes, and hands.

relationships, the basic operations required for visual identification systems. Expressed more formally, they provide attractive approaches to solving problems that can be described as relaxation or constraint satisfaction searches [GEM 84]. While vision algorithms are particularly appropriate for this form of computation (often dedicating one processor to each pixel in the image), other applications are primarily artificially intelligent in nature, for example symbolic sorting, unification, and retrieval from semantic networks. Connectionist models also enjoy popularity for applications which seek to capture in explicit form the computational properties of real (animal) neural nets.

The best known commercial example of a connectionist system is Hillis's connection machine [HIL 85], which reflects the knowledge representation used by the applications which execute on it. It provides tens of thousands to millions of single-bit processors (each based on a cellular automaton [HIL 84]). The connection machine is a SIMD[1] machine, meaning that each operation applies to all processing cells in parallel[2]. However, each process will conditionally execute an instruction depending on the internal state of one of its flags. Programs for the connection machine are written in extensions of the conventional languages LISP or C, called CM LISP [STE

[1]Single-instruction stream, multiple-data stream.

[2]Each processor is too rudimentary to fetch its own instruction stream. Instead, all processors receive the same instruction stream, which is generated by a conventional serial *host* machine. In the 65536 processor connection machine prototype (manufactured by Thinking Machines, Inc. of Cambridge, Massachusetts), each single-bit processor has 4096 bits of memory. Each processor is connected directly to sixteen other processors in a *Boolean 16 cube*. However, every processor can communicate with every other processor through routers (the communication network), with a delay of only a few dozen machine cycles. Each router is hardware responsible for receiving messages from a processor or another router and forwarding the message to the destination processor or to another router closer to the destination processor. Thus, a program may establish a logical interconnection among processors which matches the way the elements of the application program are connected. A 1-bit-wide ALU is capable of performing any computation that can be done by wider ALUs, albeit more slowly. Arithmetic operations that can be performed in a single clock cycle on conventional processors take time proportional to the length of the arguments on a connection machine. For example, to add two 16-bit numbers in every processor of a connection machine would require 16 clock cycles. For this reason, connection machines are not particularly distinguished as *number crunchers*.

86] and C* [1] respectively. Any task which cannot be done in parallel is executed on the host machine. The effectiveness of the connection machine lies in its ability to represent each application problem cell by a single-bit processor, and to logically connect each processor to other processors in a manner matching the application topology[2].

Beynon and Dodd [BEY 88] have described the targeting of general connectionist networks, such as those described by Rumelhart and McClennan [RUM 86], onto the same class of execution environments as is targeted by this thesis. They have considered a small number of cells (only 16 to date) mapped onto an equal number of Transputers; few weights can be adjusted in parallel[3]. On the other hand, Hillis's massively parallel connection machine can adjust many tens of thousands of weights simultaneously by means of dedicated microchips.

The way in which knowledge is represented in a connectionist system divides current connectionist research into two distinct approaches. The *distributionist* approach suggests that concepts or hypotheses should be represented by patterns of activations over large numbers of cells [HIN 85]. In contrast, the *localist* approach advocates the use of a unique cell to represent every item of knowledge [FEL 82]. Of the two approaches, the cells within the localist category have more in common with the model proposed in this thesis, and will be the subclass referred to for comparison purposes in the remainder of this discussion.

Since the connectionist model is applied almost exclusively to artificial intelligence, a single localist cell represents a piece of knowledge, rather than an item of data as represented by the interrupt-generating active data object of this thesis. Cells maintain an internal activation level, which is

---

[1] C* is based upon the data parallel style of programming that maps every data element to a virtual processor [HIL 86].

[2] Some other variants of the connection machine approach are described by Fahlman *et al* [FAH 83], and recent connectionist networks and their implementations are surveyed by Schoonees [SCH 88].

[3] Less general implementations of connectionist networks on transputers have been attempted by Ellison *et al* [ELL 88a] and Johannet *et al* [JOH 88].

restricted to a small finite subset of real values, and communicate with each other using a finite set of messages which is a function of the activation level. A simple cycle consists of accepting input messages, updating the activation level based on a simple function[1] of the input values, and generating output messages. Typically, several machine cycles are required for cells to reach an equilibrium over the input.

When a correctly configured network is provided with an input set, the cells spend some time exchanging messages and updating their activation levels. Eventually the network settles into a stable state. In this state, the only cells which have a high activation level are those that constitute the object that has been identified. All the other cells are at a low activation level. The cells that are active form a mutual support group and keep each other active.

In contrast, the interrupt-generating active data object represents an item of data which has no implicit affiliation to any other item of data, and whose status is calculated definitively during a single reference to the object and is independent of the status of any other object.

The two models also differ significantly in their computational relationships and interconnection structures. Cells in a localist network are connected in a hierarchical relationship with each other, whereas interrupt-generating active data objects are connected in a linear relationship with sequential processes. Connectionist networks fall into the SIMD computational model (and non-parallel imperative activities need to be performed on a sequential host machine), whereas interrupt-generating active data objects support a MIMD[2] model. As a consequence, the connectionist cells support data parallelism, whereas the model of this thesis supports partitioned parallel algorithms.

Unlike the connectionist approach, the proposal of this thesis forms a rudimentary programming

---

[1]The complexity of a computation is strictly limited to simple operations such as summation, multiplication, and thresholding.

[2]Multiple-instruction stream, multiple-data stream.

language structure. Connectionist cells are algorithmic models which, when programmed in an application language, suffer the same drawbacks as the object-based implementations described in section 2.3 for the interrupt-generating active data object. Connectionist networks in which the entire operation of a simplified and standardized cell is implemented in hardware are exceptions. One version of this type of implementation is the massively parallel Boltzmann architecture [ACK 85], which uses simple *on-off* processing units and stores all its long-term knowledge in the strengths of the connections between processors.

Both the connectionist approach and the approach of this thesis are supported in their empirical development by mathematical theory. However, the networks of connectionism are often non-linear systems for which traditional convergence proofs are inadequate in supporting the experimental observations [FEL 82], whereas all aspects of the behaviour of the interrupt-generating active data object are underpinned by the well established theory of communicating sequential processes [HOA 85].

From the above comparison it can be seen that, despite the common denominator of having a linear threshold mechanism as a primary feature, the localist cell and the interrupt-generating active data object were developed for very different application areas and neither is well suited for modelling the application area of the other. Each has engineered the data encapsulation and thresholding concepts to solve a very different problem, each with its own performance, resource management, and expressibility trade-offs.

The discussion so far has highlighted only superficial similarities and differences between the objects proposed by this thesis and the connectionist model. For a more substantial appreciation of the differences between the two computational models, the following paragraphs give a formal description of an interrupt-generating active data object in terms of the features which it has in common with the connectionist model.

Localist cells are formally defined in terms of a 7-tuple [FEL 82]

$$(s,\ i,\ o,\ a,\ f,\ g,\ h)$$

where s is a small set of possible states (typically 2 to 5),

i is a small set of input tokens (e.g. the numbers 1 to 10),

o is a small set of output values similar to the input tokens,

a is a small range of real values (for example [-1.0..1.0]) that constitutes the activation level of the cell,

$f(s, i, a) \rightarrow a$ is the next activation level function,

$g(s, i, a) \rightarrow s$ is the next state function, and

$h(s, i, a) \rightarrow o$ is the next output function, a simple real to integer conversion.

Although a theoretically has an infinite number of possible levels, once implemented on a digital machine a cell becomes a finite state automaton [HIL 84].

A link between two cells is defined by a triple [FEL 82]

$$(S, D, W)$$

where S is the source cell of the connection,

D is the destination cell of the connection, and

W is a number from a small range of real values, called the *weight* of the link.

A link works by taking the output of the source cell, multiplying it by the weight and delivering it as input to the destination cell.

Ignoring for the moment the data storage, non-deterministic access, and mutual exclusion needs of the interrupt-generating active data object, a simplified form of the proposal can be defined, using the connectionist mould, in terms of a 5-tuple

$$(s, i, o, g, h)$$

where s is the set of possible states corresponding to the range of potential interrupt condition values,

i is a set of input tokens for which it is possible to distinguish between a normal

input value and a *set* value which alters the cell state,

o is a small set of output values corresponding to all possible interrupt destinations in the general case[1],

$g(s, i) \rightarrow s$ is the next state function, for which the input token will be a *set* value, and

$h(s, i) \rightarrow o$ is the next output function, which is a simple logical operation.

When compared to the original 7-tuple, a simplification is evident in the omission of an activation level and the set of operations required to calculate it (a and f in the original 7-tuple). This means that it is impossible to perform a computation for which it is necessary to integrate the input over a number of machine cycles, making this simplified structure unattractive for modelling real neural networks, such as those which will cope with noise in the input.

A link between the simplified interrupt-generating active data object and its owner process is defined by a simple pair

$$(S, D)$$

where S and D represent the source and destination as before. The absence of a weight W precludes the usual means of training a connectionist network to recognize an object by specifying the relative strengths of interconnections between cells. The interrupt-generating active data object *learns* by receiving an input which it is able to distinguish as a signal intended to *set* its state. This is an added complexity in the definition of the set of input tokens compared to the original 7-tuple.

While demonstrating that a simple binary interrupt-generating active data object would be able to simulate some of the activity of a connectionist cell, this comparison highlights the deficiencies in the connectionist model for dealing with concurrent references to data objects. The

[1]Just as a strict hierarchical structure in a connection machine will restrict the number of output values for each cell to one, so the ownership rule introduced for interrupt-generating objects will restrict this set to one element.

connectionist-like 5-tuple permits a computation to make simultaneous use of a number of input tokens, a precept which is directly at odds with the guarantee of the interrupt-generating active data object that input tokens which arrive simultaneously will be handled mutually exclusively. Furthermore, the 5-tuple does not make any provision for the large set of possible states required to represent any possible value of the data item which the data object will represent (in addition to the set of interrupt condition values represented by *s* in the 5-tuple), it does not cater for input tokens which must be interpreted as requests to read the stored value (a form of output which should not be confused with *o* in the 5-tuple above), and, more importantly, it is unable to cope with the non-deterministic arrival of input tokens. This thesis shows that it is possible to formulate an uncomplicated imperative programming model which takes all of these factors into account.

2.5 Process decoupling, shared information, and specific application topics

Like most of the object-based languages and connectionist networks discussed in previous sections, the programming model proposed by this thesis binds concurrent processes (objects and sequential processes) tightly together, either by explicit or implicit language constructs, or, as in the case of the connection machine, implicitly through the intermediation of a parallel data structure. In the proposal of this thesis, communication between a process and an interrupt-generating active data object is implicitly symmetrical. Were the communication to be made asymmetrical, there would be a degree of spatial decoupling between communicating processes in that a called object would not know the identity of the caller. Ada and *MOD are examples of languages which make use of asymmetrical communication. The implementation of interrupt-generating active data objects in such languages would enjoy the linguistic advantage that the shared data object would not need to name each process (or port through which each process communicates) which makes a reference to it. However, such a modification would be at the expense of placing yet another responsibility upon an application programmer, this time the burden of ensuring that the lexical scope rules governing access to the interrupt-generating active data object are adhered to. The implementation effort would also be increased since it is generally accepted [ROB 81] [COO 80] that one-way naming complicates task identification and termination.

The proposal of this thesis uses synchronous (unbuffered) message passing in common with communicating sequential processes (CSP) [HOA 78], Occam [MAY 87], distributed processes (DP) [BRI 78], and Ada [ADA 83]. Buffered messages allow a degree of temporal decoupling, and have been used as the process interaction mechanism in Actors [AGH 86], Gypsy [GOO 79], PLITS [FEL 79], and Argus [LIS 83]. The temporal decoupling of the objects of this thesis from their client processes would be most undesirable since all inherent arbitration and selection facilities would be lost.

An approach which decouples parallel processes entirely (both spatially and temporally), called *tuple space communication* (alternatively *generative communication*), has been proposed by Gelernter as a communication model for Linda [GEL 85]. The tuple space model contrasts sharply with tightly bound process models in that processes aspire to know as little about each other as possible; for this reason, the tuple space environment is sometimes called an *un-connection* machine. Tuple space is a transport layer which provides a synthetic shared memory facility as the only communication mechanism. Processes or objects never interact with each other directly, they deal only with tuple space. The tuple space supplies infinite buffering, and, in contrast to the proposal of this thesis, provides a transparent operating system level rather than a language facility. Linda consists of a set of operators (calls to the tuple space) which are designed to turn any host language into a parallel programming language, and which are something akin to the standard environment libraries found in many existing language implementations. The tuple space approach has aroused much interest among the programmers of distributed memory multicomputers because it affords implicit communication, process mapping, and load balancing, although explicit algorithm partitioning is still a necessity[1]. In contrast, the proposal of this thesis retains the synchronization advantages of temporal coupling, with their associated load balancing and process mapping difficulties.

[1]Tuple space communication lends itself to the spatial decomposition of tasks, in which each processing node is able to solve the entire problem and applies its capacity to that portion of the problem which is within its spatial neighbourhood.

Very recently, details of the experimental language PARLE[1] have been published [EBE 89], which, like the experimental programming language (HUL) described in this thesis, provides two primitive communication mechanisms for distributed memory multiprocessors: A synchronized mechanism through message passing, and a system level shared memory mechanism. Like HUL, PARLE attempts to design out some of the limitations of static languages such as Occam without burdening the implementation with unacceptable inefficiencies. However, other design objectives of PARLE are rather different from those of this thesis. PARLE has been designed as a target machine language for a broad range of high level languages, including a parallel logic language, a parallel functional language, and a parallel object-oriented language [REF 89], but is also intended for use as a parallel systems programming language. The low level generality of PARLE means that it lacks the linguistic support for application programming which HUL possesses, and the flexibility of its shared data mechanism leads to compromises in the safety of programs when compared with static languages like Occam or the structured interrupt-generating active data object of HUL. The shared data mechanism in PARLE possess no form of synchronization or other safety properties, whereas condition synchronization and safe access are an integral feature of the design of the interrupt-generating active data object.

It should be noted that while both HUL and PARLE are distributed programming languages, their intended areas of usage are quite different. PARLE is a systems programming language while HUL is an application programming language. As a result, the abstractions supported by the two languages are quite different.

Having covered both monitors and one-way naming in this chapter, it is possible to present a further disadvantage of interrupt-generating active data object implementation in a language such as Ada. This lies in the way in which requests for access to a common facility are queued. At least three entry points would be needed to service calls for assigning data, fetching data, and setting an interrupt condition. In Ada, waiting calls for each entry name in a task are placed on different queues [WEG 83] (whereas waiting procedure calls on a conventional monitor are all placed on the same queue, and the selection of synchronous communications in the interrupt-

[1]The Parallel Architectures Research Language for the Esprit-1588 project [EBE 89] [REF 89].

generating active data object of this thesis is non-deterministic). Since an Ada entry call can execute only if its task has reached the control point where the entry is expected [GEH 84b], it is likely that such an implementation will be biased in favour of servicing one of the entry points.

An application in the field of production management systems which has made novel use of the concept of active, distributed, object-based communications is the HUTDPMS[1] project [HÄM 87], which uses a high level generic object known as an *InFo* (Intelligent Form). This project is aimed at bringing the advantages of distributed active objects to the end user level through the development of a form-based applications generator to create *InFos* which communicate using standard electronic mail mechanisms [ELO 85a]. Because of its specific end user predilection, the HUTDPMS applications generator is more comfortably classed along with production tools such as Lotus 1-2-3™ than as a programming language. The *InFo* model itself is based upon an earlier active form concept for end users [TSI 79], into which *intelligence* is introduced by means of a simple grammar which allows the user to specify basic arithmetic, relational, and routing operations. In addition to providing inherent facilities for maintaining data, logic, and routing information, an *InFo* provides a triggering predicate, which functions very differently to the signalling mechanism of the interrupt-generating active data model. An *InFo* trigger is used to initialize one or more (hitherto non-executing) *InFos* [ELO 85b], thereby enabling organizational parallelism. In contrast to the static relationships which characterize the life cycle of an interrupt-generating active data object, the *InFo* model operates by means of dynamic binding. Because of this, it is unable to offer any form of protection to applications which make use of it, or provide any guarantees which ensure that the implementation meets its specification.

2.6 Overview

This chapter has discussed a broad range of programming approaches and execution environments to show that existing techniques do not adequately address the aims of this thesis. None of the connectionist, object-based, or imperative exception handling approaches is able to provide an integrated encapsulation facility which can interact with a clean process model for unanticipated

[1]Helsinki University of Technology Distributed Production Management System.

communications. Moreover, none of the existing approaches is able to support the proposal as an elemental language construct.

In handling shared data, the interrupt-generating active data object of this thesis is a process's aide-de-camp; it does not get in the way, requires no briefing of its role, is accomplished at coping with any injudicious reference to it, and gives a polite tap on the shoulder when a pressing situation arises.

3. Formal models and safety properties

3.1 Introduction

There is an increasing need for the development of language constructs that are at least partially chosen for their provability. A language feature providing synchronization should be designed to provide usable axioms about the possible orders of events in a program. The language feature should guarantee that conditions needed to prove properties of programs are explicit in the axioms for the language feature.

In deriving the behaviour of the interrupt-generating active data object, it was the intention from the outset to retain the advantages of existing theoretical advances by not deviating in any fundamental way from proven notations that express concurrent computations simply, make synchronization requirements explicit, and facilitate formal correctness proofs. This thesis uses the established theoretical foundation of CSP [HOA 85] to formulate an interrupt driven computational model which exhibits reasonable safety properties. As important as the behaviour of the interrupt-generating object is the correct behaviour of processes which interact with the object.

The sections which follow present formal specifications for an interrupt-generating active data object, and show the interaction of this formal model with concurrent processes, using the mathematical approach of CSP.

3.2 A safe interrupt-generating active data object

In the paragraphs which follow, a model for the behaviour of a process which emulates an enhanced form of data object is developed from a simple definition which caters for communication with a single process in an application program, to a model capable of servicing references from n concurrently executing application processes. The formal notation is used to illustrate the need to introduce a safety requirement which restricts each memory emulation

process to servicing only one interrupt condition.

Before introducing the interrupt-generating model, the formal notation is illustrated by two conventional high level language data types which are defined in the form of processes.

A Boolean variable can be described using two processes, *TRUE* and *FALSE*, to represent its behaviour in each of its two states. These processes may engage in any of the events specified by their event alphabets:

$$\alpha\ TRUE = \{read, fetch_1, assign_0, assign_1\}$$
$$\alpha\ FALSE = \{read, fetch_0, assign_0, assign_1\}$$
$$\alpha\ BOOLEAN = \alpha\ TRUE \cup \alpha\ FALSE$$

The behaviour of the Boolean variable can be specified as a choice of events from the alphabets of processes *TRUE* and *FALSE*. The events which assign a value to the variable determine which state the variable will assume, and consequently which of the processes will describe the ensuing behaviour of the variable.

$$\begin{aligned}
BOOLEAN &= (assign_0 \rightarrow FALSE \ [] \ assign_1 \rightarrow TRUE) \\
FALSE &= (read \rightarrow fetch_0 \rightarrow FALSE \\
&\quad [] \ assign_0 \rightarrow FALSE \\
&\quad [] \ assign_1 \rightarrow TRUE) \\
TRUE &= (read \rightarrow fetch_1 \rightarrow TRUE \\
&\quad [] \ assign_1 \rightarrow TRUE \\
&\quad [] \ assign_0 \rightarrow FALSE)
\end{aligned}$$

Note that this Boolean variable refuses to allow its value to be fetched until after a value has first been assigned to it.

As a second example, consider how this definition can be extended to cater for a range of values, such as are needed to describe an integer variable capable of storing values in the range $-n$ to $+n$. The behaviour of the variable in each of its k states can be defined by a process VAL_k.

As in the case of the Boolean variable, the integer variable's behaviour can be specified as a choice of events from the alphabets of the VAL_k processes.

$$\alpha\ INTEGER = \cup\ \alpha\ VAL_k \qquad for\ -n \le k \le n$$
$$\alpha\ VAL_k = \{read\ ,\ fetch_k\ ,\ assign_i\} \qquad for\ -n \le i \le n$$

$$INTEGER = ((i:\ -n..n)assign_i \rightarrow VAL_i\ )$$
$$VAL_k = ((i:\ -n..n)assign_i \rightarrow VAL_i \qquad for\ -n \le k \le n$$
$$[]\ \ read \rightarrow fetch_k \rightarrow VAL_k\ )$$

If the current value stored in the integer variable is not of interest, the alphabet of events can be simplified to ignore the distinction between states which the variable may assume, and to consider only the actions of making references to the variable. A simplified process which is only concerned with the actions of assigning and fetching values, without regard to the magnitude of values, may be depicted as:

$$\alpha\ INTEGER = \{assign,\ fetch\}$$
$$INTEGER = (assign \rightarrow \mu\ X.(assign \rightarrow X\ \ []\ \ fetch \rightarrow X))$$

This definition consists of a single process which, once initiated with the appropriate event prefix, is able to reference an elemental nested process definition recursively to describe all possible traces of events that the variable may engage in.

Using a local variable m to represent the variable's *memory* register, this specification can be written in the form of a process which communicates with the rest of the system by passing messages[1]. The events which cause a value to be transferred to or from the variable are equivalent to synchronous communication primitives.

[1]Note that output guards have been allowed in the CSP notation to simplify the specifications. Justification for this form of symmetric communication is based on the work of Wrench [WRE 86] [WRE 88], who showed that it was feasible to implement such a construct.

$$\alpha\ INTEGER = \{assign.k\ ,\ fetch.k\}$$
$$INTEGER = (assign\ ?\ m \rightarrow \mu\ X.(assign\ ?\ m \rightarrow X \ \square\ fetch\ !\ m \rightarrow X))$$

An interrupt-generating process may be superimposed upon this simple variable definition to describe the behaviour of the interrupt-generating active data object of section 1.3. The process which emulates such a variable and which is referenced by one application process can be expressed using the alphabet of events:

$$\alpha\ VARIABLE = \{assign.k,\ fetch.k,\ set.k,\ interrupt.k\}$$

Using the local variables t and m to represent the *test* and *memory* registers of the object, and the constant *any* to represent an arbitrary value, the behaviour of the process to emulate an interrupt-generating active data object can be specified as:

$$VARIABLE = (set\ ?\ t \rightarrow VAR)$$

where $VAR =$ $(assign\ ?\ m \rightarrow TEST \ \square\ fetch\ !\ m \rightarrow VAR \ \square\ set\ ?\ t \rightarrow VAR)$

and $TEST =$ **if** $m = t$ **then** $\mu\ X.(interrupt\ !\ any \rightarrow VAR \ \square\ fetch\ !\ m \rightarrow X)$ **else** VAR

(3-2-1)

This model ensures that the variable's *test* register is set before the *memory* register can be used. Once the *memory* register of the variable has been assigned a value which equals the *test* register, the process referencing the variable may not assign another value to the variable without first accepting an interrupt signal. The set of legal traces for specification 3-2-1 excludes this possibility. On the other hand, a process may fetch the current value of the variable any number of times between assigning a value and accepting an interrupt signal, or between two successive assignments.

In communicating systems in which processes may potentially be distributed across a network, it is important to ensure that a message has been correctly received before its contents are used. In the case of the *receive-and-test* operation which results from the assignment event in specification 3-2-1, integrity can be ensured by adhering to the law [ROS 85]:

$$(assign\ ?\ m \ \wedge\ m = t \ \rightarrow\ P) \quad = \quad \textbf{if}\ m = t\ \textbf{then}\ \mu X.(assign\ ?\ m \ \rightarrow\ P)$$

The point is illustrated by the following incorrect specification, which is not equivalent to specification 3-2-1.

$$VARIABLE \ = (set\ ?\ t \ \rightarrow\ VAR)$$

where VAR =
$$\begin{aligned}
&((assign\ ?\ m \ \wedge\ m = t \ \rightarrow\ \mu X.(interrupt\ !\ any \ \rightarrow\ VAR \\
&\qquad\qquad [] \ fetch\ !\ m \ \rightarrow\ X) \\
&\quad [] \ assign\ ?\ m \ \wedge\ m \neq t \ \rightarrow\ VAR) \\
&[] \ fetch\ !\ m \ \rightarrow\ VAR \\
&[] \ set\ ?\ t \ \rightarrow\ VAR)
\end{aligned}$$

To illustrate the interaction between a process which emulates an interrupt-generating variable, and a number of concurrent application processes which communicate with it simultaneously, it is expedient to limit the number of application processes to two for simplicity, then to extend the specification to n processes.

The following specification for an interrupt-generating variable permits simultaneous access by two application processes, *P1* and *P2*.

$$\begin{aligned}
\alpha\ VARIABLE \ = \quad &\{assign_1k,\ assign_2k,\ fetch_1k,\ fetch_2k,\ set_1k,\ set_2k, \\
&interrupt_1k\ ,\ interrupt_2k\}
\end{aligned}$$

$$VARIABLE = (set_1 ? t \rightarrow VAR_1 \; [] \; set_2 ? t \rightarrow VAR_2)$$

where $VAR_1 = (assign_1 ? m \rightarrow TEST_1 \; [] \; assign_2 ? m \rightarrow TEST_1 \; [] \; fetch_1 ! m \rightarrow VAR_1 \; [] \; fetch_2 ! m \rightarrow VAR_1 \; [] \; set_1 ? t \rightarrow VAR_1)$

and $TEST_1 =$ **if** $m = t$ **then** $\mu X.(interrupt_1 ! any \rightarrow VAR_1 \; [] \; fetch_1 ! m \rightarrow X \; [] \; fetch_2 ! m \rightarrow X)$ **else** VAR_1

and VAR_2 and $TEST_2$ are similar with all 1's replaced by 2's and vice versa.

(3-2-2)

This model allows either *P1* or *P2* to set the *test* register, thereby claiming ownership of the variable, whereafter only that process may be interrupted. The other process may alter or fetch the *memory* register value, but may not set the *test* register or respond to an interrupt signal. The other process is also prevented from altering the *memory* register between the signalling of an interrupt and the acceptance of this signal by the owner process.

The *n*-process model below retains the restriction of allowing only one application process to claim ownership of the interrupt-generating process by setting the *test* register, thereby causing any ensuing interrupt signals to be directed to that owner process.

$$\alpha\, VARIABLE = \bigcup \alpha\, VAR_i \qquad for \;\; 1 \leq i \leq n$$

$$\alpha\, VAR_i = \{assign_j.k,\; fetch_j.k,\; set_i.k,\; interrupt_i.k\} \qquad for \;\; 1 \leq j \leq n$$

$$VARIABLE = ((i: 1..n)set_i ? t \rightarrow VAR_i)$$

$$\text{where} \quad VAR_i = ((j: 1..n)assign_j\ ?\ m \rightarrow TEST_i \qquad for\ 1 \le i \le n$$
$$[]\ (j: 1..n)fetch_j\ !\ m \rightarrow VAR_i$$
$$[]\ set_i\ ?\ t \rightarrow VAR_i\ )$$

$$\text{and} \quad TEST_i = \textbf{if}\ m = t \qquad for\ 1 \le i \le n$$
$$\textbf{then}\ \ \mu\ X.(interrupt_i\ !\ any \rightarrow VAR_i$$
$$[]\ (j: 1..n)fetch_j\ !\ m \rightarrow X)$$
$$\textbf{else}\ \ VAR_i$$

(3-2-3)

To illustrate the need for exclusive ownership of an interrupt-generating process, the two-process example of specification 3-2-2 is reintroduced. It is possible to allow *P2* to be interrupted once *P1* has secured ownership of the interrupt-generating process as long as the two processes agree on the value to be placed in the *test* register. In the specification which follows, p:(1..2) is a cardinal value which represents the number of application processes which can be interrupted.

$$VARIABLE = p := 1 \rightarrow (set_1\ ?\ t \rightarrow VAR_1$$
$$[]\ set_2\ ?\ t \rightarrow VAR_2\ )$$

$$\text{where} \quad VAR_1 = (assign_1\ ?\ m \rightarrow TEST_1$$
$$[]\ assign_2\ ?\ m \rightarrow TEST_1$$
$$[]\ fetch_1\ !\ m \rightarrow VAR_1$$
$$[]\ fetch_2\ !\ m \rightarrow VAR_1$$
$$[]\ set_1\ ?\ t \rightarrow UNIQUE_1$$
$$[]\ set_2\ ?\ y \rightarrow IDENTICAL_1\ )$$

$$\text{and} \quad UNIQUE_1 = \textbf{if}\ p = 1\ \textbf{then}\ VAR_1$$
$$\textbf{else}\ STOP\ \ \{\text{cannot reassign interrupt value}\}$$

and $IDENTICAL_1$ = **if** $y = t$ **then** $p := 2 \rightarrow VAR_1$
else $STOP$ {P1 and P2 disagree on the value of t}

and $TEST_1$ = **if** $m = t$ **then** **if** $p = 2$ **then** $(SENDP_1 \parallel SENDP_2) \rightarrow VAR_1$
else $SENDP_1 \rightarrow VAR_1$
else VAR_1

and $SENDP_i$ = $(interrupt_i \,!\, any \rightarrow SKIP$ *for* $1 \leq i \leq 2$
$[]\ fetch_1 \,!\, m \rightarrow SENDP_i$
$[]\ fetch_2 \,!\, m \rightarrow SENDP_i)$

and VAR_2, $TEST_2$, $UNIQUE_2$ and $IDENTICAL_2$ are similar with all 1's replaced by 2's and vice versa.

If *P1* re-assigns the *test* register after *P2* has requested an interrupt by setting the *test* register, the process will *STOP* (this might take the form of reporting a run-time error and aborting) to avoid the possibility of an incorrect interrupt signal. A similar situation arises if *P2* attempts to set the *test* register to a value which is distinct from the one assigned to it by *P1*. These conditions represent conflicts which result in deadlock. Situations such as this cannot always be predicted by the application programmer, and it is not possible to exclude them from the design of a programming notation. However, the elimination of these sources of deadlock at the system level eradicates them from the higher levels as well. The price of facilitating the correct execution of high level languages, such as the one described in chapter 4, is the imposition of the exclusive ownership property of interrupt-generating processes.

To release an interrupt-generating process from the clutches of an application process which has claimed ownership of it, it is necessary to include a facility for resetting the interrupt-generating

process once it is no longer needed[1]. Specification 3-2-3 can be modified as follows to include this facility:

$$\alpha\, VARIABLE = \bigcup \alpha\, VAR_i \cup \{reset\} \quad for\ 1 \le i \le n$$
$$\alpha\, VAR_j = \{assign_j.k,\ fetch_j.k,\ set_i.k,\ interrupt_i.k,\ reset\} \quad for\ 1 \le j \le n$$

$$VARIABLE = ((i:\ 1..n)set_i\ ?\ t \rightarrow VAR_i \ \square\ reset \rightarrow VARIABLE)$$

where $VAR_i = ((j:\ 1..n)assign_j\ ?\ m \rightarrow TEST_i \ \square\ (j:\ 1..n)fetch_j\ !\ m \rightarrow VAR_i \ \square\ set_i\ ?\ t \rightarrow VAR_i \ \square\ reset \rightarrow VARIABLE)$ *for* $1 \le i \le n$

and $TEST_i =$ **if** $m = t$ **then** $\mu\, X.(interrupt_i\ !\ any \rightarrow VAR_i \ \square\ (j:\ 1..n)fetch_j\ !\ m \rightarrow X \ \square\ reset \rightarrow VARIABLE)$ **else** VAR_i *for* $1 \le i \le n$

(3-2-4)

As an alternative to the use of *reset* events, the interrupt-generating process might be defined to automatically reset itself after an interrupt. Specification 3-2-4 would begin to behave like process *VARIABLE* again after an interrupt has been signalled. It would still be necessary to provide for the receipt of *reset* messages to relieve deadlock situations. To match this model, an application process requiring periodic interrupts would be required to set the *test* register of its

[1] It is possible in practice for higher order interrupts to cause an application process to be abandoned before it is able to reset the interrupt-generating processes which it has reserved. The receipt of an abort message should therefore cause a reset sequence to be executed before the application process terminates.

interrupt-generating process again as part of its interrupt handler.

3.3 Proper interaction with sequential processes

A sequence of statements is an essential feature of all imperative programming languages. Such a sequence is often combined with a control structure to form a repetitive sequence, with the capability of terminating on the result of a conditional statement. As an example, the following specification for a *repeat-until* loop is given.

$$\begin{aligned}
&\alpha\ LOOP && = && \cup\ \alpha\ STATEMENT_i && \text{for } 1 \le i \le n \\
&\alpha\ STATEMENT_i && = && \{statement_i\} && \text{for } 1 \le i < n \\
&\alpha\ STATEMENT_n && = && \{statement_n\ ,\ testcondition\ ,\ true\ ,\ false\}
\end{aligned}$$

$$\begin{aligned}
&LOOP && = && STATEMENT_1 \\
&STATEMENT_i && = && (statement_i \rightarrow STATEMENT_{i+1}) && \text{for } 1 \le i < n \\
&STATEMENT_n && = && (statement_n \rightarrow testcondition \rightarrow (true \rightarrow SKIP \\
& && && \qquad [] \ false \rightarrow LOOP))
\end{aligned}$$

The *SKIP* process in this specification represents successful termination of $STATEMENT_n$, and consequently of the loop.

For a sequential process to interact with the interrupt-generating model of the previous section, it is necessary to allow an interrupt to occur as an alternative event to every statement process at the most primitive level. It is convenient to design the control structure in the form of an infinite loop. Successful termination of the loop is specified as an action which results from the receipt of an interrupt signal.

$$\begin{aligned}
&\alpha\ LOOP && = && \cup\ \alpha\ STATEMENT_i && \text{for } 1 \le i \le n \\
&\alpha\ STATEMENT_i && = && \{statement_i\ ,\ interrupt\ ,\ action\}
\end{aligned}$$

$$
\begin{aligned}
LOOP \quad &= \quad STATEMENT_1 \\
STATEMENT_i \quad &= \quad (statement_i \rightarrow STATEMENT_{i+1}) \qquad for \;\; 1 \leq i < n \\
&\qquad [] \;\; interrupt \rightarrow (action \rightarrow STATEMENT_i \\
&\qquad\qquad [] \;\; action \rightarrow SKIP)) \\
STATEMENT_n \quad &= \quad (statement_n \rightarrow STATEMENT_1 \\
&\qquad [] \;\; interrupt \rightarrow (action \rightarrow STATEMENT_n \\
&\qquad\qquad [] \;\; action \rightarrow SKIP))
\end{aligned}
$$

(3-3-1)

In this model, the action occurring as a result of an interrupt event may take the form of a subroutine, or a sequence of statements which result in the successful termination of the loop (the *SKIP* process in the specification represents successful termination). These two possibilities have been represented as distinct alternatives. Specification 3-3-1 excludes the evaluation of the test condition for determining the point of termination. Typically, this would be done by a second process executing in parallel, which would cause process *LOOP* to be interrupted. However, there is no reason why one of the statements in 3-3-1 should not perform this evaluation. In this case, a language implementation should be able to avoid an unnecessary message passing overhead by transforming the message passing primitives required to implement the interrupt, into simple memory references equivalent to those required for a conventional sequential condition test.

Specification 3-3-1 can be altered from an infinite loop into a single iteration of the statement sequence by modifying process $STATEMENT_n$ to

$$
\begin{aligned}
STATEMENT_n \quad &= \quad (statement_n \rightarrow SKIP \\
&\qquad [] \;\; interrupt \rightarrow (action \rightarrow STATEMENT_n \\
&\qquad\qquad [] \;\; action \rightarrow SKIP))
\end{aligned}
$$

(3-3-2)

An application process comprising n statements may interact with m interrupt-generating processes by expanding the choice of interrupt events in 3-3-1. In the specification below, the interrupt events have been represented as input processes to accentuate the nature of the input signals.

$$\alpha\, LOOP = \cup\ \alpha\, STATEMENT_i \qquad for\ 1 \le i \le n$$
$$\alpha\, STATEMENT_i = \{statement_i\ ,\ interrupt_j k\ ,\ action_j\} \qquad for\ 1 \le j \le m$$

$$LOOP = STATEMENT_1$$
$$STATEMENT_i = (statement_i \rightarrow STATEMENT_{i+1}) \qquad for\ 1 \le i < n$$
$$[]\ (j:\ 1..m) interrupt_j\ ?\ any \rightarrow (action_j \rightarrow STATEMENT_i$$
$$[]\ action_j \rightarrow SKIP))$$
$$STATEMENT_n = (statement_n \rightarrow STATEMENT_1$$
$$[]\ (j:\ 1..m) interrupt_j\ ?\ any \rightarrow (action_j \rightarrow STATEMENT_n$$
$$[]\ action_j \rightarrow SKIP))$$

(3-3-3)

Interrupt events are associated with nested processes according to the rule:

$$((e_1 \rightarrow P\ []\ i_1 \rightarrow P)\ []\ i_2 \rightarrow P) = (e_1 \rightarrow P\ []\ i_1 \rightarrow P\ []\ i_2 \rightarrow P)$$

(3-3-4)

where e_1 is an event with i_1 as an alternative, and i_2 is a higher order event. This law can be illustrated with the following nested loop example:

$$\alpha\, LOOP_{OUT} = \{statement_{OUT\text{-}1}\ ,\ statement_{OUT\text{-}3}\ ,\ interrupt_{OUT}\ ,\ action_{OUT}\}$$
$$\cup\ \alpha\ \mathrm{LOOP_{IN}}$$

$$LOOP_{OUT} = STATEMENT_{OUT\text{-}1}$$

$$STATEMENT_{OUT\text{-}1} = (statement_{OUT\text{-}1} \rightarrow STATEMENT_{OUT\text{-}2}$$
$$[]\ interrupt_{OUT} \rightarrow (action_{OUT} \rightarrow STATEMENT_{OUT\text{-}1}$$
$$[]\ action_{OUT} \rightarrow SKIP))$$

$$STATEMENT_{OUT\text{-}2} = (LOOP_{IN} \rightarrow STATEMENT_{OUT\text{-}3} \;[]\; interrupt_{OUT} \rightarrow (action_{OUT} \rightarrow STATEMENT_{OUT\text{-}2} \;[]\; action_{OUT} \rightarrow SKIP))$$

$$STATEMENT_{OUT\text{-}3} = (statement_{OUT\text{-}3} \rightarrow LOOP_{OUT} \;[]\; interrupt_{OUT} \rightarrow (action_{OUT} \rightarrow STATEMENT_{OUT\text{-}3} \;[]\; action_{OUT} \rightarrow SKIP))$$

and $\alpha\, LOOP_{IN} = \{statement_{IN\text{-}1}, interrupt_{IN}, action_{IN}\}$

$$LOOP_{IN} = (statement_{IN\text{-}1} \rightarrow LOOP_{IN} \;[]\; interrupt_{IN} \rightarrow (action_{IN} \rightarrow LOOP_{IN} \;[]\; action_{IN} \rightarrow SKIP))$$

which is equivalent to:

$$\alpha\, LOOP_{OUT} = \{statement_{OUT\text{-}1}, statement_{IN\text{-}1}, statement_{OUT\text{-}3}, interrupt_{IN}, interrupt_{OUT}, action_{IN}, action_{OUT}\}$$

$$LOOP_{OUT} = STATEMENT_{OUT\text{-}1}$$

$$STATEMENT_{OUT\text{-}1} = (statement_{OUT\text{-}1} \rightarrow STATEMENT_{OUT\text{-}2} \;[]\; interrupt_{OUT} \rightarrow (action_{OUT} \rightarrow STATEMENT_{OUT\text{-}1} \;[]\; action_{OUT} \rightarrow SKIP))$$

$$STATEMENT_{OUT\text{-}2} = (statement_{IN\text{-}1} \rightarrow STATEMENT_{OUT\text{-}2} \;[]\; interrupt_{IN} \rightarrow (action_{IN} \rightarrow STATEMENT_{OUT\text{-}2} \;[]\; action_{IN} \rightarrow STATEMENT_{OUT\text{-}3}) \;[]\; interrupt_{OUT} \rightarrow (action_{OUT} \rightarrow STATEMENT_{OUT\text{-}2} \;[]\; action_{OUT} \rightarrow SKIP))$$

$$STATEMENT_{OUT\text{-}3} = (statement_{OUT\text{-}3} \rightarrow LOOP_{OUT} \;\square\; interrupt_{OUT} \rightarrow (action_{OUT} \rightarrow STATEMENT_{OUT\text{-}3} \;\square\; action_{OUT} \rightarrow SKIP))$$

Note that the inner loop's *SKIP* process behaves like a *return* operation.

It is frequently desirable to regard a sequence of events as being an atomic operation. A critical region and the *P* operation on a semaphore [BEN 82] are examples. To simulate loops in which an inseparable sequence of actions must be completed before an interrupt is serviced, it is necessary to specify a means of delaying the interrupt until the desired action is completed.

$$\alpha\, LOOP = \bigcup \alpha\, STATEMENT_i \qquad for\ 1 \le i \le n$$
$$\alpha\, STATEMENT_i = \{statement_i, interrupt, action\}$$

$$LOOP = STATEMENT_1$$
$$STATEMENT_i = (statement_i \rightarrow STATEMENT_{i+1}) \qquad for\ 1 \le i < n$$
$$STATEMENT_n = (statement_n \rightarrow STATEMENT_1 \;\square\; interrupt \rightarrow statement_n \rightarrow (action \rightarrow STATEMENT_1 \;\square\; action \rightarrow SKIP))$$

(3-3-5)

The delaying action of 3-3-5 should be implemented in such a way as to enable an interrupt condition to be noted sufficiently promptly to avoid further program execution altering the circumstances which gave rise to the condition. The interrupt-generating model of specification 3-2-3 avoids the potential loss of information by insisting that an interrupt it has signalled should be serviced before its register contents may be altered. This works reasonably for other parallel processes, but may lead to deadlock with the process which has claimed ownership of the interrupt-generating object. For example, when a process P_1

$$P_1 = (set_1\,!\,k \rightarrow LOOP)$$

$$LOOP \quad = \quad (x \rightarrow assign_1 \,!\, k \rightarrow (x \rightarrow LOOP \;\; [] \;\; interrupt_1 \,?\, any \rightarrow SKIP))$$

$$\exists \; assign_1 k \in \alpha\, LOOP \qquad \text{and} \qquad x \in \alpha\, LOOP$$

is executing in parallel with the process from specification 3-2-3

$$(P_1 \quad \| \quad VARIABLE)$$

the possibility exists for the *VARIABLE* to signal an interrupt before P_1 has reached the $assign_1 \,!\, k$ process. P_1 then refuses to acknowledge the interrupt until the assignment has been received; while *VARIABLE* refuses to service the assignment until the interrupt has been acknowledged. To prevent this situation arising, the implementation must make provision for the interrupt to be noted, but only acted on later. A successful method for achieving this is discussed in section 6.4[1].

[1]The implementation described in section 6.4 services the interrupt immediately according to the model of specification 3-3-3 using a dummy service routine to note the interrupt, although the execution of the action code is delayed as in specification 3-3-5.

4. The interrupt-generating model of concurrent computation in the experimental programming language HUL

4.1 Introduction

The design of a computation model is not usually considered complete unless it has been subjected to empirical analysis within an implementation. The practical application of the computation model of this thesis was tested by designing and implementing a programming language which presupposes the proposed programming structures. This experimental programming language, called HUL[1], is introduced in this chapter. HUL is intended to execute in a highly concurrent (and possibly distributed) environment, and, in embracing the interrupt-generating active data object, it adopts the primary objective of reducing the programming effort for applications involving shared data and conditional signalling, and the ancillary objective of sustaining an abstract programming level through the isolation of the concurrent source language from hardware considerations in distributed parallel processing environments.

This chapter concentrates on the principal and novel aspects of the language, those features which are directly concerned with the use of the interrupt-generating active data model in supporting shared information and in accommodating the expression of occasional events. A few additional aspects of the syntax of HUL (which are not directly concerned with data encapsulation and sharing or with interrupt signalling) are discussed briefly so that the programming examples presented later will be fully understood. Since the proposed interrupt-generating object requires a distinctive class of programming construct for proper interaction, this chapter uses the syntax of HUL to provide details of the semantics and transformation laws for actual programming constructs which falls into this class.

The concept of processes is fundamental to the structure of the HUL language. A HUL

[1]HUL is an acronym for HUMAN-LIKE.

program should be viewed as a hierarchy of processes; a complete program is considered to be a single process which is composed of sub-processes, each of which is decomposed further into component processes. The basic model of a network of communicating processes may exist at any level. Each process itself can be large and complex or, at the other extreme, a process may be a single primitive operation.

The primary control structure proposed in HUL is based on a common feature of human communication, the use of implied repetition in conjunction with interruption. *Move forward until you reach the target* implies a repeated *move forward* process. To emphasize the interrupt feature, the command can be rephrased as

Move forward.
When you reach the target
stop.

To encourage the use of concurrency, implied parallelism is a further feature adopted by HUL's primary control structure. Unless a particular succession of component processes is explicitly stated, the construct assumes that the programmer wants all component processes executed simultaneously, or if insufficient processors are available for this to be possible, that the order in which they are executed is not important. HUL uses the reserved word *do* as a leading symbol for introducing this construct.

```
Do
  Move_forward
  Swing_your_arms
  When you_reach_the_target
    stop
```

The code to evaluate the condition for terminating this infinite loop is placed outside the bounds of the loop by implementing *you_reach_the_target* as an interrupt-generating Boolean variable. In this way, HUL eliminates condition evaluation on each pass through a loop.

To further facilitate the natural modelling of the user's problem domain, a number of novel definitions have been designed into the grammar of HUL to enable the programmer to make the

source code more natural to read.

Wirth has pointed out that what is omitted from a programming language is as important as what is included [WIR 85a]. In the case of HUL, all conventional forms of constructing loops have been omitted. Apart from the *do* construct, only one other control structure has been included, an extended form of the conditional *if*. The other major omission is in the area of data types. HUL is in fact a very sparse language which does not offer the use of pointers or structured data types, demanding considerable effort for programming applications requiring data structures more elaborate than one dimensional arrays. No apology is made for the meagre range of data types; HUL has been designed for the purpose of experimenting with interrupt-generating objects, and is not intended to be a rival for Occam or Ada.

The features of HUL that are more familiar have been based primarily on Occam [MAY 83] and CSP [HOA 78]. In particular, the innovative notation put forward by Hoare in CSP has been adopted to achieve synchronous communication. HUL permits synchronization to be accomplished using interrupts as well. Non-synchronized communication is possible via shared interrupt-generating objects, as well as via declared channels.

Following the design of Occam, the structure of a HUL program is denoted by indenting the source code from the left margin, rather than by statement separators and block indicators[1]. The declaration of variables, constants, channels, and procedures is based on the equivalent features of Occam, with the syntax modified in accordance with the aim of allowing a more natural expression of the source program.

Nevertheless, there are several instances in which further refinement could improve those elements of the language which have remained rather unnatural. When selecting language features to support the control structures, compromises were made which reduced the extent to which the syntax of some features reflected natural expression. This was done to accommodate forms which

[1]Readers unfamiliar with this method of denoting the bounds of code blocks are referred to the example programs listed in chapter 5.

had widespread acceptance, or which eased the implementation effort. For example, the CSP form of the output primitive, *a_channel ! result*, has been retained, although it would be more natural to say *send result to a_channel*; consistency between the declaration of variables and formal parameters has been maintained by adopting the form *variable integer I* rather than the more natural *integer variable I*, or the succinct *integer I*; parsing has been simplified by including the ":" symbol at the end of definitions and declarations, as in Occam.

In discussing HUL, this chapter pays more attention to the control structures that are the novel and powerful features of the language than to the declaration of identifiers and the evaluation of expressions. Consequently many of the examples are rather trite, having been chosen to illustrate the control structures of the language with as little obscuring support code as possible.

4.2 The principal features of HUL

This section provides an overview of the principal features of HUL. The reader's attention is drawn to the full syntax definition of HUL presented in appendix C and to the description of the semantic interpretations of control structures in section 4.4.

4.2.1 Control structures

Where most procedural programming languages use a range of different control structures, HUL supports only two.

The primary structuring mechanism of HUL, the *do* control structure, has already been mentioned in the introduction to this chapter. By allowing a set of counters and qualifiers to be coupled to it, the *do* control structure may be extended to construct all loops and single iteration sequences, and to specify parallel and sequential execution. An unqualified *do* statement presumes that the programmer wishes its component processes to be executed in parallel, or if insufficient processors are available for this to be possible, that the order in which they are executed is not significant. The execution of each component process is repeated forever, with synchronization of the component processes occurring between each pass of the structure. Accordingly

```
do          implies          do forever in parallel[1]
```

Counters and sequence qualifiers may be included to restrict the number of times that the process body is executed, or impose a sequential succession on the execution of the specified processes. Examples are:

```
Do once in sequence      Do twice          Do hour times in sequence
  Keyboard ? char          Write ["do"]      Chime
  Value <- char - 48                         Reset_the_chime
```

The *do in sequence* is a restricted form of the control structure which should only be used when the algorithm being implemented requires it. As all actions in HUL are undertaken by processes, it is the *in parallel* qualifier that is the safer option[2], which allows synchronization and communication to be determined by the dynamics of the program's execution rather than by a predefined sequence[3].

The following example of a process to control two independent traffic signals illustrates the use if *do in parallel* and *do in sequence*. The process *green[i]* switches on the green light in the i^{th} traffic signal for a predetermined period of time, then switches it off again. *Amber[i]* and *red[i]* perform similar functions.

---

[1]Although *do* defaults to *do in parallel*, the *in parallel* qualifier is explicitly declared in many of the examples in this thesis to avoid any possible confusion between concurrent and sequential execution.

[2]Section 5.5 identifies inherent parallelism as a means of preventing message passing deadlock by always anticipating a non-deterministic ordering of input and output messages.

[3]There is no advantage in executing a sequence of simple assignments in parallel. Although they might be independent, they all terminate immediately. The HUL compiler uses the transformation laws of section 4.5 to collapse this form of fine grained concurrency into a single sequential process.

```
do
  do in sequence
    green[1]
    amber[1]
    red[1]
  do in sequence
    green[2]
    amber[2]
    red[2]
```

The example which accompanies figure 2 in section 4.2.3 illustrates the use of the *do* control structure in defining interacting processes.

One or more *when* statements may be tagged onto the process body of the *do* control structure. Each *when* statement sets up an interrupt condition supported by an interrupt-generating variable. Implementation is such that the state of the interrupt-generating variable is not examined on each pass of the loop. When the interrupt-generating variable is assigned a value which agrees with the value of the interrupt condition which it supports, the loop body is automatically suspended and the appropriate programmer defined action is taken. Some examples are:

```
When store_empty                          When seconds = 60
  wait until item_count = 1                 minutes <- minutes + 1

When time_up
  stop {stop terminates the control structure, not the program}
```

An *intact* qualifier may be used in conjunction with the *do* control structure to ensure that the current iteration of the structure will execute to completion before any interrupt action is taken. This option only has meaning in a control structure which includes a *when* statement, or in a component process of such a structure which might be aborted by the premature termination of the structure.

In the example below, a sequential loop to cycle through the processes which control the lights on a traffic signal has had a *when* statement added to its process body to show how an interrupt-generating Boolean variable *no_cars_allowed* can be used to switch the traffic signal permanently to red. The *intact* qualifier guarantees that a green-amber-red cycle is completed, to avoid the

possibility of a direct change from green to red, or of two lights being switched on simultaneously.

```
do intact in sequence
  green[1]
  amber[1]
  red[1]
  when no_cars_allowed
    stay_red[1] {switch the red light on and leave it on}
```

HUL's second structuring mechanism is an *if* statement of the type used in Brinch Hansen's Distributed Processes [BRI 78], in EDISON [BRI 81], and in Occam [MAY 83]. The HUL conditional *if* clause extends Dijkstra's *guard* [DIJ 75] to a more general structure which includes the Boolean expression and the channel communication. In this context, communication is restricted to the standard, non-blocking Boolean function, *ready*[1], which indicates whether a *receive* message channel is ready to communicate. With this facility, HUL's *if* control structure replaces both the conditional and the *PRI ALT* structures of Occam.

```
If
  { condition }
    { process body }
  { condition }
    { process body }
  {... as many condition-process pairs as are needed }
```

An *if* control structure can contain any number of branches. Only one branch is executed. On execution of the *if*, the first condition is evaluated; if it evaluates to *true* then its process body is executed; if it evaluates to *false* then the next condition is evaluated in the same way. This procedure is repeated until one of the condition clauses evaluates to *true* and its process body is executed. Termination of this component process is synonymous with termination of the *if* statement. Unlike Occam, the situation in which all the condition options evaluate to *false* is not

[1]The standard system function *ready* is listed with the reserved identifiers in appendix B, and its use with the *if* control structure in providing a command that will allow a server process to make a guarded choice between a number of possible communications is described in section 4.2.3.

deemed to be an error condition resulting in infinite postponement. In this circumstance, the conditional control structure of HUL terminates.

To provide an *otherwise* option, HUL incorporates a pre-declared Boolean constant *otherwise*, which has the value *true*. For example

```
If
  wound = "scratch"
    write ["cat"]
  wound = "bite"
    write ["dog"]
  otherwise
    write ["unknown attacker"]
```

Should the programmer wish to suspend execution if all conditions fail, then this must be stated explicitly:

```
If
  X <> 0
    Z <- Y / X
  otherwise
    do {forever}
      skip
```

Since a process body can itself be a control structure, a branch of an *if* control structure can itself be an *if*.

4.2.2 Interrupt-generating active data objects

HUL adopts the model of specification 3-2-4 for its interrupt-generating active data objects, and the *do* control structure has been designed to exploit this model. Interrupt-generating objects may be defined to encapsulate integers, Booleans, and characters, the three data types supported by HUL.

In designing the syntax of HUL, an attempt has been made to make the low level operation of

interrupt-generating objects as transparent as possible. The objects are declared as variables, so that the programmer does not need to conceptualize an interrupt-generating object as being different from an ordinary variable. This is emphasized by the omission from the syntax of HUL of any reserved word which designates a variable as being interrupt-generating. To identify interrupt-generating variables for code generation purposes the HUL compiler makes use of a first pass which locates instances of identifiers used in interrupt-generating contexts, and distinguishes them from normal variables in the compiler's symbol table, even though they were all declared in exactly the same way. Thus there should be no conceptual difference between the use of interrupt-generating objects and ordinary variables. The difference is in the way the two classes of objects are implemented. The programmer simply has to be aware that only those variables which are used to form interrupt conditions may be declared globally and referenced as shared data by concurrent processes[1], and that two constraints apply to these variables: they may appear in only one interrupt condition at a time[2], and they may not be referenced unless they are part of an active interrupt condition. The former constraint avoids situations which could cause deadlock[3], and the latter constraint ensures that shared variables are implemented as interrupt-generating objects. In programs for which efficient object code is required, the programmer may have to take cognisance of the fact that references to interrupt-generating variables incur a higher run-time cost than normal memory references (because they are implemented as communicating processes).

To prevent concurrent processes from demanding simultaneous interrupt signals from the same

---

[1]To facilitate process placement in distributed parallel processing environments, normal variables may only be accessed as local variables or as parameters.

[2]This is ensured by the exclusive ownership rule.

[3]Section 3.2 illustrates the need for restricting the number of processes which can be interrupted by a single interrupt-generating variable.

interrupt-generating variable, HUL enforces the following strict ownership rule[1]:

> ***The first process to set the interrupt value of an interrupt-generating variable is given exclusive ownership of the variable until it explicitly relinquishes ownership. Only the owner process will receive interrupt signals from an interrupt-generating variable, although the variable may be referenced for fetching and storing by other processes executing concurrently with the owner process.***

Since interrupt-generating variables are implemented as active objects[2], ownership is claimed by sending the interrupt-generating object a *set* message, which specifies the interrupt value to be stored in the *test* register. A *reset* message relinquishes ownership. HUL's *do* control structure automatically generates these signals to claim ownership (for the duration of the control structure) of any interrupt-generating variables which are named in *when* statements within its scope. The following example illustrates this mechanism.

```
Do
  {process body}
  when X = 3
    {action}
```

The interrupt value is evaluated (the simple constant 3 in this example) and a *set* message is produced to send this value to the interrupt-generating variable *X* before the control structure begins executing. This sets up an interrupt condition sustained by *X*, thereby claiming exclusive ownership of *X*. The process body of the control structure is executed (repeatedly in this

---

[1]Because the difference between normal and interrupt-generating variables is apparent only from their appearance in particular programming constructs, the compiler needs to issue a warning when an interrupt-generating variable is used to provide interrupt conditions for two or more processes which may be executed in parallel. Although some elaborate algorithms might make deliberate use of the ownership rule, this is not normally the case.

[2]The reader is referred to figure 1 in section 1.3, which illustrates the functional components of a process which emulates an interrupt-generating object. In particular, the inclusion of a test register in addition to the standard memory register should be noted. In the ownership rule, "set the interrupt" corresponds to a reference to the test register, while "fetching and storing" are references to the memory register.

example). During the execution of the process body, an interrupt signal is generated each time *X* is set to the value 3, which suspends the process body and executes the action sequence of the *when* statement. A *reset* message relinquishes ownership of *X* when the control structure terminates.

The reader is reminded that each HUL control structure is regarded as a hierarchy of processes, and that a process body might comprise a number of nested processes. For example, the parent process will claim ownership of *X* in the *do* statement below; the nested processes simply constitute the code body of the parent process.

```
Do  {parent process}
  {nested process 1}
  {nested process 2}
  when X = 3
    {action}
```

When a process attempts to claim ownership of an interrupt-generating variable which is already owned by another process, it will become suspended until the other process gives up its ownership of the variable. The following examples illustrate the ownership rule.

```
Variable integer X :
Do in parallel
  Do
    {some statements which alter X}
    when X = 3
      {action for X = 3}
  Do
    {some statements which do not alter X}
    when X = 10
      {action for X = 10}
```

The program listed above will not execute the two nested processes in parallel, but will execute whichever of these processes manages to claim ownership of the shared interrupt-generating variable *X*. Once this process has executed to completion, the other one will be able to claim ownership of *X* and proceed. To execute the processes in parallel, the code might be modified as follows:

```
Variable integer X,Y :
Do in parallel
  {some statements which alter X}
  {some statements which do not alter X, but which make
   use of its value by assigning it to Y and referencing Y}
  when X = 3
    {action for X = 3}
  when Y = 10
    {action for Y = 10}
```

Similarly, code such as

```
Do
  {process body}
  when X = 3
    {action for X = 3}
  when X = 10}
    {action for X = 10}
```

will result in an error condition because the interrupt-generating process which emulates *X* is unable to sustain two interrupt conditions[1].

The interrupt-generating model of specification 3-2-4 allows its owner process to alter the value of its *test* register, although this may not be done by other processes. Because the *do* control structure is regarded as a hierarchy of processes, it is impossible for one of the component processes of a *do* construct to send an updated value to the *test* register, since it would be regarded as emanating from a different process to the control structure. Accordingly, there is no facility in HUL for altering the value of an interrupt condition during execution of the process which it will later interrupt. For example, it is unlikely that the effect of the following program segment will be in accordance with the programmer's intentions, since the interrupt value *X - (X mod 10)* will only be calculated once (before execution of the loop), not on every pass of the loop.

[1]The HUL compiler is able to detect errors such as this (and that of the previous example), and warn against a possible conflict arising from multiple ownership claims of the same variable.

```
Do
  {statements to update X}
  when X = X - (X mod 10)
    Tens <- tens + 1
```

A possible correction for this program segment might be:

```
Do
  {statements to update X}
  If
    X mod 10 = 0
      Tens <- tens + 1
```

Interrupt-generating variables may be passed as reference parameters or as value parameters in the normal way. To the programmer, there is no conceptual difference between passing interrupt-generating variables as parameters and passing normal variables as parameters. However, if a procedure accepts an interrupt-generating variable as an actual parameter passed by reference, implementation considerations prevent any normal variable being passed to the corresponding formal parameter in any other part of the program. The current implementations are able to cope with this restriction without resorting to the declaration of interrupt-generating variables as special data types, because only complete programs may be compiled. Passing interrupt-generating parameters by reference would clearly cause a problem in implementations which permit the separate compilation of sub-programs, and an adjustment to the HUL syntax would be needed to resolve this problem.

Interrupt-generating reference parameters are implemented by passing to the called procedure the system identifier of the active object created to emulate the interrupt-generating variable. This enables the called procedure to access the channels for communicating with the interrupt-generating object. The communication layer described in section 6.3 plays an important role in supporting the passing of interrupt-generating variables as reference parameters. Since the ownership of an interrupt-generating variable will have already been claimed to enable the calling process to reference it, it follows that a procedure should never attempt to set up an interrupt

condition involving an interrupt-generating variable passed to it by reference[1].

For interrupt-generating variables passed by value, the normal method of expression evaluation is followed in the calling code, and the current value of the variable is passed, as for any other variable passed by value.

Interrupt-generating variables are subject to the normal scope rules which apply to all declarations, and are not visible to processes declared outside their scope.

4.2.3 Communication and synchronization

Following the design of CSP and Occam, a synchronous communication method, the rendezvous, was chosen to combine the needs of data communication and synchronization in a single primitive process. Like Occam, HUL makes use of channels which are unidirectional and can be used for point-to-point communication between one calling process and one called process. The commands for reading and writing to a channel have a terse syntactical form as in CSP, as well as a more readable (and verbose) equivalent. To write the value contained in the variable *sum* to a channel *result*, one of the following forms is used:

```
result ! sum
result send sum
```

To read from this channel into variable *partial_sum* in another process, one would use either of the following:

```
result ? partial_sum
result receive partial_sum
```

As the communication is synchronous, the first process to execute one of the above communication pairs will be suspended until the other process reaches the matching

[1]The compiler is able to detect this type of error.

communication primitive. Only when both processes are ready to communicate will data pass from *sum* to *partial_sum*. Both processes will then proceed, independently and concurrently. The effect of the rendezvous is to cause an assignment across the boundary of two parallel processes, and therefore the variable, *v*, and expression, *e*, must be of the same type in a matching pair

ch send *e*
ch receive *v*

The standard system channels *screen* and *keyboard* (described in appendix B) are used in the same way as user declared channels, and transmit items of type *character*.

The following program is a HUL version of the standard two element buffer example [BUR 88b].

```
Channel in, through, out:
Do in parallel
  Variable integer X:
  Do in sequence
    in ? X
    through ! X
  Variable integer Y:
  Do in sequence
    through ? Y
    out ! Y
```

This program may be represented as in figure 2.

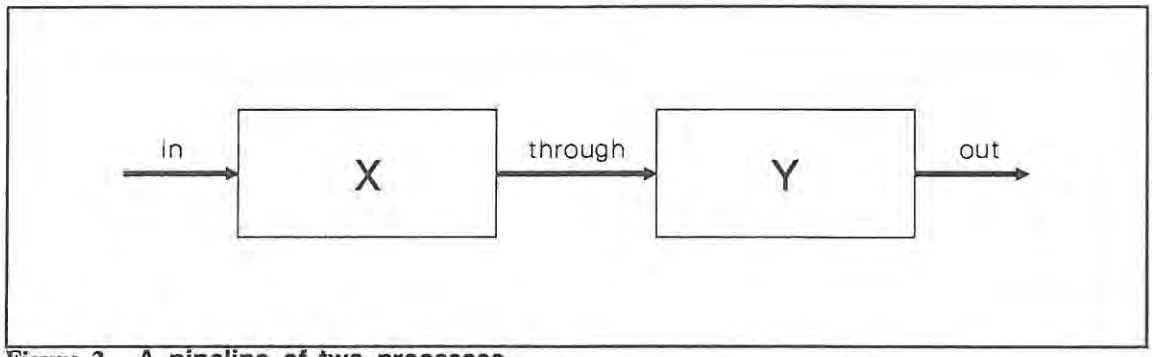


Figure 2. A pipeline of two processes.

In any concurrent programming language basing its interprocess communication facilities on synchronous message passing there is a need to allow a process to wait for one of a number of possible communications. In the absence of a non-determinate guarded control structure (such as the alternate structure of CSP, the *ALT* statement of Occam, or the *select* statement of Ada), HUL provides the standard function *ready*, which, preceded by a channel identifier, returns a Boolean value to indicate whether there is a value waiting to be input from the channel (there will be another process committed to sending the value to this process) or not. The function takes the place of a *condition*, and no message is actually passed. A subsequent input primitive is needed to effect communication. For example, the following code allows two sequences of values to arrive down channels *input_1* and *input_2*, and then be merged into a single *output* channel.

```
Variable integer X :
Do {forever}
  If
    input_1 ready
      input_1 receive X
      output send X
    input_2 ready
      input_2 receive X
      output send X
```

The *ready* function extends the power of the *if* control structure to include a prioritized version of CSP's adaptation of Dijkstra's guarded command, in an alternative structure. The example above caters for non-determinacy in the order in which messages are received, but adheres to the designated order of evaluation of the condition clauses to make a non-arbitrary selection in the case when messages arrive down both input channels simultaneously.

From their experience of writing Ada programs, Gehani and Cargill have discovered that the lack of appropriate facilities to support the rendezvous mechanism can result in a polling bias in concurrent programming languages based on the rendezvous mechanism [GEH 84a][1]. An *if* control structure which makes use of *ready* condition clauses (a less sophisticated structure than

[1]Indeed, the programming language Joyce, also based on CSP, has an explicit polling construct [BRI 87].

Ada's *selective wait* statement or *condition entry* call) is semantically similar to the use of the *else* clause in an Ada *wait* statement, one of the specific causes of polling identified by Gehani and Cargill. If neither input channel is *ready* in the example above (i.e. a rendezvous is not possible), the *if* statement terminates, requiring the programmer to place the construct within a busy waiting loop. This would be an unsatisfactory solution if the process were expected to perform some other task while it waited for input. An interrupt driven construct used in place of the rendezvous avoids program segments which poll. The use of an interrupt-generating variable to perform the communication requires the process receiving the message to know what the contents of the message will be so that it can set up an interrupt condition. For this reason, a Boolean interrupt-generating variable would typically be used to signal the presence of an input message, which would be received using a rendezvous. For example, the polling in the following program segment

```
Variable integer X :
Do {forever}
  If
    input_1 ready
      input_1 receive X
      output send X
    input_2 ready
      input_2 receive X
      output send X
    otherwise
      {Perform some small task that will not
       delay input polling for too long}
```

can be avoided by introducing an interrupt driven control structure:

```
Variable integer X :
Do
  {Perform any sort of task}

  When input_1_available
    input_1 receive X
    output send X

  When input_2_available
    input_2 receive X
    output send X
```

HUL offers no equivalent of the *ready* function for output processes. Once a process attempts to write to a channel it will be suspended unless and until another process commits itself to reading from that channel. Unlike Ada, in which an exception is raised in a calling task if its partner in a rendezvous has terminated, allowing the caller to continue [BOO 87], a HUL writer process will remain suspended even if the receiving process has terminated. Since HUL has no specific facility for expressing a timeout on a receive or send operation, the HUL programmer is responsible for implementing the cancellation of an output primitive which is not accepted within a specified time interval, should a timeout be desired. A programmer designed timeout facility can be constructed from an *if* control structure, combined with the use of the *ready* function, which terminates when all its condition clauses evaluate to *false*. For normal synchronous communication, a timeout facility for either an input or an output process can be implemented by setting an interrupt-generating variable which causes the process to be aborted.

An alternative means of synchronizing processes is offered by the *wait* statement. This option makes use of interrupt-generating variables, and although the effect is the same as using message passing channels, the *wait* statement often has the advantage of increased expressive power. For example, it is more meaningful to say

```
Wait until box_full
```

than to use

```
Box_full ? message     {the box is now full}
```

On occasions when only synchronization is required, a dummy piece of data must be communicated. The standard constant and variable pair with the same name, *message*[1], (equivalent to Occam's *ANY*), may be used in this situation.

---

[1] *Message* behaves as a standard system constant in output statements, and as a standard system variable in input statements.

```
on_chan_1 send message
on_chan_1 receive message
```

Conversely, where non-blocking message passing is needed, the synchronization feature can be suppressed by the standard technique of introducing a buffer process between the communicating processes, which is always ready to receive the sender's message [HOA 78].

A shared interrupt-generating variable can be used to communicate between two processes, one assigning a value to it and the other fetching the value. Low level synchronization is lost because the interrupt-generating variable acts as a buffer. For example, there is no synchronization between the parallel processes in the following program segment:

```
Variable integer X : {shared interrupt-generating variable}
Do in parallel

  Variable integer Z :
  Do in sequence  {Process 1}
    {some statements to set Z}
    X <- Z
    {some more statements}

  Variable integer Y :
  Do in sequence  {Process 2}
    {some statements}
    Y <- X
    {some statements which use Y}

  {A when statement which references X}
```

The buffering effect of the interrupt-generating variable is equivalent to the buffer process below:

```
Channel in, out, request :
Do in parallel

  Variable integer Z :
  Do in sequence  {Process 1}
    {some statements to set Z}
    In  |  Z
    {some more statements}
```

```
Variable integer X :
Do                {Buffer process}
  If
    In ready
      in ? X
    Request ready   {see footnote[1]}
      request ? message
      out ! X

Variable integer Y :
Do in sequence  {Process 2}
  {some statements}
  Request ! message
  Out ? Y
  {some statements which use Y}
```

For communicating with the outside world, the current HUL implementations support a set of standard system channels and predeclared procedures, which are described in appendix B.

4.2.4 Primitive processes

HUL supports five general primitive processes: input, output, assignment, *wait* and *skip*. A sixth primitive process, *stop*, may only be used in the interrupt handling sequence of a *when* statement. HUL shares CSP and Occam's novel and important feature, that of viewing process communication and synchronization as inseparable activities (discussed in section 4.2.3) at the same primitive level as assignment.

A numeric or string assignment in HUL has the form

$$v \ <- \ e$$

where v is a variable and e is an expression of the same type. Because of the left arrow's

[1]Because the *ready* function is not defined for output processes, process 2 has to request a value.

symbolic representation of the movement of the *r-value* to the *l-value*[1] of an assignment statement, "<–" is the preferred operator for assignment. Two standard alternatives for "<–" have been included in HUL: ":=" because of its widespread usage in computer languages, and *is* for instances in which a more natural language form is desired.

A boolean assignment may be expressed either as one of the standard assignment forms

v <– *c*
v := *c*
v is *c*

or, if the *r-value* is a simple constant, as one of

v
not *v*

where *v* is a Boolean variable and *c* is a *condition*. Thus

`Full`	is equivalent to	`Full <- true`
`Not Full`	is equivalent to	`Full <- false`

The *r-value* of an assignment in HUL is an expression involving variables, constants and operators. The assignment can therefore be viewed as a process in its own right, which will terminate after a fixed time unless interrupted. HUL does not make provision for calls to user defined functions, which might cause assignment processes to be delayed.

The *wait* statement was mentioned in section 4.2.3 as a means of synchronizing processes on an interrupt-generating variable. An abbreviated form of the *wait* statement may be used in the action sequence of a *when* statement, to take as its implicit condition the logical inverse of the

[1] The terms "l-value" and "r-value" are used to refer to values that are appropriate to the left and right sides of the assignment, respectively. "R-values" are values while "l-values" are addresses [AHO 86].

when statement's interrupt condition. For example, the following statements are equivalent:

```
When store_full                    When store_full
  wait                               wait until not store_full
```

The *wait until* primitive process may be used anywhere where a primitive process may be used, and is equivalent to a trivial *do* control structure used in conjunction with a *when* statement whose action is *wait*. For example, the following two program segments are equivalent.

```
wait until I = 0                   Do
                                     skip
                                     when I = 0
                                       stop
```

The *skip* process is always ready to execute, and terminates immediately. It is comparable to Dijkstra's empty statement [DIJ 76], and is used as a null process in places where some syntactical form requires a process that is not needed in the application program.

The reserved word *stop* is used very differently in HUL from its use in Occam and CSP. Rather than representing a non-terminating process that has no action, the *stop* statement in HUL must be contained within the action body of a *when* statement, and behaves like the *EXIT* statement in Modula-2 [WIR 85b], by transferring control to the statement after the end of the loop of which its parent *when* structure was a statement. This action is so named because it *stops* the current activity. A procedure to perform the drastic function of an Occam *stop* statement can be coded as an infinite loop:

```
Procedure suspend =
  do  { forever }
    skip :
```

4.2.5 Procedures and parameters

The only form of modularity supported in HUL is the procedure. Procedures are named processes with parameters. A call to a procedure creates an instance of that procedure. A procedure is re-entrant in the sense that more than one process can call it concurrently as long as there are no global variables manipulated within the procedure's process body[1].

Formal parameters must have their types specified. The programmer is required to specify explicitly whether they will be passed by reference (*variable* | *var* parameters) or by *value*. The actual parameter of a *value* formal parameter may take the form of an expression. Channel parameters may be passed in the same way as variables, and vector parameters are catered for.

There is no conceptual difference between passing interrupt-generating variables as parameters and passing normal variables as parameters, as was explained in section 4.2.2. Because interrupt-generating variables are implemented and referenced differently from normal variables, a formal parameter which is mapped by reference onto an interrupt-generating variable in one part of the program may not be mapped onto a normal variable in another[2].

In concurrent programming languages such as HUL, parallel control structures form part of a parent process at run-time, which spawns a parallel child process for each component process of the control structure. Variable declarations global to a parallel control structure cause an obvious problem if the child processes are to be distributed across a multiprocessor network. This prohibits the global declaration of variables, unless they are interrupt-generating variables. For normal variables, child processes may only reference local variables, forcing sibling processes to communicate with each other, with their parent, and with any other process, via channels.

The HUL implementations alleviate this restriction for parallel processes declared and called as

---

[1]The compiler ensures that global variables are not shared by parallel processes.

[2]Because the current implementations only compile complete programs, any contravention of this restriction can be detected.

procedures. The programmer is allowed to import, as actual parameters to a child process, all global variables used by that child, including any children it spawns, and to state explicitly whether they will be used as reference or value parameters. This greatly simplifies the compiler, since only those variables used as actual parameters need be passed to the processor executing the child process.

Two sibling processes can be prevented from simultaneously attempting to modify a global variable by introducing the following rule: ***a variable may be passed as a reference parameter to only one child process.*** The following simple example illustrates the legal application of this rule.

```
Procedure P1 [Variable integer x] =
  Do
    x <- x + 1 :

Procedure P2 [Value integer x] =
  Do
    x <- x + 1 :

Variable integer y :
Do once in sequence
  y <- 1
  Do once in parallel
    P1 [y]
    P2 [y]
```

The value of *y* is clearly defined at all times, and should *P1* and *P2* be placed on different processors for execution, *P1* should be the child process which shares the parent processor, to simplify passing *y* by reference.

Since unnamed non-primitive processes cannot be replicated, an important application of HUL procedures is to represent sub-processes of concurrent control structures. The control variable of a replicator may only be passed as a value parameter.

Procedures can be defined at any place where a variable declaration is allowed. A procedure may therefore be defined within another procedure, and, like Joyce, but unlike Occam, recursive and mutually recursive declarations are allowed. However, this feature is suppressed in current

implementations unless explicitly enabled by the programmer[1]. This is because the compile-time calculation of workspace size and number of processes is impossible if recursion is permitted. Programmers who enable recursion[2] should do so in the knowledge that an estimate will be made of the run-time workspace, which is likely to be unsuitable for the application. Programmers should in addition ensure that procedures called recursively do not create parallel processes[3].

4.2.6 Replicators

The *varying* replicator is used to instantiate a component process a specific number of times, and is useful in making the source program more compact. It may be used in conjunction with any primitive statement, procedure call, *when* statement, or condition option of an *if* control structure. When used in conjunction with a code segment executed in sequence, a replicator is similar to the familiar *for* loop, found in many programming languages.

```
Do in sequence
  Service_request [i], varying i from 1 to 5
```

is equivalent to

```
Do in sequence
  Service_request [1]
  Service_request [2]
  Service_request [3]
  Service_request [4]
  Service_request [5]
```

[1]This restriction is a compromise to ease the burden of process placement (see section 4.2.8).

[2]Recursion is enabled with a compiler directive.

[3]It is not always possible for the compiler to detect whether a procedure called recursively will create additional parallel processes.

To enable the compiler to identify how many sequential processes make up the program when a replicator causes a number of instances of a process to be created in parallel, the current implementations restrict the initial and final values of the replicator statement to expressions which can be calculated at compile-time[1]. This makes it possible to employ a static process allocation algorithm to decide before run-time where to execute each processes. Prior knowledge of the number of processes also enables memory allocation and queue length calculations to be undertaken before execution.

Rather than attaching the replicator to a control structure, as is the practice in Occam, HUL attaches the replicator to a component process of a control structure. This structuring improves the readability of the source program, and simplifies the implementation by excluding unnamed non-primitive parallel processes from replication. Thus rather than the Occam structure

```
IF I = 1 FOR 80
  Column [I] = CR
    CR.found := TRUE
```

HUL allows

```
If
  column [I] = CR, varying I from 1 to 80
    CR_found
```

which is similar in structure to the CSP notation

$$[(i{:}\ 1..80)\ \mathit{Column}(i) = \mathit{CR} \quad \rightarrow \quad \mathit{CRfound} := \mathit{true}]$$

A common application of replicators is to create several instances of a declared procedure as concurrent processes, and to distinguish them by passing the value of the replicator control variable associated with each instance of the procedure as a parameter. Since it is forbidden to alter the replicator value from within the replicated process, the control variable may only be

[1]Bypassing this restriction by creating more processes than are needed is demonstrated in section 5.4.

passed as a value parameter[1].

A zero replicator applied to any statement is equivalent to a *skip* (there is nothing to execute if the number of statements is zero).

4.2.7 HUL names and data types

Identifiers follow the common convention of a sequence of alphanumeric characters, the first of which must be an alphabetic character. HUL is not case sensitive (references to the same identifier may appear with an initial upper-case letter at the start of a *sentence*, and with a lower-case letter later on), and the underscore character may be used to improve readability.

HUL has three predefined data types:

int | *integer* -	an integer type, defined to have the range that the implementation can best support.
char | *character* -	an integer type with a range most appropriate for representing the character set (for example 0 to 255 in ASCII implementations).
bool | *boolean* -	Boolean type with values *true* and *false*.

These simple data types, together with the *chan* | *channel* type, form the complete set of types in HUL. All variables and channels must be declared, and are associated with a single type. Declared identifiers take the form of simple objects or vectors of simple objects[2].

The scope of a named entity is from the point of its declaration to the end of the process in which it is declared. It is visible from within any procedures or nested processes declared within its scope. Names must be unique and cannot be redefined within their scope.

---

[1]Once again, the compiler is able to enforce this requirement.

[2]The syntax of declarations is presented in appendix C.

The three simple data types have the assignment operator defined. The assignment operator is also defined for vectors of characters, for use in manipulating strings.

The usual arithmetic operations are available for integers, including the *mod* operator. Run-time code is included by the current implementations to detect and report any overflow condition which occurs during expression evaluation. As in Occam [MAY 87], the programmer is encouraged to make use of parentheses to delimit an operand which is a sub-expression of a larger expression, although the precedence of operators is defined by the syntax of HUL.

Literals can be used with all data types, and simple character and integer data objects may be mixed in expressions. Run-time range error reporting covers inappropriate assignments of integer values to character variables. Integer literals are simple signed or unsigned integers. Character literals may be unsigned integers or characters enclosed in single quotation marks. A character vector literal (string) is a sequence of characters enclosed in double quotation marks.

The only structured data type in HUL is the vector (a one dimensional array). Only the upper bound is specified when declaring a vector; indexing is from zero. Since there is no provision for dynamic memory allocation in HUL, the upper bound must be determinable at compile-time, and must be positive to ensure that every vector contains at least one component. Each element in a vector may be referenced by means of a subscript in the usual way. A run-time check on out-of-range subscripts is included in current implementations. Multidimensional arrays, or arrays of arrays, are not catered for. With the exception of character vectors, vectors may not be assigned or manipulated, other than one element at a time, but may be passed as parameters.

Strings are implemented as character vectors, with element zero reserved to store the current length of the string. Character vector identifiers may be used without a qualifying subscript in any context where strings may be used, to denote a string value or string variable. The assignment of a string to a character vector whose upper bound is less than the string length, causes a run-time overflow condition, and is regarded as an error.

When the operator + is used within the context of a string expression, it denotes concatenation.

Character vector variables and literals, simple character variables and literals, and unsigned integer constants may be mixed in concatenation expressions. Integer variables may not be included (although the values of integer variables may be assigned to character variables[1]), and the use of an inappropriate integer constant is detected at compilation time.

The *define* | *def* definition allows an identifier to be associated with a constant value or a vector of constant values.

4.2.8 Process Allocation

The design and implementation of HUL addresses the issue of shifting the responsibility for distributing parallel sub-programs across a network from the programmer or user of the application program to a system utility. Although this is essentially an implementation issue, the topic could not go unmentioned in this chapter, since the provision for the automatic allocation of processes to processors imposes a number of restrictions on the HUL language. The implementation facilities which support this feature are described in chapter 6.

There is no syntactical provision for process placement in HUL, and all component processes of a parallel processing construct are regarded as being distributable across a multiprocessor network. In the HUL implementations, the run-time loader is replaced by a post-linkage configuration utility, which has detailed knowledge of its host hardware, and which is able to allocate processes to processors automatically during program loading. This objective has two related benefits. The automatic allocation of processes in the configuration phase absolves the programmer of having to deal with the architectural details of the specific target parallel processing network. More importantly, it offers the potential for computer users to purchase ready-written software for a general parallel processing environment, and to execute it on their own processor network without having to reconfigure it at the source level, an exercise requiring detailed knowledge of both the application program and the target network.

---

[1]During the assignment of integers to character variables, a run-time check is able to trap inappropriate values for characters.

By contrast, Occam includes a cumbersome, non-portable *PLACED PAR* construct for specifying process placement in the source program[1] [INM 87b]. This feature has handicapped the principle of post-compilation fragmentation of Occam programs. For programmers concerned with stringent timing details, explicit control over a program's configuration is an advantage. This is not true for the many applications that require that a multiprocessor network simply be a fast general purpose machine.

The privilege of distributing processes automatically before execution imposes two significant restrictions on the language; parallel processes may not share global variable declarations, nor may they be created dynamically.

The non-interference condition imposed on the *PAR* construct in Occam [INM 87b], which prohibits more than one concurrent process from modifying a global variable, could not be adopted by HUL. The constraint imposed on Occam's *PLACED PAR* construct had to be extended to all parallel control structures to support HUL's freedom to distribute processes across all available processors on any network on which it is run. This constraint permits parallel processes to reference only local variables. Section 4.2.5 describes how the implementation of HUL alleviates this restriction for parallel processes declared and called as procedures, by passing global variables as parameters. This allows more freedom in the use of global variables than is allowed by the non-interference rule of Occam. The interrupt-generating variable concept further eases this restriction by allowing this special form of variable to be declared globally. Interrupt-generating variables are implemented as separate processes which communicate with concurrent processes in their scope by means of messages.

The syntax of HUL allows procedure definitions to be replicated as parallel processes. Because process placement is undertaken before run-time, the HUL implementation precludes the dynamic creation of processes so that the total number of processes will be known at compile-time. HUL

[1]Occam is not the only high level language which requires the programmer to specify process allocation in the source code. *MOD [COO 80], a language derived from Modula [WIR 77], and which is intended for execution in distributed parallel processing networks, uses a similar approach to Occam.

therefore restricts the initial and final values of replicator statements to expressions which can be evaluated at compile-time. Section 5.4 illustrates how dynamic process creation may be simulated by creating more processes than are needed. Naturally, this scheme is only adequate for applications in which the maximum possible number of processes is known at compile-time.

The prohibition on dynamic process creation forms the basis for discouraging direct and indirect recursion, although the current HUL implementations do allow the programmer to override this restriction. If recursion is used, the programmer is responsible for ensuring that procedures called recursively do not create parallel processes.

Besides making the automatic placement of processes possible, the prohibition on dynamic process creation also makes the compile-time calculation of workspace size possible.

At any instant a typical HUL program will have a number of processes that could execute simultaneously. Although in reality several processes may execute on each available processor, such considerations need not be of concern to the programmer. It is useful for software developers to visualize concurrent processes as each having their own physical processor to execute on. The HUL compiler produces relative estimates of the processor load required by each process (such as the likely communication and processing loads) that can be used as allocation indicators for the static configuration utility at load-time.

4.2.9 Clean termination of processes

Typically, a programmer must ensure that all child processes terminate cleanly before their parent process is able to terminate cleanly. This is often easier said than done. Concurrent programs frequently end up as a collection of suspended processes, none of which can become executable again, once they have produced their final result. The programmer is faced with the choice of either contriving an elaborate scheme of sending messages to all processes to terminate them cleanly, or reloading the operating system to rid it of the deadlock.

This problem has largely been overcome in HUL by forcing the termination of child processes on termination of their parent. Natural termination, wherein all actions have been successfully

performed and all sub-processes have terminated, can also occur and is regarded as the *normal* case. For example, the process

```
Do once in parallel
  Process_A
  Process_B
  Process_C
  When time_to_terminate
    stop
```

will terminate normally if all three component processes terminate. Alternatively, if the *when* statement's condition should become true, its *stop* action will terminate the main process, thereby aborting all component processes.

Although the semantics of the *do* control structure are of assistance in designing programs with clean termination, the programmer is still responsible for avoiding common programming errors which prevent clean termination, such as terminating a process before its neighbour has finished communicating with it, aborting a process which should be allowed to complete its current iteration before termination, or closing down a buffer process before it has had time to clear its data stream.

4.2.10 Support for embedded systems

The model of concurrency in HUL is extended so that external devices are defined to be processes outside the scope of the program. An interrupt from an external device can be modelled using channels (as in the case of the standard *screen* and *keyboard* channels described in appendix B) or using interrupt-generating variables. The infinitely looping *do* control structure with built-in interrupt handling makes HUL attractive for writing embedded systems. Channels which communicate with external processes are mapped onto specific predefined memory addresses. Interrupt-generating variables are implemented in essentially the same way at a low level, but provide the programmer of the embedded system with a useful memory-mapped

communication model which incorporates a more readily recognizable representation of interrupt handling.

Clearly, when interfacing a HUL program to an environment in which some of the processes are implemented in hardware, or in an alien language, it is impossible to isolate the programmer from hardware and system considerations. However, some measure of hardware independence is achieved by requiring the channel and address mapping to be made after compilation.

4.3 Linguistic support for the proposed model

Although systems which understand natural language are the subject of much current research, techniques for parsing natural language have not advanced sufficiently to allow computers to be programmed in this way. The control structures of high level programming languages are often unnatural to read. Professor E.W. Dijkstra was prompted to write "... most programs are presented in a way fit for mechanical execution but, even if of any beauty at all, totally unfit for human appreciation" [DIJ 76]. As an example, it would be rather unnatural when communicating with a human, to issue an instruction such as

```
WHILE NOT you_reach_the_intersection DO drive;
turn_right
```

or

```
REPEAT
  drive
UNTIL you_reach_the_intersection;
turn_right
```

Rather, one would say

```
drive
when  you_reach_the_intersection
  turn_right
```

This is the form of HUL's *do* control structure. Although the reserved word *do*[1] and the use of indentation for denoting the scope of blocks diminish the natural language qualities of the structure, the flavour of human expression is retained without incurring an inordinate increase in the complexity of parsing.

To complement the natural form of this control structure, a number of compile-time enhancements have been designed into the HUL language to permit the programmer to make the source code more natural to read. All of the features described in this section can be implemented fairly simply in the initial phases of the compiler, and have no negative effect on the run-time efficiency of programs. These options[2] allow programs to be expressed concisely as in Occam, or in a rather more wordy English form.

Several terminal symbols in the language have standard alternatives. For example, the preferred symbol for assignment is "<−", while ":=" has been retained due to its wide acceptance and *is* has been added for programmers who wish to keep their programs as natural as possible. Thus, *age <− 25*, *age := 25* and *age is 25* are equivalent. The output and input symbols "!" and "?" have the alternatives *send* and *receive*. Several reserved words have abbreviated alternatives: *proc* for *procedure*, *var* for *variable*, *alt* for *alternate*, *def* for *define*, *par* for *parallel*, *seq* for *sequence*.

Programmers may also define their own alternative names for any terminal symbol of the language. For example,

```
Alternate to_be = = :
```

will allow a definition such as

---

[1] A leading symbol is necessary to simplify the task of parsing the language.

[2] As far as the author is aware, all compile-time options described in this section are original.

```
Define A to_be 1, B to_be 2, C to_be 3,
       top to_be not bottom,
       response to_be "thank you",
       initial_values to_be table[9,4,8,3,5] :
```

Should a HUL programmer prefer the Modula-2 term *EXIT* to HUL's *stop* for terminating an infinite loop, he may create an alternate reserved word for *stop* with the definition:

```
Alternate EXIT = stop :
```

The *alternate* definition should clearly be used with care to enhance the readability of the program, and not as a reckless tool for making the program incomprehensible by altering all the reserved words. The following definitions and assignment illustrate the spirit which is intended:

```
Alternate are = is :
Variable boolean rumours :
  {......}
  Rumours are true
```

All names defined by HUL declarations and definitions must be unique and cannot be redefined within their scope. Although the word *if* might seem attractive as a synonym for "<—" in Boolean assignments such as

```
Empty if top = max,
```

the compiler would not allow the word *if* (which is already a terminal symbol) to be redefined.

In a further effort to produce more easily readable source code, an *ignore* definition has been included to allow the programmer to define a set of words to be removed from the source code at the lexical analysis stage. This facility can be used to take the place of comments in improving readability, although a conventional form of comment is available as well (all text within curly brackets is ignored by the compiler). For example, the definitions

```
Ignore initialize :
Alternate to_be = <- :
```

will allow the initialization of a ten element array to be specified as

```
Initialize A[i] to_be 0, varying i from 0 to 9
```

To promote the formation of natural phrases, a special form of Boolean assignment has been included which may take the form of the Boolean identifier alone.

`floor_is_clean`	is equivalent to	`floor_is_clean <- true`
`not floor_is_clean`	is equivalent to	`floor_is_clean <- false`

Boolean antonyms may be specified in much the same way as compile-time constants are defined, to produce Boolean identifiers whose states are always opposite. The declaration

```
Variable boolean up :
Define down = not up :
```

is best described in terms of logical equivalence:

$$(down \Rightarrow \neg\ up) \wedge (up \Rightarrow \neg\ down)$$

It is important to note that a Boolean antonym (*down* in the above example) is a compile-time convenience for which no run-time space is allocated.

In contrast to the trend in many modern programming languages for identifiers and reserved words to appear in a particular case or style, HUL is not case sensitive in any way. The intention is to allow programmers to write code in the form of English-like sentences should they so wish.

As an example of the combined effects of HUL's compile-time enhancements, the code

```
Floor_is_dirty := TRUE
DO
  Sweep_the_floor
  WHEN NOT Floor_is_dirty
    STOP
```

can be improved by using the compile-time enhancements

```
Ignore the, following :
Define floor_is_clean = not floor_is_dirty :
```

and the lack of differentiation between upper and lower case characters, to produce the more elegant form:

```
The floor_is_dirty
Do the following
  Sweep_the_floor
  When the floor_is_clean
    stop
```

4.4 Formal semantics of HUL control structures

Although only the *do* construct is able to specify concurrency, the *if* construct is also inextricably linked to the behaviour of interrupt-generating active data objects by the provision of *when* clauses in the bodies of its constituent processes. This section describes the semantics of the *if* and the *do* control structures of HUL. Examples of the distinct facilities offered by the structures are presented along with Petri net models to illustrate their execution.

4.4.1 The DO control structure

The essential syntactical alternatives of the *do* control structure are given by[1]

do = **do** [**intact**] [*count*] [*qualifier*]
process-body

count = **once** | **twice** | *simple-value* **times**

qualifier = **in parallel** | **in sequence** | **parallel** | **sequence** | **par** | **seq**

process-body = *statement*
{ *statement* }
{ *when-statement* }

when-statement = **when** (*variable* = *expression* | [**not**] *boolean-variable*)
action

action = **wait** |
{ *primitive* }
[**stop**]

[1]For the complete syntax definition of the *do* control structure, the reader is referred to appendix C.

The fundamental sequential and concurrent statements are provided by *do* control structures which have no *when* statements. *Do once in sequence* causes simple sequential flow of control which terminates when the last component in the sequence terminates. *Do once in parallel* terminates when the longest executing of its concurrent sub-processes terminates. An important characteristic of the parallel control structure is that the component processes are synchronized at the start and at the end of the structure. When multiple iterations of a parallel control structure are specified, component processes synchronize at the end of each iteration. Figure 3 describes the semantics of the following example:

```
Do once in parallel
  { parallel processes }
```

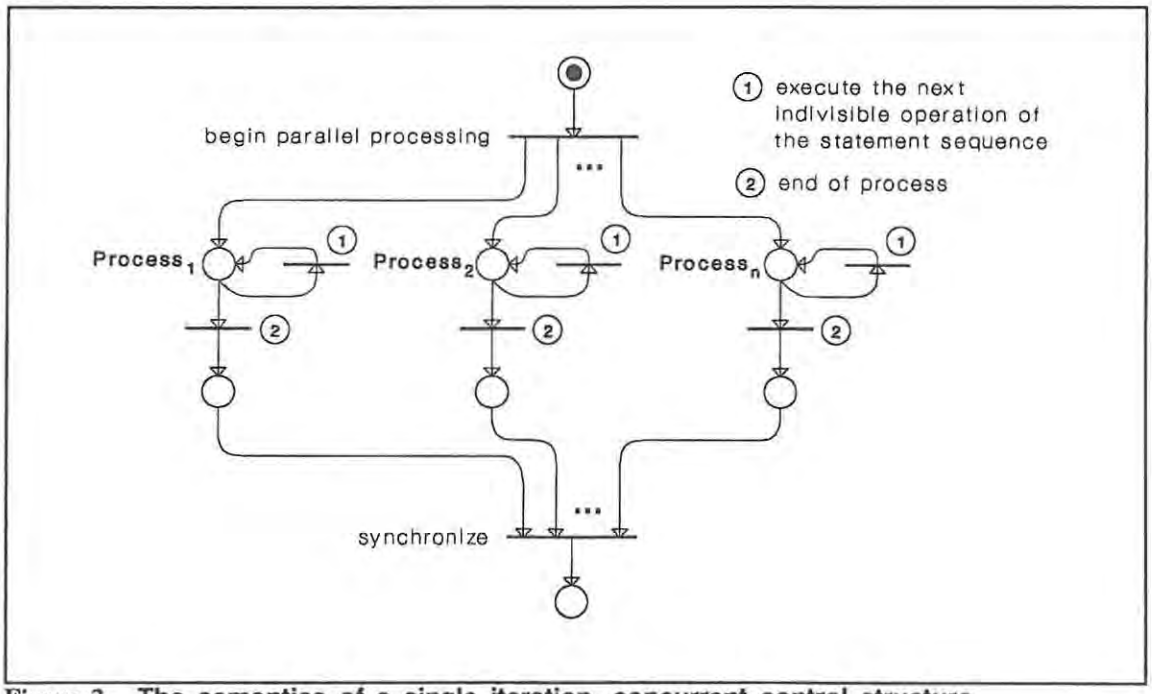


Figure 3. The semantics of a single iteration, concurrent control structure.

The semantics of a loop which is executed a specified number of times follows that of the well

established *for* loop. The control variable is a hidden variable and cannot be referenced by the HUL programmer. Figure 4 specifies the behaviour of the following code:

```
Do N times
  { process body }
```

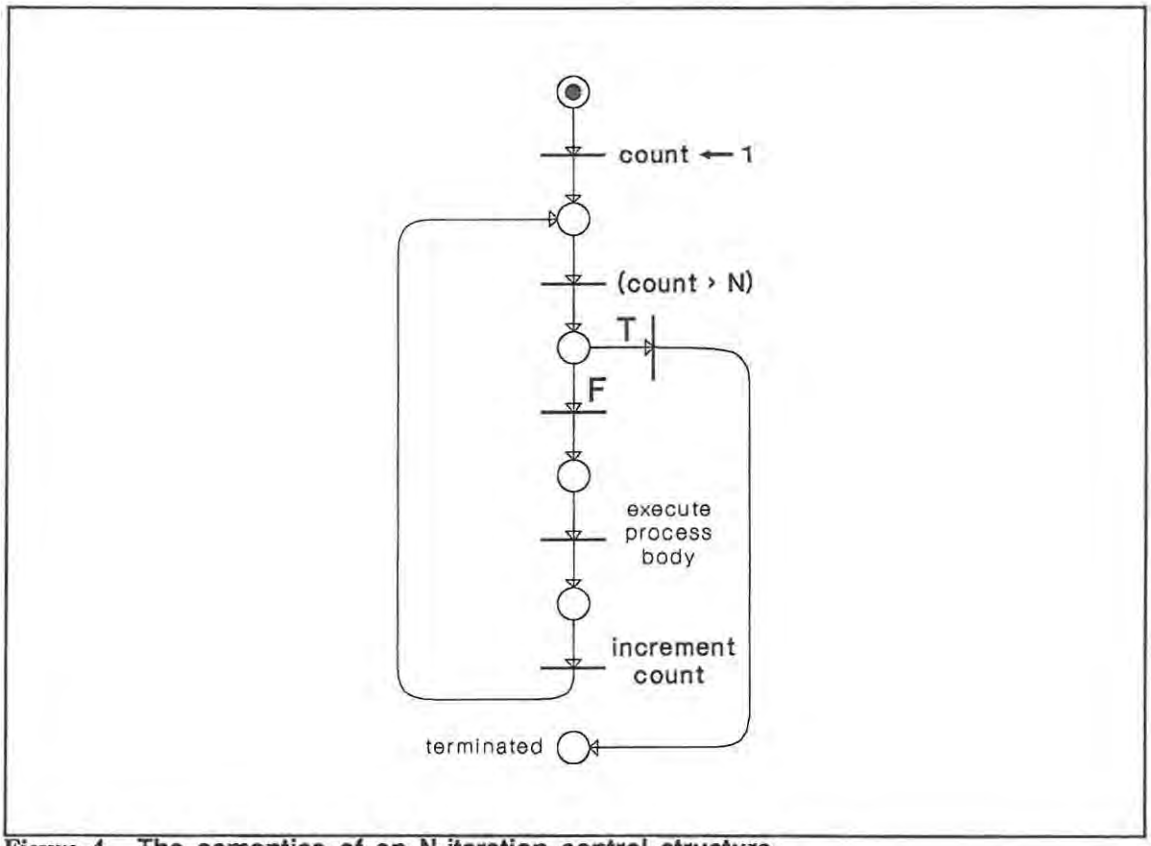


Figure 4. The semantics of an N-iteration control structure.

The remainder of this sub-section describes the use of *do* control structures in conjunction with *when* statements, and considers the general case of an infinite looping structure. The semantics of single iteration structures and loops executed a specified number of times have equivalent descriptions in each instance.

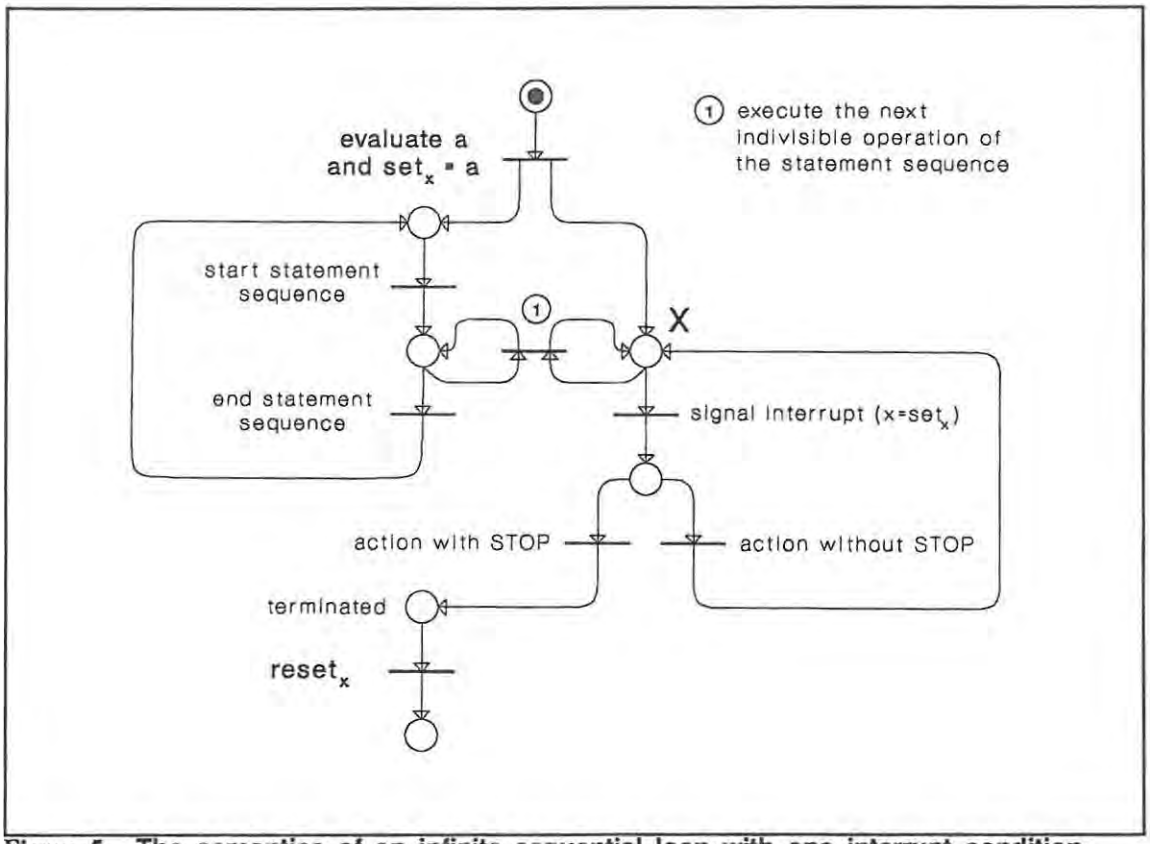


Figure 5. The semantics of an infinite sequential loop with one interrupt condition.

Formally, the semantics of a multiple iteration, sequential loop (the *do in sequence* control structure) are described by specification 3-3-3. The *test* register of the interrupt-generating variable named in a *when* statement is assigned its interrupt condition value at the start of the control structure. This action claims exclusive ownership of the variable[1] until the control structure terminates. The interrupt value of the *test* register cannot be altered during the execution of the loop. Figure 5 describes the semantics for the following example:

```
do in sequence
  { statement sequence }
  when x = a
    { action }
```

[1]Exclusive ownership allows other processes executing concurrently to reference the interrupt-generating variable (as an l-value or an r-value), but not to receive an interrupt signal from it.

Once the transition which signals an interrupt has fired in the Petri net of figure 5, the token is removed from the place labelled *x*, preventing the transition which executes the next indivisible operation of the statement sequence from firing.

Only one interrupt action may be executed at a time should interrupts occur simultaneously in a process which has a number of interrupt conditions. The interrupt action sequences are regarded as indivisible, and so cannot be interrupted by further interrupt signals. The order in which interrupt condition values are evaluated and sent to the interrupt-generating variables is non-deterministic, which prevents the programmer from assuming a particular order of execution. Figure 6 shows exclusive execution of the interrupt actions for the sequential loop listed below, which has two interrupt conditions.

```
do in sequence
  { statement sequence }
  when x = a
    { action }
  when y = b
    { action }
```

Formally, the semantic effect of the *intact* qualifier can be described as in specification 3-3-5. Any interrupt action is deferred until the end of each iteration. The process to be interrupted shares the same processor as the interrupt action sequences, but is given initial exclusive execution privilege. The previous example, modified for *intact* execution, is listed below, and the semantics which this modification directs are shown in figure 7. In this Petri net, the process body of the control structure ensures its own exclusive execution by providing the token for the *exclusive execution* (of interrupt actions) place at the end of an iteration.

```
do intact in sequence
  { statement sequence }
  when x = a
    { action }
  when y = b
    { action }
```

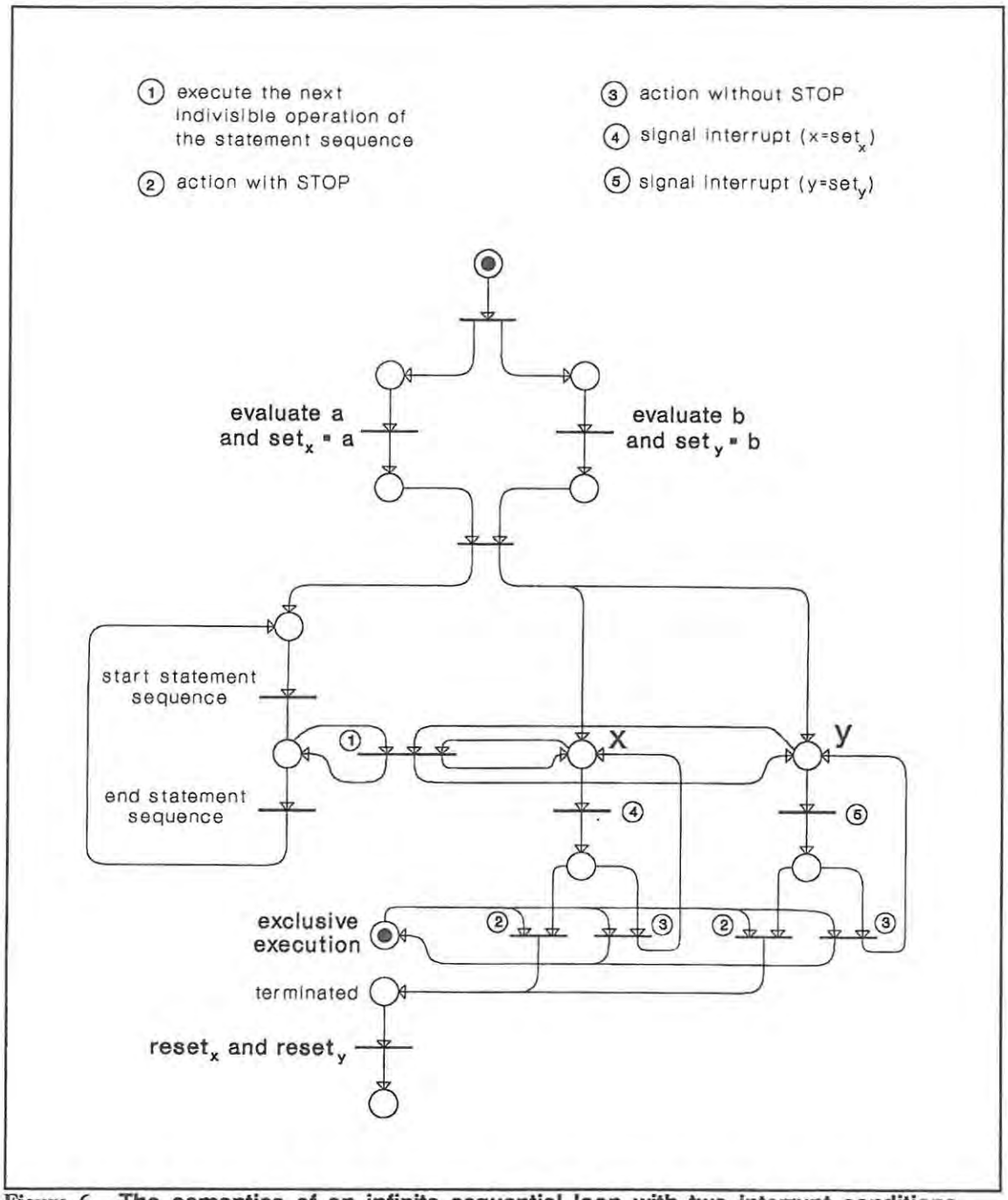


Figure 6. The semantics of an infinite sequential loop with two interrupt conditions.

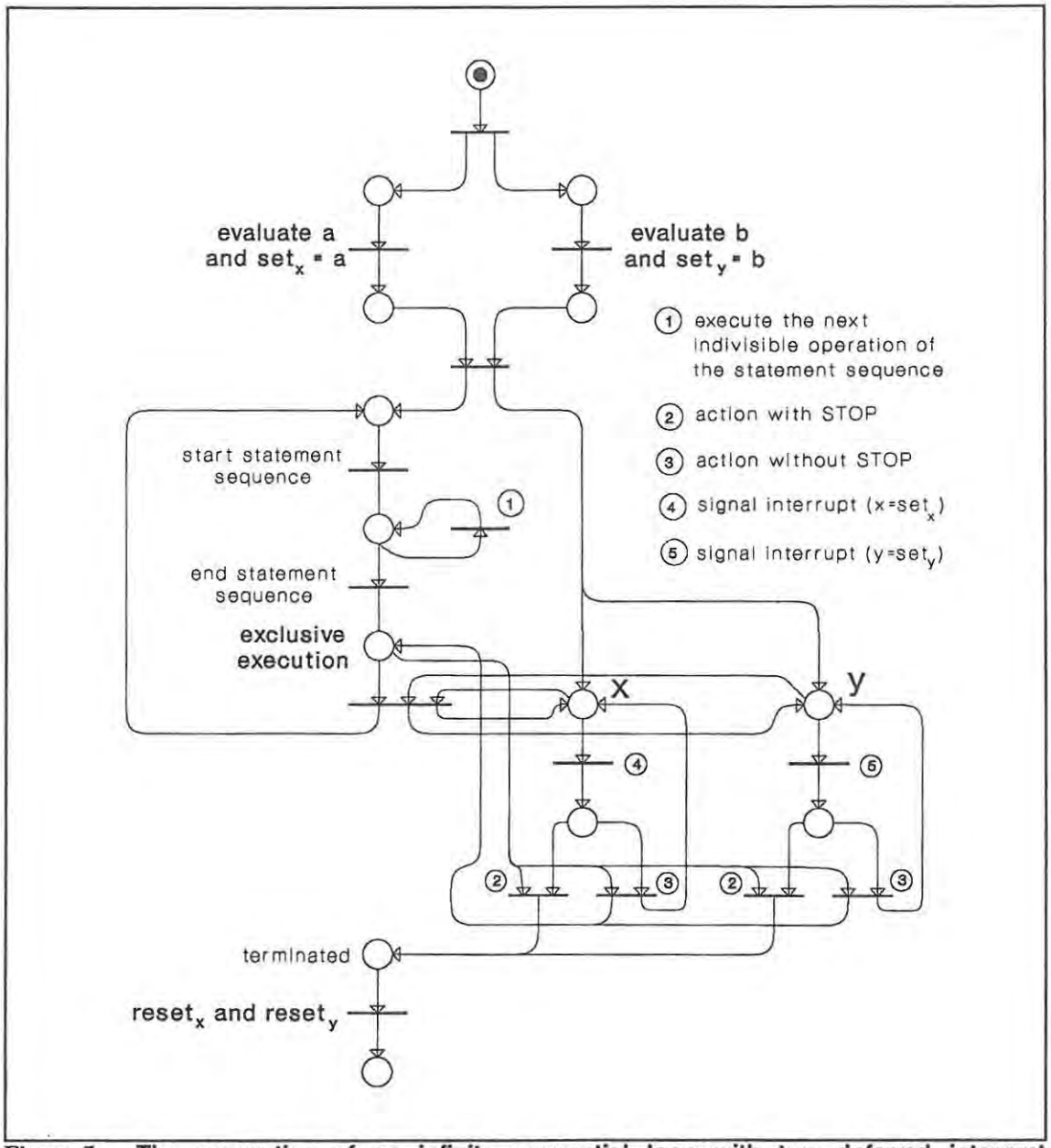


Figure 7. The semantics of an infinite sequential loop with two deferred interrupt conditions.

The *intact* option is used with parallel control structures to produce the same form of deferment as for the sequential case. The action of any interrupt is deferred until the last concurrent sub-process has executed to completion, within a particular iteration. The semantics of a concurrent loop with one or more normal interrupt conditions allow for the execution of each of the

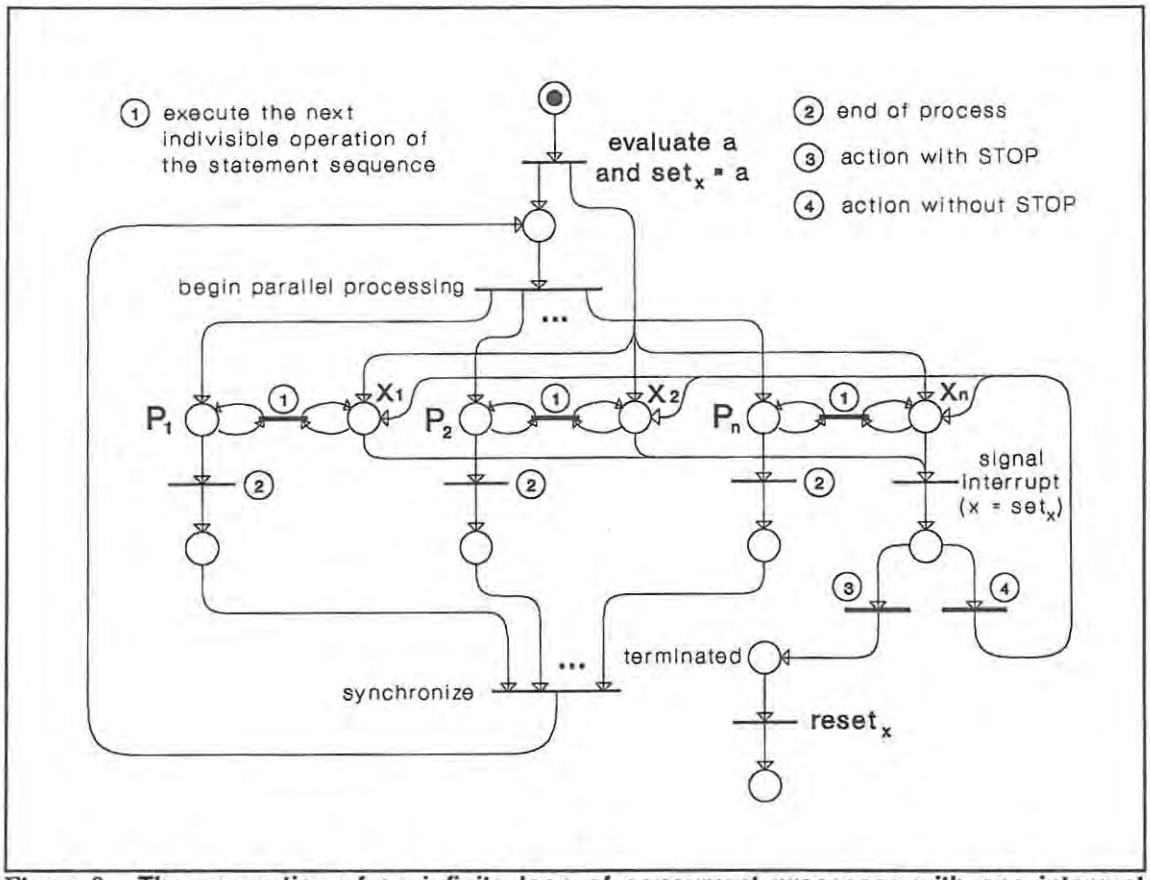


Figure 8. The semantics of an infinite loop of concurrent processes with one interrupt condition.

component sub-processes to be held up while the appropriate action is taken on an interrupt. The programmer should therefore never assume concurrent execution of an interrupt action sequence and any of the sub-processes of the parallel control structure to which it belongs[1]. Figure 8 illustrates the semantics which are applied to the following example.

```
do in parallel
  { parallel processes }
  when x = a
    { action }
```

[1]Although it is practical for some of the component processes of a parallel control structure to execute through one of the control structure's interrupt action sequences, the implementation will usually place the interrupt action sequences on the same processor as one or more of the component processes.

The semantics of the ownership rule of interrupt-generating objects are described by the following example:

```
Variable boolean SHARED :
DO IN PARALLEL

  Do in sequence  { PROCESS 1 }
    { statement sequence }
    when SHARED
      { action }

  Do in sequence  { PROCESS 2 }
    { statement sequence }
    when SHARED
      { action }
```

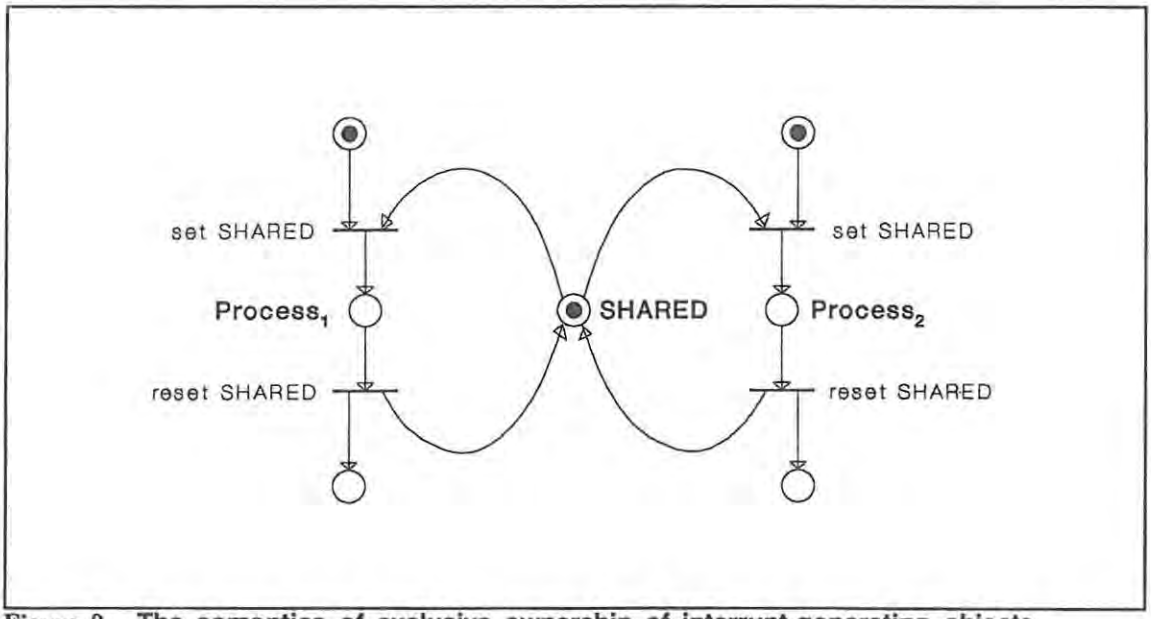


Figure 9. The semantics of exclusive ownership of interrupt-generating objects.

Figure 9 shows that the ownership of *SHARED* is exclusive. The infinite postponement of one of the processes is clearly shown should the other process not relinquish ownership of the shared interrupt-generating variable (if it is an infinite loop which only terminates at the end of the

program). It also follows that two processes, one nested within the other, which both try to claim ownership of the same interrupt-generating variable, will cause deadlock.

4.4.2 The IF control structure

The syntax for the conditional *if* is given by[1]

if = **if**
condition
process-body
{ *condition*
process-body }

condition = [**not**] *boolean-expression* { (**and** | **or**) *boolean-expression* }

boolean-expression = *boolean-variable* |
boolean-constant |
expression ("=" | "<>" | "<" | ">" | "<=" | ">=")
expression |
channel-identifier **ready** |
"(" *condition* ")"

process-body = *statement*
{ *statement* }
{ *when-statement* }

Formally, the semantics of the *if* statement can be described as follows.

[1]For the complete definition, which includes opportunities for replication, the reader is referred to appendix C.

The component branches of the *if* control structure have the alphabet:

$$\forall\ BRANCH_i \in IF\ .\ (\ \alpha\ BRANCH_i\ =\ \{evaluate_i\ ,\ true\ ,\ false\}\ \cup\ \alpha\ PROCESS_i\ )$$

representing the event of evaluating $CONDITION_i$, generating the value *true* or *false*, and executing $PROCESS_i$. Let the *if* process with n branches have the alphabet:

$$\alpha\ IF\ =\ \cup\ \alpha\ BRANCH_i \qquad for\ \ 1 \le i \le n$$

then

$$\begin{aligned} IF &= BRANCH_1 \\ BRANCH_i &= evaluate_i \rightarrow (true \rightarrow PROCESS_i \rightarrow SKIP \qquad for\ i < n \\ &\qquad\qquad [] \ \ false \rightarrow BRANCH_{i+1}\ ) \\ BRANCH_n &= evaluate_n \rightarrow (true \rightarrow PROCESS_n \rightarrow SKIP \\ &\qquad\qquad [] \ \ false \rightarrow SKIP\ ) \end{aligned}$$

Only one of the component processes can be executed, and termination of this processes is synonymous with termination of the *if* control structure. The control structure terminates if all the condition options evaluate to *false*. The Petri net model of figure 10 represents the behaviour of the following example:

```
If
  { condition 1 }
    { process 1 }
  { condition 2 }
    { process 2 }
```

An *otherwise* option is treated as a normal condition which evaluates to *true*, as can be seen in figure 11 for the example below. It must logically be the last condition clause. Since the *otherwise* option must evaluate to *true*, the control structure will not terminate without executing

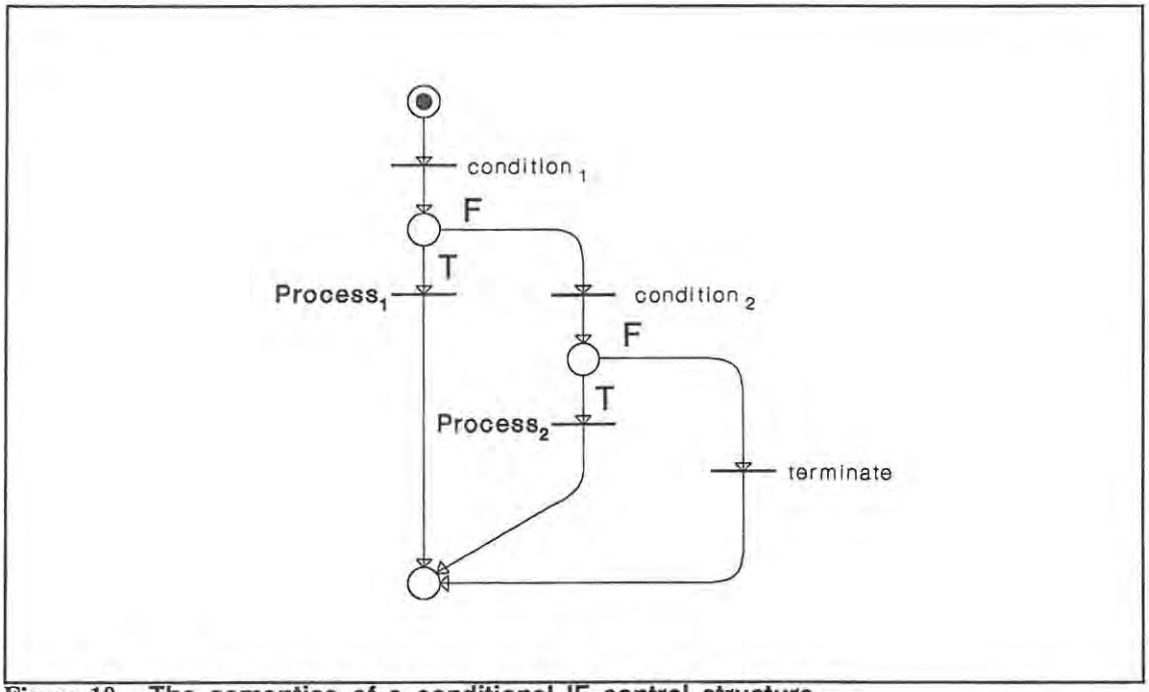


Figure 10. The semantics of a conditional IF control structure.

one of its component processes.

```
If
  { condition 1 }
    { process 1 }
  { condition 2 }
    { process 2 }
  otherwise
    { process 3 }
```

Figure 12 demonstrates the ease with which the semantics of the Occam conditional control structure can be simulated using HUL's *otherwise* option. In practice, when a condition which is not regarded as a desirable option in the program logic is considered to be an error (such as might occur in the program code below), it is more meaningful for the program to report an error than simply to suspend itself.

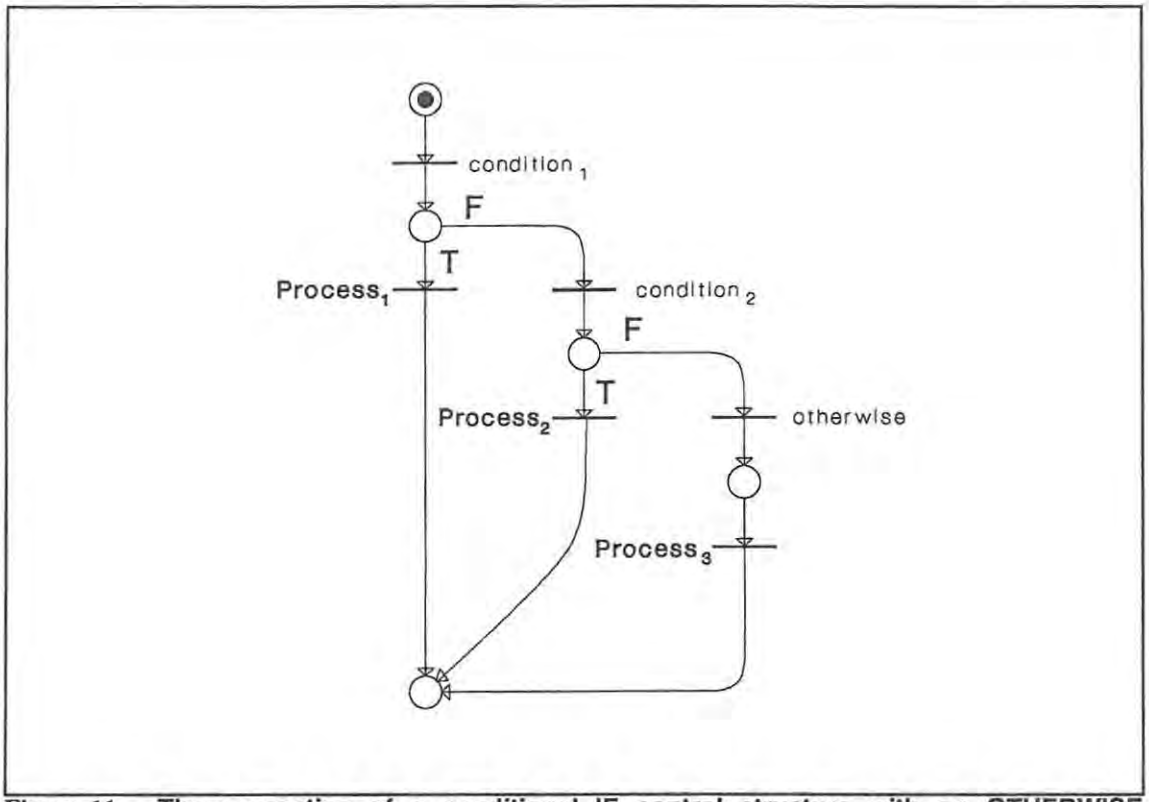


Figure 11. The semantics of a conditional IF control structure with an OTHERWISE option.

```
{ HUL }
If
  X <> 0
    Z <- Y / X
  otherwise
    suspend  {see footnote[1]}
```

```
-- Occam
IF
  X <> 0
    Z := Y / X
```

A component process of an *if* control structure which has an interrupt condition specified by a *when* statement assumes the semantics of the *do once in sequence* structure.

[1]The declaration of a procedure to implement suspension is listed in section 4.2.4.

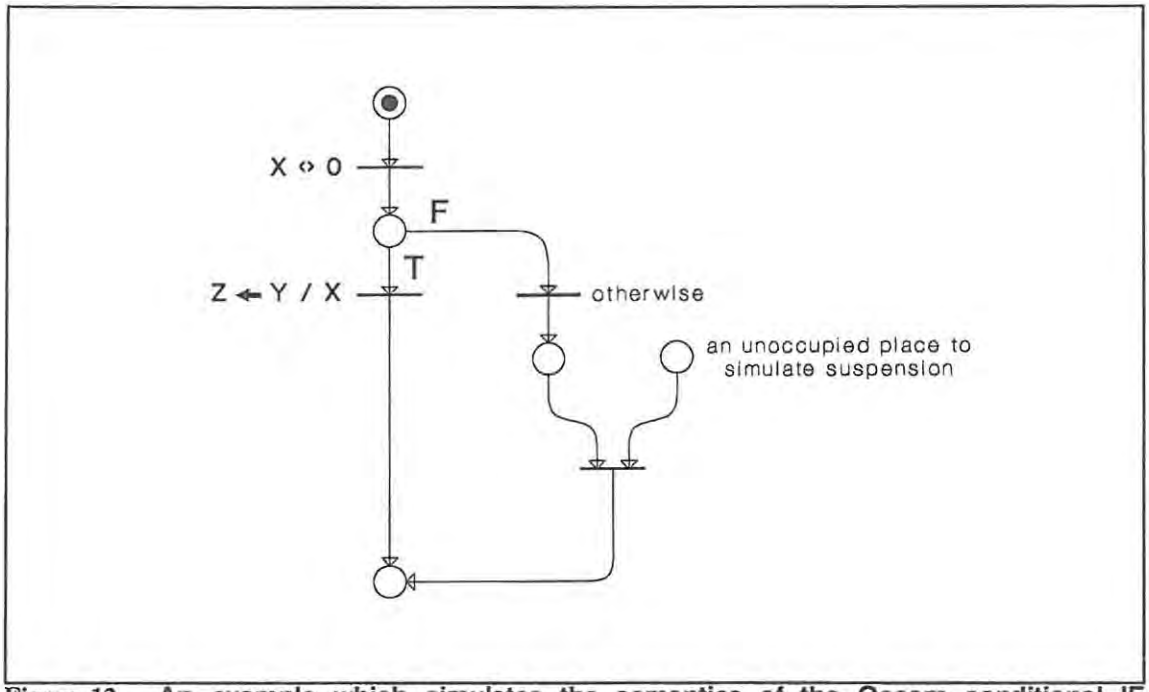


Figure 12. An example which simulates the semantics of the Occam conditional IF construct.

4.5 The primary laws which govern HUL programs

In this section, a number of laws are introduced which allow program transformation to be applied to small HUL program segments. These laws can be formally specified, which makes it possible for some of the transformations to be undertaken by appropriate software tools. In particular, some of the laws are used to make simple transformations during the compilation process to produce a more efficient object program.

The single iteration parallel and sequential *do* control structures obey the rule of association. A sequence of sequences is itself a sequence. For example, the following three program segments are equivalent.

```
Do once in sequence          Do once in sequence          Do once in sequence
  Do once in sequence          Process_A                    Process_A
    Process_A                  Do once in sequence          Process_B
    Process_B                    Process_B                  Process_C
  Process_C                      Process_C
```

This can be expressed more formally as:

$$\textbf{do once in sequence}(P_A\ ,\ \textbf{do once in sequence}(P_B\ ,\ P_C))$$
$$= \textbf{do once in sequence}(P_A\ ,\ P_B\ ,\ P_C) \qquad \textbf{(4-5-1)}$$

A nested set of concurrent processes executes concurrently with the set of concurrent processes of which its parent process is a member. In addition, the parallel *do* structure obeys the rule of commutativity, enabling component processes to be rearranged in whatever order is convenient. For example, the following three program segments are equivalent.

```
Do once                      Do once                      Do once
  Do once                      Process_A                    Process_C
    Process_A                  Do once                      Process_B
    Process_B                    Process_B                  Process_A
  Process_C                      Process_C
```

Two formal rules can be expressed for this example:

Associativity **do once in parallel**(P_A , **do once in parallel**(P_B , P_C))
= **do once in parallel**(P_A , P_B , P_C) **(4-5-2)**

Commutativity **do once in parallel**(P_A , P_B , P_C)
= **do once in parallel**(P_C , P_B , P_A) **(4-5-3)**

When multiple iterations are indicated, neither the sequential nor the parallel *do* control structures obey the rule of associativity, and commutativity is restricted for nested parallel structures.

do(P_A , **do**(P_B , P_C)) ≠ **do**($P_{,A}$, P_B , P_C)
where *do* represents either *do in parallel* or *do in sequence*,
and **do in sequence**(P_A , P_B , P_C) ≠ **do in sequence**(P_B , P_C , P_A)

Commutativity is naturally antithetical to an explicit sequential specification, and because of nested iteration, associativity cannot be applied to sequential structures which involve multiple iterations. The nested iteration restriction on associativity applies to the parallel option too. Concurrent processes in HUL synchronize when they are initiated and again when they terminate (the parallel control structure terminates when the last component process terminates), so even in the case of infinite repetition it is possible that the application of associativity to nested structures might make the program behave differently. Commutativity can still be applied within individual parallel control structures, but not across the boundaries between levels of nesting.

When one of the branches of an *if* statement is itself an *if* statement, it is possible to remove the nesting:

if((C_1 , P_1) (C_2 , **if**(C_3 , P_3))) = **if**((C_1 , P_1) (C_2 **and** C_3 , P_3)) **(4-5-4)**
where (C_i , P_i) represents a conditional clause and its sub-process.

The *skip* process has no effect on either a sequence of processes or a set of concurrent processes.

$$\textbf{do in sequence(skip , } P_A \textbf{ , } P_B\textbf{)} = \textbf{do in sequence(}P_A \textbf{ , } P_B\textbf{)} \qquad \textbf{(4-5-5)}$$

$$\textbf{do in parallel(skip , } P_A \textbf{ , } P_B\textbf{)} = \textbf{do in parallel(}P_A \textbf{ , } P_B\textbf{)} \qquad \textbf{(4-5-6)}$$

The single process of a simple parallel or sequential control structure may stand alone.

$$\textbf{do once in sequence(}P\textbf{)} = \textbf{do once in parallel(}P\textbf{)} = P \qquad \textbf{(4-5-7)}$$

In nested *do* control structures which are subject to 4-5-7, a replicated *do* structure supercedes a single iteration *do* structure with identical qualifiers.

$$\textbf{do(do once(}P\textbf{))} = \textbf{do once(do(}P\textbf{))} = \textbf{do(}P\textbf{)} \qquad \textbf{(4-5-8)}$$

where *do* may be followed consistently by a *parallel* or *sequence* qualifier.

When statements are not executed on each iteration of a *do* control structure, and are not affected by the number of iterations of the *do* structure. They have the same effect in both parallel and sequential control structures. Because their execution is not determined by their order of appearance in the source code, commutativity may be applied to their definition.

$$\textbf{do(}P\textbf{, when A, when B)} = \textbf{do(}P\textbf{, when B, when A)} \qquad \textbf{(4-5-9)}$$

It should be noted that nested *do* control structures which are subject to 4-5-7 are effective in establishing the scope of an interrupt condition in a source program. Because the nesting of *do* control structures influences the range of code which can be interrupted and the order in which ownership is claimed of interrupt-generating variables, *do* structures which contain *when* statements might behave differently if the associativity of laws 4-5-1 and 4-5-2 are applied to them, or if nested control structures are transformed according to law 4-5-7. Stated formally:

$$\textbf{do(}P_A \textbf{ , do(}P_B \textbf{ , when B))} \neq \textbf{do(}P_A \textbf{ , } P_B \textbf{ , when B)}$$

$$\text{and} \quad \textbf{do(}P_A \textbf{ , do(}P_B \textbf{ , when B) , when A)} \neq \textbf{do(}P_A \textbf{ , } P_B \textbf{ , when B , when A)}$$

Exclusive ownership of an interrupt-generating object will cause two concurrent processes, each

attempting to claim ownership of the same interrupt-generating object, to execute in sequence.

do once in parallel(**do**(P, **when** A), **do**(Q, **when** A))
= (**do once in sequence**(**do**(P, **when** A), **do**(Q, **when** A))
[] **do once in sequence**(**do**(Q, **when** A), **do**(P, **when** A))) **(4-5-10)**

The motivation for transforming a HUL program during the compilation process is usually to change a concurrent program segment whose parallel grain is very fine into a sequential program segment for more efficient execution on a single processor. For this transformation, the following equivalence can be assumed subject to certain conditions:

do once in parallel(P_A , P_B , P_C) = **do once in sequence**(P_A , P_B , P_C) **(4-5-11)**

When transforming from a sequential control structure into a parallel one, this equivalence is correct provided the sets of variables and the channel names and directions used in P_A, P_B and P_C are disjoint. Transforming the other way, the constraints are less stringent. Care must be taken to avoid the inappropriate sequential ordering of matching input and output primitives (which could cause deadlock), and where possible, such input and output pairs should be transformed into references to common variables. This can be represented as a law as follows:

do once in parallel(C ! X , C ? Y) = Y <- X **(4-5-12)**

This can be carried further to specify the combination of two disjoint concurrent processes which synchronize and communicate data during their execution.

do once in parallel (**do once in sequence**(P_1 , C ! Y , P_2),
do once in sequence(Q_1 , C ? X , Q_2))
= **do once in sequence** (**do once in parallel**(P_1 , Q_1),
X <- Y,
do once in parallel(P_2 , Q_2)) **(4-5-13)**

As an example, a transformation is presented below to demonstrate the synchronization which takes place at the beginning and the end of each iteration of a parallel control structure.

```
Channel begin, end :
Do in parallel
  Do once in sequence
    Begin ! message
    Process_A
    End ! message
  Do once in sequence
    Begin ? message
    Process_B
    End ? message
```

```
        =
(by 4-5-8)
```

```
Channel begin, end :
Do in parallel
  Do once in parallel
    Do once in sequence
      Begin ! message
      Process_A
      End ! message
    Do once in sequence
      Begin ? message
      Process_B
      End ? message
```

```
        =
(by 4-5-13)
```

```
Do in parallel
  Do once in sequence
    skip   {message <- message}
    Do once in parallel
      Process_A
      Process_B
    skip   {message <- message}
```

```
        =
(by 4-5-5,
   4-5-7
and 4-5-8)
```

```
Do in parallel
  Process_A
  Process_B
```

Laws 4-5-11, 4-5-12 and 4-5-13 may be used by programmers who wish to increase the degree of parallelism of their program by transforming some sequential code into concurrent code. For example a single stage in a pipeline of processes might execute a code sequence which could be represented as:

```
Do in sequence
  From_last_stage receive W
  Calculate_X [W, X]
  Y <- X mod 12
  Calculate_Z [Y, Z]
  To_next_stage send Z
```

If it was found that the amount of computation required by this stage was in excess of the other stages in the pipeline, the two calculation steps of the sequence could be executed as separate processes in the pipeline by transforming the sub-program using law 4-5-13 (and some of the other primary laws, notably 4-5-6), to yield:

```
Do once in parallel

  Do in sequence
    From_last_stage receive W
    Calculate_X [W, X]
    New_chan send X mod 12

  Do in sequence
    New_chan receive Y
    Calculate_Z [Y, Z]
    To_next_stage send Z
```

5. Programming techniques

A range of standard concurrency problems is presented in this chapter using the syntax of HUL, to illustrate the application and programming techniques of the interrupt-generating active data object. The examples have been chosen to highlight the benefits and shortcomings of the proposal.

HUL allows and encourages fine grained parallelism, to the level of the most primitive process. Although the implementation of the language is able to compensate for the negative effects of the overheads associated with initiating parallel processes in cases involving simple processes[1], the programmer remains responsible in most situations for choosing a sensible grain of parallelism. It is important to ensure that the processing loads of parallel processes are large enough to justify the communication overhead incurred by their parallel execution. It is equally important to ensure that the components of a parallel control structure are independent of one another, or interact in a well defined manner.

Most concurrent programs can be coded using the message passing features of HUL, which are similar to those of several other concurrent programming languages[2]. This section will concentrate on ways in which HUL's novel feature, the interrupt-generating object, can be applied.

The use of indentation to denote the program structure results in program listings which are sometimes difficult to follow, particularly if they extend over the end of a page. The example programs presented in this chapter have been written using declared procedures rather than

---

[1]Simple parallel processes which terminate in a finite time are collapsed into a single sequential process (see section 6.2).

[2]Numerous programming notations have been devised which are based on implicit or explicit forms of message exchange. These include Ada [WEL 81], CHILL [FID 83], Concurrent-C [GEH 86], COSPOL [ROP 81], CSP [HOA 78], CSP-80 [JAZ 80], DP (Distributed Processes) [BRI 78], ECSP [BAI 84], Input Tools [VAN 81], Joyce [BRI 87], Nil [STR 83], Occam [MAY 87], Planet [CRO 84], PLITS [FEL 79], Smalltalk [GOL 83], SR (Synchronizing Resources) [AND 82], and the many expressly port-based languages, such as CP (Communication Ports) [MAO 80], Directed Ports [SIL 81], Ordered Ports [BAS 87], and Port Language [KER 86].

unnamed processes, or have had dashed lines inserted within comments to mark the boundaries of parallel processes. The HUL development environment incorporates a folding editor [HAR 86b], similar to the Occam folding editor distributed with the TDS software [INM 87a], which enables a HUL programmer to hide the details of a process body and view the macrostructure of the process. Not all of the HUL examples in this chapter attempt to use a natural form of expressing the solution they represent, some because a less wordy representation better illustrates the concept being described, and others because they explore the fringe effects of the semantic rules.

5.1 Using interrupt-generating active data objects

Although channels can be declared to handle all the necessary communications between concurrent processes, the model of a shared interrupt-generating object is particularly useful for solving that class of problems in which several parallel processes make references to the same piece of information in a non-deterministic order.

The following procedure to control a bounded buffer, such as the type utilized by the producer-consumer problem [BEN 82], is presented as an example of an interrupt-generating variable acting as a shared data object between distributed parallel processes. The buffer is implemented as a stack in this example. It is possible (though not good programming practice as far as object-oriented programming is concerned) to make the stack pointer both visible to the producer process, and useful in synchronizing the producer's activities, by declaring a shared variable for use in an interrupt driven structure.

```
Variable integer top :

Procedure warehouse =
  Variable character stack [stack_max] :
  Do  {to repeat the if construct}
    If
      request ready
        Request receive message
        Consumer send stack [top]
        Top <- top - 1
      producer ready
        Top <- top + 1
        Producer receive stack [top]   :

Procedure producer_process =
  Variable character item :
  Do in sequence
    {manufacture character item}
    Producer send item
    when top = stack_max
      wait    :
```

In this example, both processes have access to variable *top*, even if they are running on separate processors. The producer process is given exclusive ownership, enabling the interrupt-generating mechanism of *top* to delay the production of items while the condition *top* = *stack_max* is true. Because of the presence of the interrupt generation mechanism, there is no need for the producer to test the value of *stack_max* after each item has been manufactured[1].

To illustrate the use of the HUL control structures in a truly parallel environment (in which each process is intended to run on a separate processor), an original implementation is presented of a simplified controller for a numerically controlled tool (NC-tool), such as a milling machine. The NC-tool is moved under the control of an operator process (which could be a special purpose computer, such as a computer aided design and manufacturing workstation). It has three degrees of freedom, each of which is manipulated by a separate hydraulic pump and piston. It is convenient to use a separate processor for controlling each hydraulic pump, since the real-time feedback loop, which senses piston movement and adjusts the current to the pump, requires

[1] A slightly amended version of the producer-consumer problem is listed in section 5.6.

a variable load compensation facility to cope with tools of different weight. A fifth process monitors the co-ordinate position of the tool and adjusts the angle of a nozzle which sprays cooling fluid onto the tool. The interaction of processes in the HUL solution for the controller is depicted in figure 13.

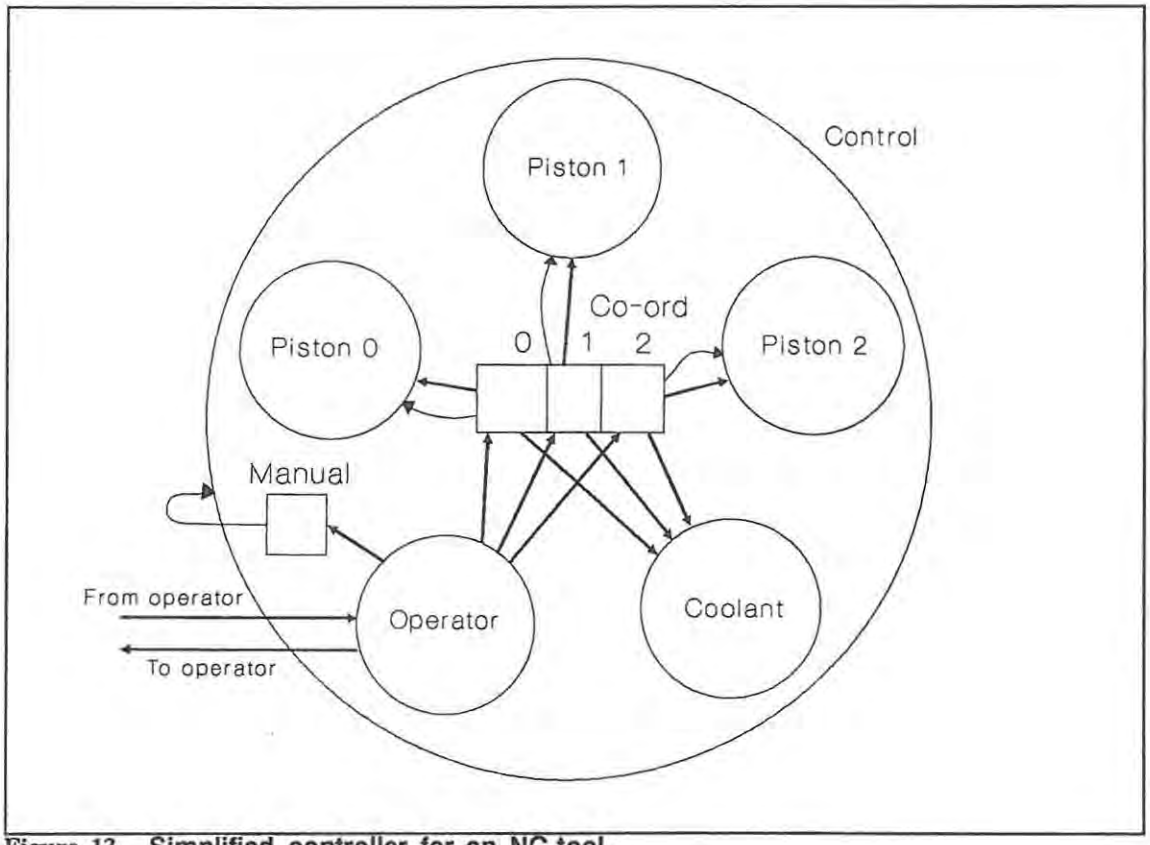


Figure 13. Simplified controller for an NC-tool.

In the HUL solution below, local variables are used to accommodate frequent accesses to the values contained in shared variables without incurring an unnecessarily high message passing overhead. The shared co-ordinate variables, *co_ord*, are set by the *operator* process, and simply monitored by the three *piston* processes. The *coolant* process can also eavesdrop on the values of the shared co-ordinate variables for its own purpose. Because the shared variables act as a buffer, causing synchronization to be lost, a synchronizing message is needed if the piston processes are required to keep in step. Each piston process claims ownership of a shared interrupt-generating co-ordinate variable to interrupt it as soon as the co-ordinate reaches its

maximum distance (to prevent a data error from causing the tool to be driven into the working platform).

```
Variable integer co_ord[2], boolean manual :   {shared variables}
Define max_distance = table [2800, 3280, 1500] :
Channel to_synchronize[2] :
Channel from_operator, to_operator :
Variable integer i :

Procedure operator =
  Variable integer local_co_ord[2], command[4], i :

  Do in sequence
    From_operator receive command[i], varying i from 0 to 4
    {Calculate local co-ordinates from the tool movement command}
    If
      local_co_ord[i] > max_distance[i], varying i from 0 to 2
        local_co_ord[i] <- max_distance[i]
    Co_ord[i] <- local_co_ord[i], varying i from 0 to 2
    Do once in parallel {non-deterministic order of arrival}
      To_synchronize[i] receive message, varying i from 0 to 2
    If
      local_co_ord[i] = max_distance[i], varying i from 0 to 2
        write["*** alarm *** axis ",i," - switching to manual"]
        writeln
        manual  :


Procedure piston [value integer i] =
  Variable integer old_co_ord, new_co_ord :

  Do once in sequence
    Old_co_ord <- 0
    Do in sequence
      To_synchronize[i] send message
      New_co_ord <- co_ord[i]
      Do
        { Feedback loop which uses the difference
          between old_co_ord and new_co_ord to move
          the tool, compensating for the tool load.}

      Old_co_ord <- new_co_ord
      when co_ord[i] = max_distance[i]
        stop    :
```

```
Procedure coolant =
  { The coolant nozzle has two degrees of freedom, controlled
    by two stepper motors, each of which requires to know the
    direction of motion and the number of steps through which
    the motor should be made to travel.}
  Variable integer local_co_ord[2], i :

  Do in sequence
    local_co_ord[i] <- co_ord[i], varying i from 0 to 2

    {Use local co-ordinates to calculate and transmit
    the direction and number of steps for each motor.}   :


Do once in sequence
  Not manual
  Do once in parallel
    Operator
    Piston [i], varying i from 0 to 2
    Coolant
  when manual
    stop
```

In this solution, the *operator* process will be blocked should it attempt to assign a value to any of the shared co-ordinate variables before a *piston* process has begun executing the *do* loop which claims ownership of that variable. It will remain blocked (in accordance with specification 3-2-4, which disallows any assignments or fetches to be made to an interrupt-generating object until a *set* signal has been received) until all piston processes have begun executing the loops which reference the shared co-ordinates in *when* statements.

5.2 Highly parallel algorithms

Although the primary use of the interrupt-generating object in concurrent programs is as a means of sharing information between a number of parallel processes on a small scale, the notation of HUL can be used to provide a terse form of expressing parallel algorithms for memory intensive data processing applications, with an apparently low order of time complexity. This is done by making use of large vectors of interrupt-generating variables. As an example, the problem is presented of selecting the smallest ten items from a list of N items known to be in the range 1

to 1000. If the items are stored in the integer array *A*, a concurrent HUL algorithm, making use of an auxiliary vector of interrupt-generating variables, *TEST*, could be coded as:

```
Do once in sequence
  count <- 0
  no_of_terms <- 0
  Do once            {find the 10 smallest items}
    Do in sequence
      count <- count + 1
      do once in parallel
        TEST[i] <- count, varying i from 0 to N-1
    when TEST[i] = A[i], varying i from 0 to N-1
      write[A[i]]
      no_of_terms <- no_of_terms + 1
  when no_of_terms = 10
    stop
```

Figure 14 depicts the action of this parallel selection algorithm. The *when TEST[i] = A[i]* statement causes an initialization sequence to be generated which is executed before the first iteration of the loop to find the ten smallest items. This sequence sets the *test* register of each element *TEST[i]* to the value of the expression *A[i]*. These values cannot be altered until the loop terminates. On each pass through the loop, the current value of *count* is assigned in parallel to the *memory* register of each element *TEST[i]*, and an associated interrupt signal is produced should the contents of any of the *test* and *memory* register pairs be found to be equal.

Although this algorithm might at first appear to have a space complexity of $2N$ because of its memory usage, it should be remembered that the N interrupt-generating variables are actually active objects. So the algorithm requires $2N$ parallel processes, N to handle the parallel assignment of *count* to the interrupt-generating objects, and N for the objects themselves. The worst case running time will be recorded when all the values of A are 1000, and the loop (to find the 10 smallest items) is executed 1000 times. If references to interrupt-generating objects are recognized as elementary operations (which they are in HUL), and it is assumed that the messages which result from the references to these objects will not be blocked (which is true in this instance, since the parallel selection process has itself claimed ownership of all *TEST[i]* cells) and will take a fixed time to execute, then the loop can be regarded as containing a constant

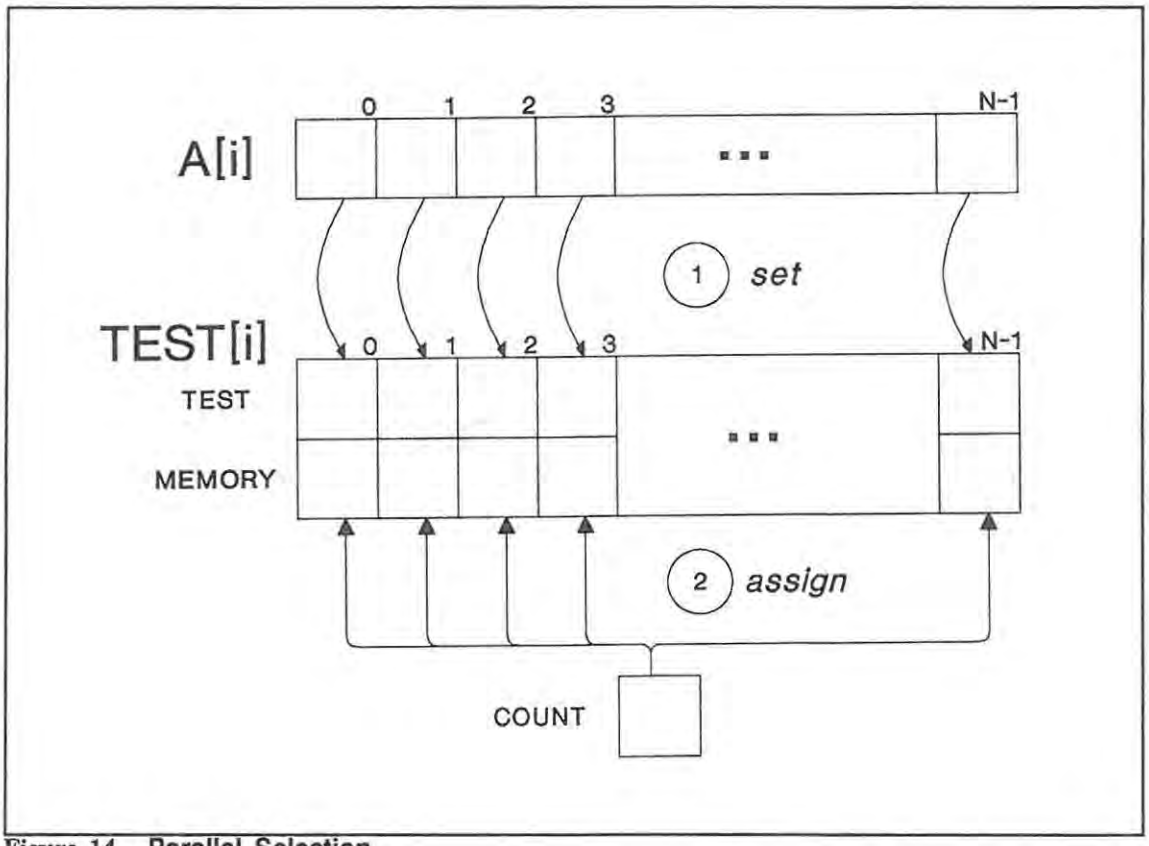


Figure 14. Parallel Selection.

number of elementary instructions. The actions resulting from the *when* statements may be disregarded, since they too are elementary instructions which terminate in a short, finite time. Included in the constant number of instructions within the loop (to find the 10 smallest items) is a parallel control structure which executes N parallel processes to assign the value of *count* to the elements of *TEST*. This yields a worst case time complexity for the loop of $1000C_1N$, if it is executed on one processor. The *test* registers of the interrupt-generating vector are set at the commencement of this loop by N parallel *set* messages, making the total time complexity for the algorithm executing on one processor $C_2N + 1000C_1N$, which can be simplified to $2CN$ for the case $C_2 << 1000C_1 = 2C$.

For P processors, a time complexity of $O(N/P)$ can be expected, and in the case when $P = 2N$, a worst case running time $2CN/P$ suggests a lower bound time complexity $\Omega(1)$. The processor-complexity product measure of $O(1)*O(N) = O(N)$ matches the lower bound on this algorithm's

equivalent sequential-time complexity.

The above algorithm suffers form the common problem challenging good performance on distributed parallel architectures, the difficulty of getting the correct data to the proper place in the desired time. Although this algorithm reduces time complexity at the expense of an increased number of processors, a distributed solution would require a good deal of work to be done by the system which supports the language. The large number of messages exchanged is complicated by the fact that physical processors have an upper limit to the number of communications links available to them, necessitating the use of funnel processes or the rerouting of messages[1]. The assumption that unblocked message passing instructions will take a fixed time to execute is not valid, since the system which currently implements distributed HUL does not enjoy a linear speedup factor as the number of processors is increased, primarily as a result of message rerouting. For the above algorithm, the processor-complexity product of $O(N)$ is subject to a large big-O constant, which is at least as great as the interval required for a message to be routed between the furthest processors apart in the network. To illustrate the extent of this low level activity, a second algorithm is presented.

An example often used to demonstrate concurrent programming languages which make use of message passing primitives to exploit distributed processing architectures, is that of a binary parallel search [BIT 84]. Figure 15 shows the principle by which a tree of processes can be used to break down the search of a large data set into a collection of searches with smaller scope that can be undertaken simultaneously. Each leaf node in the tree of processes interrogates a single element in the data set (though possibly a multifarious one representing a collection of data). The interior branch nodes are used to route the request down to the leaf nodes, and to route the result back to the root.

A message passing version of this search algorithm hinges around the actions of the two types of node in the tree. Each branch node requires six channels to communicate with its parent node (*request* and *result* in the code below) and its two child nodes (*left_request*, *left_result*,

[1]The rerouting of messages in the current distributed HUL implementation is discussed in section 6.3.

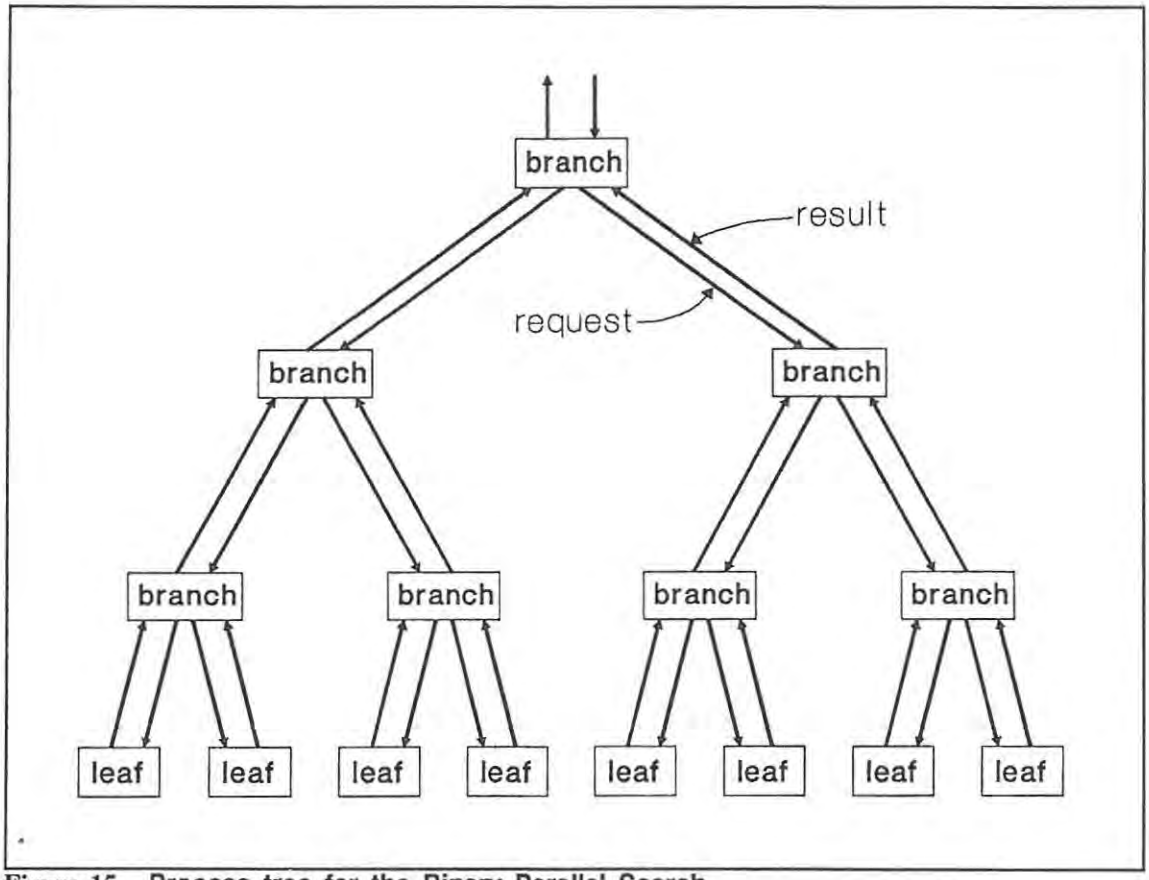


Figure 15. Process tree for the Binary Parallel Search.

right_request and *right_result*), and can be described by the HUL procedure:

```
Procedure branch [channel request,result,left_request,left_result,
                                        right_request,right_result] =

  Variable integer key, boolean result_1,result_2 :

  Do in sequence
    Request receive key
    Do once
      left_request send key
      right_request send key
    Do once
      left_result receive result_1
      right_result receive result_2
    Result send (result_1 or result_2)   :
```

The leaf node requires only two channels, one to obtain a request from its parent node, and the other to return a result. This process may be described by the HUL procedure:

```
Procedure leaf [channel request,result] =

  Variable integer key, data :

  Do in sequence
    Load [data]     {implementation specific}
    Do once in sequence
      Request receive key
      Result send (key = data)   :
```

A considerable amount of channel activity is clearly visible in this solution. N leaf processes and N-1 branch processes need to be initiated in parallel, with $2N$-1 request channels and $2N$-1 result channels mapped as actual parameters onto the appropriate procedures.

An alternative solution, rather different in concept to the one above, can be written in HUL, making use of the interrupt-generating facility. This solution is more compact and efficient than the one which uses explicit message passing, but also induces a considerable amount of low-level activity, which is not immediately obvious, to support inter-processor communication.

The N leaf processes can be retained in the modified form:

```
Procedure leaf [value integer i] =

  Variable integer data :

  Do once in sequence
    Load [data]
    TEST[i] <- data  :
```

By using separately declared leaf processes, each with its own local workspace, this example avoids the awkward conceptual problem which exists in the parallel selection algorithm, that of deciding how to distribute the elements of the static integer array across a fully distributed multiprocessor system. An N-element auxiliary vector of interrupt-generating variables, *TEST*, can be used to code the operation in HUL as:

```
Do once in sequence
  Request receive key
  Not found
  Do once                {perform the search}
    Do once
      Leaf[i], varying i from 0 to N-1
    when TEST[i] = key, varying i from 0 to N-1
      Found
  Result send found
```

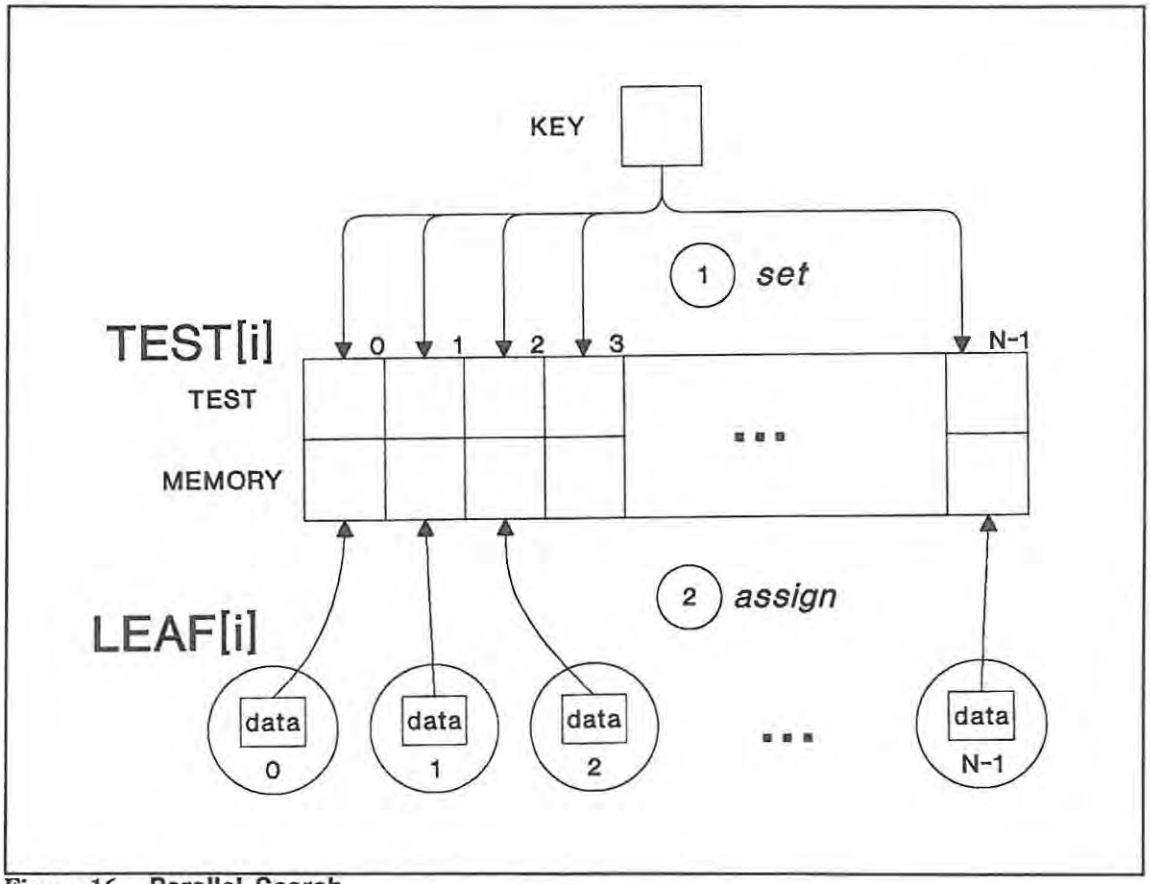


Figure 16. Parallel Search.

Figure 16 shows how the *when TEST[i] = key* statement causes an initializing sequence to set the *test* register of each element of *TEST* to the value of *key* before execution of the *do* control structure which performs the search. All *N* instances of the leaf procedure are then executed as concurrent processes, to assign their *data* values in parallel to the corresponding *memory*

registers of the *TEST[i]* cells. A Boolean result[1] is returned on termination of this sequence. The product of the processor and time complexities yields $O(N)$ for this solution, as opposed to $O(N\log N)$ for the earlier message passing version. Whereas the interrupt-generating notation is more concise, the processor and message passing complexities are of the same order in both versions.

5.3 Realizing conventional shared variables

To allow for the maximum degree of truly concurrent execution of a program, the compiler ensures that all component processes of a parallel control structure (child processes) are written as potentially distributable object modules. They are therefore not allowed to share a global variable definition, unless it is an interrupt-generating variable.

To illustrate this point, consider the following HUL program:

```
Variable integer shared_data :
Channel from_somewhere :
Do once in sequence
  From_somewhere receive shared_data
  Do once in parallel
    {Process 1 which makes use of shared_data's value}
    {Process 2 which makes use of shared_data's value}
```

(5-3-1)

The above program will cause an error during one of the compilation phases, since the global variable cannot be referenced by both parallel processes. However, if the program is rewritten as

[1]The index of the element containing the key could be returned in place of a simple Boolean result in both the message passing and the interrupt-generating solutions.

```
Channel from_somewhere, start :
Do once in parallel
  Variable integer shared_data :
  Do once in sequence                          {Process 1}
    From_somewhere receive shared_data
    To_start send shared_data
    {Code which makes use of shared_data's value}

  Variable integer shared_data :
  Do once in sequence                          {Process 2}
    To_start receive shared_data
    {Code which makes use of shared_data's value}
```

(5-3-2)

then no global variable exists and the two child processes can be distributed onto separate processors. The program now has the potential to be configured for a multiprocessor network of two processors. The compile-time checks which restrict the use of global variables ensure that the programmer is not able to develop a program which might not be configurable for such an environment. For this reason, the programmer will be forced to write code in the form of program 5-3-2, even though program 5-3-1 is a far more lucid expression of the algorithm.

It may be worthwhile in cases such as this to declare a global variable for use as an interrupt-generating object to preserve the style of program 5-3-1, even though no explicit interrupt signalling is needed. Although its presence seems rather inept, a dummy *when* statement must be included to conform to the semantic rules of HUL, as illustrated in the modified form of program 5-3-1 below:

```
Variable integer shared_data :
Channel from_somewhere :
Define never = -999 {some unlikely value} :
Do once in sequence
  From_somewhere receive shared_data
  Do once in parallel
    {Process 1 which makes use of shared_data's value}
    {Process 2 which makes use of shared_data's value}
  when shared_data = never     {dummy interrupt}
    skip
```

5.4 Simulating dynamic process creation

When a process which resides within a parallel control structure is replicated, the number of processes created is one more than the difference between the final and the initial expressions of the replicator statement. In order for the compiler to know how many processes are contained within the program, HUL restricts the final and initial values of the replicator source code to expressions which can be calculated at compile-time. They often take the form of a constant. Although this restriction is significant, it enables memory allocation and queue length calculations to be undertaken prior to execution, and makes it possible to employ static process allocation methods to distribute processes automatically across the processor network in support of the ancillary aim of this thesis. HUL's implementation allays the adverse effect of this restriction for applications in which the maximum possible number of processes is known at compile-time. For example, the successful compilation of the program segment

```
Input ? N
Do in parallel
  Predefined_process, varying I from 1 to N
```

will be blocked in existing implementations, because the value of N is not known at compile-time; but the following HUL code segment would be acceptable to the compiler.

```
Define max = 200 :

Procedure new_predefined_process[Variable integer I] =
  If
    I <= N
      predefined_process
    otherwise
      skip       :

Do in sequence
  Input ? N
  Do in parallel
    New_predefined_process[I], varying I from 1 to max
```

The maximum number of processes is always generated, but those not required (when I exceeds

N) are only *skip* processes that terminate immediately[1]. Although the implementation is unable to detect which processes will be trivial, the only overhead incurred will be the protocol for initiating processes.

5.5 Deadlock and infinite postponement

The state in which it is impossible for any process in a group of processes to proceed, known as *deadlock*, is one of the most serious error conditions which can arise in a working system. The testing of software rarely removes other than the most obvious deadlocks. Subtle deadlocks may occur infrequently, but with devastating results.

Although it is not practical to design a concurrent programming language in which it is impossible to construct deadlocks, interrupt-generating active data object provides inherent protection for the programmer. The ownership rule of interrupt-generating objects has been designed to enable deadlock free access of shared variables by several processes (see section 3.2). While strict enforcement of the ownership rule avoids contention in the use of the interrupt signalling mechanism, the programmer is not protected from allocating ownership of a variable to one process for the duration of the program, and, in so doing, causing infinite postponement in other processes which attempt to gain ownership of the same variable[2].

In the syntax of HUL, inherent parallelism is intended to prevent message passing deadlock by always anticipating a non-deterministic ordering of input and output messages. The sequential ordering of synchronized input and output messages provides classic grounds for deadlock. Consider two processes which exchange data via channels *chan_1* and *chan_2*. The following order of the communication pairs will lead to both processes being suspended indefinitely:

---

[1] It would be advisable, in this example, to include code to detect the possibility of a value of *N* being received which exceeds *max*.

[2] This property can be usefully exploited, as is described in section 5.7.

```
{process 1}
Do once in sequence
  chan_1 send A
  chan_2 receive B
```

```
{process 2}
Do once in sequence
  chan_2 send Y
  chan_1 receive X
```

Process 1 cannot proceed until process 2 has read from *chan_1*; process 2 cannot do this until it has written to *chan_2*, which it cannot do until process 1 has read from *chan_2*. The problem is easily solved in HUL by omitting the *in sequence* qualifier.

It is desirable for designers of a program to illustrate or prove that the possibility of deadlock has been removed from their software. In message passing languages such as HUL, CSP [HOA 85] is a useful formalism for proving the absence of deadlock. Section 5.7 presents the HUL solution of the Dining Philosophers' problem, and uses the notation of CSP to prove that the solution avoids deadlock.

A situation as acute as deadlock, termed *livelock*, arises when interacting processes are still executing, but are inhibited from proceeding. A typical example of livelock is a group of interacting processes which are stuck in a loop in which they are doing no useful work. The use of an *otherwise* option in an *if* control structure which is imbedded within an infinite loop is particularly prone to livelock in HUL. A busy wait loop, such as the one below, will livelock if no other process commits itself to sending a value to channel *waiting_for_input*.

```
Do  {forever}
  If
    Waiting_for_input ready
      Waiting_for_input receive N
      Do_something_with_it
    Otherwise
      skip
```

This code has no advantage over the ordinary input primitive.

5.6 Mutual exclusion

Mutual exclusion is frequently necessary when allowing simultaneous access to a resource by concurrent processes. The interrupt-generating active data object provides inherent exclusive access to simple data items. Mutual exclusion can be neatly handled in the wider features of HUL by including in the same process all critical code sections which reference the shared resource, and which should not be executed simultaneously. If process *P* is the only process which can access a resource, exclusive access to the resource is ensured as long as only one instance of process *P* is allowed to exist. Condition synchronization is used when a process wishes to perform an operation that can only sensibly, or safely, be performed if another process has undertaken some action or is in some defined state. Condition synchronization is naturally supported with shared interrupt-generating active data objects.

The bounded buffer producer-consumer solution [BEN 82] listed below illustrates mutual exclusion and condition synchronization. Instead of using procedure-parameter programming, message passing is used to synchronize with a special *warehouse* process which controls a buffer and all references to it. This process must make sure that storage and retrieval never occur simultaneously. The *warehouse* process gives priority to the receiving code (putting the customer first is sound business policy), and uses the channels *request*, *producer*, and *consumer* to synchronize with the producer and consumer processes.

In this version of the producer-consumer problem, the buffer is modelled as a simple stack which transmits characters of the alphabet from the producer to the consumer. Rather than declaring *top* globally as was suggested in section 5.1, an interrupt-generating Boolean variable, *warehouse_full*, is used to interrupt the producer when the stack is full. Because of the exclusive ownership rule of interrupt-generating variables, one such variable cannot interrupt both the consumer and the producer, so a second interrupt-generating variable, *warehouse_empty*, is used to interrupt the consumer.

```
{ Producer-Consumer.
  In this example, not all characters produced will be printed
  as the program can terminate with characters in the buffer.}
```

```
Alternate becomes = <-, whenever = <- : {use "becomes" and
                      "whenever" as assignment operators}
Define stack_max = 10 :
Variable boolean warehouse_empty, warehouse_full,
               finished {terminator} :
Channel producer, consumer, request :

Procedure warehouse =
  Variable character stack [stack_max] :   {0..stack_max}
  Variable integer top :
  Do once in sequence
    Top becomes -1
    Warehouse_empty
    Not warehouse_full
    Do in parallel {repeat}
      Warehouse_empty whenever top = -1
      Warehouse_full whenever top = stack_max
      If
        request ready
          Request receive message
          Consumer send stack [top]
          Top becomes top - 1
        producer ready
          Top becomes top + 1
          Producer receive stack [top] :



Procedure producer_process =
  {This process produces letters of the alphabet.}
  Define alphabet = "abcdefghijklmnopqrstuvwxyz" :
  Variable integer i :
  Do once in sequence
    i becomes 1
    Do in sequence
      Producer send alphabet [i]
      i becomes (i + 1) mod 27
      when warehouse_full
        wait :
```

```
Procedure consumer_process =
  { This process consumes letters of the alphabet and writes
    them to the screen.}
  Variable character item :
  Do in sequence
    Request send message  {see footnote[1]}
    Consumer receive item
    Screen send item
    when warehouse_empty
      wait :


Procedure terminate =
  {Set the flag to terminate the program when a key is pressed.}
  Do in sequence
    Keyboard receive message
    Finished  :



Do once in sequence {main program}
  Not finished
  do once in parallel
    warehouse
    producer_process
    consumer_process
    terminate
  when finished
    stop
```

A second version of the bounded buffer producer-consumer program is listed below in a more concise form of HUL for comparison with the previous solution This version includes code to empty the buffer before terminating. To do this, the producer and consumer processes are terminated before the warehouse process by placing them in a nested parallel control structure.

[1]Note that the consumer has to ask before it receives, because the *ready* function is only defined for input processes. This corresponds to the use of input guards in Occam and CSP.

```
{ Producer-Consumer.}
Def stack_max = 10 :
Var bool warehouse_empty, warehouse_full :
Var bool finished, kill_program : {terminators}
Chan producer, consumer, request, clear_warehouse :


Do once seq
  Not finished
  Not kill_program
  Do once par  {outer parallel control structure}


{------------    process 1 of 3 parallel processes    -----------}

    Var char stack [stack_max], int top :
    Do once seq {warehouse}
      Top <- -1
      Warehouse_empty
      Not warehouse_full
      Do in parallel {repeat}
        Warehouse_empty <- top = -1
        Warehouse_full <- top = stack_max
        If
          request ready
            Request ? message
            Consumer ! stack [top]
            Top <- top - 1
          producer ready
            Top <- top + 1
            Producer ? stack [top]
          clear_warehouse ready
            Clear_warehouse ! stack[top]
            Top <- top - 1

{--------------------    process 2    --------------------------}

    Do once par     {nested parallel control structure}
      Def alphabet = "abcdefghijklmnopqrstuvwxyz" :
      Var int i :
      Do once seq  {producer}
        i <- 1
        Do seq
          Producer ! alphabet [i]
          i <- (i + 1) mod 27
          when warehouse_full
            wait
```

```
      Var char item :
      Do seq       {consumer}
        Request ! message
        Consumer ? item
        Screen ! item
        when warehouse_empty
          wait
      when finished
        stop  {producer and consumer}

{--------------------    process 3    ---------------------------}

    Var char item :
    Do seq  {await termination signal}
      Keyboard ? message
      Finished
      writeln
      write ["The following items are left in the warehouse: "]
      do intact seq           {intact ensures completion}
        clear_warehouse ? item
        write [item]
        when warehouse_empty  {This process can claim ownership}
          writeln             {of warehouse_empty once consumer}
          Kill_program        {has terminated}

{-------------------------------------------------------------------}

  when kill_program  {interrupt outermost process}
    stop
```

The use of buffers to decrease the coupling between active processes is a common feature of systems written in a concurrent programming language. If the buffer itself, and its associated pointers, are open to concurrent access, then it is necessary to provide mutual exclusion to give a reliable implementation. Moreover, two condition synchronizations also apply to buffers, those needed to stop a process reading from an empty buffer or writing to a full one[1].

[1]The producer-consumer solution presented above uses two such condition synchronizations, *warehouse_full* and *warehouse_empty*.

In languages which make use of shared variables, mutual exclusion is an important class of synchronization for preventing simultaneous access. A more complex example, the readers and writers problem [GEH 84b], illustrates simultaneous and exclusive access. Several processes, which only read data, do not require exclusive access to the shared resource, in this case a buffer. However, any process which alters the data in the buffer must be given exclusive access.

Since the critical sections of code in which the reader processes access the buffer may be executed concurrently, it is logical that they remain within the reader processes. A process to control access to the buffer is still necessary so that when the writer requests access, new reader processes are made to wait until the modification to the buffer is complete. Since a writer demands exclusive access, its critical section can be included in the process which controls access to the buffer.

In the example below, there are three readers and one writer executing concurrently and accessing the buffer. The readers cycle through the buffer, displaying the characters contained in its elements. The writer updates specific elements of the buffer, and is given preference over requests for access from readers.

```
{ Readers and Writers.}
Ignore a_terminator_has :
Define dummy = 0 :      {some unlikely value}
Channel request_read [2], end_read [2] :
Channel exclusive_write :
Variable character buffer [4] :    {will store characters in 0..4}
Variable boolean A_terminator_has been_typed :   {A...has ignored}
Variable integer i :


Procedure control_buffer_access =
  Variable boolean writing :     {set if write process wants access}
  Variable integer no_of_readers :   {number of readers with access}
  Variable boolean ok_to_write :              {interrupt-generating}
  Variable integer i, index :
```

```
  Do once in sequence
    Not writing
    No_of_readers <- 0

    If
      exclusive_write ready   {writer requests access}
        Exclusive_write ? Index
        Writing               {hold up any future read requests}
        Do      {allow currently active read processes to finish}
          If
            end_read [i] ready, varying i from 0 to 2
              End_read [i] ? message
              No_of_readers <- no_of_readers - 1
              Ok_to_write <- no_of_readers = 0
          When ok_to_write
            Exclusive_write ? Buffer [index]   {critical section}
            Not writing                        {end of write}
            Stop

      end_read [i] ready, varying i from 0 to 2
        End_read [i] ? message    {End of reader access}
        No_of_readers <- no_of_readers - 1

      not writing and request_read [i] ready, varying i from 0 to 2
        Request_read [i] ? message    {Request for reader access}
        No_of_readers <- no_of_readers + 1 :

Procedure read_from_buffer [ value integer i ] =
  {instance i of this process successively reads each
  element of the buffer and writes it to the screen}
  Variable integer index, character element :
  Do once in sequence
    Index <- i
    Do in sequence
      Request_read [i] ! message
      Element <- buffer [index]  {critical section}
      End_read [i] ! message
      Write [element]
      Index <- (index + 1) mod 5 :
```

```
Procedure write_to_buffer =
  {Allow the user to type a digit and a character.  Store the
   character in the buffer element numbered by digit.}
  Variable integer index, character ch :

  Do in sequence
    Keyboard ? ch                  {type buffer index '0'..'4'[1]}
    Index <- ch - '0'              {ascii to numeric conversion}
    Keyboard ? ch                  {character to store in buffer}
    Exclusive_write ! index
    Exclusive_write ! ch
    When ch = '*'
      a_terminator_has been_typed  :


Do once in sequence          {readers and writers}
  A_terminator_has not been_typed
  Buffer [i] <- '0' + i, varying i from 0 to 4
                             {set initial buffer values }
  Do once in parallel
    Read_from_buffer [i], varying i from 0 to 2     {3 instances}
    Write_to_buffer
    Control_buffer_access
  When buffer[i] = dummy, varying i from 0 to 4
    skip                     {see footnote[2]}
  When a_terminator_has been_typed
    stop
```

Notice that procedure *control_buffer_access* controls access to the buffer, but only references the buffer on the writer's behalf, not on behalf of the readers. A shared buffer must be implemented. The action of *no_of_readers* is similar to that of a semaphore, forcing the writer to wait until all current reading access has ceased.

The interrupt-generating model of HUL has been designed with built-in exclusive access, which

---

[1] It would be advisable to include a check to validate this data.

[2] To enable the buffer to behave as shared memory in a distributed environment, the array *buffer* is declared globally and each element is placed in a *when* statement in the parent process, so that it is implemented as an interrupt-generating object.

makes truly simultaneous references to a shared interrupt-generating variable impossible. The model also features an exclusive ownership rule for interrupt purposes. The use of this property as a tool for mutual exclusion is discussed in section 5.7.

5.7 Exploiting the ownership rule of interrupt-generating active data objects

Once a programmer has fully understood the properties of interrupt-generating active data objects and their interaction with the *do* control structure in a distributed environment, it is possible for him or her to make use of the ownership rule as an exclusion facility[1]. It must be emphasized that this application of HUL's high level interrupt facility is not intended to be an illustration of easily readable program code. The use of interrupt-generating variables without assigning values to them in the program listing below does not exhibit the kind of natural expression which HUL tries to encourage, but does provide a concise notation for mutual exclusion.

The example presented below is Dijkstra's Dining Philosophers' problem [DIJ 68], in which each philosopher is a process and each fork is an interrupt-generating Boolean variable, shared between two processes. The solution has the form:

```
Variable integer fork [4] :

Procedure simulate_philosopher [value integer i] =
  Do in sequence
    Think
    Do once in sequence  {claim exclusive ownership of forks}
      Eat
      when fork [i]
        skip
      when fork [(i+1) mod 5]
        skip
    {release exclusive ownership of forks} :
```

Although no data is passed between the philosopher processes via the shared fork variables, only

[1] Law 4-5-10 of section 4.5 is applied.

one process may claim ownership of a fork at any time. Possession of a fork variable by one philosopher locks out other philosophers until ownership of the shared fork variable has been relinquished.

In the source code, no mention is made of the fork variables until after the code which represents the philosopher eating; it should be remembered that ownership of a variable mentioned in a *when* statement is claimed at the start of a control structure and relinquished when the control structure terminates.

The full text of the HUL solution to the dining philosophers' problem is listed below, using the ownership rule of interrupt-generating objects as an exclusion facility.

```
Variable boolean fork [4], in_room [4] :
Variable integer k, boolean time_to_stop :
Channel start_philosophers :


Procedure simulate_philosopher [value integer i] =

  Procedure think =
    Do once in sequence
      write ["philosopher ", i, " thinks"]
      writeln :

  Procedure eat =
    Do once in sequence
      write ["philosopher ", i, " eats"]
      writeln :

  Do in sequence
    Think
    Do once  {gain entry to the room}
      Do once in sequence  {establish exclusive ownership of forks}
        Eat
        when fork [i]
          skip
        when fork [(i+1) mod 5]
          skip
      when in_room [i]
        skip :
```

```
Procedure Simulate_doorman =
  Variable integer i :
  Channel synchronize[1] : {channels to synchronize the parallel
                            processes within the doorman procedure}
  Procedure claim_room[value integer i,channel to_other,from_other]=
    Do once in sequence  {claim ownership of in_room[i]}
      To_other send message
      From_other receive message
      when in_room [i]
        skip  :

  Do once in sequence
    i <- 0
    Do {two parallel processes}
      Do {forever} in sequence
        Claim_room [i, synchronize[0], synchronize[1]]
        i <- (i+1) mod 5
      Do once in sequence
        Synchronize[0] receive message  {offset the message order}
        Start_philosophers send message {start philosopher processes}
        i <- 1
        Do {forever} in sequence
          Claim_room [i, synchronize[1], synchronize[0]]
          i <- (i+1) mod 5 :

Procedure await_terminator =
  { when a '*' is typed, all activity ceases immediately }
  Variable char ch :
  Do in sequence
    keyboard receive ch
    if
      ch = '*'
        time_to_stop :

Do once in sequence
  Not time_to_stop
  Do once in parallel
    Simulate_doorman
    Do once in sequence
      {Allow the doorman to claim ownership of an in-room
                  variable before the philosophers begin}
      Start_philosophers receive message
      Do once in parallel
        Simulate_philosopher [k], varying k from 0 to 4
        Await_terminator
  when time_to_stop
    stop
```

The popular means of avoiding deadlock (which might occur were each philosopher to claim ownership of one fork), the introduction of a doorman process to allow at most N-1 philosophers into the dining room [HOA 85], has also been implemented using the ownership rule of interrupt-generating variables. Each philosopher process shares an interrupt-generating Boolean variable, *in_room[i]*, with the doorman process. By ensuring that he always owns one of the *in_room* variables, the doorman guarantees that at least one philosopher is kept out of the room. The order of claiming ownership of room and fork variables by the philosopher process is significant. To prevent philosophers from picking up forks before they have been allowed into the dining room, the code which claims ownership of the forks is contained within the control structure which gains entrance to the room.

The doorman process is more complicated than might at first be imagined. To prevent the N^{th} philosopher from sneaking into the dining room between the doorman relinquishing one *in_room* variable and claiming the next, the doorman process is written as two parallel processes which synchronize using messages to ensure that the first of these processes does not relinquish ownership of an *in_room* variable before the second has managed to claim ownership of one. The action of the doorman is depicted in figure 17. To ensure that the doorman is able to claim at least one *in_room* variable before the philosophers begin their monotonous life, some synchronizing code must be included (such as the message from the doorman in the code above).

Since the possibility of deadlock is seldom obvious, particularly in programs which are as enigmatic as the one above, a programmer should be able to prove that a program is free from deadlock. It is possible to prove that the doorman process prevents deadlock by using the formalism of CSP [HOA 85] to identify the set of events in which each parallel process in the program may engage. Because the events which involve only one process cannot affect the interaction between processes, such events (for example *eat* and *sleep* for the philosopher process) can be ignored for the purpose of this proof. The specifications below are confined to events which involve the simultaneous participation of a number of processes.

Each philosopher process can be described by the following succession of events:

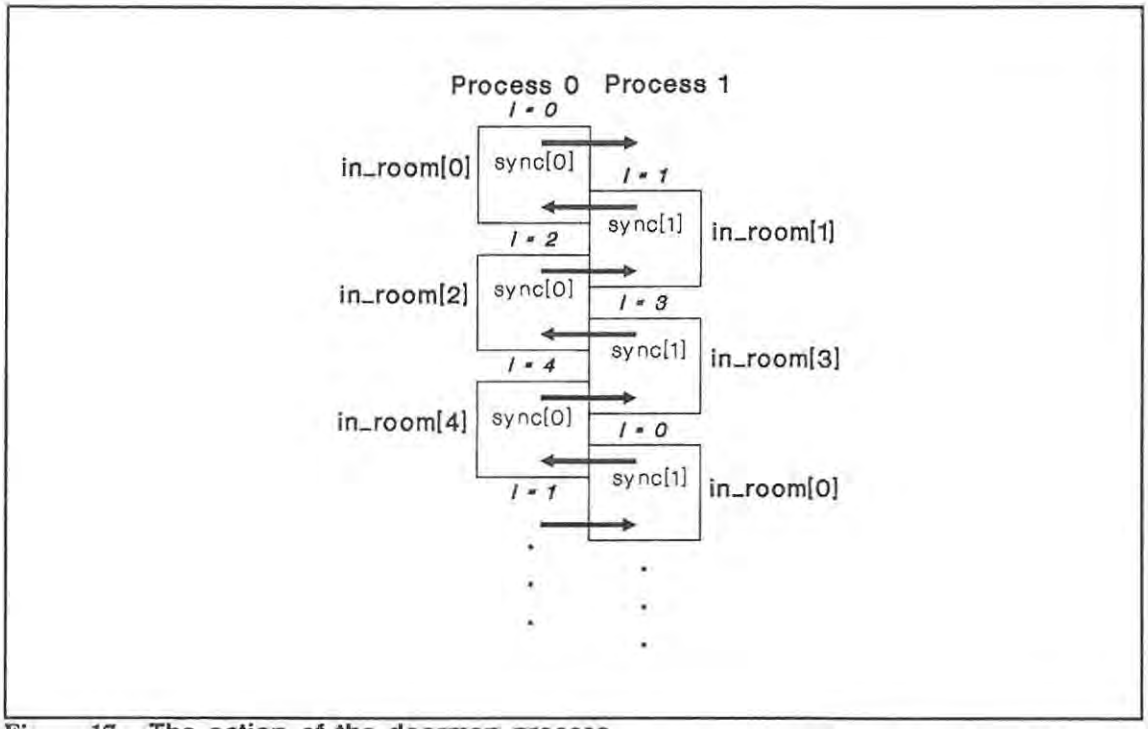


Figure 17. The action of the doorman process.

$$PHIL_i = (i.claims.in\text{-}room_i \rightarrow (i.claims.fork_i \rightarrow i.claims.fork_{(i+1) \bmod 5} \;[]\; i.claims.fork_{(i+1) \bmod 5} \rightarrow i.claims.fork_i) \rightarrow (i.releases.fork_i \rightarrow i.releases.fork_{(i+1) \bmod 5} \;[]\; i.releases.fork_{(i+1) \bmod 5} \rightarrow i.releases.fork_i) \rightarrow i.releases.in\text{-}room_i \rightarrow PHIL_i) \qquad \text{for } 0 \leq i \leq 4$$

The picking up (and putting down) of the two forks may occur in either order, but no fork may be claimed until the philosopher has gained access to the room (by claiming a corresponding *in_room* variable).

Because none of the philosophers assign values to (or fetch values from) the fork variables, the interrupt-generating processes which implement them can be described simply as:

$$FORK_i = (i.claims.fork_i \rightarrow i.releases.fork_i \rightarrow FORK_i \quad \text{for } 0 \leq i \leq 4$$
$$[] \ ((i+1) \bmod 5).claims.fork_i \rightarrow ((i+1) \bmod 5).releases.fork_i \rightarrow$$
$$FORK_i)$$

The doorman process ensures that at least one philosopher is kept out of the dining room, to prevent all five philosophers claiming one fork in quick succession before any of them is able to claim a second fork as well.

$$DOOR = (i := 0 \rightarrow$$
$$(\mu X.((i: doorman.claims.in\text{-}room_i \rightarrow doorman.claims.in\text{-}room_{(i+1) \bmod 5} \rightarrow$$
$$doorman.releases.in\text{-}room_i) \rightarrow i := i+1 \rightarrow X)$$
$$\| \ (doorman.claims.in\text{-}room_0 \rightarrow start.philosophers \rightarrow i := 1 \rightarrow$$
$$\mu Y.((i: doorman.claims.in\text{-}room_i \rightarrow doorman.claims.in\text{-}room_{(i+1) \bmod 5} \rightarrow$$
$$doorman.releases.in\text{-}room_i) \rightarrow i := i+1 \rightarrow Y)))$$

Ignoring the events which assign values to i (which are of interest only to *DOOR*), a typical trace of *DOOR* would be:

$$doorman.claims.in\text{-}room_0 \rightarrow start.philosophers \rightarrow doorman.claims.in\text{-}room_1 \rightarrow$$
$$doorman.releases.in\text{-}room_0 \rightarrow doorman.claims.in\text{-}room_2 \rightarrow doorman.releases.in\text{-}room_1 \rightarrow$$
$$doorman.claims.in\text{-}room_3 \rightarrow doorman.releases.in\text{-}room_2 \rightarrow \ \ldots$$

The processes which describe the *IN-ROOM* variables have a similar specification to the forks.

$$IN\text{-}ROOM_i = (i.claims.in\text{-}room_i \rightarrow i.releases.in\text{-}room_i \quad \text{for } 0 \leq i \leq 4$$
$$\rightarrow IN\text{-}ROOM_i$$
$$[] \ doorman.claims.in\text{-}room_i \rightarrow doorman.releases.in\text{-}room_i$$
$$\rightarrow IN\text{-}ROOM_i)$$

The entire system is described by

$$
\begin{aligned}
PHILOSOPHERS &= (PHIL_0 \parallel PHIL_1 \parallel PHIL_2 \parallel PHIL_3 \parallel PHIL_4 \parallel \\
&\quad FORK_0 \parallel FORK_1 \parallel FORK_2 \parallel FORK_3 \parallel FORK_4) \\
DOORMAN &= (DOOR \parallel IN\text{-}ROOM_0 \parallel IN\text{-}ROOM_1 \parallel IN\text{-}ROOM_2 \parallel \\
&\quad IN\text{-}ROOM_3 \parallel IN\text{-}ROOM_4) \\
SYSTEM &= ((start.philosophers \rightarrow PHILOSOPHERS) \parallel DOORMAN)
\end{aligned}
$$

To prove the absence of deadlock, it is necessary to show that there is at least one event by which a legal trace can be extended in all cases. The specification for *DOOR* is such that this process will never own less than one and more than two *IN-ROOM* variables. This ensures that the number of philosophers in the room can never exceed four. If the number is less than four, there is no deadlock because a philosopher not in the dining room may engage in the event of entering the room, or the doorman may engage in the event of releasing an *IN-ROOM* variable. If the number is equal to four then two possibilities exist: If all five forks have been claimed, then at least one of the four philosophers in the room must be able to eat and will soon engage in an *i.releases.fork* event. Since a philosopher may pick up his forks in either order, and since all fork positions are within reach of one of the four philosophers in the dining room, if there is an unclaimed fork, a philosopher seated next to it will be able to engage in an *i.claims.fork* event. Hence the doorman process is successful in preventing deadlock.

For comparison with the HUL solution to the dining philosophers' problem, in which the ownership rule of interrupt-generating variables is used as an exclusion facility, the following conventional message passing solution is presented. This version also makes use of a doorman process to prevent deadlock. The doorman process can be made simpler using message passing, since the programmer has explicit control over the order in which rendezvous occur, at the expense of having to deal with all messages in the source program.

```
Channel grab_left_fork [4], grab_right_fork [4] :
Channel release_left_fork [4], release_right_fork [4] :
Channel try_to_enter [4], try_to_leave [4] :
Variable integer j,k, boolean time_to_stop :
```

```
Procedure simulate_philosopher [value integer i] =

  Procedure think =
    Do once in sequence
      write ["philosopher ", i, " thinks"]
      writeln :

  Procedure eat =
    Do once in sequence
      write ["philosopher ", i, " eats"]
      writeln :

  Do in sequence
    Think
    Try_to_enter [i] ! message
    Grab_left_fork [i] ! message
    Grab_right_fork [i] ! message
    Eat
    Release_left_fork [i] ! message
    Release_right_fork [i] ! message
    Try_to_leave [i] ! message :



Procedure control_possession_of_fork [value integer i] =
  Do                  {fork i is philosopher i's right fork}
    If                {fork i is philosopher i+1's left fork}
      grab_right_fork [i] ready
        Grab_right_fork [i] ? message     {philosopher i has fork}
        Release_right_fork [i] ? message  {until he puts it down}
      grab_left_fork [(i+1) mod 5] ready
        Grab_left_fork [(i+1) mod 5] ? message
        Release_left_fork [(i+1) mod 5] ? message :
```

```
Procedure simulate_doorman =
  {this process allows a maximum of 4 people into the dining room
   at one time.}
  Variable integer philosophers_in_room, i :
  Do once in sequence
    Philosophers_in_room <- 0
    Do
      If
        philosophers_in_room < 4  and  try_to_enter [i] ready,
                                       varying i from 0 to 4
          Try_to_enter [i] ? message
          Philosophers_in_room <- philosophers_in_room + 1
        try_to_leave [i] ready, varying i from 0 to 4
          Try_to_leave [i] ? message
          Philosophers_in_room <- philosophers_in_room - 1 :


Procedure await_terminator =
  { when a '*' is typed, all activity ceases immediately }
  Variable char ch :
  Do in sequence
    keyboard ? ch
    if
      ch = '*'
        time_to_stop :


Do once in sequence
  Not time_to_stop
  Do once in parallel
    Simulate_philosopher [k], varying k from 0 to 4
    Control_possession_of_fork [j], varying j from 0 to 4
    Simulate_doorman
    Await_terminator
  when time_to_stop
    stop
```

5.8 Simulating conventional loops

HUL, in which the only loop control structure is one for interacting with interrupt-generating active data objects, requires a programming style which is rather different to that of high level languages with more conventional loop structures. The *do* control structure is better suited to programming communicating sequential processes than stand-alone sequential code; nevertheless, the two primary forms of the *do* control structure are similar in concept to conventional programming structures for conditional loops and counted loops[1].

The interruptible *do* control structure of HUL is similar in form to the Modula-2 *LOOP* control structure used in conjunction with an *EXIT* statement [WIR 85b].

```
(* Modula-2 *)
LOOP
  (* some statements *)
  IF SomeCondition THEN
    EXIT
  END;
  (* some statements *)
END
```

```
{ HUL }
Do in sequence
  { some statements }
  If
    some-condition
      Exit  { set a boolean }
  { some statements }
  When exit
    stop
```

The *do* control structure with an iteration qualifier models a conventional FOR loop without a control variable. If the control variable is needed, it must be explicitly implemented.

```
(* Modula-2 *)
FOR I := 1 TO N DO
  (* some statements *)
END
```

```
{ HUL }
I <- 1
Do N times in sequence
  { some statements }
  I <- I + 1
```

Direct translations into HUL of other conventional programming loop structures, such as the Modula-2 *WHILE* and *REPEAT* loops, yield rather clumsy code.

---

[1] Interestingly, the language Turing, originally designed as a language for teaching computer programming, makes use of the same two structured forms as its only loop control structures, a LOOP-EXIT construct and a FOR loop [HOL 83a].

```
(* Modula-2 *)
WHILE SomeCondition DO
  (* some statements *)
END
```

```
{ HUL }
Do in sequence
  If
    not some_condition
      Exit { set a boolean }
  { some statements }
  When exit
    stop
```

```
(* Modula-2 *)
REPEAT
  (* some statements *)
UNTIL SomeCondition
```

```
{ HUL }
Do in sequence
  { some statements }
  If
    some_condition
      Exit { set a boolean }
  When exit
    stop
```

Implementations of both the *WHILE* and *REPEAT* loops could be attempted using the *do* control structure's *intact* qualifier, but this is only safe as long as the programmer is able to guarantee that the condition for terminating the control structure will remain valid until the end of the loop iteration in which it is set.

```
{ possible HUL REPEAT loop }
Do intact in sequence
  { some statements which do not alter x }
  x <- some_value
  { some statements which do not alter x }
  When x = 10
    stop

{ possible HUL WHILE loop }
If
  x <> 10
    Do intact in sequence
      { some statements which do not alter x }
      x <- some_value
      { some statements which do not alter x }
      When x = 10
        stop
```

In the above loops, if there is a chance that x could be set to 10 (which would generate an interrupt) but reset to some other value before the end of the loop body, an explicit *if* statement must be used to set the interrupt condition, as is done in the previous versions of the *REPEAT* and *WHILE* loops.

Although it might seem that an advantage in speed would be gained if an interrupt-generating variable were used in place of code to evaluate the condition for terminating a loop, in practice this is not necessarily true, firstly because genuine parallel processing is not viable for such fine grained parallelism[1], and secondly because synchronization cannot be guaranteed. The second point is illustrated by the HUL implementation of a Pascal [BSI 82] loop to place the input window over the first character after the next '*' in the input stream.

```
(* Pascal *)
READ(ch);
WHILE (ch <> '*') DO
  READ(ch);
```

```
{ HUL }
Do
  Read[ch]
  when ch = '*'
    stop
```

Although this code might execute correctly most of the time, there is no guarantee that the reading of the next character will not be initiated before the loop is terminated, resulting in the loss of information. The possibility of this type of error arising during parallel execution frequently causes programmers who are used to the semantics of sequential programming languages to flinch. The problem is made apparent using the notation of CSP [HOA 85] to describe the interaction between the two processes during execution of the loop body.

$$HUL.PROCESS = ((read \ [] \ interrupt \rightarrow SKIP) \rightarrow (assign \rightarrow HUL.PROCESS \ [] \ interrupt \rightarrow SKIP))$$

$$CH = (assign \rightarrow \textbf{if ch = '*' then } interrupt \rightarrow SKIP \textbf{ else } CH)$$

The alphabets of the two processes have the events *assign* and *interrupt* in common; the occurrence of one of these events involves the simultaneous participation of both processes. A

[1]Section 6.2 shows experimental results which demonstrate the folly of making the grain of parallelism too fine.

legal trace of the composite process *(HUL.PROCESS || CH)* in which the character '*' is immediately present might be:

read → *assign* → *interrupt* → *SKIP*

but another legal trace might be

read → *assign* → *read* → *interrupt* → *SKIP*

The following program segment is a safe non-parallel HUL equivalent for the Pascal loop:

```
Read[ch]
Do in sequence
  If
    ch <> '*'
      read[ch]
    otherwise
      exit    {set a boolean}
  When exit
    stop
```

Though clumsy for sequential programming[1], the infinitely looping *do* control structure with built-in interrupt handling works well for programming imbedded systems. These systems are characterized by concurrent processes, and large numbers of *watch-dog* and exception handlers. The interrupt facility removes the evaluation of the conditions, which determine whether an exception has occurred, from the critical execution path. The detection of a termination character in the writer process of the readers and writers solution listed in section 5.6 is an example of this style of programming.

[1]HUL would yield more convenient code for simulating conventional sequential loops if a simple extension to its syntax were to allow the *stop* primitive to form the process body of an *if* condition. This would avoid the need to use an interrupt to exit a loop.

6. Implementation notes

Much of the implementation of HUL is able to make use of standard compilation techniques. However, the novel features of the interrupt-generating active data object require novel implementation techniques, and the ancillary aim of this thesis to divorce the programming environment from specific hardware considerations demands an uncommon approach to language implementation. This chapter discusses the innovative techniques which were necessary to support the proposal of this thesis in the implementation of HUL.

HUL was implemented using a pseudo-code compiler and a set of stack-based interpreters. The implementation effort was directed towards discovering any implementation difficulties or inefficiencies inherent in the proposal, and providing a system on which demonstrations and experiments could be carried out. For economic and practical implementation reasons, the initial system was implemented on a network of IBM PC compatible microcomputers, which formed a distributed multiprocessor environment in which identical copies of the interpreter were executed by each processor in the network. Each processor was capable of executing one or more HUL processes, and each microcomputer contained four serial ports and was functionally equivalent to a transputer[1]. The system was later transported to a genuine transputer network.

In the text which follows, a brief overview is provided of the development environment which was produced to support the investigation of this thesis, and of the uncommon compilation issues which were addressed. The communication layer, which forms the foundation for supporting interrupt-generating active data objects in distributed environments and for cushioning language features from hardware details, is outlined and compared to other similar efforts described in the literature. This chapter concludes with an evaluation of the viability of the proposal in terms of the implementations' performance.

[1]The low level communication facilities which support the communication layer of HUL on the microcomputer network were implemented as a post graduate project by D.T. Hill, under the supervision of the author. Hill also used the HUL implementation as a springboard for experimenting with process allocation in Occam [HIL 87].

6.1 The development environment

The design of HUL is intended to enable a programmer to be unconcerned about the final implementation scheme chosen for the program and, next to the endeavour to satisfy the specifications of the interrupt-generating variable concept, it was in this area that implementation efforts were concentrated.

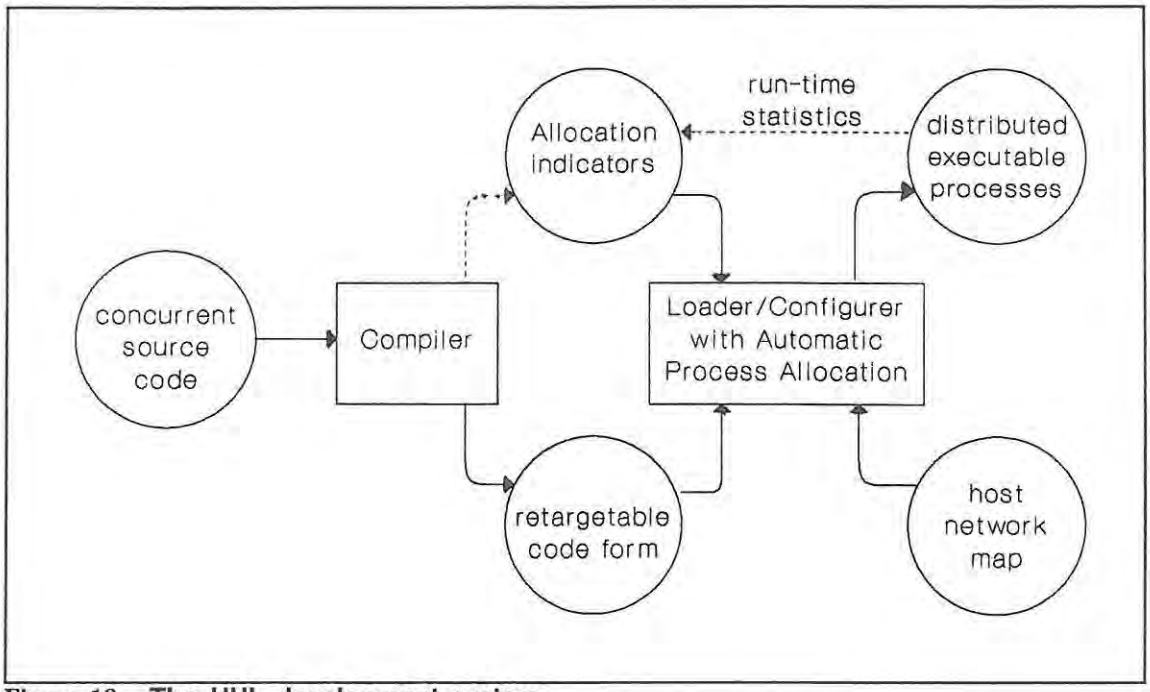


Figure 18. The HUL development system.

Figure 18 depicts the phases of the HUL development system. The HUL compiler produces an intermediate language code module for each stand-alone process in the source program, in a form which can be executed on any one of the processors in the host network. It also provides rough estimates of the communication and processing cost functions for each process in the system, to act as initial allocation indicators. At the intermediate language code stage, interrupt-generating variables are represented by process modules which have the same form as normal stand-alone processes.

In order to avoid unnecessary recompilation of the source code when adjusting the load balance of processors, or when transferring the program to an execution environment with a different

processor topology, the development system has a separate program configuration phase. This post-compilation phase gives the program its *run-time configuration* by allocating processes of the source program (including system processes which emulate interrupt-generating objects) to processors of the host network for execution. In the current development system, the program's run-time configuration may be altered each time it is executed, and consequently the configuration and loading stages are not separated. Since it is impossible to predict the run-time behaviour of the program at compile-time with any great accuracy, the allocation indicators obtained from the compilation phase have to be regarded as extremely speculative. Reliable cost functions are only obtained for programs in which identical processes are replicated. Unfortunately, most programs do not fit into this mould and the user is required to refine the allocation indicators to obtain a reasonable load balance across the host processor network. To facilitate the refinement process, the execution environment is designed to record run-time statistics and report them in the form of tables of cost functions for each process and processor.

Some researchers [CHO 82] [CHU 80] consider that interprocess communication overheads should be minimized by placing communicating processes on common processors, whereas others [BAR 85] [BOK 81] [EFE 82] [OUS 81] [STA 84] consider that processes which communicate should, where possible, be allocated to adjacent processors to minimize both multiprogramming and interprocess communication overheads. To achieve consistently good mappings, accurate measurements of the relative interprocess communication of constituent processes must be provided. Unfortunately such detailed information is hard to derive and is often absent.

In the HUL development system, the post-compilation configurer allocates a group of process modules to each processor in the distributed network. In doing this, it makes use of a simple allocation algorithm in which process allocation is resolved according the allocation indicators obtained from the compilation phase and refined by the user, and a map of the host network supplied by the user. The network map provides the mapping of virtual links (to connect processors which might not be adjacent to each other) to physical links. In order to avoid having to calculate the shortest paths between processors, the map is required to provide the set of port numbers which will route a message from one processor to another, for each ordered pair of processes. The contents of the network map for a network of N processors can be represented by the following extended BNF description:

$$network\text{-}map = N \{ \{ route\text{-}from\text{-}processor\text{-}i\text{-}to\text{-}processor\text{-}j \}_{j=1}^{N} \}_{i=1}^{N}$$

$$route\text{-}from\text{-}processor\text{-}i\text{-}to\text{-}processor\text{-}j = \{ port\text{-}number \} \text{ "0"}$$

$$port\text{-}number = unsigned\text{-}integer$$

$$N = unsigned\text{-}integer$$

Figure 19 shows a host network and its particular network map.

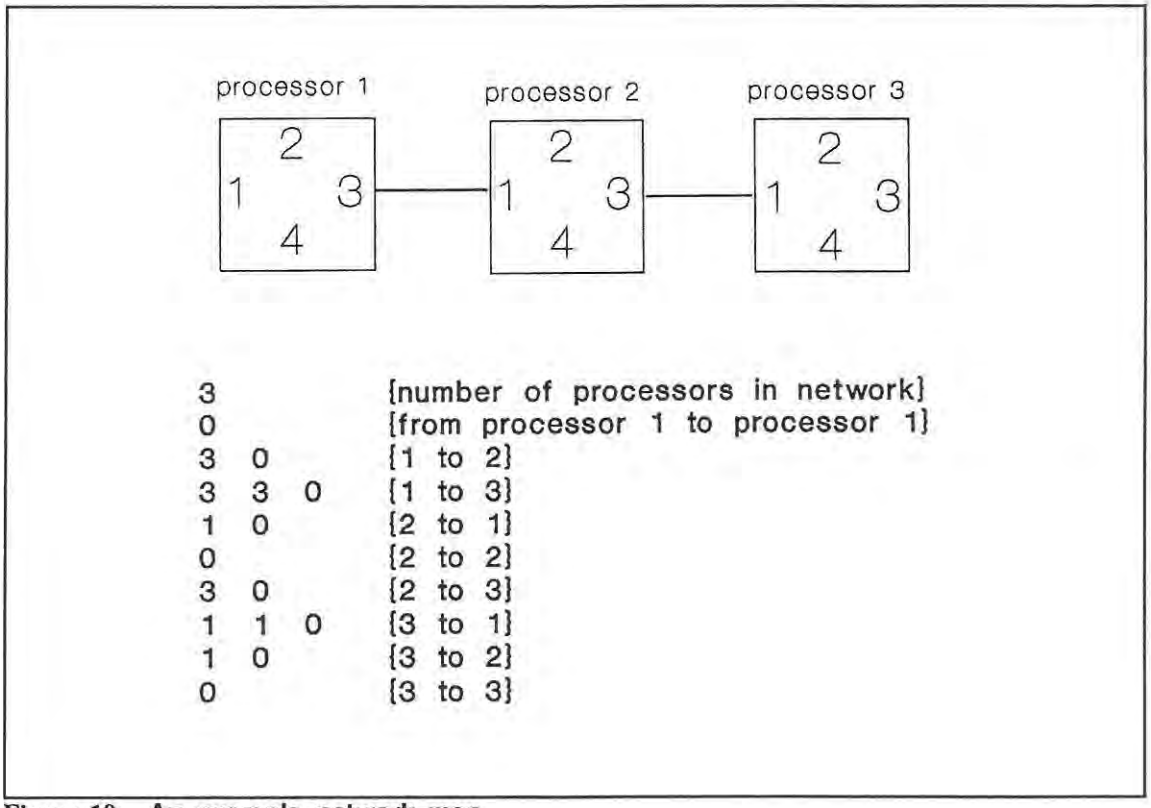


Figure 19. An example network map.

The result of the allocation algorithm is a configuration table which is distributed to each processor along with the its process modules. The configuration table records the placement of each process in the network, and the message routing procedure that is to be followed for each logical channel (for the benefit of processes which do not reside on adjacent processors). The logical connections between processes are divorced from the physical connections between

processors by a communication layer which resides on each processor.

6.2 Compilation issues

The unconstrained allocation proposals for HUL encourage very fine grained parallelism, by treating every component process of a parallel control structure as a distributable sub-program. This does not necessarily result in increased execution speed, and often results in the very opposite, due to the delays caused by message passing. These delays can result in a considerable overhead in initiating parallel processes.

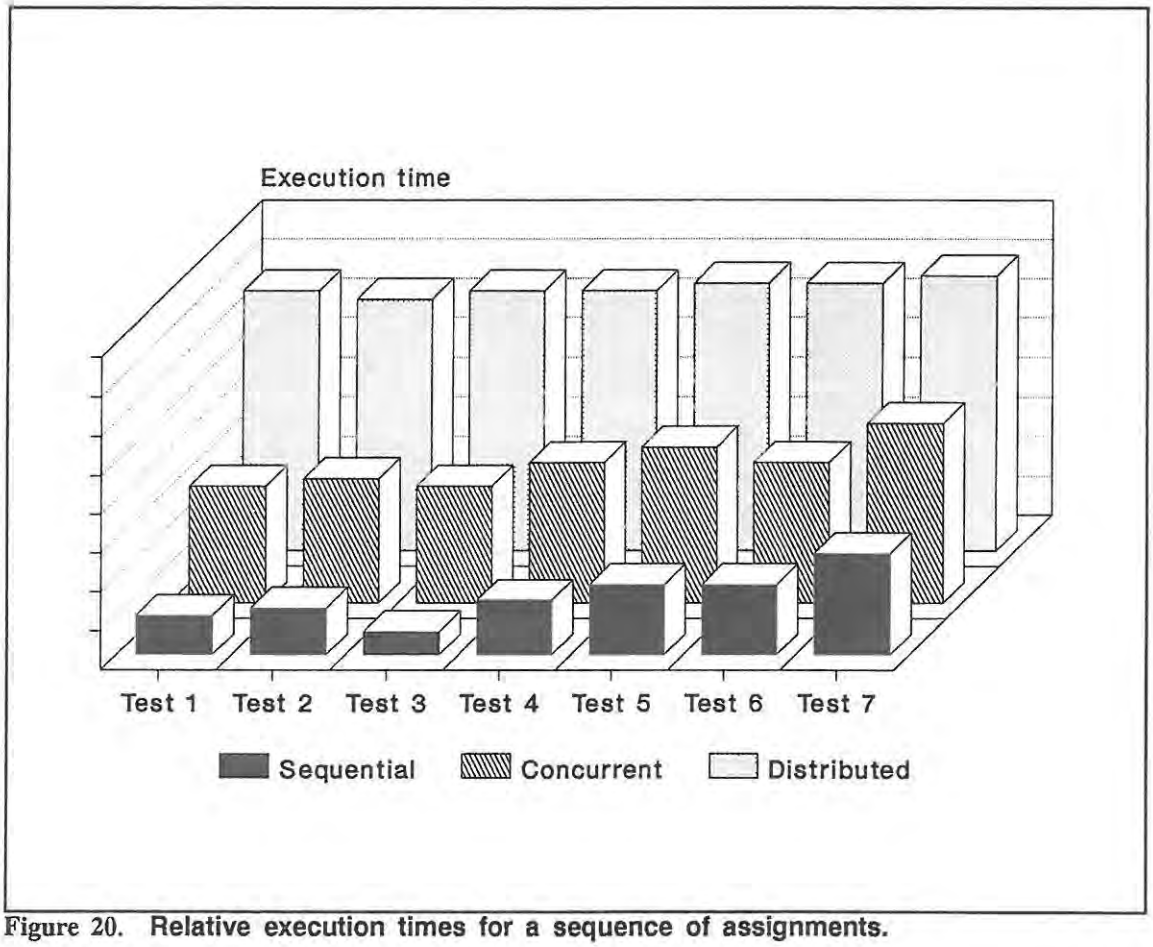


Figure 20. Relative execution times for a sequence of assignments.

Figure 20 shows the relative execution times for seven program segments, each comprising a

sequence of assignment statements executed sequentially, then executed concurrently on a single processor using time slicing, and finally executed in parallel on a network of processors. The amount of computational work required to evaluate the r-value of each assignment was increased in test sequences 4 and 5, while the number of assignments was increased in test sequences 6 and 7[1]. In all cases, the overhead of the time slicing mechanism, as well as that of the message passing necessary to initiate distributed processes, resulted in these alternatives executing considerably less efficiently (factors of about 2.5 and 6 respectively) than the simple sequential version.

Because of the negative effect of fine grained parallelism on the efficiency of the processing network, the implementation of HUL for distributed systems collapses those component processes of parallel control structures which terminate in a finite time[2] into a sequentially executed process, in accordance with the laws outlined in section 4.5.

This form of program transformation cannot be performed on input and output statements (other than in very obvious cases), although they are primitive operations. This is because it is not always possible to determine the order in which communication will take place with processes not involved in the transformation, and because there is a possibility that the execution of an input or output primitive will take infinitely long. Instead, the compiler tags primitive input and output processes and their sibling processes to indicate to the post-compilation configurer that they should be placed on the same processor. This avoids the potential overhead of initiating trivial parallel processes on a remote processor, yet causes no significant degradation in overall execution time should the input or output process never communicate, as suspended processes are never allocated a processor time slice.

---

[1]As might be expected, an increase in the number of component processes in the program segment affected both the sequential and time sliced versions, but had no significant effect on the overall execution time of the distributed option. Although similar results are shown for an increase in the computational complexity of the component processes, this can be ascribed to the order of magnitude difference between the message passing component and the processing component of the execution time measured for the distributed version.

[2]They contain no input, output, *wait* or *when* statements, or procedure calls.

The compiler checks the use of actual parameters by child processes according to the restrictions defined in section 4.2.5. Warning messages are issued when an interrupt-generating variable is used to provide interrupt conditions for two or more processes which may be executed in parallel. Checks are also made on the use of channel parameters by child processes to ensure, as far as possible, the legal use of channels and to detect some obvious cases of deadlock. Since the HUL development system only compiles complete programs, and since there is no facility in HUL for specifying the creation of dynamic processes, a process tree can be constructed at compile-time to record the run-time interdependencies of processes in order to implement these checks. The interdependencies are used in conjunction with estimates of communication and processing cost, based on the number and types of intermediate language codes generated for each process, to provide initial allocation indicators.

The pre-processor which locates instances of identifiers used in interrupt-generating contexts, and builds up a table of interrupt-generating identifiers for use by the parser, does not cause a notable degradation in compilation speed, although it makes an additional pass of the source program. Only the initial characters of each line need to be scanned to detect *when* or *wait* statements. To ensure that a proper correspondence is obtained with variable declarations during the parsing phase of the compiler, the interrupt-generating variable identifiers are stored in a tree structure which mirrors the symbol table scope structure.

6.3 The communication layer

The ability to provide global communication is a fundamental requirement of the interrupt-generating active data object for facilitating information sharing, and is essential in the wider programming structures if the programmer is to be absolved of hardware considerations and yet be allowed the facility to make optimal use of the available hardware to execute the program. In the HUL system, a communication scheme is used which resides on each processor, and provides a *virtual link* between every pair of processors in the network. Those pairs which are not directly connected by a single physical link communicate via other processors which pass the message on. The communication facilities also allow any number of logical channels to be mapped to a virtual link. This communication layer forms a natural means of implementing

interrupt-generating objects as active processes, since an interrupt-generating object must be able to communicate with all concurrent processes in its scope.

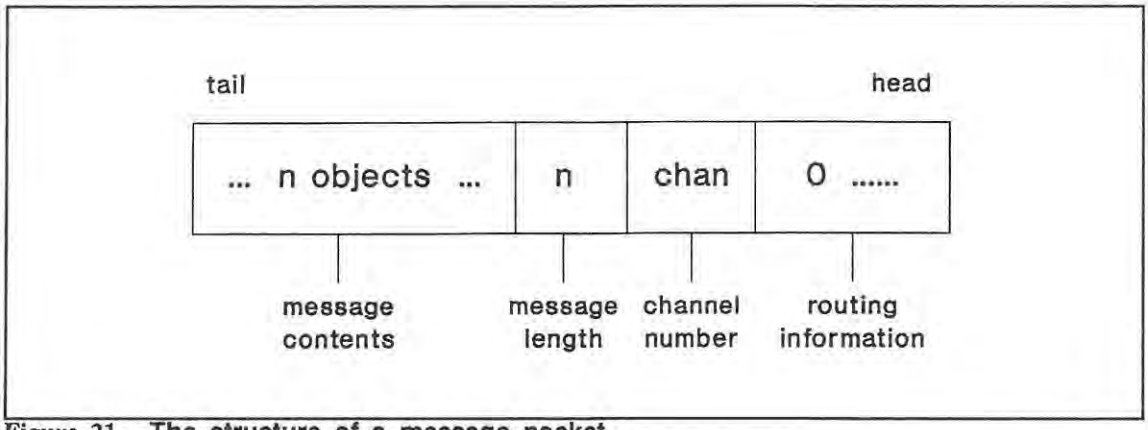


Figure 21. The structure of a message packet.

In order to facilitate the multiplexing of many channels onto a single link, all messages are sent in the form of packets. The structure of a message packet is shown in Figure 21.

Only complete packets may be multiplexed, not elements within a packet. At the head of the message packet is a set of port numbers, obtained from the configuration table, which routes the message through the network. On receiving a message packet, the communication routine of the processor decodes the message if the leading port number is a zero; otherwise it uses the leading port number to determine through which of its serial ports to reroute the rest of the message. Figure 22 shows a typical sequence of port numbers needed to create a virtual link.

The implementation of synchronized communication on a multiprocessor system requires that a number of asynchronous protocol messages must be exchanged at run-time for each simple message exchange at the application program level, in order to ensure synchronization. Allowing any child process to execute on any processor introduces a major difficulty in implementing channels. A child process may execute on a different processor from its parent, and yet both may make use of channels declared globally to them. As a result, a channel does not necessarily remain mapped to the same virtual link throughout program execution.

Figure 23 illustrates this point. The code sequences enclosed in blocks mark four child processes,

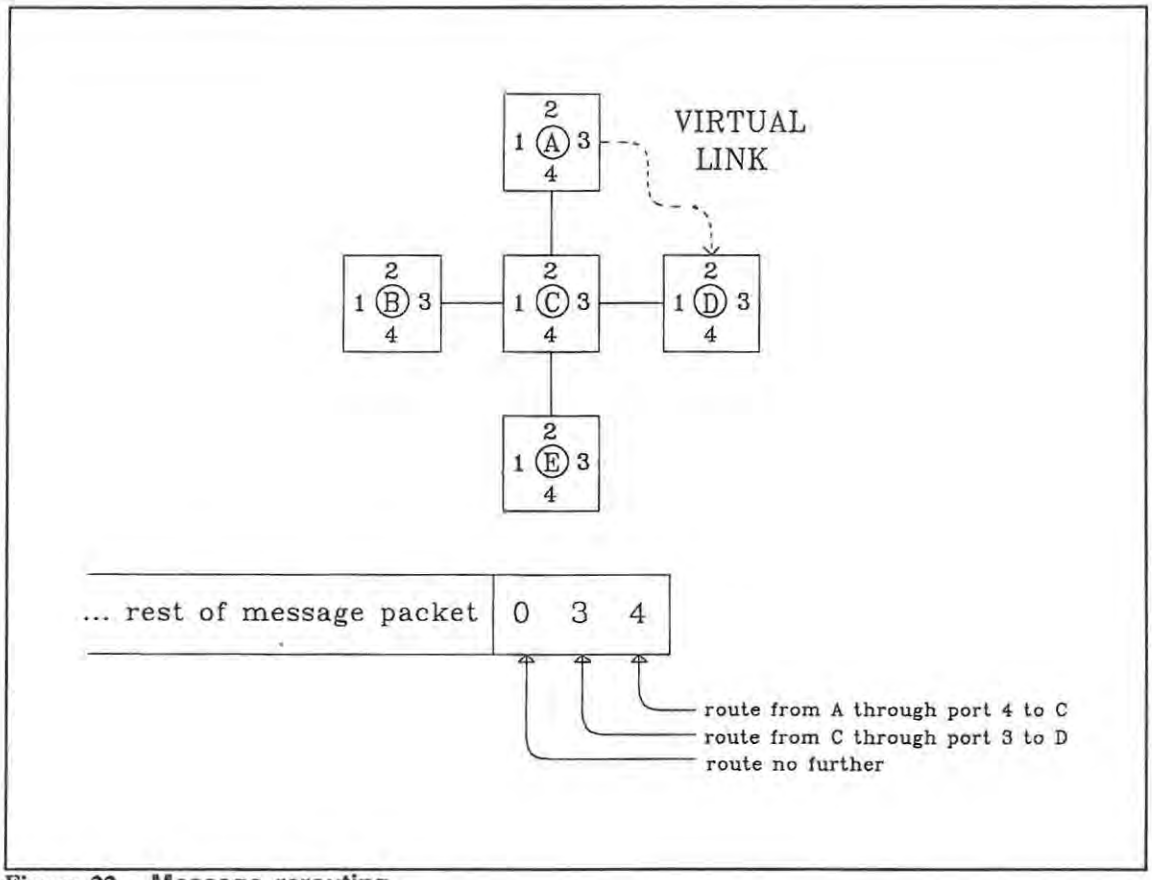


Figure 22. Message rerouting.

each of which could execute on a different processor. In this example, process {iii} receives on *from_process_ii* which is also used by its parent process {i}. Processes {i}, {ii} and {iii} are mapped to the separate processors (1), (2) and (3) respectively. The first communication occurs when process {i} executes the input primitive in line 5 and process {ii} executes the output primitive in line 10. For this communication, *from_process_ii* is mapped to the virtual link between processor (1) and processor (2). However, when process {iii} executes its input primitive (line 7) and process {ii} executes the output primitive of line 11, channel *from_process_ii* is mapped to the virtual link between processor (2) and processor (3). This change in channel mapping occurs at some stage during the execution of process {i} and cannot be predicted by processor (2). Hence the channel mapping for a particular communication can only be determined when the synchronization between the two communicating processes has been established.

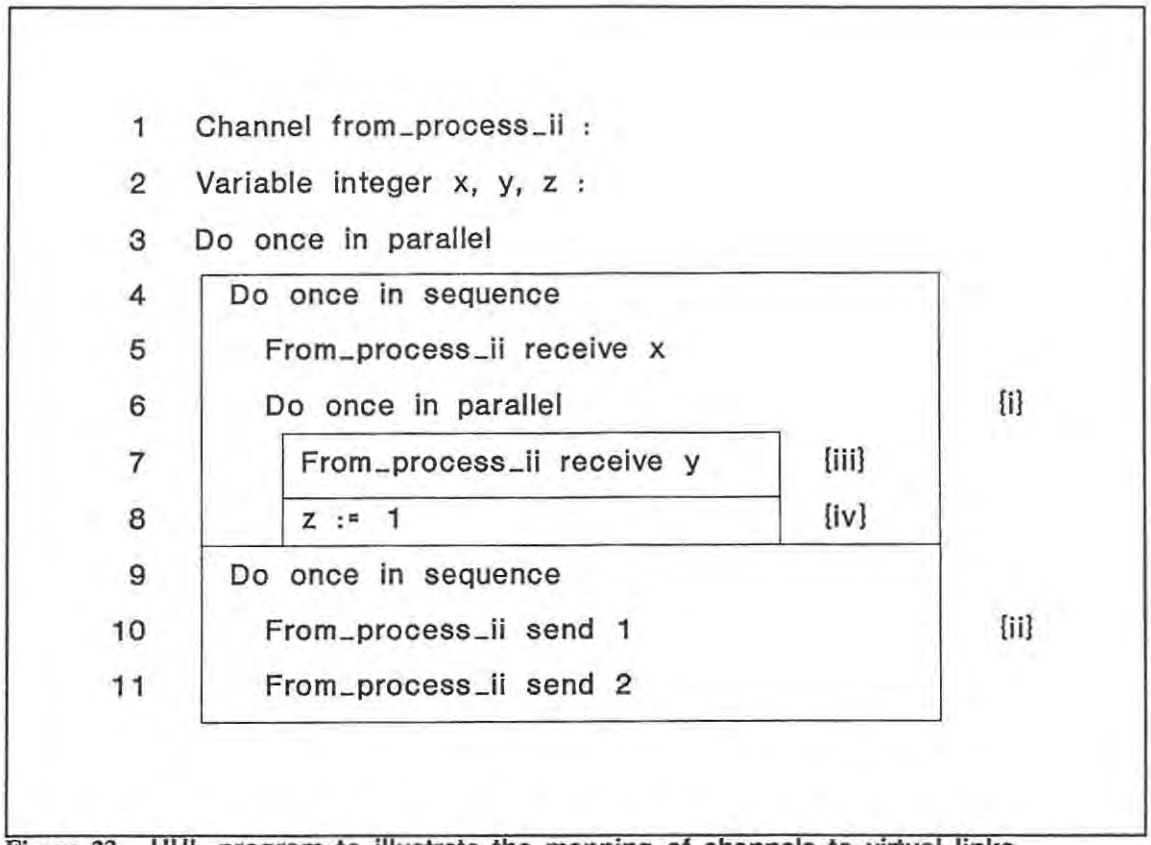


Figure 23. HUL program to illustrate the mapping of channels to virtual links.

The problem of dynamically changing the mapping of channels to virtual links has been solved by making use of a third-party channel matcher, whose task is to monitor the status of a channel and to co-ordinate synchronization. The third party matcher is similar to the matcher agent proposed by Bornat [BOR 86], although Bornat's application is rather different to that of HUL (Bornat uses his matcher to pair off partner processes in an implementation of the Occam ALT construct which caters for output guards). In HUL, the sender and receiver processes use the matcher to negotiate a synchronous message exchange between them, by sending all their protocol messages to the matcher. When synchronization has occurred, the matcher can inform the sender of the receiver's location, and the sender can then send the message directly to the receiver's processor. The three components involved are illustrated in figure 24. The matcher may exist on a processor which does not execute either the sender or the receiver. The processor which executes the process in which one or more channels are declared is assigned the task of matcher for those channels. Since the matcher remains on the same processor throughout the existence

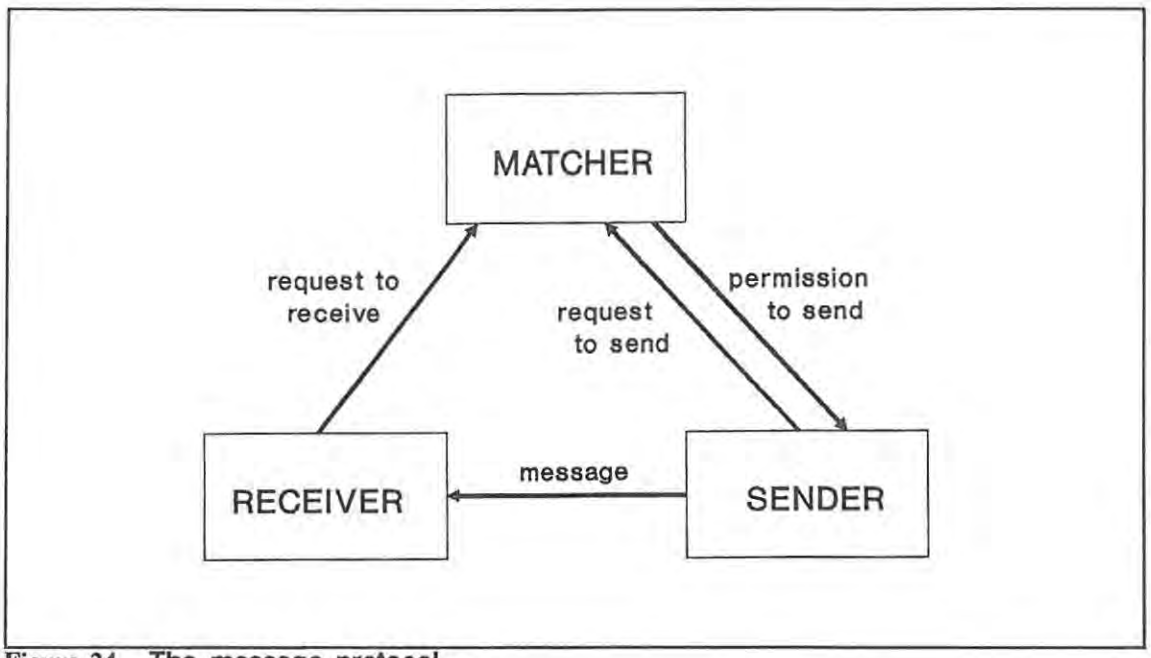


Figure 24. The message protocol.

of a channel, the mapping of the protocol messages to a virtual link is static, and can be determined at the loading stage.

An extension to the basic message protocol is required for the receiver to establish whether the sender has committed itself to sending a message or not, without the receiver itself becoming suspended. This is achieved at the source language level by the standard function *ready*[1]. When the receiver executes a *ready* function, it sends a *request for status* message to the matcher process. The matcher replies with an *able to send* message if it has already received a *request to send* from the sender, as shown in figure 25(a). The receiver may then issue a *request to receive* without risking suspension. Figure 25(b) shows an *unable to send* reply from a matcher which has not yet received the appropriate *request to send* message from the sender.

A communication layer which divorces logical connections between processes from physical connections between processors is a common feature of wide area networks. However, the

[1]The standard function *ready* is defined in appendix B, and discussed in section 4.2.3.

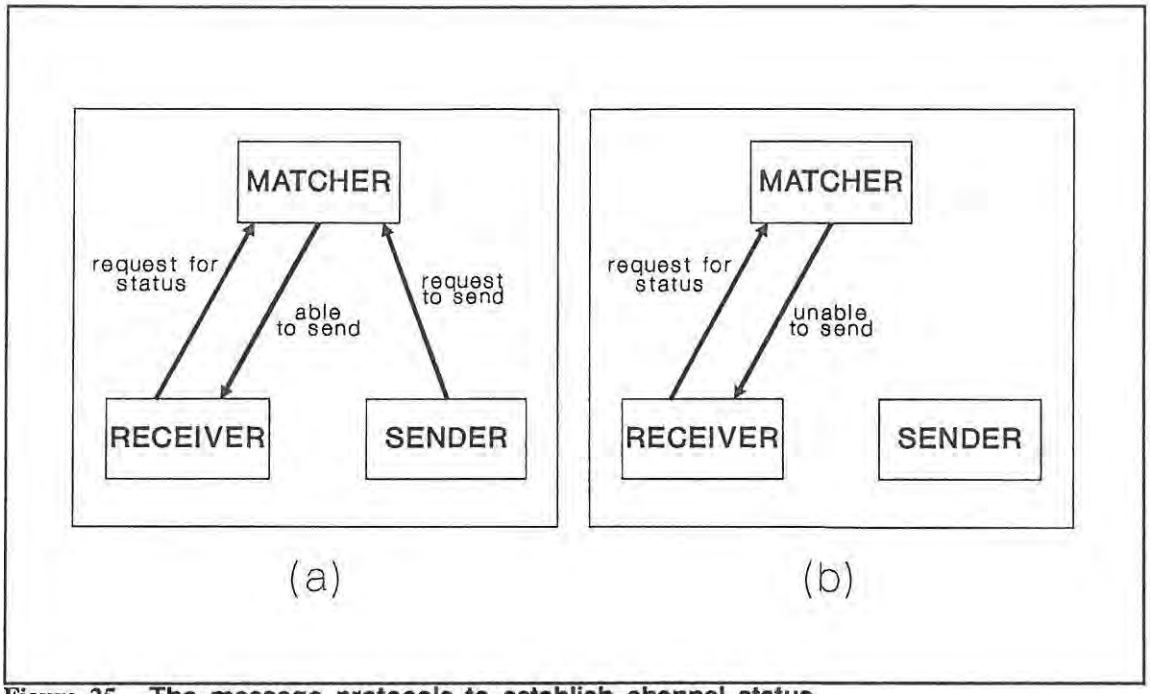


Figure 25. The message protocols to establish channel status.

traditional routing table approach of wide area networks is unsuitable for intra-program communication because it degrades the rerouting service. Recently, much effort has been invested in designing virtual communications layers for synchronous message passing languages [CAP 88] [FIS 86] [JON 88] [KAL 87] [KNO 89] [NOR 88] [ROS 88]. Few of these attempts constitute inherent features of a programming language. PLITS [FEL 79] includes a simple scheme for performing the automatic routing of messages, and the object-based languages Argus [LIS 88a] (based on CLU [LIS 77]) and the Eden Programming Language (EPL) [BLA 85] (based on Concurrent Euclid [HOL 83b]) provide location-independent invocation of distributed objects. Fisher [FIS 86] has implemented Occam on a multiprocessor token ring network which allows for abstraction between logical and physical connections by multiplexing Occam channels onto the ring. He reports that the performance of this communication ring is degraded even for small networks as the number of processors is increased.

The multilayered transport system developed at the University of Manchester [KNO 89] uses a similar form of single packet channel communication to the proposal of this thesis. Data is preceded by a very simple routing header, which is used to index routing tables at each node. Murray and Wellings have developed a message rerouting module as part of a distributed

operating system for a network of transputers [MUR 88]. This rerouting module resides on each processor, as is the case in the HUL system. Whereas the HUL development system avoids keeping routing tables on each processor by placing the routing information at the head of a message, Murray and Wellings avoid the use of tables by imposing a fixed Euclidean grid topology on the transputer network so that processors can be addressed using x and y co-ordinates. The more flexible routing mechanism of HUL would not be feasible for Murray and Wellings's operating system because the HUL method requires that final process placement be known before run-time. The routing module of Murray and Wellings supports an operating system function which establishes the destination of a message at run-time.

6.4 Code structure for interrupt conditions

The *when* statement in HUL is responsible for denoting the creation and scope of its associated interrupt-generating object. To enable the setup code for *when* statements to be executed before a *do* control structure begins its execution, a branch to the start of the control structure is

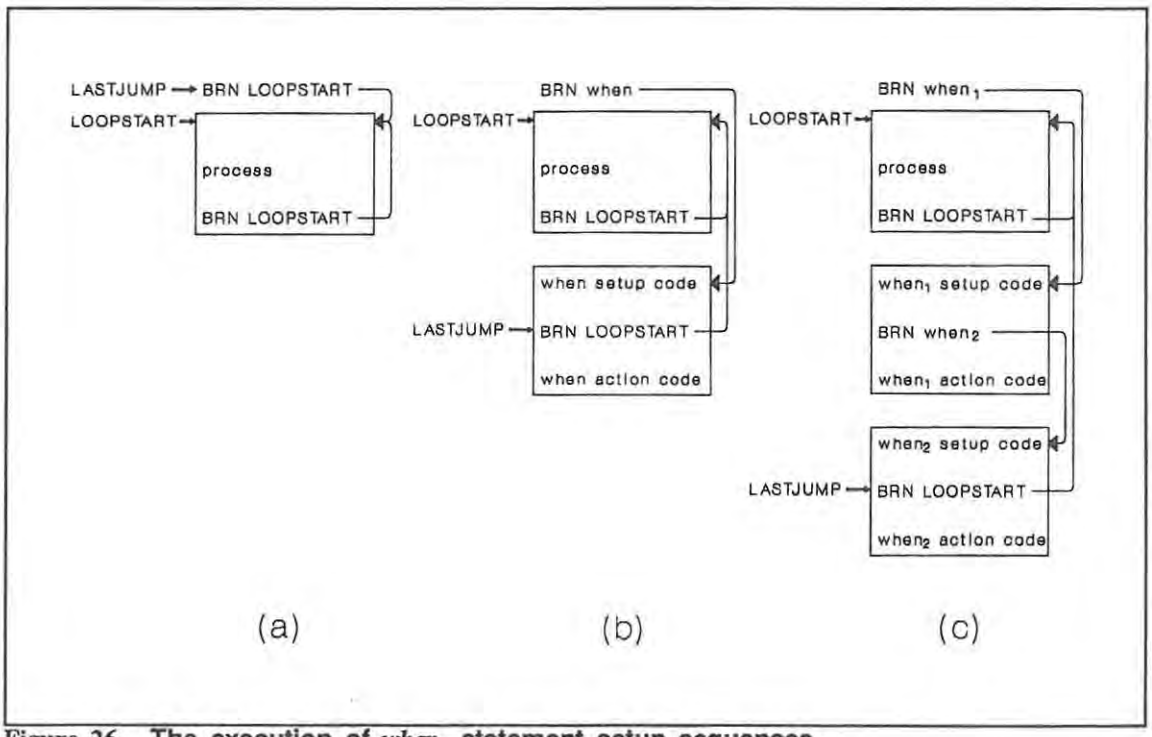


Figure 26. The execution of *when* statement setup sequences.

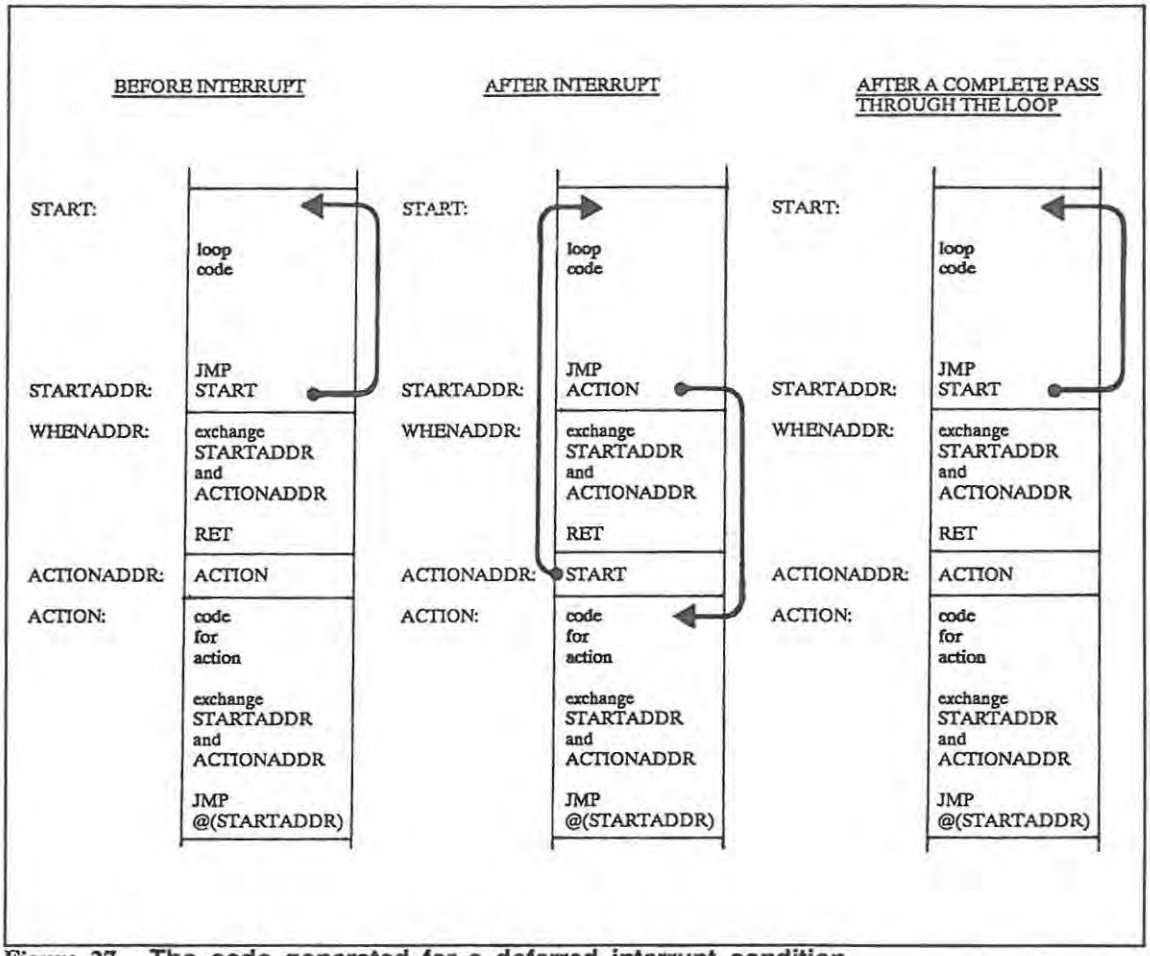


Figure 27. The code generated for a deferred interrupt condition.

generated before the code for the control structure itself. The leading branch operation code can be patched at a later time, should a *when* statement be encountered, to point to the setup code for the interrupt condition. The form of this code for an infinite loop is shown in figure 26(a). Each time a *when* statement is encountered, the branch statement at *LASTJUMP* is patched to jump to the start of the *when* statement's setup code. The pointer *LASTJUMP* is repositioned to point to the branch statement at the end of the setup sequence. This algorithm is repeated for each *when* statement in the control structure, as is illustrated by figures 26(b) and (c). The setup code evaluates the expression of the interrupt condition, and communicates with the interrupt-generating object to secure exclusive ownership of it and to transmit to it the interrupt condition value. Code (not shown in figure 26) to reset all interrupt-generating objects claimed by the control structure is placed immediately after the point of exit from the loop.

The code for an *intact* qualifier which designates a deferred interrupt condition also requires special attention. To prevent loss of information during the execution of this structure, an interrupt signal must be noted, but not acted on until the end of a complete iteration of the interrupted control structure. To achieve this, a surrogate interrupt handling code sequence is executed to modify the code at run-time. The surrogate action simply reassigns the jump address at the end of the loop structure to point to the authentic interrupt action. The interrupt action is not called as the interrupt handler, but is tagged onto the end of the code which forms the control structure. Figure 27 illustrates the structure of this code for an infinite loop. The code at *WHENADDR* is the surrogate interrupt handler which sets up the deferred interrupt action.

Should several interrupt signals arrive for different *when* statements of the same *do intact* loop, the action of the surrogate interrupt handlers results in a linked list of action code being tagged onto the end of the loop code. This situation is shown for two interrupts in figure 28. The order of actions in this run-time linked list causes deferred interrupts to be handled in the reverse order to which they were signalled.

Also illustrated in figure 28 is a call to an initializing sequence which restores to their initial values all addresses which can be altered dynamically. This code is necessary to cope with instances in which processes are aborted by their parent processes after run-time addresses have been altered, but before the interrupt actions have been executed.

6.5 Process and object creation and management

The implementation of interrupt-generating data objects as active processes forms a natural extension to the view of a HUL program as a hierarchy of processes. Parallel control structures form part of a parent process at run-time, which spawns a child process for each component process of the control structure and each interrupt-generating object in its lexical scope. The configuration table is used to determine the processor to which a particular instance of a child process has been allocated. Having determined where a child process is to execute, the parent process sends that processor a system message instructing it to initiate execution of the child. A new, separate stack is allocated to each child process, whose size is determined by the

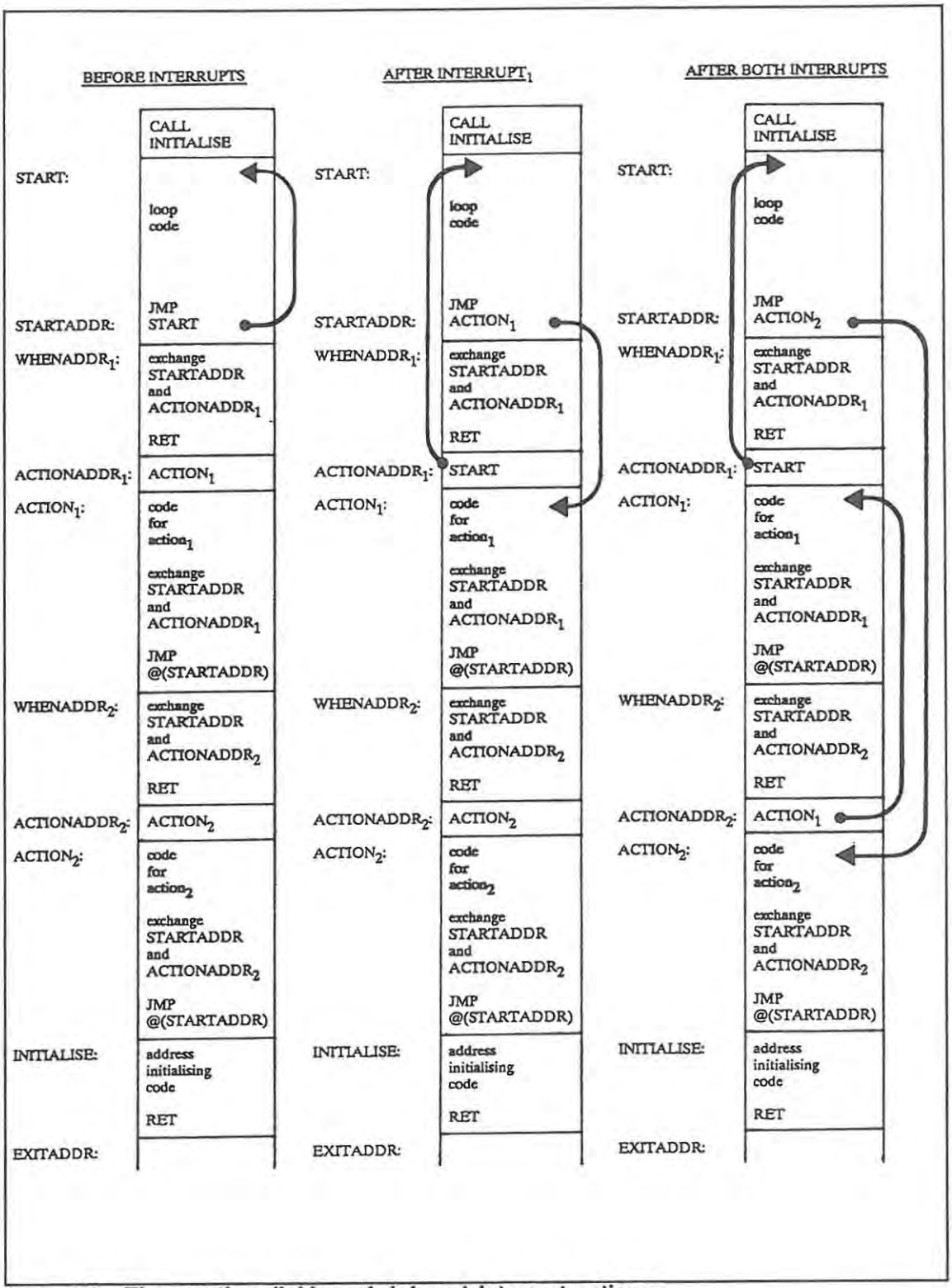


Figure 28. The run-time linking of deferred interrupt actions.

workspace requirements calculated at compile-time[1]. If the programmer has enabled recursion, the run-time stack is simply assigned a default size, regardless of the child's space requirements for locally declared variables and channels. This simplistic approach clearly makes inefficient use of stack memory.

The compiler automatically declares a pair of system channels between a parent process and each of its children. These channels are used to send parameters to the child process after activating it, and to receive the new values of any parameters passed by reference, before the child terminates. These channels also carry synchronization signals, and interrupt signals to abort the child process if the parent process is terminated. A component process of a control structure which may be prematurely terminated is implemented as if it had been declared in the source program as

```
Do once
  { child process }
  when parent_terminates
    stop
```

After spawning its children, the parent process will be suspended until a synchronizing message has been received from each child in turn, or an interrupt condition has been met.

The code generated for procedures, which are able to form named parallel processes, warrants special mention. A declared procedure may be called as a sequential process or as the child process of a parallel control structure. This requires that code be generated to cater for both types of call. When parsing a procedure, the compiler determines whether it makes any reference to a non-local variable. If so, the procedure is disqualified from being called as a child process in terms of the restrictions laid down in section 4.2.5. In this case, code is generated which is suitable only for sequential execution, and the compiler ensures that only such references are made to the procedure. If the procedure references only local variables, it is possible for it to

[1]It is difficult to calculate exact space requirements for run-time expression evaluation and for parameters, since HUL allows vectors of unspecified length to be used as formal parameters. Standard values are added to the run-time areas to cater for these needs.

be called as a sequential procedure or as a concurrent process, and code must be generated to support both cases. The difference between the code for a sequential procedure and for a concurrent procedure process is that the concurrent process must include code for parameter passing and synchronization via channels, as well as for parameter passing via the run-time stack.

6.6 Implementation appraisal

When configuration issues are encompassed by the source language as is the case with Occam, then programmers are disadvantaged in three ways. Firstly, they are not absolved of having to deal with hardware considerations and are thereby restricted in their use of the language. In current commercial Occam compilers, programmers are required to map logical channels to transputer links using explicit *port allocation* clauses. Only two Occam channels may be mapped onto a single transputer link, providing they communicate in opposite directions. Thus two separately compiled processes, which are mapped to separate transputers, may only share as many channels as can be mapped to the links connecting the two transputers directly. As a result, the designers of Occam programs must minimize the number of channels between potential separately compiled processes so as not to fall foul of the limited number of hardware links available, even though their algorithm may be more clearly expressed by using more channels. The alternative is for them to create (for each transputer) an Occam process which multiplexes many channels onto a single channel, with the attendant loss of low level synchronization due to the multiplexing process's buffer action.

Secondly, the source code of a distributed parallel processing program which includes configuration constructs has to be written for the specific hardware resources which it will use. For example, if an Occam program is developed for a 12-transputer network, it will not be able to run on one with only 10 transputers. Similarly, if it is written for 4 transputers, it is only able to make use of 4 even if it is offered 10. In order to adapt a program to run on a different number of processors, it has to be altered at the source code level, recompiled, relinked and reconfigured. This is clearly not commercially desirable, as authors are then required to sell a program in source form, with a compiler and a set of instructions as to how to configure the program.

Thirdly, specifying allocation in the source code becomes excessively tedious when programming a system of many processors (such as a processor farm of over 100 processing units).

Leaving allocation to the implementation, or to some standard run-time support environment, allows for program portability, abstraction from the hardware, and convenience, but significantly complicates the implementation.

The communication layer suggested by the HUL development environment succeeds in absolving the programmer of having to take into account processor interconnection details and the upper limit on the number of physical communication links available to each processor, by allowing any logical network of channels to be mapped onto any physical communication topology. In addition, the post-compilation configurer makes some progress towards the automatic distribution of processes over as many processors as are available.

The HUL implementations employ a static allocation scheme, which involves the mapping of processes to processors before the execution of the program. The advantage of this approach is that there is no run-time scheduling overhead as there would be for dynamic processor balancing, while the disadvantage is that it is impossible to predict the run-time behaviour of the program at compile-time. The allocation should enable the program to make the best possible use of the available processor network, by minimizing communication overheads and by sharing the processing load between processors as fairly as possible. Efficiency in a distributed parallel processing environment is directly related to load balancing.

The post-compilation placement of processes in distributed parallel processing environments has been an area of considerable interest in the past two years. Much of the work done in this area has been prompted by commercial interests, and is scantily documented in the literature as a result. A British company, 3L Ltd., has developed a special purpose configurer, known as a *flood-filling configurer*, for use with its parallel-C and parallel-FORTRAN implementations for the transputer. This product features packet rerouting software and allows particular classes of problems to farm out processing tasks [CUL 88]. The developers of the Meiko Computing Surface are experimenting with a *dynamic* allocation mechanism in which each transputer will execute a small Occam process which will either accept data sent to it for processing, or pass

it on to the next transputer [BOT 86]. Both of these approaches are dynamic, and provide a more general form of process allocation than is offered by the static scheme of the HUL development system, but for a restricted class of applications. At least two other development firms are working on transputer based operating systems which will enable programs to be reconfigured while they are running[1] [OAK 88].

Automatic allocation and allocation specified in the source program represent opposite ends of a spectrum. In the middle of the allocation spectrum lie the approaches of Bishop *et al* [BIS 87] and Mellor *et al* [MEL 86], who have adopted static allocation schemes in which allocation has been separated from the source code but must still be supplied by the user. A degree of portability is achieved in that the source code need not be directly modified for reallocation. However, user-supplied allocation essentially amounts to indirect modification of the source code with the user still requiring an understanding of the software in order to perform an intelligent allocation. The HUL development system carries the static allocation approach further, by providing a simple allocation algorithm and rough cost functions for automatic program configuration. Cost functions produced by the compiler are inadequate for all but the most obvious configurations, and user intervention is still necessary to refine the allocation indicators. Run-time statistics are of considerable assistance to the user in this respect. Aided by them, a user who is relatively unfamiliar with the source language structure can achieve a reasonable run-time load balance. An obvious area for development is the provision of a more sophisticated placement algorithm which automatically records run-time statistics using suitable cost functions, to eliminate the need for user intervention. The allocation should be viewed as an iterative process, with the program being configured before each execution using cumulative statistics averaged over the previous runs, to optimize the configuration for a typical data set.

The system communication layer which supports HUL allows communication between any two processors in the network, and consequently has the potential for increasing the message

[1]These two operating systems, Helios and Mercury, are not expected to become commercially viable products for some time. Helios is available in a preliminary form which does not feature the dynamic reconfiguration of programs.

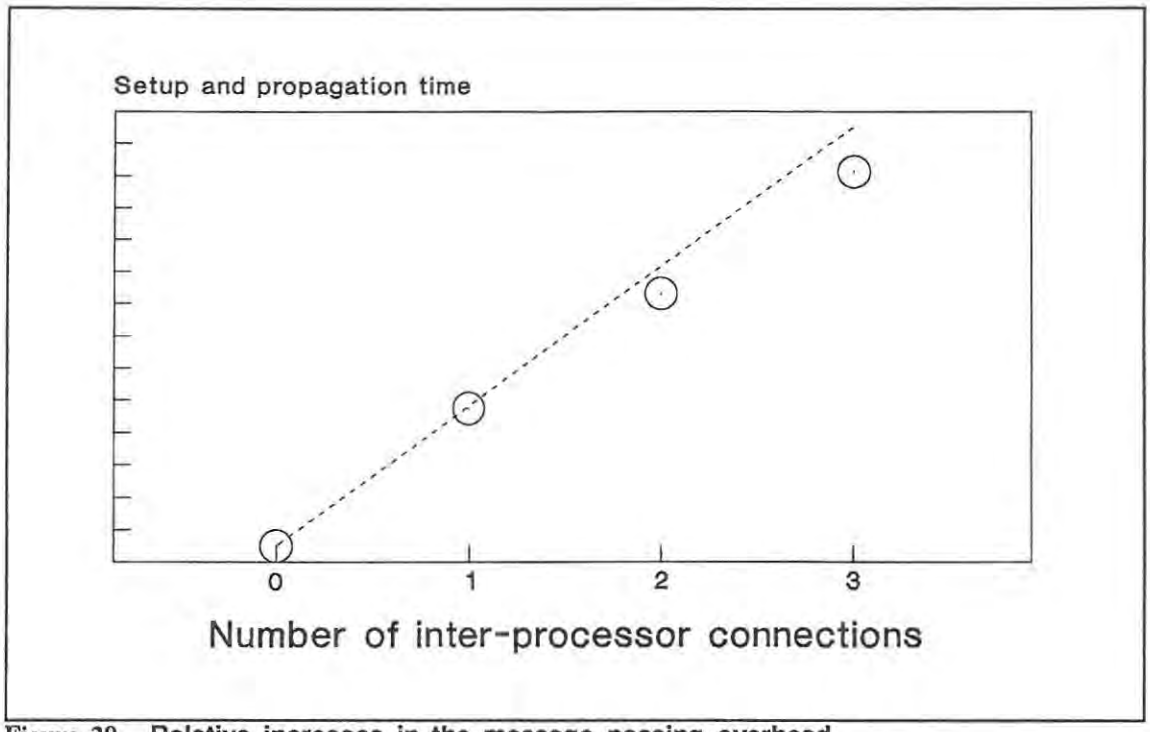


Figure 29. Relative increases in the message passing overhead.

forwarding overhead for each processor as the network size increases. Figure 29[1] shows an almost linear increase in the setup and propagation time for a message between two processes, as the number of rerouting stages is increased by placing the processes further apart on the processor network.

It is imperative that the program configuration adopted should attempt to minimize this overhead. To do this, the allocation should reflect a clustering, in the network, of processes which communicate with each other.

It is worth noting that the time taken to accomplish a message transfer is made up primarily of the message setup and propagation times; the rerouting module and matcher processes (which are used whether messages are rerouted or not) create a relatively small execution overhead for each processor. For problems in which loads are well balanced and a functional clustering of

[1]The dashed line in figure 29 indicates what a linear increase in the setup and propagation time would be.

processes in the network can be achieved (to avoid message rerouting), the test system shows a reasonable speed-up factor when additional processors are added.

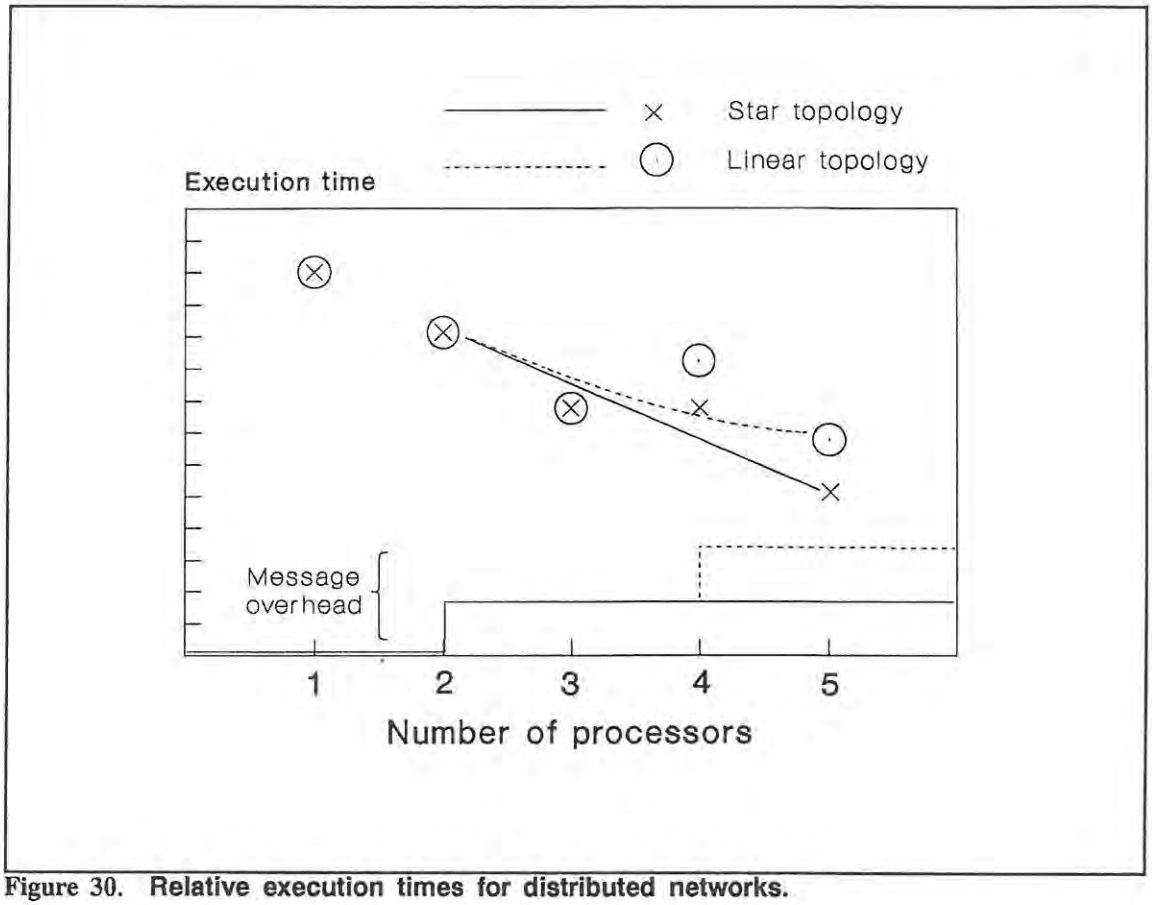


Figure 30. Relative execution times for distributed networks.

Figure 30 compares the execution time of a simple program to sum the elements of a 100x100 matrix, executed using increasing numbers of processors on two processor networks, one connected in a star topology (in which 5 processes can be directly connected) and the other in a linear topology (in which rerouting is necessary if more than 3 processors are used). A concurrent solution was formulated with five identical processes, each manipulating a 20x100 matrix. For the star topology, the solution was well balanced and no message rerouting was required, resulting

in a reasonable speedup factor as additional processors were added[1]. With the grain of parallelism that this solution exhibits, the negative effect of rerouting in the linear topology is unmistakable.

The combined execution overhead introduced by interrupt-generating processes and matcher processes was measured by comparing the execution times of programs which used interrupt-generating variables to share information between their processes, with equivalent message passing versions which were specially tailored to run on the system without matcher processes. By placing interrupt-generating processes and matcher processes carefully in relation to the application processes which they served, an increase in the execution times of between 8% and 40% over the specially tailored message passing versions was observed for the shared variable test programs[2]. For the test programs which exhibited the largest increases in execution times, the major contributing factor was the additional system level message traffic required to support the liberal referencing of interrupt-generating variables. Figure 31 shows typical execution times for one of the test programs.

The execution overheads were measured on relatively small networks (the test facility was limited to eight processors). It is anticipated that the combined overhead of the two types of system processes will increase as the size of the network is increased, due primarily to the increased difficulty of placing system processes economically in the network as the number of application processes they serve is increased. The additional execution time overhead will exhibit an imbalance in the processing/communication ratio for processors, and is likely to encourage program configurations to reflect a courser grain of parallelism.

The best execution times for processes which shared a common processor were measured when

---

[1]It should be noted that there was no facility for dynamic load balancing, so that the busiest processor had to execute two processes to completion in both the 3- and the 4-processor tests, resulting in the same overall execution time on the star topology.

[2]These results do not exhibit any degradation due to message rerouting because the message passing versions of the test programs also featured the rerouting of messages (in the source program) where necessary.

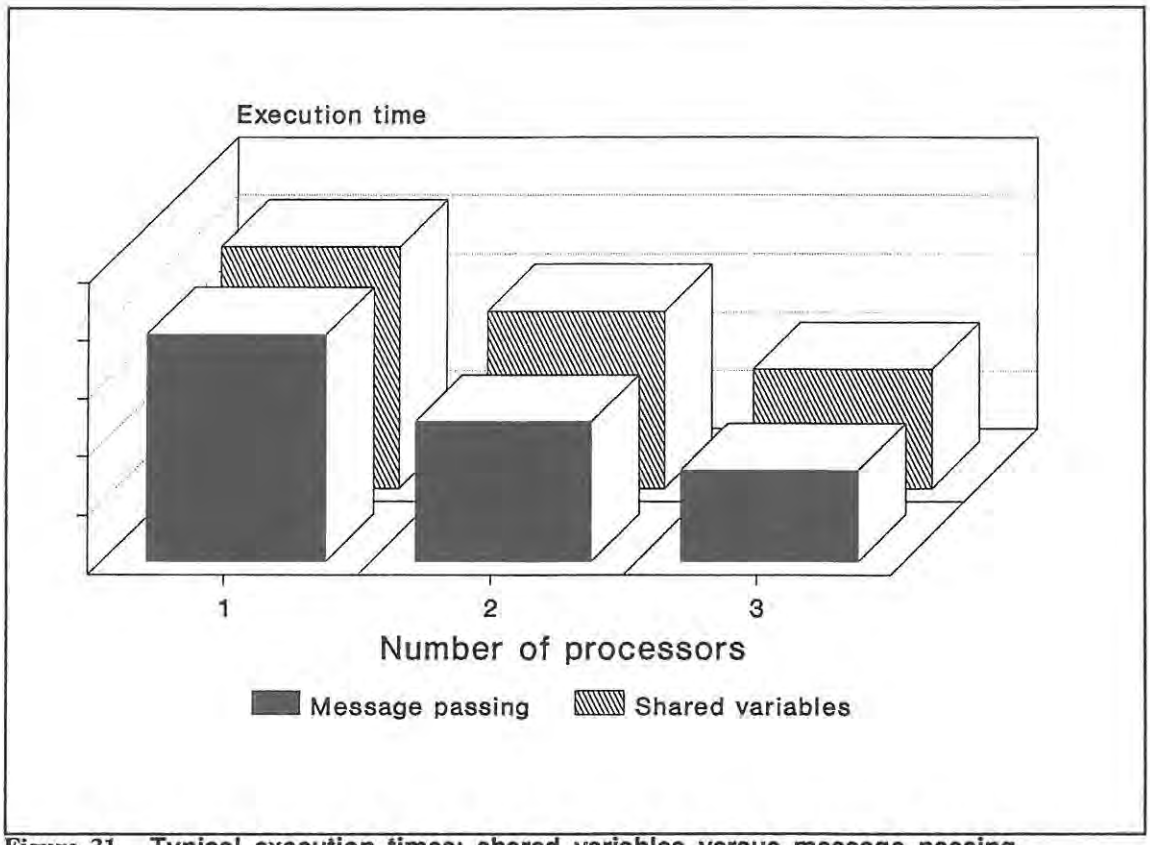


Figure 31. Typical execution times: shared variables versus message passing.

the time slice was made large enough to ensure that most context switches occurred as a result of processes becoming blocked (because of an unmatched message passing primitive or an unserviced *wait-until* statement), or terminating.

The test system revealed that the implementation proposals are successful in absolving the programmer of hardware considerations, but would not create an efficient environment for massively parallel programming.

The implementations developed so far have brought to light an overhead not immediately obvious from the initial proposals. The fine partitioning of programs results in the communication path for a particular channel changing during execution. The introduction of matcher processes to deal with this results in more protocol messages for a single language-level communication. Furthermore, the fact that two communicating processes reside on the same processor does not

imply that their communication places no burden on the communication hardware. The matcher function might be provided by a separate processor and protocol messages may, therefore, be external communications; only the actual message transfer is guaranteed to be simply a memory transfer.

The implementations have also highlighted an inefficiency in the syntax of HUL message passing; data communication between processes in HUL takes place a single object at a time. A string value is the only multi-element object catered for. Thus if two items have to be passed down a channel, then two distinct rendezvous are needed, each incurring the concomitant overhead cost of system messages. This inevitably leads to inefficiencies, and deviates from the hardware design of most modern processing units, notably the transputer [INM 84b] and the microcomputer network described in this thesis, which have facilities to handle block transfers. There is also no type checking associated with communication via channels in HUL, which places a greater responsibility on the programmer to ensure that data received on a channel is interpreted correctly. The definition of Occam 2 [MAY 87] offers a communication model which is an improvement on HUL; data communication is efficient, and the syntax includes a means for defining channel protocols which, though rather verbose and complicated, introduces some measure of protection against the accidental use of a channel in a way which is not in accordance with the programmer's intentions. Fisher has proposed an alternative syntax for defining channel protocols [FIS 88], based on regular expressions. His proposal is more flexible than the existing Occam 2 protocol definitions, and provides a greater measure of protection against run-time communication errors than is afforded by Occam 2. Besides introducing stronger typing rules for channels, Fisher's notation is less verbose and more easily understood than that of Occam 2, and would provide an attractive basis for introducing type checking to HUL channel communications.

The goal of generating object modules which can be distributed across a multiprocessor network after compilation has been facilitated by the careful handling of global variable declarations in the source program. HUL permits parallel processes to reference normal variables only if they are defined locally. However, the provision for importing global variables as parameters to parallel processes declared and called as procedures affords the programmer more freedom in the use of global variables than is allowed by the non-interference rule of Occam. The interrupt-generating variable further eases this restriction by allowing this special form of variable to be

declared globally as a shared data item.

The HUL implementation also suggests a method of abstract communication, which allows any logical network of channels to be mapped to any physical communication topology. Although the message rerouting overhead resulting from this communication layer cannot be ignored, a reasonable speed-up factor is feasible for problems in which a tight clustering of communicating processes in the network is possible. It is plausible that the link adapter of future transputer designs could be modified to handle message forwarding and channel multiplexing autonomously. Barbosa and Franca have included this type of hardware support in their proposal for a virtual communication processor which serves as the co-processor of a distributed processing node [BAR 88].

The communication layer forms a natural support environment for implementing interrupt-generating processes and matcher processes. The implementation of these two forms of system process have much in common. Both are small, transparent processes, which promote the isolation of the application program level from considerations of the host hardware. Both incur a reasonable execution overhead if placed wisely in relation to the processes they serve in the network, and a high overhead caused by communication delays if placed badly.

7. Conclusions

This thesis has proposed a data-encapsulating, interrupt-generating object and a structured interface for its interaction with active processes, which together support information sharing and a predicate triggering mechanism. The proposal forms the basis of a novel, easily used, structured programming paradigm, which allows programmers of non-shared memory multiprocessors a significant measure of abstraction in modelling entities which interrupt one another as they make use of shared resources. Seen against the severe shortage of understood programming techniques which research on distributed parallel architectures currently suffers from, the proposal provides a tangible step towards reducing the effort of programming such computing environments. The thesis has also introduced techniques which enable the proposal to be realized with reasonable efficiency, and has demonstrated that the resulting programs possess desirable safety properties.

A number of desirable qualities have been demonstrated for the interrupt-generating active data object.

- It provides an easily used facility for shared memory concurrency in multiprocessor systems in which processors have no memory in common. This facility relieves programmers of having to map concurrent solutions in which shared variables are a fundamental aspect of the problem area onto sets of message exchanges. However, point to point communication and pipelined applications are still more appropriately handled by channels.

- It provides for exclusive access of an encapsulated shared item as an axiomatic feature, and supports condition synchronization through its interrupt generation mechanism. The proposal does not support imperative synchronization elegantly, and the established message passing notation of CSP was incorporated into the experimental language, HUL, for this purpose.

- The proposal is sufficiently simple to be implemented in the form of an

active process with reasonable efficiency, and to be mapped precisely onto a multicomputer network. The implementation appraisal of section 6.6 showed that the combined run-time overhead of interrupt-generating processes and matcher processes was not prohibitively large, provided these processes were placed wisely in relation to the application processes they served. However, the proposal does introduce a hidden execution cost (as was demonstrated in section 5.2), a factor which is not always favoured by software designers [SNY 86a].

- The object's method invocation mechanism exhibits inherent safety properties. Built-in mutual exclusion avoids interference in references to shared data controlled by this structure, and the ownership rule facilitates contention free condition synchronization.

- The model of interrupt-generating active data objects is based on the established theoretical foundation of CSP, which can be used to prove properties of programs based on the inherent axioms of the model.

In its interface with active processes of an application program, the interrupt-generating active data model requires that its client processes are able to cope with unanticipated high level signals. This has necessitated the introduction of a novel class of control structure to be used in conjunction with interrupt-generating active data objects.

The thesis has succeeded in introducing the proposal as an elemental language structure in the experimental programming language, HUL. The design of HUL has provided an example of an actual programming construct which falls into the class of control structure required for use with the interrupt-generating object model. This implementation has proved effective for demonstrating the application of interrupt-generating active data objects to concurrent solutions which are naturally expressed using shared variables. The interrupt-generating object and its associated control structures have been shown to provide a versatile programming approach, which improves on existing concurrent control structures in accommodating unanticipated communications. The proposal also avoids the polling bias evident in many applications which require a process to

choose from a number of possible rendezvous.

It is essential that a programming language should have a precise mathematical meaning. The mathematical semantics of the programming structures in this thesis can be specified in terms of events and processes using the meta-language of CSP [HOA 85]. In particular, this notation can be used to specify the interaction of sequential processes of an application program and system-generated active data objects, as was demonstrated in section 3.3. The high level interrupt of this thesis is a well defined event involving the simultaneous participation of two processes; the potential of the participating processes to engage in the event can be predicted, although the interrupt need not necessarily occur. By identifying the set of events in which each component process in the program may engage, writing out the behaviour of each set of parallel sub-processes as a composite process, and examining an arbitrary trace to show that in all cases there is at least one legal event by which the trace can be extended, the programmer is able to prove that the program conforms to safety properties (as was done for the example program in section 5.7). Important conditions to detect in programs based on the interrupt-generating object model are the presence of a potential deadlock (a situation in which the component parallel processes are prepared to engage in some further action, but cannot agree on which action this should be), and the infinite postponement of a process due to inappropriate ownership of interrupt-generating active data objects. Not only are mathematical semantics useful in assisting to establish the correctness of a program by formally proving safety properties, but they also give a stable and precise specification of the program which promotes a reliable implementation. Regrettably, the degree of rigour required for a detailed CSP analysis is extremely tedious for all but the most trivial of HUL programs, and would be beyond the ability of many potential users of the language.

The primary aim of the research was accomplished more effectively than the ancillary aim. In designing an experimental language which would sustain the abstract programming level facilitated by the interrupt-generating structure, suitable language support facilities were chosen which would divorce the abstract problem solving environment from specific hardware considerations of the distributed multiprocessor host. The automatic configuration stage in the HUL development system provides the advantage of distributing copies of child processes to as many processors as are available, without it being necessary to recompile the source program. Despite the need for

the user to refine the allocation indicators, the method does exhibit the potential for applying distributed parallel processing networks to a vastly increased range of applications. This does not absolve the programmer of making sensible decisions about the relative loads of sibling processes and the level of granularity at which to place parallelism; automatic process allocation is not expected to prove to be as effective as hand tailored parallelism.

On the negative side, HUL has not succeeded in making the differences between normal variables and interrupt-generating active data objects fully transparent to the programmer. Although the physical appearance of program listings does not distinguish between the two types of data items, the programmer is made aware of the existence of the ownership rule if an attempt is made to use the same object to support two interrupt conditions simultaneously, or to reference an interrupt-generating active data object which is not *owned*.

A programmer wishing to perform formal proofs on a program which makes use of interrupt-generating active data objects will have to be aware of their distinctive behaviour. For this reason, a strong case exists for declaring interrupt-generating active data objects explicitly so as to avoid any potential conceptual problem which the implicit declaration might cause. An explicit declaration would also allow interrupt-generating active data objects to be used as normal shared variables without the need for a dummy *when* statement, which looks rather awkward in the source program[1].

The extent to which HUL programs are able to reflect natural expression is restricted by the constraints imposed on HUL's grammar, which enable it to be implemented as a structured imperative language, and by those aspects of its design which are weak in supporting natural expression. The attempt to provide facilities which allow solutions to be expressed naturally has concentrated on the control structures of the languages. The syntax for some of the primitive processes and data declarations have remained rather unnatural and could be improved. Some of the more interesting solutions that the author has managed to code in HUL (for example the

[1]The ability to reference an interrupt-generating variable which does not appear in a *when* statement would require the relaxation of the constraint which permits references to such variables only if they are supporting an active interrupt condition (refer to specification 3-2-4).

Dining Philosophers' program listed in section 5.7) have distinctly unnatural looking source listings. The ownership rule of interrupt-generating active data objects contributes to this effect. If evoked deliberately, this rule is useful for implementing mutual exclusion in the source program, but it yields programs which are enigmatic and unnatural.

It is advisable for an author commenting on his own work to distinguish subjective results from objective results. No formal research has been done to measure the extent to which the proposal reduces the effort of programming a distributed memory multicomputer network, and the comments made about the proposed notation's ease of use have been subjective ones. The extent to which the interrupt-generating active data concept is pertinent to natural problem domains, is also a subjective assessment. Objectively, the interrupt-generating active data object has used an established theoretical foundation to provide a structured information sharing mechanism with built-in safety properties. The direct linguistic support and axiomatic guarantees which the proposal provides are improvements over existing programming structures with similar features. The automatic distribution of processes introduced to support the ancillary aim of the thesis makes positive progress towards program portability and ease of use by improving upon the crude form of post-compilation fragmentation of languages such as Occam. Objective measurements have shown that the run-time overheads introduced by the proposal need not be prohibitively large.

The research described in this thesis provides fertile ground for further research in several areas. Advantage has not yet been taken of the potential for extending the object model to richer classes of data. Further research is required to determine how efficiency would be affected by such an extension and whether all the safety guarantees provided for simple data objects could be retained. In the same vein, further examination is needed to establish whether the modification of the interrupt generation method to provide an assortment of signalling methods (such as signalling on a range of condition values) would increase ease of expression and reduce the complexity of the application program still further.

Beyond these obvious extensions to the object model, ideas for future research include modifying the object for integration into a persistent database query language. By overlaying the database structure, the object proposed in this thesis would provide a novel mechanism for implementing

persistent access by several processes to distributed database entities, provided the data encapsulated by the object were not created when the object was created, but *adopted* for the life of the object. In addition to providing a structured access mechanism for simultaneous access to persistent data in a distributed system, this would provide a conditional signalling method for particular status values within the database. It is envisaged that the object model would provide assistance in enforcing data integrity and (with suitable adaptation) would provide a simple method of providing relational triggers[1].

There is considerable room for improving the linguistic support for the interrupt-generating active data object in the wider constructs of HUL, and for improving the degree to which the abstract HUL programming environment has been isolated from the hardware. The preceding discussion has provided some indication of the need to investigate the use of interrupt-generating active data objects in a language in which they are explicitly declared and invoked. The extension of the object model of this thesis to provide user defined methods and data types, more in the vein of conventional object-based languages, also provides an area for further research. The current model is not expected to inhibit class inheritance. Such a line of investigation would provide the opportunity for using an interrupt generation mechanism to overcome the problems which exist in current object-based languages in handling unanticipated object interaction (although the extended object would naturally no longer be an elemental structure of the language).

At the implementation level, the interrupt-generating active data object has led to further research into process placement at the author's university. The efforts to isolate source programs from hardware considerations in distributed parallel processing environments have opened up a large area in need of further investigation. The research has already been carried over to other programming languages [HIL 87] [HAN 89]; current research is directed towards devising an automatic process allocation method which improves on the one used in the HUL development system by providing better run-time cost functions. The functionality of the interrupt-generating active data object in supporting (memory-mapped) embedded systems could also be improved through the refinement of the language support features.

---

[1]The application of triggers to database systems is described by Eswaran [ESW 76].

At the conceptual level, a more ambitious project planned at the author's university is an investigation into incorporating interrupt-generating active data objects into Gelernter's *tuple space* model [GEL 85]. The retention of the interrupt-generating method and its built-in safety properties, with the added matching and unification functionality and data handling richness of tuple space, is expected to provide a communication mechanism which retains the decoupling advantages of LINDA, but allows for the invocation of built-in safety properties and the avoidance of the necessity for processes to poll tuple-space or wait upon a tuple match.

Bibliography

[ACK 85] Ackley, D.H., Hinton, G.E. and Sejnowski, T.J. (1985), A Learning Algorithm for Boltzmann Machines, *Cognitive Science*, **9**, 147-169.

[ADA 83] *Ada Reference Manual*, Ichbiah, J. (1983), Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A.

[AGH 86] Agha, G.A. (1986), *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA.

[AGH 87] Agha, G.A. and Hewitt, C.E. (1987), Concurrent Programming Using Actors, In: Yonezawa, A. and Tokoro, M. (*eds*), *Object-Oriented Concurrent Programming*, MIT Press, Cambridge, MA, 37-54.

[AGH 89] Agha, G.A. (1989), Foundational Issues in Concurrent Computing, Proc. SIGPLAN Workshop: Object-Based Concurrent Programming, *ACM Sigplan Notices*, **24**(4), 60-65.

[AHO 86] Aho, A.V., Sethi, R. and Ullman, J.D. (1986), *Compilers - Principles, Techniques, and Tools*, Addison-Wesley, Reading, Massachusetts.

[ALB 71] Albus, J.S. (1971), A Theory of Cerebellar Functions, *Mathematical Biosciences*, 9, 25-61.

[AME 87] America, P. (1987), POOL-T: A Parallel Object-Oriented Language, In: Yonezawa, A. and Tokoro, M. (*eds*), *Object-Oriented Concurrent Programming*, MIT Press, Cambridge, MA, 199-220.

[AND 82] Andrews, G.R. (1982), The Distributed Programming Language SR - Mechanisms, Design and Implementation, *Software - Practice and Experience*, **12**(8), 719-754.

[AND 83] Andrews, G.R., and Schneider, F.B. (1983), Concepts and Notations for Concurrent Programming, *Computing Surveys*, **15**(1), 3-43.

[ANS 66] American National Standards Institute (1966), *American National Standard Programming Language FORTRAN*, ANSI X3J3, NY.

[APP 86] Apple Computer Inc., *Object Pascal Reference Manual*, Apple Computer Inc., Cupertino, CA.

[ATK 89] Atkinson, R., Demers, A., Hauser, C., Jacobi, C., Kessler, P. and Weiser, M. (1989), Experiences Creating a Portable Cedar, Proc. SIGPLAN[89] Conf.: Programming Language Design and Implementation, *ACM Sigplan Notices*, **24**(7), 322-329.

[BAC 86] Bach, M.J. (1986), *The Design of the UNIX™ Operating System*, Prentice-Hall, Englewood Cliffs, NJ.

[BAI 84] Baiardi, F., Ricci, L. and Vanneschi, M. (1984), Static Checking of Interprocess Communication in ECSP, Proc. SIGPLAN[84] Symposium: Compiler Construction, *ACM SIGPLAN Notices*, **19**(6), 290-299.

[BAR 85] Barak, K. and Shiloh, A. (1985), A Distributed Load Balancing Policy for a Multicomputer, *Software - Practice and Experience*, **15**(9), 901-913.

[BAR 87] Barry, B., Altoft, J., Thomas, D. and Wilson, M. (1987), Using Objects to Design and Build Radar ESM Systems, Proc. OOPSLA[87] (Object-Oriented Programming Systems, Languages and Applications), *ACM Sigplan Notices*, **22**(12), 192-201.

[BAR 88] Barbosa, V.C., and Franca, F.M.G. (1988), Specification of a Communication Virtual Processor for Parallel Processing Systems, *Microprocessing and Microprogramming*, **24**(1-5), 511-518.

[BAR 89] Barth, G. (1989), Object-Oriented Systems: So What?, In: Kritzinger, P. (*ed*), *Proceedings of the 5th Southern African Computer Symposium*, Johannesburg, November.

[BAS 87] Basu, J., Paitnaik, L.M. and Goswam, A.K. (1987), Ordered Ports - A Language Concept for High-Level Distributed Programming, *Computer Journal*, **30**(6), 487-497.

[BAY 86] Bayan, R., Bonnet, C., Kung, A., Kirchgassner, W., Landwehr, R. and Schwartz, B. (1986), *Transition Towards Embedded Real-Time Applications in Ada*, Proc. Ada-Europe Conference, Edinburgh, May.

[BEN 82] Ben Ari, M. (1982), *Principles of Concurrent Programming*, Prentice-Hall, Englewood Cliffs, NJ.

[BEN 87] Bennett, J.K. (1987), The Design and Implementation of Distributed Smalltalk, Proc. OOPSLA[87] (Object-Oriented Programming Systems, Languages and Applications), *ACM Sigplan Notices*, **22**(12), 318-330.

[BEY 88] Beynon, T. and Dodd, N. (1988), The Implementation of Multi-Layer Perceptrons on Transputer Networks, In: Muntean, T. (*ed*), *Parallel Programming of Transputer Based Machines*, Proc. OUG-7, IOS, Amsterdam, 108-119.

[BIR 73] Birtwistle, G.M., Dahl, O.J., Myhrhaug, B. and Nygaard, K. (1973), *Simula Begin*, Van Nostrand Reinhold, NY.

[BIS 86] Bishop, J.M. (1986), *Data Abstraction in Programming Languages*, Addison-Wesley, Wokingham, England.

[BIS 87] Bishop, J.M., Adams, S.R. and Prichard, D.J. (1987) Distributing Concurrent Ada Programs by Source Translation, *Software - Practice and Experience*, 17(12), 859-884.

[BIT 84] Bitton, D., DeWitt, D.J., Hsiao, D.K., and Menon, J. (1984), A Taxonomy of Parallel Sorting, *Computing Surveys*, 16(3), 287-318.

[BLA 85] Black, A.P. (1985), *The Eden Programming Language*, TR 85-09-01, Department of Computer Science, University of Washington, Seattle, WA.

[BLA 86] Black, A.P., Hutchinson, N., Jul, E. and Levy, H. (1986), Object Structure in the Emerald System, Proc. OOPSLA^{86} (Object-Oriented Programming Systems, Languages and Applications), *ACM Sigplan Notices*, 21(11), 78-86.

[BOB 86] Bobrow, D.G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M. and Zdybel, F. (1986), CommonLoops, Merging Lisp and Object-Oriented Programming, Proc. OOPSLA^{86} (Object-Oriented Programming Systems, Languages and Applications), *ACM Sigplan Notices*, 21(11), 17-29.

[BOK 81] Bokhari, S.H. (1981), On the Mapping Problem, *IEEE Trans. Computers*, C-30(3), 207-214.

[BOO 87] Booch, G. (1987), *Software Engineering with Ada*, Benjamin/Cummings, Menlo Park, C.A.

[BOR 86] Bornat, R. (1986), A Protocol for Generalized Occam, *Software - Practice and Experience*, 16(9), 783-799.

[BOT 86] Bottomley, R. (1986), The Meiko Computing Surface : a Reconfigurable Supercomputer, IEE Colloquium : The Transputer : Applications and Case Studies, *IEE Digest*, 91.

[BRI 70] Brinch Hansen, P. (1970), The Nucleus of a Multiprogramming System, *Comm. ACM*, **13**(4), 238-241, 250.

[BRI 73] Brinch Hansen, P. (1973), *Operating System Principles*, Prentice-Hall, Englewood Cliffs, NJ.

[BRI 75] Brinch Hansen, P. (1975), The Programming Language Concurrent Pascal, *IEEE Trans. Software Eng.*, **SE-1**(2), 199-206.

[BRI 78] Brinch Hansen, P. (1978), Distributed Processes: A Concurrent Programming Concept, *Comm. ACM*, **21**(11), 934-941.

[BRI 81] Brinch Hansen, P. (1981), Edison - a Multiprocessor Language, *Software - Practice and Experience*, **11**(4), 325-361.

[BRI 87] Brinch Hansen, P. (1987), Joyce - A Programming Language for Distributed Systems, *Software - Practice and Experience*, **17**(1), 29-50.

[BRI 89a] Brinch Hansen, P. (1989), The Joyce Language Report, *Software - Practice and Experience*, **19**(6), 553-578.

[BRI 89b] Brinch Hansen, P. (1989), A Multiprocessor Implementation of Joyce, *Software - Practice and Experience*, **19**(6), 579-592.

[BRO 84] Brookes, S.D., Hoare, C.A.R. and Roscoe, A.W. (1984), A Theory of Communicating Sequential Processes, *Journal of the ACM*, **31**(3), 560-599.

[BSI 82] British Standard Specification (1982), *Computer Programming Language Pascal*, BS 6192, BSI, London.

[BUR 85a] Burns, A. (1985), *Concurrent Programming in Ada*, Ada Companion Series, Cambridge University Press.

[BUR 85b] Burns, A., Lister, A.M. and Wellings, A.J. (1985), *A Review of Ada Tasking*, YCS.78, Department of Computer Science, University of York.

[BUR 88a] Burke, M., Cytron, R., Ferrante, J., Hsieh, W., Sarkar, V. and Shields, D. (1988), Automatic Discovery of Parallelism: A Tool and an Experiment, *ACM Sigplan Notices*, **23**(9), 77-84.

[BUR 88b] Burns, A. (1988), *Programming in Occam 2*, Addison-Wesley, Wokingham, England.

[BUS 88] Bustard, D., Elder, J. and Welsh, J. (1988), *Concurrent Program Structures*, Prentice-Hall, Englewood Cliffs, NJ.

[CAM 74] Campbell, R.H. and Habermann, A.N. (1974), The Specification of Process Synchronization by Path Expressions, In: Gelenbe, E. and Kaiser, C. (*eds*), *Operating Systems*, Lecture Notes in Computer Science, **16**, Springer-Verlag, Berlin, 89-102.

[CAM 80] Campbell, R.H. and Kolstad, R.B. (1980), An Overview of Path Pascal's Design and Path Pascal User Manual, *ACM Sigplan Notices*, **15**(9), 13-24.

[CAP 88] Capon, P.C. and West, A.J. (1988), Monitoring Occam Channels by Program Transformation, In: Muntean, T. (*ed*), *Parallel Programming of Transputer Based Machines*, Proc. OUG-7, IOS, Amsterdam, 160-169.

[CAR 88a] Carpenter, G.F., Holding, D.J. and Tyrrell, A.M. (1988), The Design and Simulation of Software Fault Tolerant Mechanisms for Application in Distributed Processing Systems, *Microprocessing and Microprogramming*, **22**(3), 175-185.

[CAR 88b] Carreiro, N. and Gelernter, D. (1988), Applications Experience with Linda, Proc. Parallel Programming: Experience with Applications, Languages and Systems, *ACM Sigplan Notices*, **23**(9), 173-187.

[CHO 82] Chow, T.C.K. and Abraham, J.A. (1982), Load Balancing in Distributed Systems, *IEEE Trans. Software Eng.*, **SE-8**, 401-412.

[CHU 80] Chu, W.W., Holloway, L.J., Min-Tsung, L. and Efe, K., (1980), Task Allocation in Distributed Data Processing, *IEEE Computer*, **13**(11), 57-69.

[CLA 86] Clayton, P.G. (1986), *A Notation for Programming with the use of Human-Like Control Structures*, Tech. Doc. 86/1, Department of Computer Science, Rhodes University.

[CLA 87a] Clayton, P.G. (1987), Hands-on Microprogramming for Computer Science Students, *Quaestiones Informaticae*, **5**(2), 63-67.

[CLA 87b] Clayton, P.G. (1987), *HUL - Language Definition*, Tech. Doc. 87/21, Department of Computer Science, Rhodes University.

[CLA 87c] Clayton, P.G. (1987), *Programming Language Constructs for Interrupt-Generating Memory*, Proc. 2nd Research Students' Conference, Stellenbosch, September.

[CLA 89a] Clayton, P.G., Handler, C., and Hill, D.T. (1989), Abstract Process Placement for Distributed Parallel Processing Environments, In: Neishlos, H. (*ed*), *Parallel Processing: Technology and Applications*, IOS, Amsterdam, 123-130.

[CLA 89b] Clayton, P.G. (1989), An Interrupt Driven Paradigm of Concurrent Programming, In: Kritzinger, P. (*ed*), *Proceedings of the 5th Southern African Computer Symposium*, Johannesburg, November, 69-85; also to be published in the *South African Computer Journal* (1990).

[CLA 89c] Clayton, P.G. (1989), *Microprogrammed Emulation of an Automatic Interrupt Generation Mechanism*, Tech. Doc. 89/11, Department of Computer Science, Rhodes University.

[CLA 89d] Clayton, P.G. (1989), *An Investigation into the Use of Interrupt Driven Data Objects with Sequential Programming Language Control Structures*, Tech. Doc. 89/10, Department of Computer Science, Rhodes University.

[COL 82] Cole, A.J. and Morrison, R. (1982), *An Introduction to Programming with S-ALGOL*, Cambridge University Press.

[CON 79] Conway, R. and Gries, D. (1979), *An Introduction to Programming - A Structured Approach Using PL/1 and PL/C*, 3rd ed., Winthrop, N.Y.

[COO 80] Cook, R.P. (1988), *MOD - A Language for Distributed Programming, *IEEE Trans. Software Eng.*, **SE-6**(6), 563-571.

[COX 86] Cox, B.J. (1986), *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, Reading, MA.

[CRO 84] Crookes, D. and Elder, J.W.G. (1984), An Experiment in Language Design for Distributed Systems, *Software - Practice and Experience*, **14**(10), 957-971.

[CRO 87] Crookes, D., Morrow, P.J., Milligan, P., Scott, N.S. and Kilpatrick, P.L. (1987), Notes on Implementing a Language for Transputer Networks, *Microprocessing and Microprogramming*, **21**(1-5), 559-566.

[CUL 88] Culloch, A.D. (1988), Parallel Programming Toolkit for 3L-C, Fortran and Pascal, *Proc. OUG Technical Meeting 8*, IOS, Amsterdam, 23-30.

[CUN 88] Cunningham, W. (1988), An Essential Error Handler, *HOOPLA* (Hooray for Object-Oriented Programming Languages), **1**(3), Object-Oriented Programming for Smalltalk Applications Developers Association.

[CUR 82] Curry, G., Baer, L., Lipkie, D. and Lee, B. (1982), Traits: An Approach to Multiple-Inheritance Subclassing, Proc. Conf. Office Information Systems, *ACM SIGOA*, **9**(2), 1-9.

[DEB 86] Debaere, E.H. and Van Campenhout, J.M. (1986), A Shared-Memory Modula-2 Multiprocessor for Real-Time Control Applications, *Microprocessing and Microprogramming*, **18**(1-5), 213-220.

[DIJ 68] Dijkstra, E.W. (1968), Cooperating Sequential Processes, In: Genuys, F. (*ed*), *Programming Languages*, Academic, N.Y., 43-112.

[DIJ 75] Dijkstra, E.W. (1975), Guarded Commands, Non-Determinacy, and Formal Derivation of Programs, *Comm. ACM*, **18**(8), 453-457.

[DIJ 76] Dijkstra, E.W. (1976), *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ.

[DUR 89] Durbin, R., Miall, C. and Mitchinson, G. (*eds*) (1989), *The Computing Neuron*, Addison-Wesley, Reading, MA.

[DYE 83] Dyer, M.G. (1983), *In-Depth Understanding: A Computer Model of Integrated Processing for Narrative Comprehension*, MIT Press, Cambridge, MA.

[EBE 89] Eberbach, E., McCabe, S.C. and Refenes, A.N. (1989), PARLE: A Language for Expressing Parallelism and Integrating Symbolic and Numeric Computations, *Microprocessing and Microprogramming*, **27**(1-5), 207-214.

[EFE 82] Efe, K. (1982), Heuristic Models of Task Assignment Scheduling in Distributed Systems, *IEEE Computer*, **15**(6), 50-56.

[ELL 88a] Ellison, D. and Natanson, L. (1988), A Talking Bee on the Transputer, In: Kerridge, J. (*ed*), *Developments Using Occam*, Proc. OUG-8, IOS, Amsterdam, 63-75.

[ELL 88b] Ellison, D. (1988), The Perceptron and AI: A New Old Way Forward?, *Computer Bulletin*, March, 27-30.

[ELO 85a] Eloranta, E., Hynynen, J., Hämmäinen, H., Jahkola, J., Kyhälä, A. and Räisänen, J. (1985), A Workbench for Distributed Production Management Systems, Proc. IFIP WG 5.7 on Decent Production Management Systems, *Computers in Industry*, **6**(6), 413-425

[ELO 85b] Eloranta, E., Hynynen, J., Hämmäinen, H. and Jahkola, J. (1985), Models and Design of Distributed Production Management Systems: A Framework for Future Development, In: Falster, P. and Mazumder, R.B. (*eds*), Proc. IFIP WG 5.7, *Modelling Production Management Systems*, North-Holland, Amsterdam, 107-122.

[ESW 76] Eswaran, K.P. (1976), *Specifications, Implementations and Interactions of a Trigger Subsystem in an Integrated Database System*, IBM Research Report RJ1820.

[EVA 82] Evans, C. (1982), *Psychology: A Dictionary of the Mind, Brain and Behaviour*, Arrow, London.

[FAH 83] Fahlman, S.E., Hinton, G.E. and Sejnowski, T.J. (1983), Massively Parallel Architectures for Artificial Intelligence, *Proc. Third National Conference on Artificial Intelligence*, Washington, D.C., 109-113.

[FAY 87] Fay, D.Q.M. and Das, P.K. (1987), Hardware Reconfiguration of Transputer Networks for Distributed Object-Orientated Programming, *Microprocessing and Microprogramming*, **21**(1-5), 623-628.

[FEL 79] Feldman, J.A. (1979), High Level Programming for Distributed Computing, *Comm. ACM*, **22**(6), 353-359.

[FEL 82] Feldman, J.A. and Ballard, D.H. (1982), Connectionist Models and their Properties, *Cognitive Science*, **6**, 205-254.

[FID 83] Fidge, C.J., and Pascoe, R.S.V. (1983), A Comparison of the Concurrency Constructs and Module Facilities of CHILL and Ada, *Australian Computer J.*, **15**(1), 17-27.

[FIS 86] Fisher, A.J. (1986), A Multiprocessor Implementation of Occam, *Software - Practice and Experience*, **16**(10), 875-892.

[FIS 88] Fisher, A.J. (1988), A Critique of Occam Channel Types, *Computer Languages*, **13**(2), 95-105.

[GAJ 85] Gajski, D.D. and Peir, J. (1985), Essential Issues in Multiprocessor Systems, *Computer*, **18**(6), 9-27.

[GEH 84a] Gehani, N.H. and Cargill, T.A. (1984), Concurrent Programming in Ada Language: The Polling Bias, *Software - Practice and Experience*, **14**(5), 413-427.

[GEH 84b] Gehani, N.H. (1984), *Ada - Concurrent Programming*, Prentice-Hall, Englewood Cliffs, NJ.

[GEH 86] Gehani, N.H. and Roome, W.D. (1986), Concurrent C, *Software - Practice and Experience*, **16**(9), 821-844.

[GEL 85] Gelernter, D. (1985), Generative Communication in Linda, *ACM Trans. Programming Languages and Systems*, **7**(1), 80-112.

[GEM 84] Geman, G.E. and Geman, D. (1984), Stochastic Relaxation, Gibbs Distribution, and the Bayesian Restoration of Images, *IEEE Trans. Pattern Analysis and Machine Intelligence*, **PAMI-6**, 721-741.

[GEN 81] Gentleman, W.M. (1981), Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept, *Software - Practice and Experience*, **11**(5), 435-466.

[GOL 83] Goldberg, A. and Robson, D. (1983), *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA.

[GOL 84] Goldberg, A. (1984), *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, MA.

[GOO 75] Goodenough, J.B. (1975), Exception Handling: Issues and a Proposed Notation, *Comm. ACM*, **18**(12), 683-696.

[GOO 79] Good, D.I., Cohen, R.M. and Keeton-Williams, J. (1979), *Principles of Proving Concurrent Programs in Gypsy*, Proc. 6th Annual ACM Symposium: Principles of Programming Languages, San Antonio, January, 42-52.

[GRE 75] Greif, I. and Hewitt, C.E. (1975), *Actor Semantics of PLANNER-73*, Proc. ACM SIGPLAN-SIGACT Conf., Palo Alto, CA, January.

[GRE 87] Gregory, S. (1987), *Parallel Logic Programming in PARLOG*, Addison-Wesley, Wokingham, England.

[GRO 88] Grossman, T., Meir, R. and Domany, E. (1988), Learning by Choice of Internal Representations, *Complex Systems*, **2**, 555-575.

[HÄM 87] Hämmäinen, H. and Eloranta, E. (1987), Object-Oriented Data Communication for Loosely Coupled Control, Proc. IFIP WG 5.7 on Information Flow in Automated Manufacturing Systems, *Computers in Industry*, **9**(4), 319-328.

[HAN 89] Handler, C. (1989), *Parallel Process Placement*, MSc Thesis, Department of Computer Science, Rhodes University.

[HAR 86a] Harland, D.M. (1986), *Concurrency and Programming Languages*, Ellis Horwood, Chichester, England.

[HAR 86b] Hartman, M.H. (1986), *The HIDE User Manual*, Tech. Doc. 86/02, Department of Computer Science, Rhodes University.

[HAR 87] Harel, D. (1987), *Algorithmics - The Spirit of Computing*, Addison-Wesley, Wokingham, England.

[HEA 87] Heath, M.T. (*ed*) (1987), *Hypercube Multiprocessors 1987: Proc. 2nd Conference on Hypercube Multiprocessors*, SIAM Press, Philadelphia, PA.

[HEB 49] Hebb, D.O. (1949), *The Organization of Behaviour*, Wiley, NY.

[HEN 86] Henderson, K.A. (1986), *The LALR(1) HUL Parser*, Honours Project, Department of Computer Science, Rhodes University.

[HEW 69] Hewitt, C.E. (1969), *PLANNER: A Language for Proving Theorems in Robots*, Proc. First Int. Joint Conf. Artificial Intelligence, Washington, DC, 295-301.

[HEW 72] Hewitt, C.E. (1972), *Description and Theoretical Analysis of PLANNER, a Language for Proving Theorems and Manipulating Models in a Robot*, MIT AI Laboratory, **TR-258.**

[HEW 77] Hewitt, C.E. (1977), Viewing Control Structures as Patterns of Passing Messages, *Journal of Artificial Intelligence*, **8**(3), 323-364.

[HEW 79a] Hewitt, C.E., and Atkinson, R.R. (1979), Specification and Proof Techniques for Serializers, *IEEE Trans. Software Eng.*, **SE-5**(1), 10-23.

[HEW 79b] Hewitt, C.E., Attardi, G. and Lieberman, H. (1979), Specifying and Proving Properties of Guardians for Distributed Systems, In: Kahn, G. (*ed*), *Semantics of Concurrent Computation*, Lecture Notes in Computer Science, **70**, Springer-Verlag, Berlin, 316-336.

[HEW 80] Hewitt, C.E., Attardi, G. and Simi, M. (1980), *Knowledge Embedding with a Description System*, Proc. First National Annual Conf. Artificial Intelligence, Stanford, CA, August, 157-164.

[HIL 84] Hillis, W.D. (1984), The Connection Machine: A Computer Architecture based on Cellular Automata, *Physica*, **10D**, 213-228.

[HIL 85] Hillis, W.D. (1985), *The Connection Machine*, MIT Press, Cambridge, MA.

[HIL 86] Hillis, W.D. and Steele, G.L. Jr. (1986), Data Parallel Algorithms, *Comm. ACM*, **29**(12), 1170-1183.

[HIL 87] Hill, D.T. (1987), *Towards a Portable Occam*, MSc Thesis, Department of Computer Science, Rhodes University; also in Tech. Doc. 88/04, Department of Computer Science, Rhodes University.

[HIN 85] Hinton, G.E., Sejnowski, T. and Ackley, D. (1985), Boltzmann Machines: Constraint Satisfaction Machines that Learn, *Cognitive Science*, **9**, 147-169.

[HOA 74] Hoare, C.A.R. (1974), Monitors: An Operating System Structuring Concept, *Comm. ACM*, **17**(10), 549-557.

[HOA 76] Hoare, C.A.R. (1976), The Structure of an Operating System, In: Bauer, F.L. and Samelson, K. (*eds*), *Language Hierarchies and Interfaces*, Lecture Notes In Computer Science, **46**, Springer-Verlag, Berlin.

[HOA 78] Hoare, C.A.R. (1978), Communicating Sequential Processes, *Comm. ACM*, **21**(8), 666-677.

[HOA 85] Hoare, C.A.R. (1985), *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, NJ.

[HOL 83a] Holt, R.C. and Cordy, J.R. (1983), *The Turing Language Report*, CSRI, University of Toronto.

[HOL 83b] Holt, R.C. (1983), *Concurrent Euclid, the Unix System, and Tunis*, Addison-Wesley, Reading, MA.

[HOP 89] Hoptroff, R.G. and Hall, T.J. (1989), Learning by Diffusion for Multilayer Perceptron, *Electronics Letters*, **25**(8), 531-533.

[HOR 89] Horwat, W., Chien, A.A. and Dally, W.J. (1989), Experience with CST: Programming and Implementation, Proc. SIGPLAN^{89} Conf.: Programming Language Design and Implementation, *ACM Sigplan Notices*, **24**(7), 101-109.

[HUR 87] Hur, J.H. and Chon, K. (1987), *Overview of a Parallel Object-Oriented Language CLIX*, Proc. ECOOP^{87} (European Conference on Object-Oriented Programming), Lecture Notes in Computer Science, **276**, Springer-Verlag, Berlin, 265-273.

[HWA 85] Hwang, K. and Briggs, F.A. (1985), *Computer Architecture and Parallel Processing*, McGraw-Hill, Singapore.

[IAN 89] Ianello, G., Mazzeo, A. and Ventre, G. (1989), Definition of the DISC Concurrent Language, *SIGPLAN Notices*, **24**(6), 59-68.

[ING 78] Ingalls, P.Z. (1978), *The Smalltalk-76 Programming System: Design and Implementation*, Proc. 5th Annual ACM Symposium: Principles of Programming Languages, Tucson, AZ, January, 9-16.

[INM 83] INMOS Ltd. (1983), *Occam Programming Manual*, Inmos Ltd., Bristol.

[INM 84a] INMOS Ltd. (1984), *Occam Programming Manual*, Prentice-Hall, Englewood Cliffs, NJ.

[INM 84b] INMOS Ltd. (1984), *Transputer Reference Manual*, Inmos Ltd., Bristol.

[INM 84c] INMOS Ltd. (1984), *Occam Evaluation Kit*, Inmos Ltd., Bristol.

[INM 87a] INMOS Ltd. (1987), *The TDS Reference Manual*, Inmos Ltd., Bristol.

[INM 87b] INMOS Ltd. (1987), *Occam 2 Product Definition*, Inmos Ltd., Bristol.

[INM 88] INMOS Ltd. (1988), *The Transputer Instruction Set - A Compiler Writers' Guide*, Prentice-Hall, Englewood Cliffs, NJ.

[INT 88] Intel, *A Technical Summary of the iPSC/2 Concurrent Computer*, Intel Scientific Computers, Beaverton, CA.

[ISH 87] Ishikawa, Y. and Tokoro, M. (1987), Orient 84/K: An Object-Oriented Concurrent Programming Language for Knowledge Representation, In: Yonezawa, A. and Tokoro, M. (*eds*), *Object-Oriented Concurrent Programming*, MIT Press, Cambridge, MA.

[JAZ 80] Jazayeri, M. (1980), *CSP/80: A Language for Communicating Sequential Processes*, Proc. Fall IEEE COMPCON80, IEEE, NY, 736-740.

[JOH 88] Johannet, A., Lohéac, G., Personnaz, L., Guyon, I. and Dreyfus, G. (1988), A Transputer-Based Neurocomputer, In: Muntean, T. (*ed*), *Parallel Programming of Transputer Based Machines*, Proc. OUG-7, IOS, Amsterdam, 120-127.

[JON 85] Jones, G. (1985), *Programming in Occam*, Oxford University Programming Research Group, **PRG-43**; also published under the same title (1987) by Prentice-Hall, Englewood Cliffs, NJ.

[JON 88] Jones, G. (1988), On Guards, In: Muntean, T. (*ed*), *Parallel Programming of Transputer Based Machines*, Proc. OUG-7, IOS, Amsterdam, 15-24.

[KAF 89] Kafura, D.G. and Lee, K.H. (1989), Inheritance in Actor Based Concurrent Object-Oriented Languages, *The Computer Journal*, **32**(4), 297-304.

[KAH 82] Kahn, K. (1982), Intermission - Actors in Prolog, In: *Logic Programming*, Academic Press, NY, 213-230.

[KAH 86] Kahn, K., Tribble, E.D., Miller, M.S. and Bobrow, D.G. (1986), Objects in Concurrent Logic Programming Languages, Proc. OOPSLA^{86} (Object-Oriented Programming Systems, Languages and Applications), *ACM Sigplan Notices*, **21**(11), 242-257.

[KAL 87] Kallstrom, M. (1987), *Channel Multiplexing for Occam 2 Programs*, PSF/MU/WP4/87/2, Department of Computer Science, University of Manchester.

[KAY 77] Kay, A. and Goldberg, A (1977), Personal Dynamic Media, *IEEE Computer*, **10**(3), 31-39.

[KER 86] Kerridge, J. and Simpson, D. (1986), Communicating Parallel Processes, *Software - Practice and Experience*, **16**(1), 63-86.

[KIM 89] Kim, Y.J. and Kim, G.C. (1989), Coordinator: A Modification of the Monitor Concept, *Information Processing Letters*, **32**(2), 73-80.

[KIR 89] Kirkerud, B. (1989), *Object-Oriented programming with Simula*, Addison-Wesley, Wokingham, England.

[KNO 89] Knowles, A. and Kantchev, T. (1989), Message Passing in a Transputer System, *Microprocessors and Microsystems*, **13**(2), 113-123.

[KNU 73] Knuth, D.E. (1973), *The Art of Computer Programming, Vol. 3 : Sorting and Searching*, Addison-Wesley, Reading, Massachusetts.

[LAM 80] Lampson, B.W. and Redell, D.D. (1980), Experience with Processes and Monitors in Mesa, *Comm. ACM*, **23**(2), 105-117.

[LIE 87] Lieberman, H. (1987), Concurrent Object-Oriented Programming in Act 1, In: Yonezawa, A. and Tokoro, M. (*eds*), *Object-Oriented Concurrent Programming*, MIT Press, Cambridge, MA, 9-36.

[LIS 77] Liskov, B.H. and Snyder, A. (1977), Abstraction Mechanisms in CLU, *Comm. ACM*, **20**(8), 564-576.

[LIS 79] Liskov, B.H. and Snyder, A. (1979), Exception Handling in CLU, *IEEE Trans. Software Eng.*, **SE-5**(6), 546-558.

[LIS 82] Liskov, B.H. (1982), On Linguistic Support for Distributed Programs, *IEEE Trans. Software Eng.*, **SE-8**(3), 203-210.

[LIS 83] Liskov, B.H. and Scheifler, R. (1983), Guardians and Actions: Linguistic Support for Robust, Distributed Programs, *ACM Trans. Programming Languages and Systems*, **5**(3), 381-404.

[LIS 88a] Liskov, B.H. (1988), Distributed Programming in Argus, *Comm. ACM*, **31**(3), 300-312.

[LIS 88b] Liskov, B.H. (1988), Data Abstraction and Hierarchy, *ACM Sigplan Notices*, **23**(5), 17-34.

[MAL 88] Malcolm, M. (1988), Exception Handling, *HOOPLA* (Hooray for Object-Oriented Programming Languages), **1**(2), Object-Oriented Programming for Smalltalk Applications Developers Association.

[MAO 80] Mao, T.W. and Yeh, R.T. (1980), Communication Ports: A Language Concept for Concurrent Programming, *IEEE Trans. Software Eng.*, **SE-6**(2), 194-204.

[MAY 83] May, D. (1983), Occam, *ACM Sigplan Notices* **18**(4), 69-79.

[MAY 84] May, D. and Taylor R. (1984), Occam, *Microprocessors and Microsystems*, **8**(2), 73-79.

[MAY 87] May, D. (1987), *Occam 2 Language Definition*, Inmos Ltd., Bristol.

[MCC 43] McCulloch, W.S. and Pitts, W. (1943), A Logical Calculus of the Ideas Immanent in Neural Nets, *Bulletin of Mathematical Biophysics*, **5**, 115-137.

[MEL 86] Mellor, P.V., Dubery, J.M. and Whitehead, D.G. (1986), Adapting Modula-2 for Distributed Systems, *Software Engineering Journal*, **1**(5), 184-189.

[MEY 87] Meyer, B. (1987), Eiffel: Programming for Reusability and Extendibility, *ACM Sigplan Notices*, **22**(2), 85-94.

[MEY 88] Meyer, B. (1988), *Object-Oriented Software Construction*, Prentice-Hall, Englewood Cliffs, NJ.

[MIN 69] Minsky, M. and Papert, S. (1969), *Perceptrons: An Introduction to Computational Geometry*, MIT Press, Cambridge, MA.

[MIT 79] Mitchell, J.G., Maybury, W. and Sweet, R. (1979), *Mesa Language Manual*, Version 5.0, Xerox PARC, Tech. Rep. **CSL-79-3**, Palo Alto, CA.

[MOO 84] Moon, D. and Weinreb, D. (1984), *Guide to Symbolics Lisp*, Symbolics, MIT Press, Cambridge, MA.

[MUR 88] Murray, K.A., and Wellings, A.J. (1988), Issues in the Design and Implementation of a Distributed Operating System for a Network of Transputers, *Microprocessing and Microprogramming*, **24**(1-5), 169-178.

[MYE 78] Myers, G.J. (1978), *Advances in Computer Architecture*, Wiley, N.Y.

[NAU 63] Naur, P. (1963), Revised Report on the Arithmetic Language Algol 60, *Comm. ACM*, **6**(1), 1-17.

[NIE 87] Nierstrasz, O.M. (1987), Active Objects in Hybrid, Proc. OOPSLA^{87} (Object-Oriented Programming Systems, Languages and Applications), *ACM Sigplan Notices*, **22**(12), 243-253.

[NG 87] Ng, K.W., and Mok, K.Y. (1987), The High Level Language and Operating System Support Features of Advanced Microprocessors, Parts 1 and 2, *Microprocessing and Microprogramming*, **19**(3), 203-218, and **19**(4), 227-289.

[NOR 88] Norman, M. and Wilson, S. (1988), *TITCH: Topology Independent Transputer Communications Harness*, Supercomputer Project Report, Edinburgh, UK, May.

[OAK 88] Oakley, H. (1988), Transputer Programming Tools, *Personal Computer World*, **11**(9), 116-118.

[OUS 81] Ousterhout, J.K. (1981), *MEDUSA: A Distributed Operating System*, UMI Research Press.

[PER 79] Perrott, R.H. (1979), Languages for Parallel Computers, In: McKeag, R.M. and Macnaghten, A.M. (*eds*), *On the Construction of Programs*, Cambridge University Press.

[PER 87] Perrott, R.H. (1987), *Parallel Programming*, Addison-Wesley, Wokingham, England.

[PET 77] Peterson, J.L. (1977), Petri Nets, *Computing Surveys*, **9**(3), 223-252.

[PET 81] Peterson, J.L. (1981), *Petri Net Theory and the Modelling of Systems*, Prentice-Hall, Englewood Cliffs, NJ.

[POW 88] Power, L. (1988), Report on the Panel Discussion: Object-Oriented Concurrency, Addendum to Proc. OOPSLA'87 (Object-Oriented Programming Systems, Languages and Applications), *ACM Sigplan Notices*, **23**(5), 119-127.

[QCA 84] QCAD, Inc. (1984), *QPARSER Translator Writing System*, QCAD Systems, Inc., San Jose, C.A.

[QUI 87] Quinn, M.J. (1987), *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, Singapore.

[RAS 86] Rasmussen, J.B. (1986), Real-Time Interrupt Handling in Ada, *Software - Practice and Experience*, **17**(3), 197-213.

[REF 89] Refenes, A.N., Eberbach E. and McCabe, S.C. (1989), *PARLE: A Parallel Target Language for Integrating Symbolic and Numeric Processing*, Proc. Conf. Parallel Architectures and Languages Europe, Eindhoven, The Netherlands; also to be published in *Parallel Architectures and Languages*, Lecture Notes in Computer Science, Springer-Verlag, Berlin.

[ROB 81] Roberts, E.S., Evans, A., Jr., Morgan, C.R. and Clarke, E.M. (1981), Task Management in Ada - A Critical Evaluation for Real-Time Multiprocessors, *Software - Practice and Experience*, **11**(10), 1019-1051.

[ROP 81] Roper, T.J. and Barter, C.J. (1981), A Communicating Sequential Process Language and Implementation, *Software - Practice and Experience*, **11**(11), 1215-1234.

[ROS 57] Rosenblatt, F. (1957), *The Perceptron, A Perceiving and Recognizing Automaton (Project PARA)*, Cornell Aeronautical Laboratory Report No. 85-460-1.

[ROS 62] Rosenblatt, F. (1962), *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*, Spartan, NY.

[ROS 85] Roscoe, A.W. and Hoare, C.A.R. (1986), *The Laws of Occam Programming*, Oxford University Programming Research Group, **PRG-53**.

[ROS 88] Roscoe, A.W. (1988), Routing Messages through Networks: An Exercise in Deadlock Avoidance, In: Muntean, T. (*ed*), *Parallel Programming of Transputer Based Machines*, Proc. OUG-7, IOS, Amsterdam, 55-79.

[RUM 86] Rumelhart, D.E., McClelland, J.L. and The PDP Research Group (*eds*) (1986), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Volumes 1 and 2, MIT Press, Cambridge, MA.

[RUS 88] Russo, V., Johnston, G. and Campbell, R. (1988), Process Management and Exception Handling in Multiprocessor Operating Systems using Object-Oriented Design Techniques, Proc. OOPSLA^{88} (Object-Oriented Programming Systems, Languages and Applications), *ACM Sigplan Notices*, **23**(11), 248-258.

[SAN 85] Sanger, W.L. (1985), *The HUL Parser*, Honours Project, Department of Computer Science, Rhodes University.

[SCH 85] Schneider, F.B. and Andrews, G.R. (1985), Concepts for Concurrent Programming, In: de Bakker, J.W., de Roever, W.P. and Rozenberg, G. (*eds*), *Current Trends in Concurrency*, Lecture Notes in Computer Science, **224**, Springer-Verlag, 669-716.

[SCH 86] Schaffert, C., Cooper, T., Bullis, B., Kilian, M. and Wilpot, C. (1986), An Introduction to Trellis/Owl, Proc. OOPSLA^{86} (Object-Oriented Programming Systems, Languages and Applications), *ACM Sigplan Notices*, **21**(11), 9-16.

[SCH 88] Schoonees, J.A. (1988), *Parallel Distributed Processing: Practical Applications of Neural Networks in Signal Processing*, Proc. COMSIG^{88} (Communications and Signal Processing), IEEE, NY, 76-80.

[SCO 89] Scott, R.B. and Trehan, R. (1989), Translating from PARLOG to Occam 2: A Methodology, *Concurrency: Practice and Experience*, **1**(1), 105-134.

[SEL 59] Selfridge, O.G. (1959), Pandemonium: A Paradigm for Learning, In: *The Mechanization of Thought Processes*, Her Majesty's Stationery Office, London.

[SIL 81] Silberschatz, A. (1981), Port Directed Communication, *Computer Journal*, **24**(1), 78-82.

[SIM 83] Simons, G.L. (1983), *Towards Fifth-Generation Computers*, NCC, Manchester.

[SNY 86a] Snyder, L. (1986), Type Architectures, Shared Memory and the Corollary of Modest Potential, In: Traub, J.F., Grosz, B.J., Lampson, B.W. and Nilsson, N.J. (*eds*), *Annual Review of Computer Science*, Volume 1, Annual Reviews Inc., Palo Alto, CA.

[SNY 86b] Snyder, A. (1986), CommonObjects: An Overview, *ACM Sigplan Notices*, **21**(10), 19-28.

[STA 84] Stankovic, J.A. and Sidhu, I.S. (1984), *An Adaptive Bidding Algorithm for Processes, Clusters, and Distributed Groups*, Proc. 4th Int. Conf. Distributed Computer Systems, San Francisco, CA, May.

[STA 85] Stammers, R.A. (1985), Ada on Distributed Hardware, In: Reijns, G.L., and Dagless, E.L. (*eds*), *Concurrent Languages in Distributed Systems*, Elsevier, North Holland.

[STE 86] Steele, G.L. Jr. and Hillis, W.D. (1986), *Connection Machine LISP: Fine-Grained Parallel Symbolic Processing*, Proc. ACM Conf. LISP and Functional Programming, Cambridge, MA, August, 279-297.

[STR 83] Strom, R. and Yemini, S. (1983), NIL: An Integrated Language and System for Distributed Programming, Proc. Symposium on Language Issues in Software Systems, *ACM Sigplan Notices*, **18**(6), 73-82.

[STR 86] Stroustrup, B. (1986), *The C++ Programming Language*, Addison-Wesley, Reading, MA.

[SWI 86] Swinehart, D., Zellweger, P., Beach, R. and Hagmann, R. (1986), A Structural View of the Cedar Programming Environment, *ACM Trans. Programming Languages and Systems*, **8**(4), 419-490.

[TER 86] Terry, P.D. (1986), *Programming Language Translation: A Practical Approach*, Addison-Wesley, Wokingham, England.

[THO 74] Thompson, K. and Ritchie, D.M. (1974), The UNIX™ Time-Sharing System, *Comm. ACM*, **17**(7), 365-375.

[THO 89] Thomas, I. (1989), Support System for Occam Objects on Transputers, *Microprocessors and Microsystems*, **13**(2), 129-137.

[TRI 89] Tripathi, A., Berge, E. and Aksit, M. (1989), An Implementation of the Object-Oriented Concurrent Programming Language SINA, *Software - Practice and Experience*, **19**(3), 235-256.

[TSI 79] Tsichritzis, D. (1979), *A Form Manipulation System*, Proc. Symposium on Automated Office Systems, New York University, Department of Computer Science, March.

[TUR 89] *Turbo Pascal 5.5®: Object-Oriented Programming Guide*, Borland Inc., Scotts Valley, CA.

[VAN 80] Van den Bos, J. (1980), Comments on Ada Process Communication, *ACM Sigplan Notices*, **15**(6), 77-81.

[VAN 81] Van den Bos, J., Plasmeijer, R. and Stroet, J. (1981), Process Communication Based on Input Specifications, *ACM Trans. Programming Languages and Systems*, **3**(3), 224-250.

[WAL 85] Walker, P. (1985), The Transputer: A Building Block for Parallel Processing, *Byte*, **10**(5), 219-235.

[WEG 83] Wegner, P. and Smolka, S.A. (1983), Processes, Tasks, and Monitors: A Comparative Study of Concurrent Programming Primitives, *IEEE Trans. Software Eng.*, **SE-9**(4), 446-462.

[WEG 87] Wegner, P. (1987), The Object-Oriented Classification Paradigm, In: Shriver, P. and Wegner, P. (*eds*), *Research Directions in Object-Oriented Programming*, MIT Press, Cambridge, MA.

[WEL 79] Welsh, J. and Bustard, D.W. (1979), Pascal Plus - Another Language for Modular Multiprogramming, *Software - Practice and Experience*, **9**, 947-957.

[WEL 81] Welsh, J. and Lister, A. (1981), A Comparative Study of Task Communication in Ada, *Software - Practice and Experience*, **11**(3), 257-290.

[WEL 84] Wellings, A.J., Keeffe, D. and Tomlinson, G.M. (1984), A Problem with Ada and Resource Allocation, *Ada Letters*, **3**(4).

[WEL 86] Welch, P.H. (1986), *A Structured Technique for Concurrent Systems Design in Ada*, Proc. Ada-Europe Conference, Edinburgh, May.

[WIN 79] Winston, P.H. (1979), *Artificial Intelligence*, Addison Wesley, Reading, Massachusetts.

[WIR 77] Wirth, N. (1977), Modula: A Language for Modular Multiprogramming, *Software - Practice and Experience*, 7(1), 3-35.

[WIR 85a] Wirth, N. (1985), From Programming Language Design to Computer Construction, 1984 ACM Turing Award Lecture, *Comm. ACM*, 28(2), 160-164.

[WIR 85b] Wirth, N. (1985), *Programming in Modula-2*, 3rd ed., Springer-Verlag, Berlin.

[WIR 88a] Wirth, N. (1988), From Modula to Oberon, *Software - Practice and Experience*, 18(7), 661-670.

[WIR 88b] Wirth, N. (1988), The Programming Language Oberon, *Software - Practice and Experience*, 18(7), 671-690.

[WOL 89] Wolf, W. (1989), A Practical Comparison of Two Object-Oriented Languages, *IEEE Software*, 6(5), 61-68.

[WRE 86] Wrench, K.L. (1986), *CSP-i: An Implementation of CSP*, MSc Thesis, Department of Computer Science, Rhodes University.

[WRE 88] Wrench, K.L. (1988), CSP-i: An Implementation of Communicating Sequential Processes, *Software - Practice and Experience*, 18(6), 545-560.

[YOK 86] Yokote, Y. and Tokoro, M. (1986), The Design and Implementation of ConcurrentSmalltalk, Proc. OOPSLA^{86} (Object-Oriented Programming Systems, Languages and Applications), *ACM Sigplan Notices*, 21(11), 331-340.

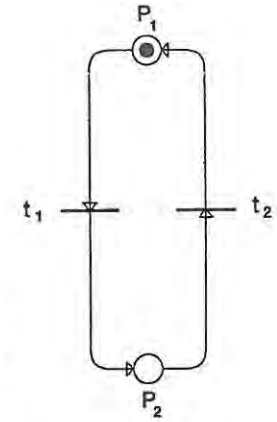
[YON 86] Yonezawa, A., Briot, J. and Shibayama, E. (1986), Object-Oriented Concurrent Programming in ABCL/1, Proc. OOPSLA^{86} (Object-Oriented Programming Systems, Languages and Applications), *ACM Sigplan Notices*, 21(11), 258-268.

Appendix A

Glossary of Symbols

Petri net theory

This thesis makes use of the pictorial representation of Petri nets as graphs.

Notation	*Meaning*	*Example*
○	*Places*, P_i, are represented by circles.	
——	*Transitions*, t_j, are represented by bars.	
⌐	Directed arcs connect places and transitions.	
●	*Tokens* move between places and control the execution of the Petri net. Tokens are moved by *firing* the transitions of the net. A transition must be *enabled* (all of its input places must have a token) in order to fire. Transitions fire by removing the enabling tokens from their input places and generating new tokens which are deposited in their output places.	

For a detailed description of the theory of Petri nets, the reader is referred to the text by Peterson [PET 81].

Efficiency of algorithms

Notation	*Meaning*	*Example*
O(N)	An algorithm has a complexity (running time when considering time complexity) of order *N* (known as the big-*O* notation)	$O(N.log_2N)$

Notation	*Meaning*	*Example*
$\Omega(N)$	The lower bound on the complexity of a solution is $O(N)$ (known as the big-Ω notation)	$\Omega(N^3)$

For more details on the study of algorithm comparison, the reader is referred to texts by Harel [HAR 87] and Knuth [KNU 73].

CSP notation

Notation	*Meaning*	*Example*
words in lower case	denote distinct events (*a, b, c*) or variables denoting events (*x, y, z*)	*add, move, signal*
words in upper case	denote specific processes (*P, Q*)	*PRODUCER, SPOOLER*
	or variables denoting processes (*X, Y*)	
	or sets of events (*A, B*)	as in $(x{:}A \rightarrow P(x))$ below
$STOP_A$	the process with alphabet A which never engages in any the events of A	
$SKIP_A$	the process with alphabet A which does nothing but terminate successfully	
$=$	equals	$x = x$
$\neq$	is distinct from	$x \neq x+1$
$\in$	is a member of	*Patrick* $\in$ *PROFESSORS*
$\notin$	is not a member of	*Patrick* $\notin$ *FEMALES*
$P \wedge Q$	P and Q (both true)	$x \leq x+1 \wedge x \neq x+1$
$P \vee Q$	P or Q (one or both true)	$x \leq y \vee y \leq x$
$\neg P$	not P	$\neg\ 3 > 5$
$P \equiv Q$	P if and only if Q	$x < y \equiv y > x$
$P \Rightarrow Q$	if P then Q	$x \leq y \Rightarrow x < y$
$\exists x.P$	there exists an x such that P	$\exists x\ .\ x > y$
$\forall x.P$	for all x, P	$\forall x\ .\ x < x + 1$
αP	the alphabet of process P	α *SUN* $=$ {*rise, set*}

Notation	*Meaning*	*Example*
$\alpha\ c$	the set of messages communicable on channel c	
$c.v$	the event on channel c which communicates the message with value v	$\alpha\, c(P) = \{v \mid c.v \in \alpha P\}$
$\cup$	union of sets	$\alpha\ P = \alpha\ Q \ \cup\ \alpha\ R$
$a \rightarrow P$	a then P	
$(a \rightarrow P \ [] \ b \rightarrow Q)$	a then P choice b then Q (for deterministic and non-deterministic traces)	
$(x{:}A \rightarrow P(x))$	(choice of) x from A then $P(x)$	
$\mu X{:}A.F(X)$	the process X with alphabet A such that $X = F(X)$	
$P \parallel Q$	P in parallel with Q	
$P \ [] \ Q$	P choice Q	
$l{:}P$	P with name l	
$x := e$	x takes on the value e	
$b\ !\ e$	on channel b output the value of e	
$b\ ?\ x$	from channel b input to x	
$P\ ;\ Q$	P successfully followed by Q	

For the complete set of symbols used in CSP, the reader is referred to the text by Hoare [HOA 85].

Appendix B

HUL Reserved Identifiers and Predeclared Procedures

Input and Output

Ready This standard system function is preceded in the source code by a channel identifier, and returns a Boolean value indicating whether a message is awaiting transmission on that channel or not.

Screen and keyboard

These two standard system channels may be used by the programmer to communicate interactively with the workstation. They are mapped onto the root processor, and conform to the rules which govern user declared channels in HUL. They facilitate communication between one process of the application program and a system process which is always ready to receive or send.

Read [*variable list*]

This standard procedure can be used to read interactively from the workstation at any point in the program that an input primitive is allowed. The *read* procedure is based on the Pascal [BSI 82] procedure with the same name, and allows any number of integer, string, or character actual parameters to be listed in the *variable list*, separated by commas. The *read* procedure may be called by any number of processes (it is not limited to use by only one process in the way *keyboard* is). *Read* is implemented using a system multiplexing process to route data from the workstation to any process of the application program via a set of transparent system channels.

Write [*list of output values*]

The *write* procedure can be called from any process in the system to write interactively to the workstation. Its implementation is similar to the *read* procedure; a system multiplexing process routes data to the workstation from any process of the application program via a set of transparent system channels which are always ready to receive. Any

number of integer or string expressions may appear in the *list of output values*, separated by commas. An output value may as an option specify a field width designator using the Pascal format. For example

```
Write["Answer is", Ans:5]
```

Readln This procedure is similar to *read*, and forces the interactive input from the workstation to begin on a new line.

Writeln This procedure is similar to *write*, and causes subsequent interactive output sent to the workstation to begin on a new line.

Message *Message* behaves as a standard system constant in output statements, and as a standard system variable in input statements. It is used when synchronization is required without the transfer of a particular message.

Constants

On, yes These are standard alternate constants representing the value *true*.

No, off These are standard alternate constants representing the value *false*.

Otherwise This is a standard alternate constant representing the value *true*. *Otherwise* is used primarily in *if* control structures as the last condition clause, to implement an *otherwise* option.

Miscellaneous

Random [*X, Min, Max*]

This standard procedure returns a random integer value in *X*, which is in the range *Min* <= *X* < *Max*. This procedure is useful in an *if* control structure for non-deterministically selecting one of a number of alternative component processes which are able to proceed.

Appendix C

The Syntax of HUL

C.1 BNF description

The following is a top down description of the syntax of HUL. The customary BNF extensions have been used in this description: { *digit* } denotes zero or more digits, [*qualifier*] indicates the optional presence of a qualifier, and ("+" | "-") represents the presence of either a plus or a minus sign. Terminal symbols appear in boldface, and single character terminal symbols are placed between quotation marks (unless the definition consists of a simple list of terminal symbols) to avoid any confusion with other symbols in the BNF notation. An attempt has been made in the layout of the BNF definitions to illustrate the program structure of HUL, which is denoted by indenting the source code from the left margin. HUL is not case sensitive, and each HUL statement must begin on a new line.

1 *program* = *process*

2 *process* = { *definitions* } *control-structure*

3 *definitions* = **ignore** *identifier* { "," *identifier* } ":" |
(**def** | **define**) *def-definition* { "," *def-definition* } ":" |
(**alt** | **alternate**) *identifier* "=" *terminal* { "," *identifier* "=" *terminal* } ":" |
(**var** | **variable**) *var-declaration* { "," *var-declaration* } ":" |
(**chan** | **channel**) *chan-declaration* { "," *chan-declaration* } ":" |
(**proc** | **procedure**) *proc-declaration* ":"

4 *def-definition* = *identifier* "=" *constant* |
identifier "=" **table** "[" *constant-list* "]" |
identifier "=" **not** *boolean-variable*

5 *proc-declaration* = *identifier* ["[" *formal-parameter* { "," *formal-parameter* } "]"] "="
process

6 *formal-parameter* = (**variable** | **var** | **value**) *var-declaration* { "," *var-declaration* } |
(**channel** | **chan**) *chan-declaration* { "," *chan-declaration* }

7 *var-declaration* = *type* *identifier-declaration* { "," *identifier-declaration* }

8 *chan-declaration* = *identifier-declaration* { "," *identifier-declaration* }

9 *identifier-declaration* = *identifier* ["[" (*number* | *numeric-defined-identifier*) "]"]

10 *type* = **integer** | **int** | **boolean** | **bool** | **character** | **char**

11 *control-structure* = *do* | *if*

12 *do* = **do** [**intact**] [*count*] [*qualifier*]
process-body

13 *if* = **if**
condition [*replicator*]
process-body
{ *condition* [*replicator*]
process-body }

14 *count* = **once** | **twice** | *simple-value* **times**

15 *qualifier* = **in parallel** | **in sequence** | **parallel** | **sequence** | **par** | **seq**

16 *process-body* = *statement*
{ *statement* }
{ *when-statement* [*replicator*] }

17 *statement* = *primitive* [*replicator*] | *procedure-call* [*replicator*] |
control-structure | *process*

18 *primitive* = *assignment* | *input* | *output* | *wait-until* | **skip**

19 *procedure-call* = *identifier* ["[" *actual-parameter* { "," *actual-parameter* } "]"]

20 *actual-parameter* = *expression* | *channel-identifier*

21 *replicator* = [","] **varying** *identifier* **from** *numeric-expression* **to** *numeric-expression*

22 *wait-until* = **wait until** ([**not**] *boolean-variable* | *variable* = *expression*)

23 *assignment* = *variable* ("<-" | ":=" | **is**) *expression* |
[**not**] *boolean-variable* |
boolean-variable ("<-" | ":=" | **is**) *condition*

24 *input* = *channel-identifier* (**receive** | "?") (**message** | *variable* { "," *variable* })

25 *output* = *channel-identifier* (**send** | "!") (**message** | *output-value* { "," *output-value* })

26 *output-value* = *expression* | *condition*

27 *when-statement* = **when** ([**not**] *boolean-variable* | *variable* = *expression*)
action

28 *action* = **wait** |
{ *primitive* }
[*stop*]

29 *condition* = [**not**] *boolean-expression* { (**and** | **or**) *boolean-expression* }

30 *boolean-expression* = *boolean-variable* |
boolean-constant |
expression ("=" | "<>" | "<" | ">" | "<=" | ">=") *expression* |
channel-identifier **ready** |
"(" *condition* ")"

31 *numeric-expression* = ["+" | "-"] *term* { ("+" | "-") *term* }

32 *expression* = *numeric-expression* |
string { "+" *string* }

33 *term* = *factor* { ("*" | "/" | **mod**) *factor* }

34 *factor* = *simple-value* | "(" *numeric-expression* ")"

35 *simple-value* = *variable* | *integer-constant* | *numeric-defined-identifier*

36 *string* = *string-variable* | *string-constant* | *character-variable* |
character-constant

37 *string-variable* = *character-array-identifier*

38 *character-variable* = *variable*

39 *boolean-variable* = *variable*

40 *variable* = *identifier* ["[" *numeric-expression* "]"]

41 *channel-identifier* = *identifier* ["[" *numeric-expression* "]"]

42 *numeric-defined-identifier* = *identifier*

43 *character-array-identifier* = *identifier*

44 *identifier* = *letter* { *letter* | *digit* | "_" }

45 *constant* = *simple-constant* | *string-constant*

46 *constant-list* = *simple-constant* { "," *simple-constant* }

47 *simple-constant* = *integer-constant* | *character-constant* | *boolean-constant*

48 *string-constant* = '"' { *character* } '"'

49 *character-constant* = "'" *character* "'" | *number*

50 *integer-constant* = ["+" | "-"] *number*

51 *number* = *digit* { *digit* }

52 *boolean-constant* = **true** | **yes** | **on** | **otherwise** | **false** | **no** | **off**

53 *terminal* = **alt** | **alternate** | **and** | **bool** | **boolean** | **char** | **character** | **chan** | **channel** | **def** | **define** | **do** | **if** | **ignore** | **in** | **intact** | **int** | **integer** | **is** | **false** | **from** | **message** | **mod** | **no** | **not** | **off** | **on** | **once** | **or** | **otherwise** | **par** | **parallel** | **proc** | **procedure** | **ready** | **receive** | **send** | **seq** | **sequence** | **skip** | **stop** | **table** | **times** | **to** | **true** | **twice** | **until** | **value** | **var** | **variable** | **varying** | **wait** | **when** | **yes** | , | : | := | <- | + | - | * | / | = | <> | < | > | <= | >= | [|] | (|) | ? | ! | ' | "

54 *character* = *letter* | *digit* | *the space character* | " | "" | @ | $ | % | + | - | * | / | < | = | > | (|) | { | } | [|] | . | , | ? | ! | : | ; | # | *other-printable-characters*

55 *letter* = **a** | **b** | ... | **z** | **A** | **B** | ... | **Z**

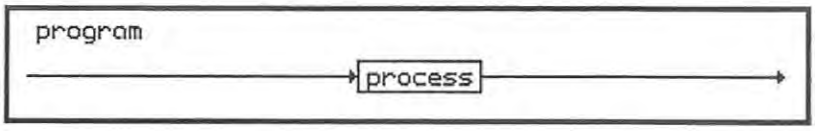
56 *digit* = **0 | 1 | ... | 9**

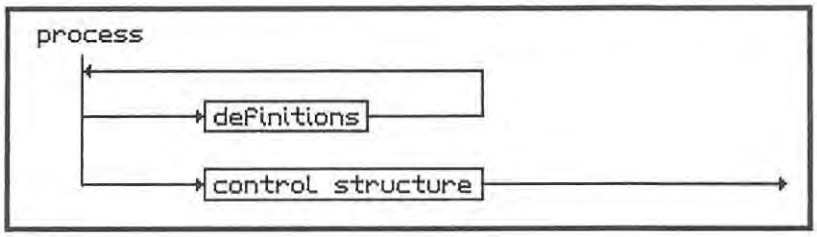
57 *other-printable-characters* = *implementation defined printable characters*

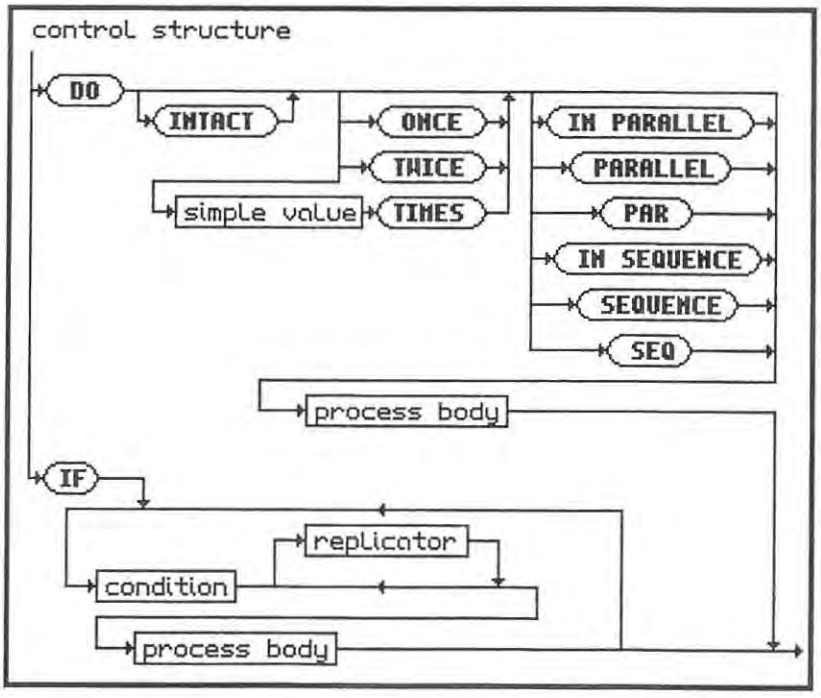
Cross references for non-terminal symbols

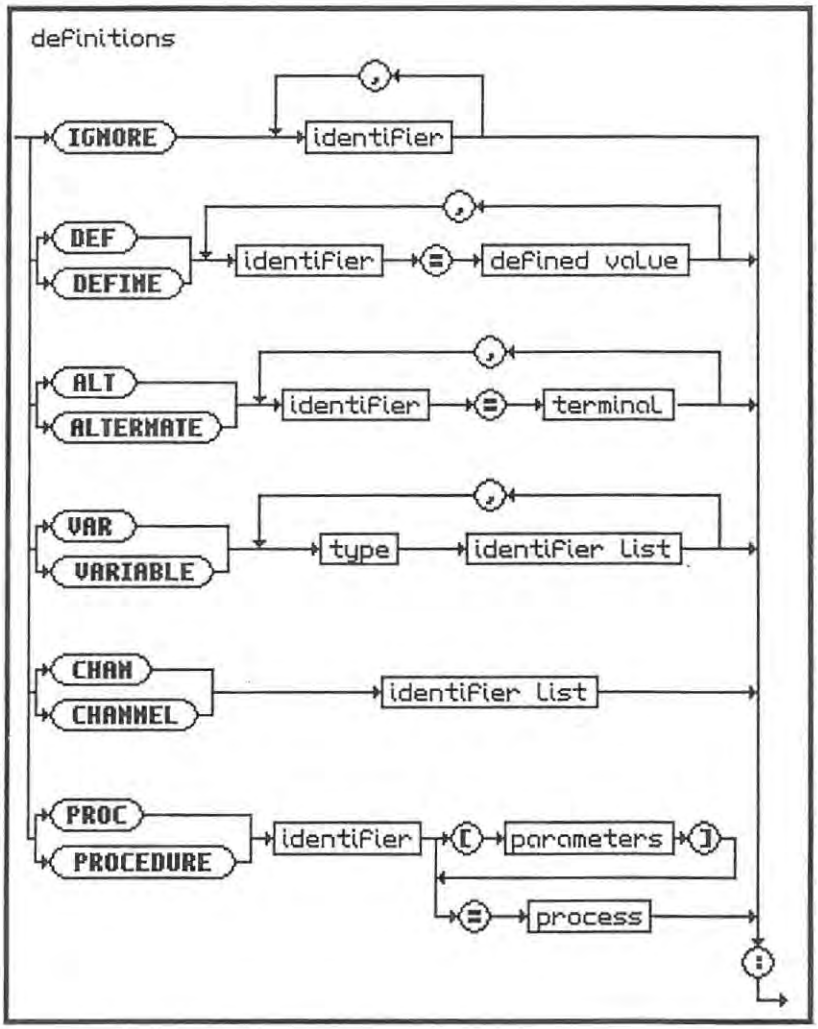
C.2 Syntax diagrams

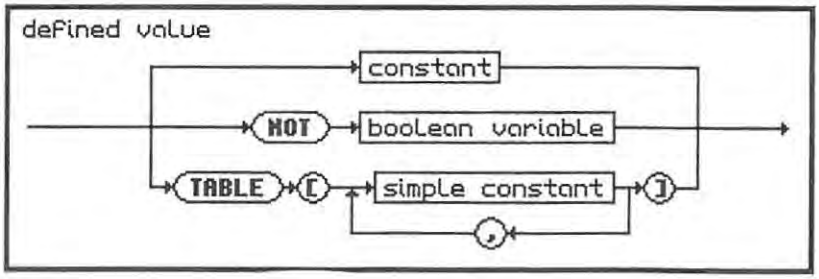
In the following HUL syntax diagrams, an attempt has been made to provide some form of semantic information as well. For example, the entities *numeric defined identifier* and *character array identifier* are syntactically equivalent to *identifier*, and *boolean variable* is syntactically equivalent to *variable*.

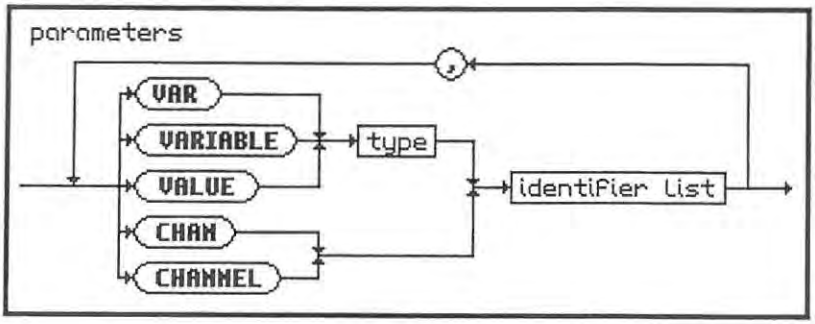


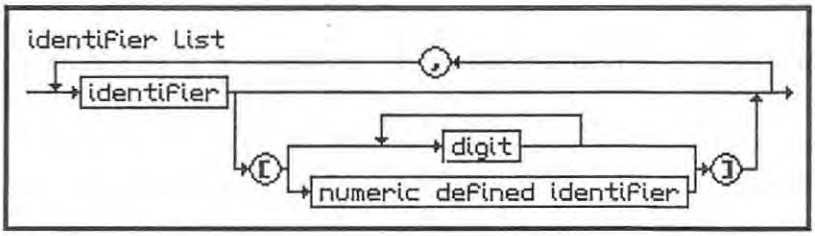


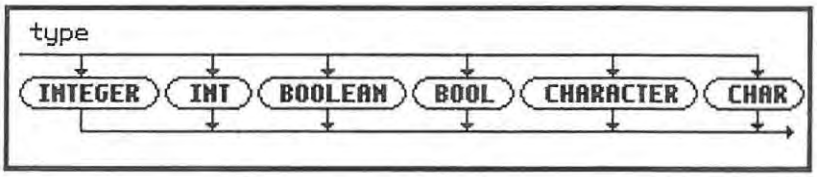


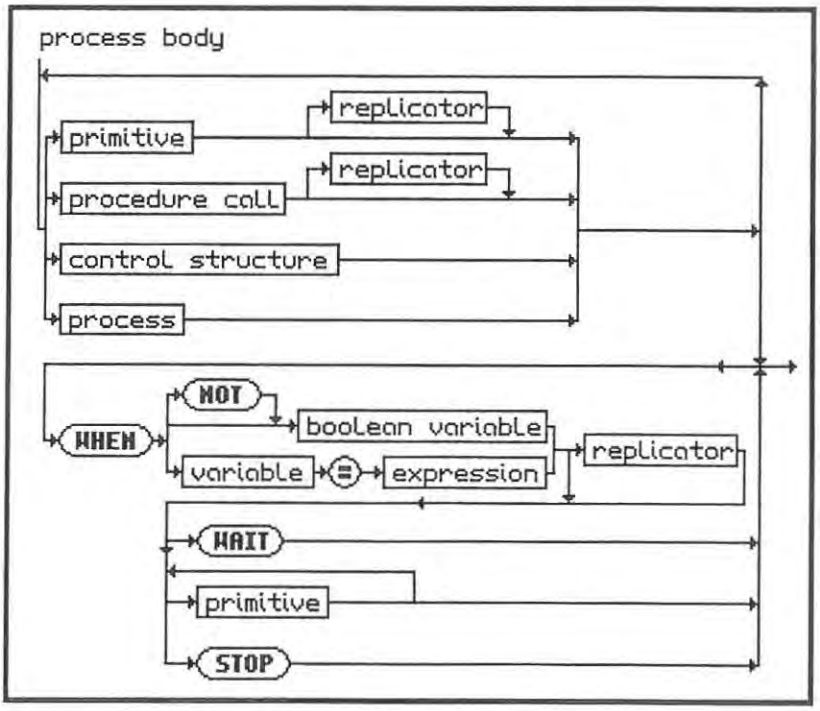

definitions
IGNORE
identifier
DEF
DEFINE
identifier
=
defined value
ALT
ALTERNATE
identifier
=
terminal
VAR
VARIABLE
type
identifier list
CHAN
CHANNEL
identifier list
PROC
PROCEDURE
identifier
(
parameters
)
=
process
:

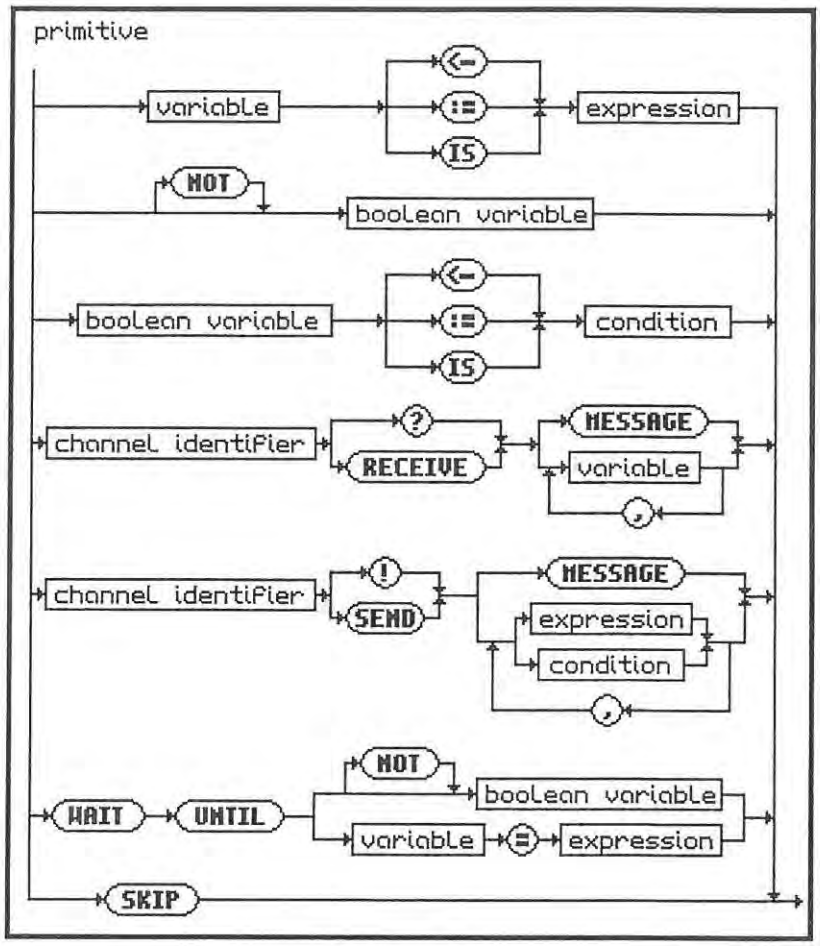

defined value
constant
NOT
boolean variable
TABLE
(
simple constant
)

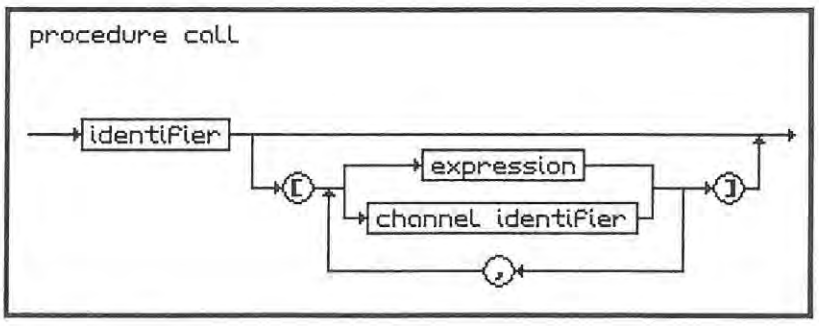

parameters
,
VAR
VARIABLE
VALUE
CHAN
CHANNEL
type
identifier list

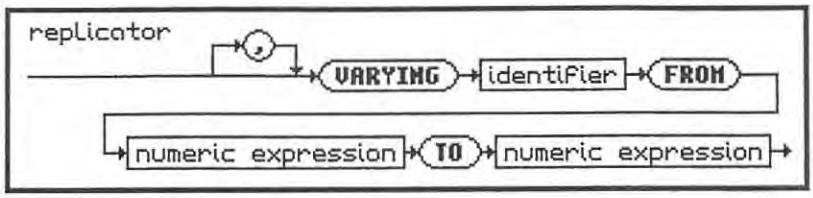

identifier list
,
identifier
digit
[
]
numeric defined identifier

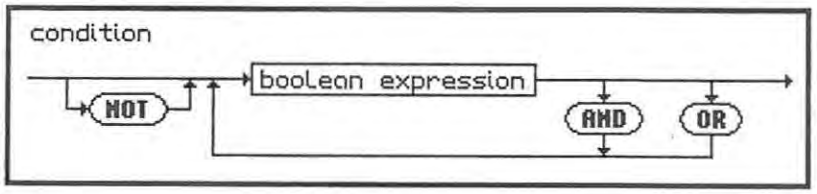

type
INTEGER
INT
BOOLEAN
BOOL
CHARACTER
CHAR

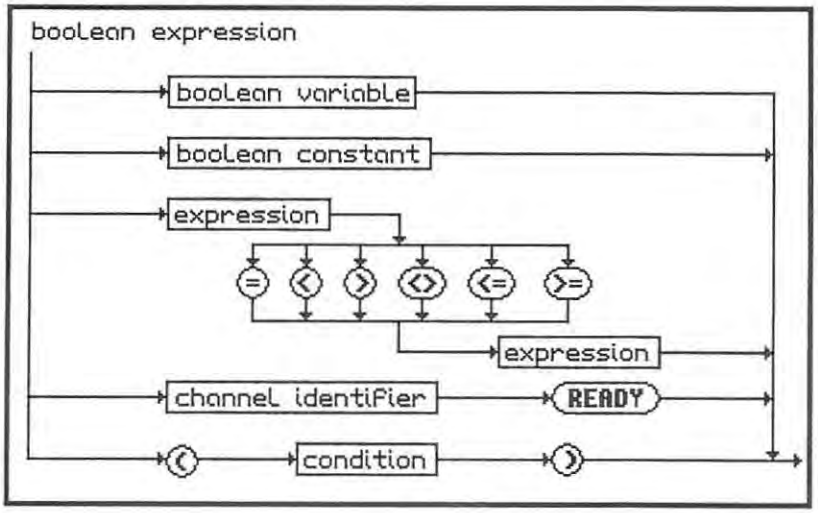

process body
primitive
replicator
procedure call
replicator
control structure
process
WHEN
NOT
boolean variable
variable
=
expression
replicator
WAIT
primitive
STOP

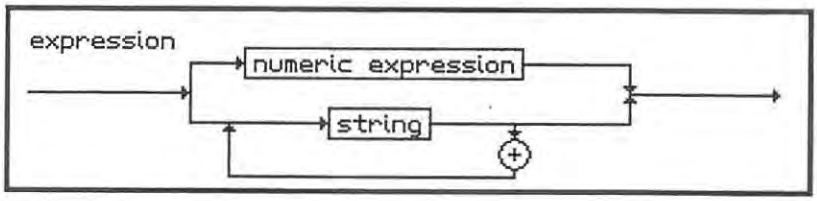

primitive
variable
<-
:=
IS
expression
NOT
boolean variable
boolean variable
<-
:=
IS
condition
channel identifier
?
RECEIVE
MESSAGE
variable
,
channel identifier
!
SEND
MESSAGE
expression
condition
,
NOT
WAIT
UNTIL
boolean variable
variable
=
expression
SKIP

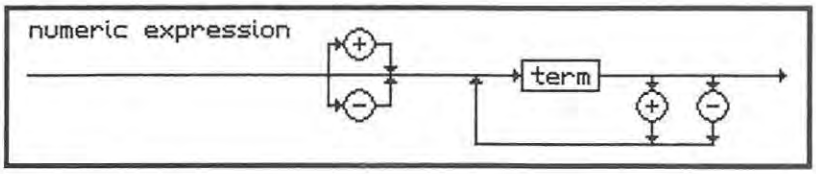

procedure call
identifier
(
expression
channel identifier
,
)

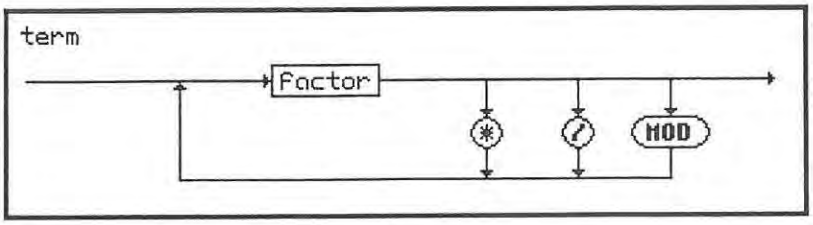

replicator
,
VARYING
identifier
FROM
numeric expression
TO
numeric expression

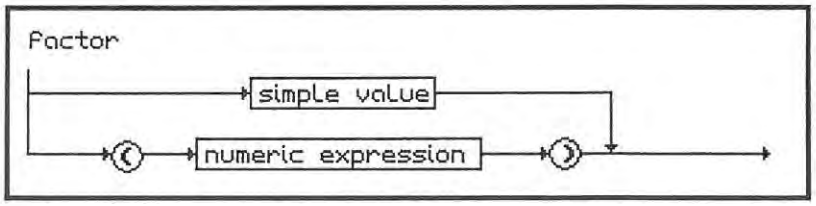

condition
NOT
boolean expression
AND
OR

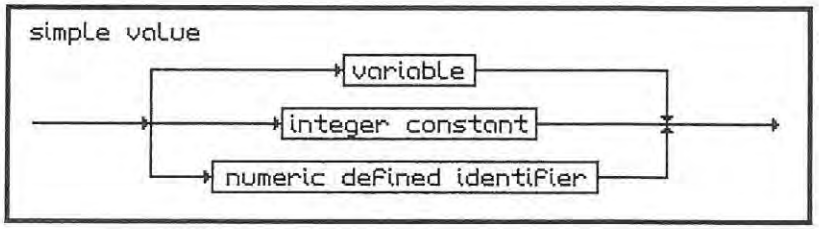

boolean expression
boolean variable
boolean constant
expression
=
<
>
<>
<=
>=
expression
channel identifier
READY
(
condition
)

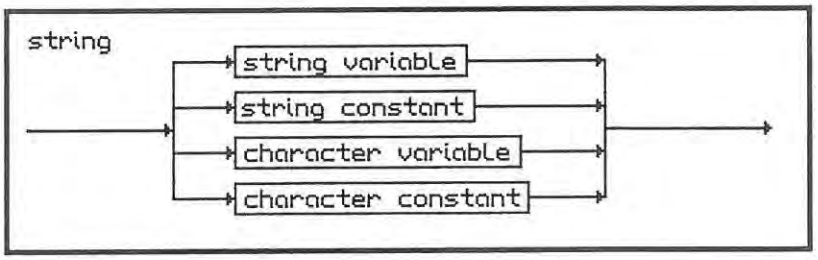

expression
numeric expression
string
+

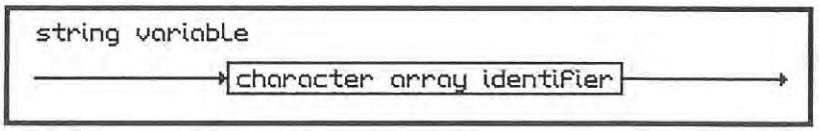

numeric expression
+
-
term
+
-

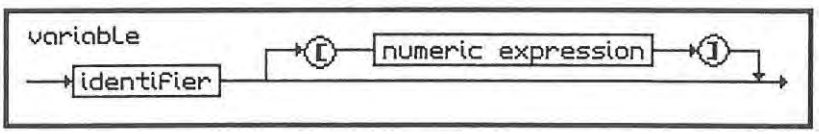

term
Factor
*
/
MOD

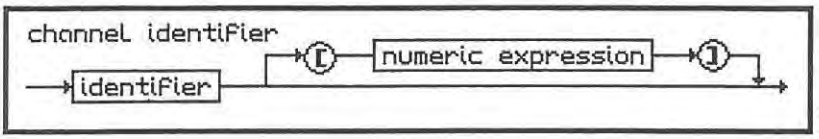

Factor
simple value
(
numeric expression
)

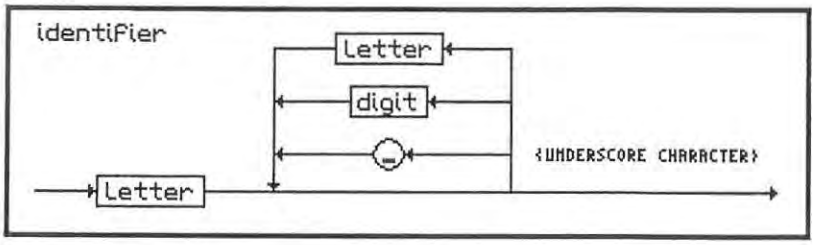

simple value
variable
integer constant
numeric defined identifier

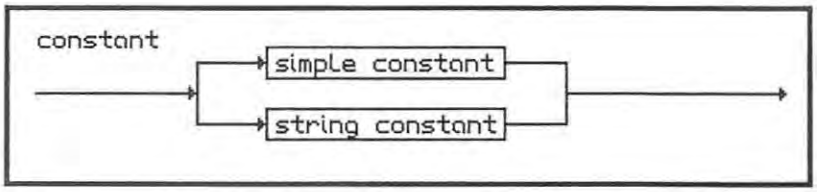

string
string variable
string constant
character variable
character constant

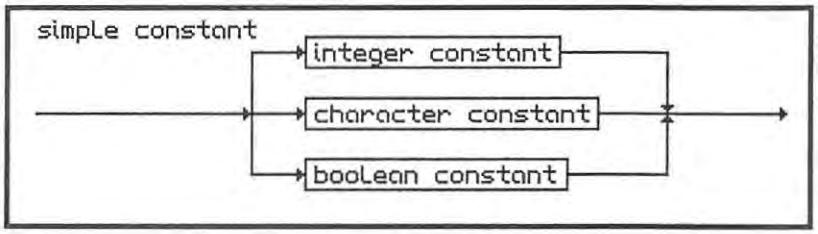

string variable
character array identifier

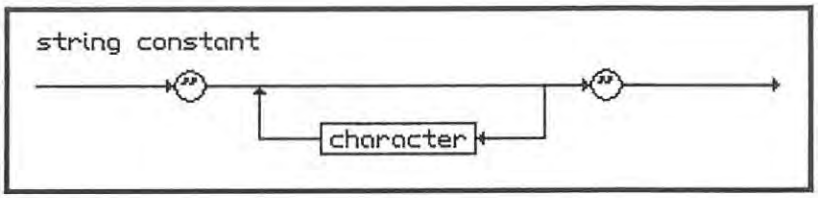

variable
identifier
(
numeric expression
)

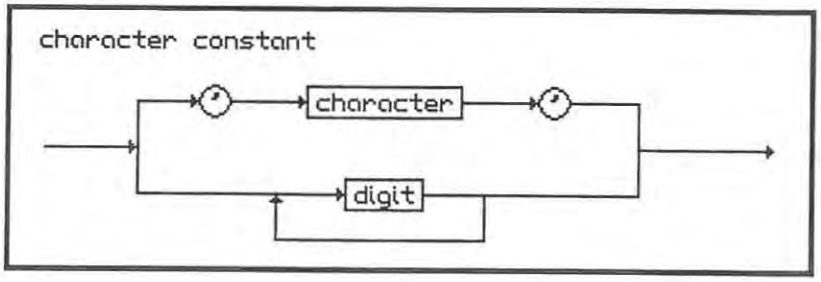

channel identifier
identifier
(
numeric expression
)

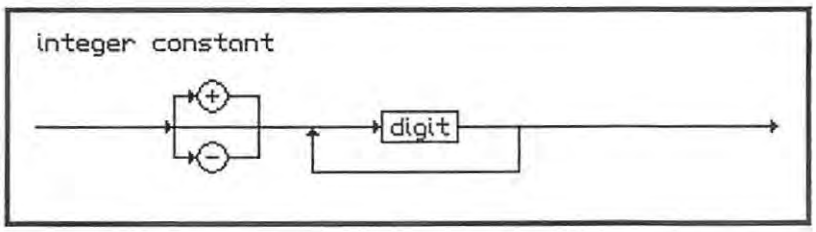

identifier
Letter
digit
_
<UNDERSCORE CHARACTER>
Letter

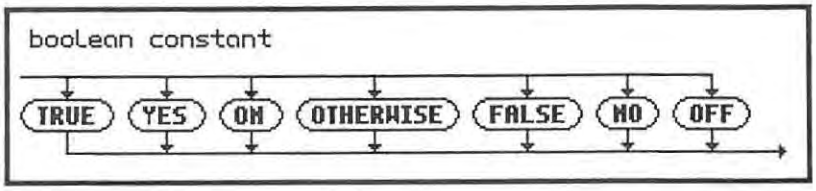

constant
simple constant
string constant

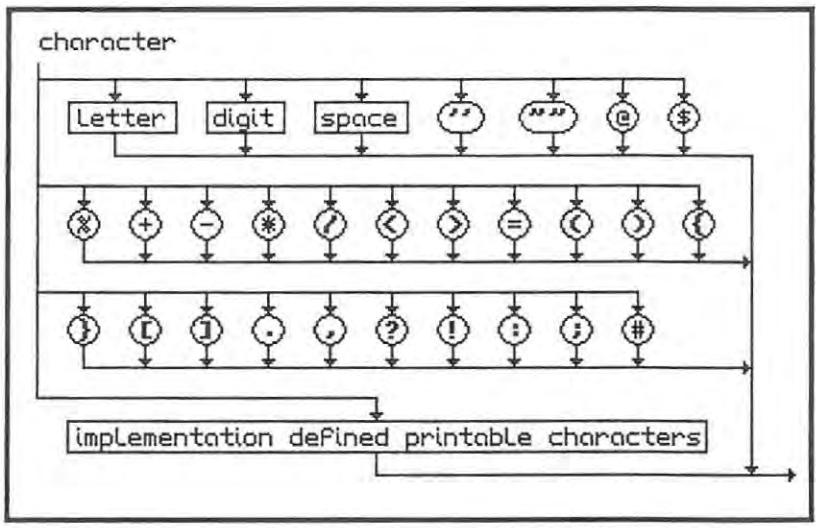

simple constant
integer constant
character constant
boolean constant

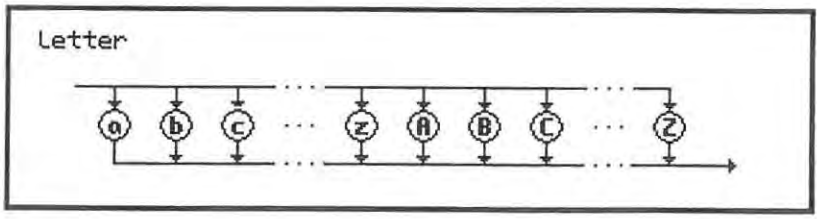

string constant
"
"
character

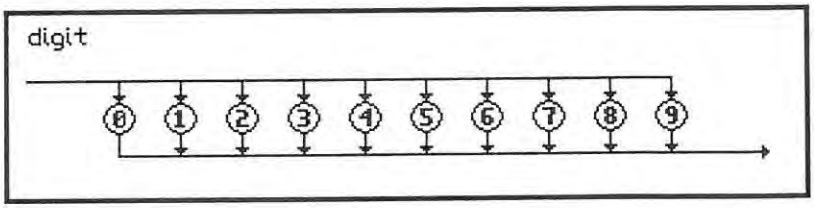

character constant
'
character
'
digit

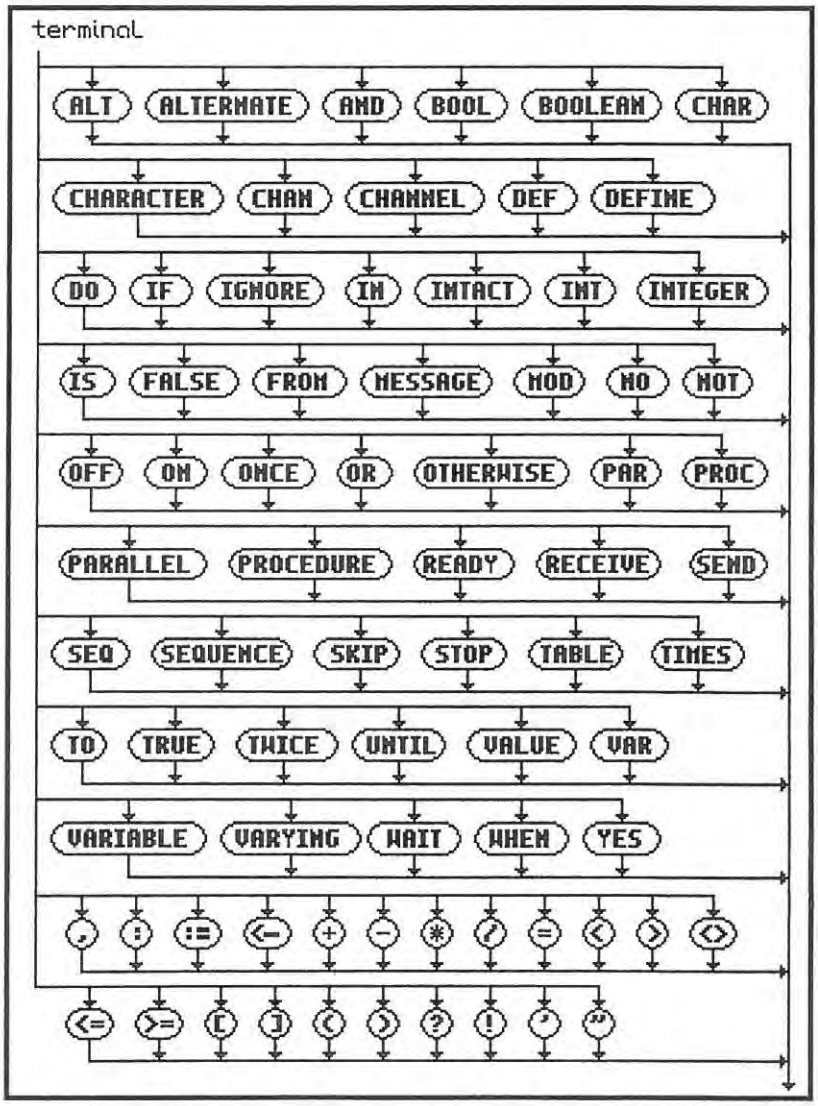

integer constant
+
-
digit


boolean constant
TRUE
YES
ON
OTHERWISE
FALSE
NO
OFF


character
Letter
digit
space
@
$
%
+
-
*
/
<
>
=
(
)
{
}
[
]
.
,
?
!
:
;
#
implementation defined printable characters


Letter
a
b
c
z
A
B
C
Z


digit
0
1
2
3
4
5
6
7
8
9


terminal
ALT
ALTERNATE
AND
BOOL
BOOLEAN
CHAR
CHARACTER
CHAN
CHANNEL
DEF
DEFINE
DO
IF
IGNORE
IN
INTACT
INT
INTEGER
IS
FALSE
FROM
MESSAGE
MOD
NO
NOT
OFF
ON
ONCE
OR
OTHERWISE
PAR
PROC
PARALLEL
PROCEDURE
READY
RECEIVE
SEND
SEQ
SEQUENCE
SKIP
STOP
TABLE
TIMES
TO
TRUE
TWICE
UNTIL
VALUE
VAR
VARIABLE
VARYING
WAIT
WHEN
YES
,
:
:=
<-
+
-
*
/
=
<
>
<>
<=
>=
[
]
(
)
?
!
'
"