GROUPING COMPLEX SYSTEMS FOR CLASSIFICATION AND PARALLEL SIMULATION

Thesis
submitted in fulfilment of the
requirements for the Degree of
Doctor of Philosophy
of Rhodes University

by

ISMAIL MOHAMED IKRAM

January 1997

Abstract

This thesis is concerned with grouping complex systems by means of concurrent model, in order to aid in *(i)* formulation of classifications and *(ii)* induction of parallel simulation programs. It observes, and seeks to formalize and then exploit, the strong structural resemblance between complex systems and occam programs.

The thesis hypothesizes that groups of complex systems may be discriminated according to shared structural and behavioural characteristics. Such an analysis of the complex systems domain may be performed in the abstract with the aid of a model for capturing interesting features of complex systems. The resulting groups would form a classification of complex systems. An additional hypothesis is that, insofar as the model is able to capture sufficient programmatic information, these groups may be used to define, automatically, algorithmic skeletons for the concurrent simulation of complex systems.

In order to test these hypotheses, a specification model and an accompanying formal notation are developed. The model expresses properties of complex systems in a mixture of object-oriented and process-oriented styles. The model is then used as the basis for performing both classification and automatic induction of parallel simulation programs. The thesis takes the view that specification models should not be overly complex, especially if the specifications are meant to be executable. Therefore the requirement for explicit consideration of concurrency on the part of specifiers is minimized.

The thesis formulates specifications of classes of cellular automata and neural networks according to the proposed model. Procedures for verification and induction of parallel simulation programs are also included.

Acknowledgements

I am very grateful to my research supervisors, Professors Peter Clayton and Peter Wentworth, for their kind professional and personal assistance. Without their encouragement, guidance and intelligent criticism, the conduct of my research and the writing of this thesis would have far less satisfying experiences.

Thanks must also go to the many members — both staff and students — of the Computer Science Department at Rhodes University, whom I had the good fortune to befriend during my stay.

Contents

Chapter 1

Introduction

1.1 Synopsis

This thesis presents a model of complex systems. Complex systems have been variously defined in the literature. These definitions are normally stated in general and qualitative terms. Fox describes them as large collections of disparate and interconnected members which evolve over time[Fox92]. According to this usage, many systems studied in the natural sciences, molecular systems and ecosystems for example, belong to the class of complex systems. Weisbuch gives a similar definition, insisting that the members of the system be of different types, rather than being of a homogeneous character[Wei91]. In addition, other definitions require that the components of the system have a simple internal structure and exhibit simple patterns of behaviour.

In general, any modelling activity can be seen as being directed towards the production of representations of systems. In computer science such target representations are formal, in the sense of adhering to the predefined syntax and semantics of a language. Just as, according to this terminology, models guide the writing of formal representations or specifications, they equally guide the interpretation of formal representations in terms of actual systems. Figure 1.1 illustrates the role of a model as a way of facilitating the conceptual transition, or abstraction, from the domain of actual systems into the domain of representations written in a language, and vice versa.

This thesis defines a model that is in many respects specific to complex systems.

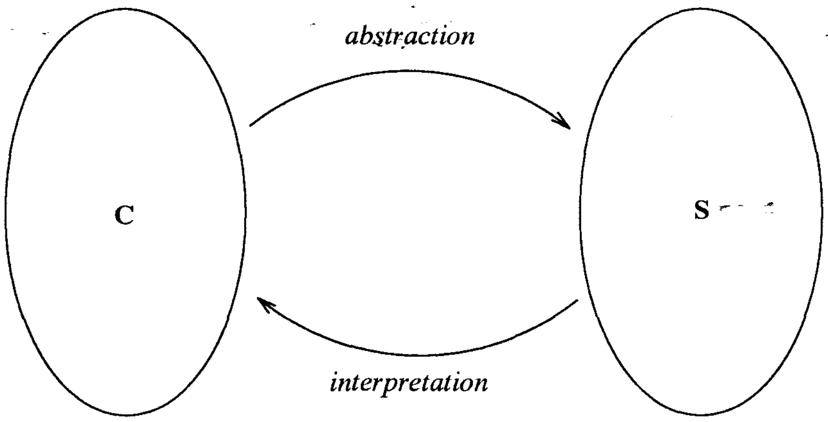

Figure 1.1: A model of a real-world domain C (in the present case, the set of complex systems of interest) consists of a formal specification language S (denoting the set of formal specifications written in the language) equipped with abstraction and interpretation mappings.

In the course of defining the meaning of various modelling aspects, this thesis proposes guidelines for performing both abstraction and interpretation. The model itself is understood to encompass a formal notation and its associated abstraction and interpretation rules.

Fundamental to the given model is that groups of similar systems discerned in the domain of complex systems should be expressed as a single instance of a special kind of formal representation, a group specification. For example systems differing simply in the spatial configuration of their components may be treated as a unit and abstracted to a single specification. However the principal importance of grouping, as will be noted shortly, lies beyond considerations of notational convenience. Stated in terms of the above diagram, it should be possible to define many-to-one abstraction relations where appropriate. The interpretation of such relations should likewise be explained according to the model.

Grouping of specifications is essential to the dual aims of the thesis, classification and parallel simulation of complex systems.

The language is so constrained that complex systems may be grouped only if they are similar in terms of either their internal organizations or their patterns of behaviour, or indeed both. Thus a group specification represents a class of systems sharing characteristics that are meaningful in the domain of actual systems. The ability to formalize concepts of similarity allows classifications of complex systems to be formulated.

Inducibility of parallel simulation programs is another property assigned to the specification model. Essentially simulation programs are an alternative representation of systems to the above formal specifications. The space of such programs constitutes a separate domain into which abstraction from the domain of complex systems may be performed, but practically this form of abstraction is more difficult to achieve because of the wider gap between the concepts of complex systems and those of a general purpose programming language.

The programming language in which simulation programs are to be generated is occam (occam-2[INM88b] is the version of the language referred to in this thesis). For this reason, the complex systems model needs to ensure that sufficient information is embodied in specifications in order to derive occam programs satisfying those specifications. However, this requirement should be balanced against the need to keep the model specific to complex systems. The choice of occam as the target programming language helps to strike such a balance since, for example, the units of modularity in occam — communicating processes — are analogous to the components of complex systems.

In this thesis, program induction may only be performed upon specific formal representations of complex systems, and not upon the more abstract representations named above as group specifications. Group specifications are nonetheless of great practical importance because they define application-specific programming interfaces. That is, they define not only the characteristics of a group of complex systems, as explained earlier, but also the characteristics that need to be further specified in order to describe an instance of that group.

In summary, the model presented here enables specification and interpretation of both groups and instances of complex systems. The terms of reference of its specification language correspond closely to those of the domain of complex systems. As a result, group specifications constitute a classification of complex systems. Associated with the language is a formal procedure which translates specifications into conforming concurrent simulation programs in occam. This, in combination with the ability to group specifications, offers a high level approach to programming simulations.

1.2 Motivations and Background

1.2.1 Introduction

The primary motive behind this work is to formalize the observation that the process-oriented approach to parallel programming exemplified by the occam language — that of constructing 'communicating process architectures'[INM88a] — is readily applied to writing parallel simulations of complex systems. Evidence for this has come from the experience of implementing simulations of complex systems using occam's parallel programming abstractions. Brinch Hansen's implementation[Han93] of cellular automata in an occam-like language is an example in the published literature.

An approach to formalization which seems feasible is to define a program induction procedure capable of accepting a description of any complex system and outputting an occam simulation program for the relevant system. In order to demonstrate a strong link between communicating process architectures and complex systems, the terms employed in the input language should express the concepts of complex systems, and not to any great degree those programming concepts which are irrelevant to complex systems.

The definition of a program induction procedure presupposes a formal language in which input descriptions of complex systems are to be stated. Now the case for the hypothesized link may be made even stronger if such a language is able to formalize classifications of complex systems; that is, if closely related complex systems, when represented in the language, yield syntactically similar formal descriptions.

1.2.2 Simulating Complex Systems

The evolution of the state of complex systems is of chief interest to researchers in this field. Complex systems are so named because of the difficulty of predicting future states of a system based on observations of previous state. The evolution of a complex system is a collective phenomenon that results from the evolution of its components. These components evolve in a mutually dependent manner; that is, the rules by which a component is changed are dependent upon the states of other components. The nonlinearity of such interactions explains the emergence of complex phenomena at the system level from the lower level interaction between simple system components.

Complex systems of particular interest in this thesis are paradigms of parallel computation found in nature, for example cellular automata[Wol84], neural networks[HKP91], genetic algorithms[For93] and immune networks[BV91]. Certain connections between complex systems and parallel computing have been elucidated in the literature. For example, Paton *et al*[PNS$^+$91] note that several biological systems provide metaphors for parallel problem solving strategies. A fundamental connection is made by Fox, who holds that parallel computer software and hardware satisfy the criteria for being considered as complex systems[Fox92]. Fox then suggests that the writing of computer simulations of natural complex systems may be viewed as a series of mappings, starting from the level of the natural system and proceeding to the level of the simulation running on a parallel computer. Intermediate mappings in this series include one from the scientific model of the complex system to an appropriate numerical formulation and a subsequent one from the numerical formulation to an implementation in software.

Computer simulations of complex systems are vital research tools. Where analytical techniques exist to model complex systems mathematically, they are typically computationally expensive to solve. An alternative is directly to simulate complex systems by computationally simulating individual components and their interactions, and examining state changes as the simulation proceeds.

One of the main intentions of this thesis is to formalize a series of mappings from complex systems to parallel simulations, so as to automate, to a large extent, the process of constructing simulations. In particular, this thesis focuses on generating simulation programs in the occam concurrent programming language. The choice of this language is due to the close structural resemblance between occam programs, which are networks of interacting processes, and complex systems.

A number of systems have been presented elsewhere for the parallel simulation of well defined categories of complex systems. Taking neural networks for example, the CuPit programming language[Pre94] is designed specifically for the purpose of writing neural learning algorithms, and compiling such programs for efficient execution on parallel computers. The Genesis system[BB92] is suited to simulating biologically realistic neural networks, which are specified by a scripting language. Genesis has facilities for distributed execution.

The Swarm simulation system[Bur94] (see section 2.2.2) is intended to simulate complex systems in general. The Echo system[FJ94] is more limited in scope, and is particularly suited to ecological simulation. It requires that complex system components and interactions adhere to a limited number of prescribed forms. At the time of writing no parallel implementations of either system are known to exist. However, Swarm in particular has been designed to reflect the distributed and concurrent nature of complex systems, and therefore a distributed version incorporating message-passing would seem straightforward to implement.

Constructing simulations using any one of the abovementioned systems entails the writing of some form of specification, for example a CuPit program or a Genesis script. This step amounts to writing a formal representation of a complex system that may then be input by the relevant simulation tool. The nature of what is expressed in specifications is dictated mainly by the model of complex systems adopted by the simulation system. For example, the Echo model assumes that systems are composed of resource-consuming and reproducing agents (system components) and that interaction between agents is limited to a two-dimensional neighbourhood.

1.2.3 Classifying Complex Systems

Returning to the proposed automated mappings from complex systems to parallel computers, one of the main issues addressed by this thesis is the construction of a model according to which complex systems may be formally represented. Complementary to the modelling issue is the consideration of an appropriate notation or language for writing formal representations. It should be noted that the task of writing specifications, or of abstraction, relies on the understanding by the writer of the complex system being specified, and of the model and its attendant notation. This abstraction task itself is not thought to be open to formalization.

Recalling Fox's sequence of mappings, it is evident that, as the series progresses, the terms of reference at successive levels become more removed from the original complex system, while simultaneously approaching those of the target parallel computer. The question then arises as to the appropriate level at which specification writers should be expected to formalize complex systems. In formulating the model and language, a reconciliation is necessary between the conflicting demands of the writer, who wishes to express systems in terms specific to complex systems, and of the implementor of the code generation system, the difficulty of whose task grows with the 'semantic gap' between the specification language and occam.

This thesis proposes, in addition to mechanisms for automating parallel simulation of complex systems, a formal way in which classification of complex systems may be performed. For this purpose, the inverse of the mapping from complex systems to specifications, which may be called the interpretation of specifications, is employed. By design, one of the characteristics of the specification language will be the ability to capture groups of related complex systems as formal representations called group specifications. Thus a group specification is to be interpreted as a collection of complex systems. If it can be shown that group specifications are always interpreted as groups or classes that are meaningful in the domain of complex systems, then the specification language becomes a tool for creating system classification schemes. This approach may be called interpretive classification, in contrast to abstractive classification, whereby systems known to be related by common properties are abstracted into a single group representation.

The capability to group specifications is also essential if the above proposal for creating simulations is to be practical. Provided that group specifications may be instantiated to actual complex system specifications, then any group specification, once written, may be used to create simulations of a variety of systems simply through appropriate instantiation. Therefore a means of modelling commonality would significantly simplify the task of modelling large numbers of systems through reuse and adaptation of a general models.

1.3 Objectives and Outline of the Thesis

The objective of this thesis is to propose criteria for grouping complex systems in a manner that facilitates both classification and parallel simulation of complex systems. Both of these ends presuppose the existence of models of complex systems. Here, a single formal model is shown to be sufficient. In the development of this thesis, the overall objective is advanced through a sequence of three sub-objectives:

1. Modelling. Define a model of complex systems consisting of a formal specification notation, and rules for abstracting real complex systems and for interpreting specifications. Define correctness of specifications and formal methods of verifying correctness.

2. Classification. Extend the model to cater for specifications of groups of complex systems. Ensure that group specifications are interpreted as related complex systems.

3. Parallel simulation. Define a mapping from specifications to conforming occam simulation programs. Propose an application-class oriented simulation system for exploiting the generality of group specifications.

It is natural to construct models by considering only the abstraction relation (to use the terminology introduced in section 1.1) in the first instance, by analysing the domain to be modelled and extracting its conceptual entities. Thereafter a formal specification notation may be defined so as to match symbols to domain concepts. Finally, the rules for interpreting specifications may be stated. This thesis follows such a sequence in developing its model.

The next three chapters are concerned with formalizing the concepts of complex systems. They propose a model according to which abstraction and interpretation of systems of interest may be performed, whether individually or in groups. A formal specification language is also presented. The two chapters following thereafter treat classification and parallel simulation, respectively, as tasks dependent upon the previously presented model.

Chapter 2 explains what is required of a model in order to satisfy the aims of the thesis. It considers various alternatives for translating informal concepts about complex systems into formal representations. Methods given in previous literature are examined, and an outline of the adopted method is presented.

Chapter 3 is an exposition of the adopted abstraction method. Individual complex systems and groups of systems are considered.

Chapter 4 gives the format and notation of specifications. The interpretation of each part of a specification is described. Formal methods for verifying correctness of specifications are given.

Chapter 5 develops group specifications for cellular automata and Hopfield neural networks. Actual examples are used to demonstrated the motivations behind some of the earlier design decisions for the model.

Chapter 6 described the procedure for the induction of occam simulation programs from specifications. An programming system for parallel simulation is outlined based on the concepts of group specification and program induction.

Chapter 7, the concluding chapter, summarizes and assesses the contributions of each of the preceding chapters of the thesis with reference to the stated objectives.

Appendix A gives the meaning of special symbols used in this thesis.

Chapter 2

Prelude to Modelling

Context

Chapter 1 explained that the aims of the present work are to propose methods for modelling, classification and parallel simulation of complex systems. Modelling was described as being fundamental to the achievement of the other two aims.

In section 1.2 the motivation for viewing complex systems as communicating process architectures was stated. This chapter determines the remaining broad characteristics that a proposed model of complex systems ought to possess in order to enable classification and parallel simulation (section 2.1).

Existing models are examined in section 2.2. Then a suitable model is outlined in section 2.3. Chapters 3 and 4 expand upon this outline. Chapters 5 and 6 demonstrate some practical applications of the model.

2.1 Requirements of the Model

2.1.1 Introduction

In section 1.1 the construction of a model was equated with the definition of a formal specification language and a pair of complementary mappings or relations called abstraction and interpretation. The two relations mediate between formal and informal views of complex systems, that is, between the set of real complex systems and the set of specifications written in a formal description language.

The requirements presented here are named information preservation, directness, syntactic grouping, parallel-program inducibility and verification.

As stated in section 1.3, it is the property of abstraction that will be examined first with regard to modelling complex systems. Considerations of a specification notation and its interpretation appear to be dependent upon the method of abstraction adopted by a model. The requirements that need to be met by a satisfactory abstraction, then, may be described first.

2.1.2 Abstraction

Abstraction effects a transition from one level of observation, or world view, for convenience referred to as the 'reality,' into another, called the specification, as depicted in figure 1.1. Abstraction is carried out with the intention of representing in formal terms interesting phenomena of the real world, in this case complex systems.

The required model of complex systems should facilitate abstraction in such a way as to ensure information preservation and a directness of modelling. These characteristics concern, respectively, the information content of the resulting specifications, and the terms of reference used in presenting that information. Both are given here as qualitative factors, and the existence of neither one is intended to be proven formally.

Information preservation is motivated by the need for accuracy in modelling. It means that any facet of informal knowledge about the system being modelled can be formally encoded through abstraction. Usually this requirement is qualified so that only knowledge that is relevant to the aims of modelling need be formalized. However, since the modelling of all complex systems is meant to be addressed here, the model should, by design, be general in purpose. Of particular importance to this thesis is that the commonality found between similar complex systems must be reflected in the formal domain.

In practice, abstractions are difficult to define whenever there is a significant divergence between the terms of reference employed in the formal and informal domains. Take, for example, an aspect of informal knowledge about complex systems, which may be framed in terms of rules of interaction between system components. Then according to some numerical model, that aspect may be formalized as a system of partial differential equations. The rules for performing abstraction in such cases would likely assume expertise in numerical methods on the part of the user of the model.

Directness is a property of models that do not exhibit divergences of this kind. It is required in order to simplify the task of learning and applying the model.

Special-purpose models have a greater freedom to define direct means of performing abstraction. If a model is devoted to members of a well defined class of systems, it would naturally be capable of making several assumptions concerning the informal domain. Then these assumptions may be incorporated into terminology of the model's abstraction relation. For example, a model of neural networks may assume that system components divide into exactly two classes, neurons and connections; its abstraction relation may then make direct reference to the number of neurons and to their inter-connection network.

Directness is further promoted by support for declarative abstraction. The declarative property is commonly associated with programming languages, where it stands in contrast to the imperative mode of expression. The difference between the two modes in the present context may be summarised as follows: where the behaviour of a system is to be formalized, declarative methods describe the effects of the behaviour, while imperative methods describe the mechanisms used to achieve those effects. Imperative abstraction relies upon the existence of a separate model according to which computations may be described. As such it detracts from the directness of modelling.

2.1.3 Specification

In the domain of complex systems, broad classes of systems are informally known, such as various classes of neural networks. With the aid of the property of information preservation, as described in section 2.1.2, knowledge about the common characteristics of groups of systems may be transferred into formal specifications.

A further requirement of the model, arising at the level of formal specifications, is called syntactic grouping. This states that formal specifications may be designed and written for members of a common class in such a way that the resulting specifications possess similar syntactic features.

To be precise, for every class of actual complex systems, let S be a set of its specifications (there is no requirement for there to be a unique S — abstraction is not necessarily a one-to-one mapping). Also assume that members of S consist of a fixed number, N, of sections uniquely indexed by integers taken from the set

$\{1, 2, \ldots, N\}$. Now S is said to vary under index i iff there exists a pair $x, y \in S$ such that the ith section of x differs textually from the corresponding section of y. Then syntactic grouping exists iff S varies under at least 1 index and at most $(N-1)$ indices.

Syntactic grouping enables abstractive classification (section 1.2.3) to be performed.

2.1.4 Interpretation

The requirements given here relate to the rules of interpretation defined by a model for its specifications.

Parallel program inducibility is said to exist when sufficient information is available in specifications in order to generate parallel programs that simulate the specified systems. Moreover, formal procedures for inducing such programs should be defined. Interestingly, such procedures can be thought to constitute interpretation mappings into the domain of parallel programs, as distinct from the usual interpretations (see section 1.1) into the domain of actual complex systems.

This requirement may be expected to conflict with the directness requirement of section 2.1.2. Specifically, declarative abstraction and program inducibility can be mutually antagonistic aims to achieve in a single model. Declarative abstraction, aiming for directness, adopts terms from the domain of the systems. In contrast, program inducibility may be achieved by incorporating a virtual parallel machine into the model, and requiring that abstraction of system behaviour be performed in terms of that virtual machine.

Another requirement appropriate to the interpretation level of a model is the ability to verify the correctness of specifications. Correctness is considered here in two senses. Firstly, specifications are said to be correct when they are interpreted as valid actual complex systems; for example, a specification describing interactions between a certain number, N, of components, while specifying an overall system consisting of fewer than N components, would be incorrect due to inconsistency. In the second sense, correctness is a property of specifications when interpreted as simulation programs; thus a specification inducing a deadlocking program, for example, would be incorrect. The two sets of correctness criteria would be expected to overlap.

2.2 Approaches to Modelling Complex Systems

2.2.1 Relational Modelling

Rashevsky proposed a mathematical formalism for describing biological systems that seems suitable to complex systems in general[Ros72, Ros91]. This method, called relational modelling, aims to determine functional components of systems and their organization. Components are modelled as mappings from input to output domains. The description of a system's organization is given by forming a network, or relational diagram, of its functional components; the network serves to relate the outputs of certain components to the inputs of others. Abstract mappings can be defined in order to represent compositions of simpler functional components.

Significantly, relational diagrams are not meant to depict the structural organization of systems. Nor do functional components necessarily correspond directly to a system's structural units. Indeed, the relational approach sets out to abstract away such structural information. For example, a neural network may be modelled independently of the relative configuration of its neurons and connections by representing the neurons collectively as a single unit, and likewise the connections as a separate unit. Then both units will be defined as functions transforming their input signal into an output signal. The relational diagram for such a system may then be constructed by letting the input of the neural unit be the output of the connective unit, and vice versa. Such a high-level representation would constitute a description of the class of neural networks. No reference has been made to the concrete structure of instances of the class.

Relational modelling possesses higher-order relations which may be employed to represent the dynamic nature of complex systems. That is, functional units may have outputs which represent other functional units. This is a useful feature for modelling systems that are capable of manufacturing new units. For example, a model of an animal colony may define such a mapping as an abstraction of the birth of new population members. Rosen presents a relational model of self-repairing systems containing higher-order mappings[Ros72].

However, since there is a deliberate exclusion of state-based and structural concepts, relational modelling seems overly abstract for the purposes of this thesis. Grouping is handled well by its formalism, by transition from the level of groups of systems to instances is not treated. Behaviour of systems over time, for example the order in which the relations are exercised, cannot be indicated in relational diagrams either.

2.2.2 Structural Modelling

Computational View of Complex Systems' Behaviour

In contrast to relational modelling, structural modelling analyses systems into state-carrying units or agents. It explicitly describes the structure of systems, for example as data-flows between agents. This form of modelling has a close association with methods normally used in computational modelling.

The observable behaviour of any system is the evolution of its visible initial state over time. Both hidden and visible states are usually possessed by systems, but observed behaviour is defined in relation to the states visible from outside the system. Computation is the logical process by which new states are generated on the basis of existing states. Therefore behaviour may be said to be the consequence of computation.

Algorithmically, complex systems are difficult to specify since they are usually large assemblies of components, or agents, which possess independent local attributes and communicate amongst themselves in intricate patterns. Much of the difficulty in modelling them in conventional computational terms arises from their complexity of their communications rather than in the computations upon state. The intense communication traffic implies an interdependence between agents. Their complex behaviour is due to this complexity in interaction to a greater degree than in the algorithmic logic of the agents themselves.

Almost without exception, agents are iterative entities. They repeatedly perform a series of operations. Certain iterated operations are bound to be nondeterministic with respect to the computations performed upon their internal state or with respect to the communications made with other agents.

Agents are active simultaneously. Concurrent activity is possible because of the distribution and independence of state information throughout the system.

Agents cooperate in small spatial groups, and only rarely in a global fashion. Communication is performed by agents in order to transmit local information or to receive information concerning their environment. In many cases, concurrency and nondeterminism combine to randomize the behaviour of complex systems. An example of this is the possibility that the information received by agents from their environment may be outdated. Such random phenomena contribute to the complexity in analysing the behaviour of these systems. It is imperative that simulations be capable of emulating the phenomena of randomization.

Concurrent Object Oriented Modelling in Swarm

A general simulation system suited to working with complex systems is Swarm[Bur94]. It adopts an abstraction of complex systems similar to the above. The Swarm specification method is explicitly modelled on object oriented concepts[Mey88]. Furthermore concurrency concepts have been added to the object oriented model. Algorithmic or coordination information for systems' agents are specified as local activity schedules. The swarm system used such schedules to deduce concurrency information which may then be used to distribute the simulation across multiple processors or to run it in a concurrent fashion on a single processor. Explicit specification of non-deterministic execution and synchronization are permitted. Communications are defined to be synchronous, whilst asynchronous simulations have to be defined in terms of synchronous primitives. Extensive support is provided for input and output to facilitate interactive simulation and graphical display.

Statically, the agents are the units of specification. They are modelled in a state-based object oriented fashion. Spatial aggregates of agents called swarms are conceptual entities that may be used to provide control over groups of agents without necessitating global control mechanisms.

Support for dynamic execution is through the ability by agents to alter schedules during run-time. To enable this, the Swarm system is implemented as a virtual machine model capable of executing schedules. Schedules, are one mechanism through which cooperative behaviour between agents is implemented, may be written in a per-swarm basis. Swarm specifications are Objective C programs. Therefore the ordinary constructs of this language, especially the 'main' function may be used to provide additional way towards centralized coordination.

Component-level genericity of agents is achieved through the Objective C language's class-based inheritance. Other forms of genericity are achieved through code libraries which implement features found to be common to a range of related complex systems.

2.3 Outline of the Proposed Model of Complex Systems

2.3.1 Abstraction

Combining Object Orientation and Process Orientation

As recognized by the Swarm system, there is a very obvious analogy between the agents of a complex system and objects in an object oriented system. There is a similarly close connection between agents and the process concept found in the Communicating Sequential Process (CSP) model[Hoa85], and in particular, the communicating process architecture model commonly associated with occam programming[Bur88]. The latter we name *process orientation* as being analogous to object orientation.

Agents are viewed as *processes* because they encapsulate *private state* and communicate with external entities by *passing messages*. In complex systems, agents are rarely homogeneous, or of the same type, rather they are instances of different classes of entity, speaking in object-oriented terms. A neural network, for example, may be thought to contain instances of two entity classes, neurons and connections, each one having quite different properties to the other.

Object orientation provides a rich source of modelling concepts centred on the class abstraction. Objects are the unit of modelling, and similar objects belong to the same class, and are said to instantiate the class. Thus a class is a generic description of a group of objects. Classes may be arranged according to the level of abstraction in a property-inheritance hierarchy. This form of hierarchical modelling promotes abstraction-based analysis. Objects encapsulate state in the form of attributes. Encapsulated state may be manipulated only from within the object by local methods. Cooperative operations between several objects is achieved by passing messages in response to which remote objects may alter their local state. Although not one of the fundamental object oriented concepts, parameterization enables the specification of a set of classes having similar structure through a single entity, the parameterized class. Commonality of structure so exploited is usually in the definition of attributes and methods.

Process orientation has analogues to such object oriented concepts. Genericity of a more restricted nature is achieved through the prototypical process specifications. These are replicated rather than instantiated. More powerful modelling of commonality may be achieved by incorporating mechanisms such as inheritance hierarchies and parameterization. In the process oriented arena, encapsulation and locality of state are even more fundamental than they are in object orientation. Since there is no common temporal reference between processes, it is absurd to describe a cooperative task in global terms. In contrast, such object oriented programming languages as C++ permit data access across object boundaries. In process orientation, cooperation between modelling units is strongly enforced to be through message passing only.

Although process orientation has less powerful mechanisms with which to express genericity, its fundamental abstractions, processes, are more suited to the task of modelling the components and behaviour of complex systems. Complex systems are inherently concurrent and their constituent agents are independent entities whose collective behaviour arises as the result of communication of information on which basis agents decide on their future state. This suggests that agents should be modelled as concurrent processes having strict data encapsulation. Furthermore, in modelling terms message passing appears to be strictly analogous to the methods of communication observed in complex systems. Another benefit of a concurrent modelling formalism is the explicit treatment of nondeterminism. Situations abound where a complex system's state and behaviour at a certain point in time cannot be feasibly predicted in advance. In sequential object oriented terms, such behaviour may only be modelled by conditional decision-making. However, process orientation offers in addition very powerful modelling concepts in the form of alternation between message sources and synchronous and asynchronous communication. These can directly model the nondeterminism in complex systems which is often given rise to by the nondeterminism in the communications medium and in the pattern of message passing between agents.

A suitable melding of object and process oriented terms into a coherent abstraction technique for complex systems is required. The common properties of state encapsulation and message passing should be retained. However, message passing in process orientation offers a richer variety of primitives, which are analogous to operations occurring in complex systems. Object orientation offers better facilities for expressing grouping, or the commonality among modelling units. Concurrency and nondeterminism should also be incorporated from process orientation.

Static and Dynamic Properties

A distinction can be drawn between properties of a system that either stay the same or change over time. When abstracting complex systems, this distinction may be said to be between *static* and *dynamic* properties, respectively, for obvious reasons.

Complex systems typically consists of agents that have identical static properties. A group of agents having identical attributes in this sense may be said to belong to the same *class*, to use the terminology of object orientation. From the process oriented view, each agent in such a group is a *replica* of the same prototypical process.

In modelling static properties, of chief interest are the state attributes of agents and their interconnection topology as well as the operations that cause state changes.

Both object orientation and process orientation are unanimous in requiring systems' state to be totally distributed among the units of specification, the objects and processes, respectively. That the state encapsulated within unit boundaries is also fundamental to both models.

In process orientation, state-changing operations are likewise strictly defined in terms of state local to objects. If an operation is activated within a certain object, and state is required from a remote object, then the operation may only proceed once the required state has been communicated to it through messages. This can be considerably tedious to specify since communication is a synchronous activity involving two processes, the sender and the receiver, thereby coupling intricately their definitions.

On the other hand, models of operations in object orientation are not uniform. Some of them define operations in similar terms to the one just described. Others do not model objects as being active in a single object, but over the set of objects constituting the arguments to the method. This latter approach is to be preferred since it frees the specifier from having to define explicitly the messages necessary to be communicated in order to communicate state. Especially since the identity of the communicating agents and of the communicated state should be deducible from the operation specification.

Although it may seem more reflective of complex systems to allow the topology to alter over time, it conflicts strongly with program inducibility as far as the static nature of the occam language is concerned. For this reason, expediency dictates that topology specifications be as general possible.

Dynamic properties refer to the sequence of state changes that take place over time, where such sequences may be either deterministic or nondeterministic.

The dynamic properties constitute an algorithmic view of the behaviour of the system. The static specification speaks of the components of the system, of their properties or attributes and of their potential avenues for change or operations. The dynamic specification is required to describe the coordinated manner in which the activity of the operations upon the attributes of the agents takes place temporally. The essential abstractions here would be temporal in nature, that is the relative timings of the activity of operations, whether such activities may take consecutively or simultaneously.

2.3.2 Specification and Interpretation

Grouping

Once an abstraction approach has been adopted, for example as described in above (section 2.3.1), it is possible to define a specification notation to capture the model in a formal way.

Typical specifications in the present model will be generic, that is, they would describe a collection of systems by virtue of being parameterized. They would be termed generic specifications. Generic specifications equipped with all necessary actual parameters would constitute an instance specification.

The sole abstraction given for expressing generic concepts is the class. This achieves generic expression with respect to the type of attributes, but not operations since operations are defined in a pairwise fashion, and not on a per-class basis, unlike attributes. Nor does it achieve genericity in an algorithmic sense, that is, in a sense related to dynamic specification, nor even in a topological sense.

The syntactic mechanism of *parameterized specification* will be adopted to perform this role. Parameterized classes are a common object oriented abstraction, but the parameterization approach may be generalized in the current model to cover all parts of a specification, from topology to operation sequence definitions.

The syntactic grouping (section 2.1.3) requirement must be satisfied by a formal specification language if it is to facilitate classification through parameterization. The parts of a group specification that vary in instance specification would be denoted by parameter names.

In describing the sections and symbols of the specification language, chapter 4 will also define the interpretation of specifications in terms of real complex systems. Largely, this form of interpretation may be simply described in terms complementary to the abstraction detailed in chapter 3.

Simulation

Interpretation of specifications in terms of parallel programs, or the writing of simulations, is less straightforward, as previously explained in connection with the directness requirement (section 2.1.2). An outline of the proposed simulation mapping now follows. Since the model has been heavily influenced by communicating process architecture concepts, it is most natural to formulate such a mapping to the programming language occam.

The agents themselves would map directly to processes. The topology of agents specified would map onto an assembly of processed interconnected by communication channels corresponding directly to the edges in the topology. The topology is therefore equivalent to a high level specification of the system's communicating process architecture. Processes simulating agents may be simple sequential components or may have nested parallelism, although permitting the latter may complicate formal validation of generated code. In any case, the process assembly as a whole must be a concurrent system.

The attributes and operations may be implemented as local variables and procedures respectively. With operations, there needs to be reconciliation between the per-process approach enforced by occam's usage rules pertaining to locality and the inter-object approach taken in specifications. This essentially means that two-way communications have to be deduced from information only specifying one-way communications. Furthermore, the unspecified low-level nature of communications, that is, whether synchronous or asynchronous needs to be made.

As far as the dynamic specification is concerned, the appropriate implementation would be in the correct coordination of all the concurrent and cooperative activity in the simulation. In occam coordination is achieved by means of algorithms embedded within the process architecture through the mechanisms of sequence, synchronous communication and alternation.

In general, once all parameters have been supplied to a specification, there is sufficient information to deduce implementation details in a language such as occam. The specific question also arises as to when non-deterministic choices allowed in the specification are to be committed. If choices are 'hard-coded' into generated programs, randomized run-time behaviour would not be observed. The intention in specifying non-determinism is normally so that such behaviour, which is typical of complex systems, will be exhibited in simulations. Therefore the preferred policy is for commitments to a particular sequence of actions to be made at run-time, as near as possible to the point where such decisions would take effect.

Chapter 3

Abstracting Complex Systems

Context

In previous chapters the utility of, and need for, formal specification of complex systems was explained. In essence it is necessary to be able to write formal specifications of systems, especially in a manner that captures the commonality found in groups of similar systems. These specifications can embody system classifications and are the base upon which induction of simulation programs may be performed.

A necessary preliminary to writing formal specifications is the abstraction technique according to which the relevant features of a system may be identified. The aim of this chapter is to define completely the proposed abstraction outlined in section 2.3.1.

Consideration of the actual notation used by specification, as well as their interpretation, is deferred to chapter 4. Chapter 5 makes use of the model's capability for abstraction in order to specify realistic groups of complex systems.

3.1 Introduction

The formulation of the following abstraction of complex systems has been motivated by a need to specify interesting systems formally and precisely. It has been deliberately furnished with a small number of concepts in order to facilitate two main aims: firstly, in order to promote ease of learning and use; and secondly, to simplify the definition of formal reasoning and manipulation procedures given in chapters 4 and

6.

It is assumed that an analysis of the system to be specified is performed initially. This analysis should uncover all relevant facts about the system and its behaviour, by making use of the abstraction guidelines presented here.

The abstraction adopts both structural and temporal views. Structurally, systems are seen as networks of communicating entities. These entities are objects in the sense of traditional object-oriented modelling, i.e. they each have distinct identities, encapsulate state information and are instances of some specific class of objects. In addition, this some of the features of communicating process architectures are adopted insofar as objects are seen to be operating concurrently and without synchronization or a common clock.

Whilst the structural view indicates the static composition of the system, the temporal modelling aspects describe the dynamic behaviour of the system. Behaviour is viewed as the pattern of state change in the system over time. State changes are viewed as events, and behaviour is modelled as restricted traces[Hoa85] over a suitable alphabet of events. Complex systems, being concurrent, are likely to be such that at any time, more than one object is actively altering its own state or communicating with other objects.

Description of groups of complex systems is achieved at the notational level, by writing parameterized specifications. Only those system aspects common to all members of a group are represented formally, and the remaining aspect that vary between members of the group are left undefined. Such under-specified group definitions may be instantiated, i.e. used to specify a certain system, simply by textually substituting system-specific formalizations for the parameter names.

The remainder of this chapter is devoted to a more detailed treatment of the above concepts.

3.2 Specification Structure

The concepts of the given abstraction are employed as an aid to the analysis of the complex system of interest. The written specification that results from such an analysis should have a structure that serves the broad purpose of modularity. Modularity of specifications would have two main benefits. Firstly, there should be a separation of concerns between the static and dynamic elements of a specification in order to mirror their conceptual separation. The dynamic aspects may only be adequately analysed after the analysis of static aspects, and a separation of the two sections of a specification would reinforce the specifier's awareness of this dependency. Secondly, the two sections are independently reusable, meaning that given a number of existing specifications, it is possible to write a new specification by adopting the static section of one existing specification and the dynamic section of another. Such reuse depends upon the existence of group specifications, parameterized in such a way as to make them adaptable to a diversity of systems.

Both sections of a specification may be parameterized independently to promote independent reuse of sections. More common, however, would be the reuse of the entire specification. There are two forms of reuse. Firstly, there is a form of reuse akin to the object-oriented concept of interface inheritance, in that an existing group specification may be specialized into a more restricted group specification, or even an instance specification by the mechanism of supplying actual parameters. Secondly, the reuse of parts of a specification to create another specification of a system that is similar, but not normally a more specialized one. This is analogous to implementation inheritance or code inheritance in object oriented programming.

The conceptual tools given in the following are to be used in analysing and then abstracting complex systems into their static and dynamic aspects. It will be noted that they permit expression of structural and temporal views of systems, though not of a detailed computational view. Computations are encapsulated by the operations concept described below. In the interest of directness of modelling (see section 2.1.2) their implementation is intentionally left out of the scope of system specification, and it is assumed that they may be defined in the form of software modules in some programming language (such as procedures in occam). However, modules would be complementary to the specification, rather than being integral to it. The implementation of each operation requires that one or more such modules be defined. In programming language terms, each event is implemented by invocations of the relevant modules (in occam, for example, as procedure calls).

3.3 Abstraction Towards Static Specifications

3.3.1 Introduction

The analysis of a system into objects is the first stage of specification. An alternative would be to determine the state contained in a system, and thereafter to divide it among objects. Such an analysis may be conducted at various levels of granularity. At the finest level, each item of state information belonging to the system is assigned to a separate object. At the coarsest level, the entire system is a single object, but complex systems, being highly distributed, would seem to defy this 'centralized' form of analysis.

Objects, their topology in the system, and operations performed upon them, are the prime static components of a specification. These characteristics are taken to be immutable over the lifetime of a system.

3.3.2 Objects

An object in a complex system is any entity having state attributes that are subject to change. The objects are denoted by unique names or identities. The set of identifiers of objects constituting the system is called the object set.

For example, in cellular automata the cells themselves are the objects. Similarly, in neural networks the neurons may be viewed as objects, however the connections between neurons also have state which they serve to transmit from one neuron to another, and so are objects as well.

The system topology is a directed graph whose node set is the object set, and whose edge set is some subset of the square of the object set (i.e. the Cartesian product of the object set with itself). An edge exists in the system's topology iff the object denoted by the source node of the edge passes messages to (interacts with) the destination node object. The source and destination of an edge may not coincide. A message refers to the communication of the local state information of one object to another. State information is private to each object and is made visible across object boundaries solely through the passing of messages between adjacent objects in the system's topology.

An object's state content is divided between a collection of one or more attributes. Like objects, attributes are uniquely named. Attributes possess values taken from a set called the type of the attribute. The state of an object at a particular time is given by the association of its attribute names with their respective values at the given instant.

An object's signature is the association its attribute names with their respective types. Objects having the same signature are said to be instances of the same class.

For example, in Conway's Game of Life cellular automaton[BCG82] the topology is a two-dimensional grid of objects. Since each cell requires access to the states of all its neighbours there are edges in the topology to each object from all four neighbouring objects (disregarding smaller neighbourhoods at the edge of the grid). According to the game a cell may be either alive or dead: this may be abstracted so that each object possesses only one attribute, the type of which is the set of two Boolean values, and whose current value indicates whether or not the cell represented by the object is alive. In this abstraction, just one class of objects exists.

Attribute values must belong to simple value types, for example, those built into common programming languages such as numeric and Boolean types. In particular, objects are not valid attribute values, and therefore the nesting of objects is not possible.

3.3.3 Operations

The components of actual complex systems engage in typically iterative processes. The number of fundamental kinds of operations, or state changes, performed upon objects are small, but complex behaviour results from their repeated application, due especially to the inter-dependence between components.

The abstraction relating to state changes is the operation. A operation is considered to be a uniquely named function whose input is the state just prior to the change — the pre-state — and whose output is the state just following the change — the post-state. In the manner of their definition, operations resemble the operation schemas of Z specifications[Spi89]. Operations thus defined are restricted to causing state change in a single object. Thus the post-state of an operation refers to the new state of exactly one object, whilst the pre-state refers to the immediate antecedent state of that object.

It is often necessary for an operation to depend upon the pre-states of more than one object. Indeed, it has already been noted that interaction between components is a defining characteristic of complex systems. This is supported in a restricted form: such an operation is allowed to depend upon the pre-states of exactly two objects, and furthermore, the post-state of the operation should refer to the one of those objects. If the operation depends on the states of two objects, there must be an edge in the system's topology between the two objects, directed toward the object undergoing change.

This restriction of the number of objects involved in an operation to at most two may necessitate the specification of artificial entities, in the form of extra attributes and operations. These artificial attributes play the role of 'buffer variables,' which are written to by appropriately defined artificial operations.

For example, consider a cellular automaton whose cells are arranged linearly such that each cell interacts with two neighbours. Further, assume that each cell possesses a single state attribute, called the principal attribute. The state updating operation for any cell depends upon the pre-state of that cell and of its two neighbours. It is not possible to abstract this update rule as a single operation since three objects are involved.

A solution is to add two extra 'buffer' attributes to the signature of all objects. In each object, these attributes will contain, respectively, the 'received' state values of its two neighbours. Three operations are required. Two of these are artificial: they serve to convey the principal attribute value of a neighbouring object to the respective 'buffer' attribute of the local object. The third operation, involving a single object, is responsible for applying the cellular automaton's update rule to the three attributes of a single object in order to set the principal attribute's new value.

The communication model assumed in specifications is a restricted form of shared memory. However, communication is not an explicitly modelled concept, although it is easily apparent that any operation depending on the state of two objects equates with an unidirectional communication across an object boundary.

In chapter 6, rules are given to permit the deduction of message-passing information from the shared-memory one abstraction given here. We work on the principle that although message passing information is required for the targeted implementation language, occam, specifiers would prefer to express shared memory concepts which are less verbose and less cluttered by considerations of the integrity of object boundaries. By regulating the shared memory concept with the rule preventing more than two objects from participating in an operation, information required by the underlying model of unidirectional message passing is deducible, yet that model is hidden from the specifier. The cost of so doing is to require specifiers to introduce artificial entities into specifications in order to formalize the more complicated operations.

3.3.4 Topology and Operation Instances

The system's topology, mentioned above, is a high level diagram of the process architecture of the system. The topology defines for each object the set of objects (its neighbours), possibly including itself, whose state it may examine.

An operation instance is a less abstract concept of an operation. An operation defines the manner in which the pre-state of one or two arbitrarily chosen objects affects the system's post-state. However, in reality, it is useful to refer uniquely to an operation acting upon specific, rather than arbitrary, objects. Such as reference to an operation, or operation instance, is made by connecting the operation definition itself to the participating objects. In practice, this may be conveniently achieved by stating the name of the operation and the identity of the participants involved in the specific instance.

A complete specification of a system should include a declaration of all operation instances.

The topology itself simply describes neighbourhoods but gives no indication as to which edges are appropriate for specific operations. The topology thus contains inadequate information to deduce the valid operation instances for a system. For example given a certain operation, there is no justification for assuming an operation instance for every edge in the topology; the system may be such that the operation only finds instances for a proper subset of the edges. Instance information is also absent from operation definitions, but this is so by the very nature of operations, which are abstract and make no reference to specific objects.

3.4 Abstraction Towards Dynamic Specifications

3.4.1 Events

The unit of behavioural description of complex systems is a specialization of the event concept as described in the Communicating Sequential Process (CSP) model[Hoa85]. The events here share all characteristics of CSP events, namely atomicity and an instantaneous nature. In addition to such CSP characteristics, the events are tied to operations. An event is a specific occurrence of a state-changing operation. Moreover, each event is defined so as to occur exactly once during the lifetime of a system. Each event is given a unique identifier.

The alphabet of a system consists of all events occurring over the lifetime of the system.

Since most specifications of complex systems describe iterative systems, most if not all operations will be repeated as the system runs. However, each operation is an abstract entity that acts as a template for generating events. In order to enumerate all the events that a particular operation gives rise to, two extra pieces of information are needed. Firstly, the identity of the object, or two objects, as the case may be, participating in the operation. Secondly, since such objects are likely to participate in the same operation repeatedly, an occurrence count. Enumerating all occurrences of each operation instance in this way will yield the alphabet of the system.

Given the above definition of events, all traces over the system's event alphabet containing exactly one occurrence of each event identifier would be possible candidates for dynamic specifications.

3.4.2 Orderings

The mechanism used to express dynamic specifications precisely is the event ordering relation. Effectively, it specifies all event traces that constitute valid behaviours of the complex system being abstracted. In a purely sequential system, that is, one whose events occur according to a known and strict order, there can only be one valid trace representing the system's lifetime. Typical complex systems would not exhibit completely sequential behaviour: in the presence of either parallelism or undefined ordering, or both, there is bound to be more than one valid trace.

Defining the order of events in valid traces of a system poses a question of the semantics of the lack of order between events, or of under-specification. Two possible interpretations of unordered events exist, which will be referred to as indeterminacy and concurrency.

Indeterminacy is very common in systems exhibiting randomized behaviour, particularly complex systems, where it is sensible the restrict a certain set of events to occur consecutively without having them adhere to a predefined sequence.

Concurrency is likewise fundamental to complex systems. This phenomenon is a less restrictive form of indeterminacy permitting a set of unordered events to occur simultaneously. Then by definition indeterminacy is a special form of concurrency. Since concurrency is the more fundamental interpretation, all events which are unordered with respect to each other will be interpreted as concurrent events. Indeterminacy is indicated in specifications by saying that the events concerned are totally ordered, while leaving unstated the precise definition of their ordering.

Therefore, lack of ordering denotes concurrency and incompletely defined ordering denotes indeterminacy.

3.4.3 Comparison of Communication Models

It is noteworthy that the distinction between asynchronous and synchronous communications has not been manifested either directly or indirectly in the given specification model. This is in contrast to the other characteristics of the CSP communication model: unidirectionality and point-to-point nature, both of which are indirectly manifested in the model. However, occasions arise where specifiers need to simulate buffered communications using the existing concepts of the model, as explained above in connection with artificial entities.

At the level of abstraction of specifications, the distinction between blocking and non-blocking communications cannot appear. It is merely taken for granted that the ordering relation between operations should hold true.

According to the present model, the temporal context assumed by operation definitions is that of the receiver. This is equivalent to a one sided specification of communications, as distinct from the two sided model of CSP. Consequently, operations implying a communication, or in other words, those operations whose effect depends upon the states of two objects, are open to different interpretations in relation to the consistency of data between the two participants.

Such an operation will naturally involve distinct transmitting and receiving objects, and will effect a state-change in the latter. Therefore an event corresponding to such an operation will only be observed in the system as alterations to some of the receiver's attributes. But how consistent should be the two components of the operation's pre-state, the states of the transmitter and receiver? Since objects take on the properties of CSP processes, running in independent temporal contexts, that is, having separate unsynchronized clocks, it is not possible from within the model to state that the data arriving at the receiver must be identical to the current state of the transmitter at the exact moment that the event occurs. This is the most desirable interpretation of data consistency.

Chapter 4

Specifying Complex Systems

Context

Chapter 3 explained the facilities of the model relating to its abstraction relation. This chapter concludes the presentation of the model by defining a formal notation for specifying complex systems. The notations assumed here for such items as function definitions and traces are given in appendix A.

The interpretation of parts and constructs of a specification are also described. Since the interpretation relation under discussion here is complementary to the abstraction relation, chapters 3 and 4 are closely tied to each other.

Chapter 6 exploits the interpretation of the notation given here in order to define a mapping from formal specifications to occam programs.

4.1 Top Level Structure

A group specification C[SF,DF,MF] is composed of two distinct parts: a static specification section S[SF,MF] and a dynamic specification section D[DF], where DF and SF are lists of formal parameters (they may be read as 'static formals' and 'dynamic formals,' respectively), and MF is a list of formal module names (section 3.2). S[SF,MF] is composed of object, topology and operation declarations, while D[DF] is composed of event and ordering declarations.

All declarations are of the form *name : definition* where *name* is a unique literal and *definition* may be a set, a first-order logical predicate, a function definition or a trace declaration, depending upon context. Definitions in either section may contain names of free variables provided that such variables belong to the relevant list of formal parameters; that is, definitions in a static specification S[SF,MF] may use free variables from SF and MF (although module names may only occur in the operation definitions of the static specification, as noted below), while definitions in a dynamic specification D[DF] may use free variables from DF.

An instance specification, C[SF,DF,MF][SA,DA,MA], supplies actual parameter lists SA, DA and MA, to a group specification. Each member of an actuals list denotes the value or name to be bound to the corresponding member of the relevant formals list. By simple textual substitution of actual values or names to formal names throughout the group specification, an instance specification is obtained.

Only an instance specification may be interpreted as an actual system. A group specification represents abstractly all those actual systems that are interpretations of its instance specifications.

A group specification describes formally and in outline the organization (in the static specification) and behaviour (in the static specification) of a group of systems. In its static section, a group specification normally describes the high-level organization, without reference to specific system topologies and specific types of state attributes. These aspects are denoted by formal parameter symbols that are to be bound upon instantiation. What is normally defined in the static part of a group specification is the structuring of the state in the system and its division among objects.

The static section also states generically the potential operations upon state in a object-wise or object-pairwise fashion, depending on whether one agent or two agents are involved (section 3.3.3). The group specification is not explicit about the actual computations upon state, just as it is not so concerning the qualitative nature (or types) of the state. Computations will be defined as modules in external programming languages (such as occam) and the relevant module names will be passed as actual parameters upon instantiation. However, for each operation, the dependency relations between the participating attributes (i.e. parts of a system's state, as given in section 3.3.2) are required to be defined generically.

In the dynamic section of a group specification is described the high level behavioural structure. This is not required to be expressed explicitly as a concurrent algorithm, but rather through the event and trace abstractions introduced in section 3.4. This latter form for expressing behaviour permits generic expression through judicious use of formal parameters. The behavioural aspect needs to be stated with reference to the static nature of systems, since system behaviour is observed as changes occurring within the system organization. What is specified concerning system behaviour is the interaction and synchronization between the operations that take effect upon the system's objects.

4.2 Objects

Objects correspond to autonomous components of complex systems. They may be identified as the smallest state-possessing structural units of complex systems. All the state information possessed by a system should be partitioned and distributed among the attributes of constituent objects. The classes of objects in a system are then declared by their respective objects' characteristic signature (section 3.3.2). The declaration of a class, c, of objects having N attributes, for example, would include the definition of a sequence of attributes:

$c \quad : \quad < a_1 : t_1, \;\; a_2 : t_2, \;\; \ldots, \;\; a_N : t_N >$

where indexed (subscripted) a symbols stand for unique attribute names and correspondingly indexed t symbols are types. Types are sets of values, but for convenience they may be written as names of primitive types in occam, such as *INT* or *BOOL*, if required.

In a group specification S[SF] it is permitted to substitute a name taken from SF for any type.

The object set of a system (section 3.3.2), Ω is declared as follows for a system composed of M objects:

$\Omega \quad : \quad \{o_1 : c_1, \;\; o_2 : c_2, \;\; \ldots, \;\; o_M : c_M\}$

where indexed (subscripted) o symbols stand for unique object names and correspondingly indexed c symbols are class names declared in the same specification.

Ω may be defined through set comprehension in order to take advantage of parameterization. For example, assuming that all objects in a system belong to a single class, c, the following declaration may be made:

$\Omega \quad : \quad \{\forall i \in \{1, 2, \ldots, O\} \;\bullet\; (o_i : c)\}$

This declares O uniquely named objects of class c. O must be a formal parameter given to the static specification.

The set of objects in the system, ω, is derived trivially from the definition of Ω. The former set is especially used in the ordering declarations of dynamic specifications (see section 4.6).

Operation declarations need to refer to the state of objects. The notation for associating an object with its state is a mapping from the name of the object to a tuple containing its current attribute values. The order of values given in the tuple respects the sequence of attributes given in the declaration of the object's class. For example, the current state of an object, o, belonging to class c:

$c \;\; : \;\; < a_1 : \{1,2,3\}, \;\; a_2 : \{4,5\} >$

may be written as, say, $(o \mapsto (1,5))$.

Speaking in terms of systems' dynamic aspect, prior to being first assigned, an arbitrary value of the type is held by each attribute. It is further assumed that no attempt will be made during the lifetime of the system to assign a value simultaneously with one or more another assignments or simultaneously with one or more read accesses. All other combinations of operations may occur simultaneously.

4.3 Topology

A system's topology is a specification abstraction in the form of a directed graph whose set of nodes is identical to the object set, Ω. Edges indicate the direction of message-passing between object pairs (section 3.operations). The topology declaration of a system defines the edge set, ϵ, of such a directed graph, where edges are ordered pairs of object names.

ϵ is not required to correspond to the smallest set of object interaction-pairs, as this is required to be performed separately in connection with binary operation instances (see section 4.4). The edge set that represents a bidirectionally fully connected graph is a valid topology specification in all cases.

4.4 Operations

An operation is defined in the abstract as a change to a single object's state, that is, to one or more of its attributes. The object undergoing change will be referred to as the target of the operation. If the change is dependent upon attributes of an object that, topologically, is a neighbour (or 'partner') of the target, then it is called a binary operation. It is called an unary operation otherwise, that is, if it is dependent solely upon the attributes of the target. No other kinds of operation are possible according to section 3.3.3.

Due to the concurrent nature of objects, with binary operations there is no implicit guarantee of temporal consistency between the attributes that exist in the two objects. If such consistency is required, it should be indirectly stated as a synchronization constraint in the dynamic specification.

Formally, an unary operation is declared as a mapping to the changed state of the target from its prior state. A binary operation is declared as a mapping to the changed state of the target from both its own prior state and that of its partner. Such functions from pre-state to post-state are of the form:

$U \quad : \quad C \longrightarrow C'$

for an unary operation U, and:

$B \quad : \quad CP \times C \longrightarrow C'$

for a binary operation B. Above, C is the instantaneous current state of the class (expressed as the set of its instances' current states as given in section 4.2) undergoing change and CP is that of the class of its partner. The notation in use by the Z specification language[Spi89] is adopted, of attaching the 'prime' symbol to a set to signify that it denotes the state of the set that arises once the operation is complete. In the present specification approach only the state of the object subject to the operation is found to be altered in the post-state set.

The precise definition of an operation should obey the form given above. It should define the new state each attribute of the target object as either the identity function or a module function, of the in-scope attributes. These attributes are simply those that belong to the target and, if appropriate, to its partner. In a group specification, all module functions would be expected to be unspecified. In fact, the specification language itself does not set out to describe the detailed computations that are more appropriately defined by programming languages. The only conception of such functions within the specification framework is that they are uniquely named, that they are parameterized by attributes and that their results are to be assigned to attributes. Module names are the only symbols pertaining to operations that may be signified by formal parameters.

For convenience, the pre-state and post-state parts of an operation definition may be referred to as its 'left' and 'right' sides, respectively.

Since an operation is an abstract function definition, it represent many possible concrete operations. As many, in fact, as there are possible argument objects. An operation that may be activated upon to one or two known objects, depending on whether it is unary or binary, respectively, is known as an operation instance (section 3.3.4). They are still static specification entities since they do not refer to the active transformation of system state in any concrete temporal context. They are, however, the link between the static and dynamic conceptions of operations.

The formal notation used to refer to instances of operations U or B, above, given specific target (t) and partner (p) objects is:

$t \mapsto U.(t)$

which is an instance of U, and

$(p, t) \mapsto B.(p, t)$

which is an instance of B. This means that, for every operation, there is a unique operation instance corresponding to a particular ordered pair of objects. However, especially for the latter case of binary operations, the converse interpretation, that is, that there is a unique operation instance for all possible ordered pairs of objects, is not true. This is clarified below.

Unary operation instances are usually implicitly defined once classes and operations have been defined. However binary operation instances require explicit specification. This is obvious from the restricted spatial nature of interaction between objects in complex systems. Most interactions are constrained by some concept of spatial distance. In other words they are neighbourhood oriented. As noted earlier, the topology provides only partial information about valid neighbourhoods. This is because the topology is operation independent, it provides the most interconnected possible interaction graph without discriminating between neighbourhoods for different operations.

Information concerning restriction of operation instances to subsets of the topology on an operation-by-operation basis is necessary in order to specify precisely the set of valid operation instances. This is achieved by specifying, for each binary operation, a mapping onto that the subset of operation instances which contains all valid instances. This is reminiscent of the function-inducing abstractions used in relational modelling, since operation instances themselves are functions. Formally, the set, b of valid operation instances for a binary operation B would be declared from the valid edge-set of the system's topology to the space of operation instances:

$b \quad : \quad \{\exists E \subseteq \epsilon,\ \forall e \in E \mid p = \pi_1.(e),\ t = \pi_2.(e) \bullet (p, t) \mapsto B.(p, t)\}$

where π_1 and π_2 are the projection functions (see appendix A).

Restricting the targets of unary operations may be similarly achieved through mappings onto function space, but this is needless as a far simpler and readable solution exists, which is to declare objects targeted by those operations to be of a unique class, and then to use this class in the declaration of the relevant unary operation.

4.5 Events

Events are an extension of operation instances into the temporal realm. An event is simply the occurrence at a certain unique point in time of the action specified by an operation instance, which is the change in state of one of the objects in the system with reference to its previous state and under certain conditions that of a neighbouring object as well.

The view of time of event occurrences is a purely relative one. Events corresponding to the same operation instance are viewed as occurring in strict succession, without any possibility of simultaneity. Under the event concept alone, there is no accounting for sequential relationships between events of different operation instances. Such capability is necessary for specifying realistic complex systems, and is the subject of the next section.

Having raised the subject of the dynamic nature of systems, it may be asked what are the precise meanings of sequence and simultaneity. In the CSP model of concurrency, events are considered to be strictly atomic entities that are instantaneous and thus do not have any extensions in the temporal dimension. In this model we relax this constraint. Since static specifications define each operation to be a mediator between one pre-state and one post-state, their concrete aspects, events, are atomic and do not conceptually encapsulate a sequence of computations. However it is still possible to conceptualize them as consuming time, during which delay other events may commence and possibly complete effecting state changes. This is allowed with the understanding that a system may not be legally specified to have multiple concurrent writes to the same object's state or to have a write concurrently with reads.

Given that events are simply unique occurrences of operation instances, it is possible to label them uniquely using a simple scheme that takes into account an occurrence count. Thus, by convention, an instance of operation O upon object t occurring for the 5th time is written $O_{t,5}$.

In instance specifications, each operation should be specified a finite maximum limit on the occurrence count of its instances' events. This number may be substituted by a formal parameter name in group specifications for purposes of generality. Consequently, in an instance specification, the number of events occurring during a system's lifetime is effectively declared.

4.6 Orderings

More complex temporal relationships between events than simply the successive ordering of the previous section will be needed. A notation which can be used express arbitrary temporal relationships between events would constitute a specification language for arbitrary behaviour patterns. Furthermore, if it were generic to a whole class of similar temporal patterns, it would constitute an algorithmic skeleton[Col89] description language.

The abstraction used to specify temporal relationships between events is the event ordering relation. This is declared as a transitive relation, $<$, over all events in an instance specification. $e1 < e2$ specifies the requirement that event $e1$ occur strictly prior to $e2$, in the same sequential fashion as, for example, two successive operation instance events. Insofar as events may be thought of as having a duration, $e2$ may not commence until the state change caused by $e1$ has taken effect. The relation being a total one, it must be possible to establish for any two events in the dynamic part of an instance specification, whether or not they are definitely sequentially ordered. The specification is not ambiguous in this sense. Since the relation is transitive, ordering is propagated through chains. In this and other factors, this specification method resembles the event structure formalism[Win89].

Other temporal relationships than strict sequence also hold in complex systems. These are indeterminacy, that is, randomized ordering, and concurrency. However, just the single primitive of sequential ordering is sufficient to discriminate unambiguously between sequence, indeterminism and concurrency. A set of events will be deduced as being mutually concurrent iff none of the events are ordered with respect to any other. Concurrency is the least temporally constrained relationship between events. A more constrained relation, though less so than sequence is indeterminism, where sets of events are to be sequentially ordered, yet are permitted to occur in any combination. Consideration of this will follow after the following discussion of notation.

Practically, the binary relation $<$ is at too low a level of expressivity to suit a specification language. A notational and conceptual convenience is to adopt a higher level language based on traces. The trace notation associated with CSP is ideal. To employ trace terminology, the ordering relation defines the set of traces that are valid observations of the behaviour of the system. For example, $e1 < e2$ for a certain system is equivalent to saying that $e1$ precedes $e2$ in all valid traces of the system.

In addition, an event-pair-by-event-pair specification of $<$ is apt to be done arbitrarily and without rigour. With the aid of traces, a more systematic, operation oriented, approach to specifying event orderings is made possible. This will assume the existence of a specification device in the form of a trace, say pre, of the lifetime of the system leading up to a certain event's occurrence, say e. Since events are unique, there will only be one such occurrence. Now the task of specifying orderings with respect to e is in characterizing the pre trace in terms of its constituent events. The precedences of every family of events should be accounted for in the dynamic specification, and this may be done by independently for each event family by specifying the precedences for each family and connecting all such clauses into a conjunctive form.

The notation for declaring precedences assumes that pre, the record of the system's lifetime up to a certain point of interest, has been advanced by the occurrence of a single event, denoted e. Then the task of writing a precedence declaration for events belonging to a particular operation requires, firstly, the assumption that e belongs to that operation, and secondly, the characterization of *pre* under that assumption. For example, these two declarations state that all events belonging to the operation $O1$ should be preceded by event d and that all events belonging to operation $O2$ are preceded by event c:

$e = O1_{t,n} \implies < d >$ **in** *pre* $(\forall t \in \omega,\ 1 \leq n \leq N_1)$

$e = O2_{t,n} \implies < c >$ **in** *pre* $(\forall t \in \omega,\ 1 \leq n \leq N_2)$

Note that N_1 and N_2 are formal parameters to the dynamic specification. In the above, they denote the number of occurrences of any instance of either $O1$ or $O2$, as the case may be.

Given this form of specification, we may return to the consideration of indeterminate sequences. This will assume that a sequence, which may be left specified as simply a formal parameter in group specifications, is to be committed as the actual sequence of events to be followed. Such commitments of random, indeterminate sequences must happen at the time of a specification's instantiation. A finite sequence of known size, say N, is a bijective function from $\mathcal{Z}_N$, the set of the integers ranging from 1 to N, to the set of members of the sequence. Commitment of a choice in this context refers to the defining of such a function, say s. Then an event belonging to an indeterminate set, and belonging also to an operation family O, may be defined by the following clause:

$e = O_{t,n} = s.(i) \implies < s.(i-1) >$ **in** *pre* $(\forall t \in \omega,\ 1 \leq n \leq N)$

where, again, N is a formal parameter to the dynamic specification.

A constraint upon ordering declarations, considered as a whole, is that any event that belongs to a binary operation family must be preceded by at least one other event. This is necessitated by the program induction procedure of chapter 6 in order to implement communications via channels.

4.7 Correctness of Specifications

The possibility of events occurring simultaneously, and by implication state changing operations too, poses a difficulty in the form of mutual exclusion of operations from shared data. Typical specifications will contain several operations that require access to the same attributes. Where unordered events are occurrences of such operations, and furthermore involve the same objects, then the same data item is being used by several operations simultaneously. This is harmless in the case where all relevant operations merely read the value of the common objects' attributes. In all other cases, that is when, at the same time, all relevant operations are ready to write or when at least one is ready to write while others are ready to read, the resultant effect is unpredictable.

Rather than giving a definite interpretation to such cases, it is decided that specifications that break the constraint of mutual exclusion are incorrect. These cases arise due to underspecification of the ordering relation. However, given the labelling scheme for objects and events presented in the next chapter, it is possible to verify formally specifications with respect to mutual exclusion, and to identify precisely the set of offending events in incorrect specifications. Although the model provides no immunity against specifications that fail the mutual exclusion constraint, formal tools can be constructed to assist in the repair of specifications that are incorrect in this respect.

A specification is also rendered incorrect if it contains a deadlock condition. This is the case when the event ordering relation contains a cycle.

In verifying specifications in the presence of indeterminacy, all possible combinations of events lying in indeterminate sequences should be checked for mutual exclusion or deadlock.

4.8 Conclusion

The language used for representing a system to be simulated should be expressive of both static and dynamic aspects. A notation has been introduced to represent each of the properties identified as necessary for the modelling of complex systems, namely state, state changing functions, message passing and deterministic and nondeterministic event ordering. A significant departure from the occam programming philosophy is the elimination of any direct representation of bidirectional message passing channels, in favour of a one sided communication model where only the receiving party is explicitly involved in the communication. This is done in order to simplify the concept of communication.

Class specifications model entire groups of related systems. A form of modularity is introduced by structuring specifications into general and specific components. The latter would consist of specific information about the system to be simulated, as opposed to properties 'inherited' from the class of systems to which it belongs. In addition this component may also specify implementation related information such as the topology of the processor network available for execution.

Chapter 5

Specification Case Studies

Context

Previous chapters have developed a specification model for complex systems and a specification language to permit its expression in a formal notation. The notation and textual format adopted by the specifications in this chapter are not meant to conform strictly to those given in chapter 4. Rather they demonstrate the application of the concepts of chapter 3 to the abstraction of actual complex systems.

Of special importance to the present thesis is that specifications exhibit generality. The practical benefits of grouping, as a mechanism for simplifying the task of the specification writer, is demonstrated. Another benefit demonstrated is the ability to exercise formal procedures for proving properties about specified systems.

The application of the specification model as a descriptive tool, then, is the main focus of this chapter. In chapter 6, its other role as means of defining a program induction procedure is demonstrated.

5.1 Cellular Automata

5.1.1 Introduction

Many of the systems classed as complex systems may be shown to be equivalent to cellular automata (CA)[Wol84]. This section presents a specification of CA. A specification of an extremely simple CA-like system is progressively refined to produce a specification of a realistic CA. Then the generality of the specification language is exploited to adapt the final specification to express particular CA-based systems.

Cellular automata (CA) are discrete models of dynamical systems that serve as convenient computational frameworks for the study of complex systems. The model of cellular automata presented here considers such aspects as system entities, state, operations, neighbourhood communication, time and synchronization.

A CA normally consists of a regular graph of *cells* where each cell may be in any one of a finite set of *states* at a particular time. The edges connecting each cell with its neighbours represent channels of communication. Cells collectively perform state transitions in discrete steps of time in the following manner: each cell inputs the current state of its neighbours and then determines its new state by computing the CA's update function. The *update function* relates the current state of a cell and that of its neighbours to its state in the subsequent step.

Starting from an initial state, the CA iteratively passes through the phases of state exchange and state update. Note that all cells are normally synchronized at the commencement of an iteration and that such behaviour is termed *bulk-synchronization*.

5.1.2 Development of a Specification

A Simple Distributed System

As an initial specification to be refined in subsequent sections, a greatly simplified form of a distributed, discrete-time system is introduced here.

Essentially, it is a CA consisting of a finite number of cells, to the exclusion of both communication channels, and by implication, the concept of neighbourhoods. To exclude any necessity for cells to communicate, bulk-synchronous updates are not required. Thus, each cell simply updates its own state iteratively, without reference to the states of neighbours. It is arguable as to whether the entire system may be called iterative under such conditions, but certainly it should be clear that the cells, by themselves, behave iteratively.

Consider this system to be a complex system. The system may be described in an abstract sense by referring to the objects composing the system as members of a set, *Object*. Similarly, let the set of states in which an object may exist be *State*. Both *Object* and *State* are finite sets. Let the update function, defined to be from *State* to *State*, be called *UpdateState*.

In order to associate objects with their state, an 'associative' function from *Object* to *State* is introduced. It may be considered to be a total function, which is to say that all objects have clearly identifiable states.

In this system the objects are expected to change their state. Since it is a dynamic system, the formal description should be capable of expressing changes in the state of objects over time. To do so using the associative function (call it *SystemF*), either index the members of *Object* with time or define a separate association for each time step. The latter approach is to be preferred as it does not necessitate altering the definition of *Object*. For the present purposes, the number of instances of *SystemF* may be reduced to just two by adopting the view of time supported by the Z specification language[Spi89]: that is, to consider a pre-state and a post-state, and describe the change in the system during a time step as a predicate relating the pre-state to the post-state. Taking the notation from Z, label the pre-state as *SystemF* and the post-state (another total function from *Object* to *State*) *SystemF'*.

For purposes of interpretation, it is to be understood that *SystemF* 'becomes' *SystemF'* instantly upon the appearance of the post-state. To be more precise, recall that in CSP terminology[Hoa85], events (only state-update operations are events of interest here) occur in a sequential order or *trace*. Now supposing that event $e1$, denoting an occurrence of the operation e, is the immediate predecessor of event $e2$ in the trace, the post-state *SystemF'* of $e1$ is taken as the the pre-state *SystemF* by $e2$. This is for the simple case where the successor event $e2$ has a sole predecessor event. In more complex cases a certain event may have more than one predecessor event. Then the event's pre-state is defined to be the union of the post-states of the predecessors.

Let *System* and *System'* be the sets of ordered pairs given by:

$$System = \{\forall ob \in Object \bullet (ob, SystemF \cdot ob)\}$$

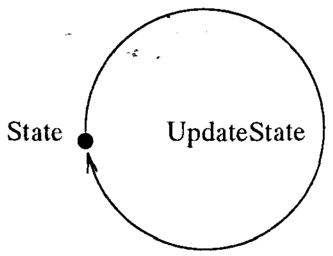

Figure 5.1: *UpdateState* as a function mapping the set *State* to itself.

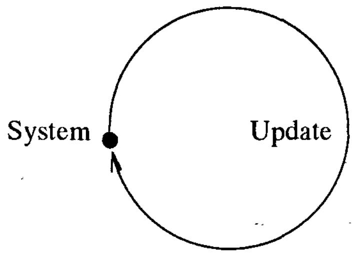

Figure 5.2: Diagram indicating the state and operations of the simple distributed system.

$$System' = \{\forall ob \in Object \bullet (ob, SystemF' \cdot ob)\}$$

Hereafter, *System* and *System'* will be used as equivalents of *SystemF* and *SystemF'* to refer to pre- and post-states, respectively.

The only operation performed by cellular automata is the update of cells' state according to some function of their current states. To illustrate the effect of the update operation upon the system's state, a diagrammatic notation is introduced as a preliminary to a formal statement.

Figure 5.1 corresponds to the signature of the *UpdateState* function, taking *State* as both its domain and range. It can be thought of as an attempt to capture the dynamics of the system in picture form. Clearly, as time progresses and the system iterates, individual states are transformed according to the update function.

It will be seen that a more effective picture of the manner in which the system is altered, as it iterates, is the relabelled diagram, figure 5.2. *Update* is an analogue of *UpdateState* as defined below.

The formal definition of the update operation is as follows:

$$Update : System \rightarrow System' : (ob \mapsto st) \mapsto (ob \mapsto UpdateState.st)$$

Update is to be interpreted as an operation that transforms the state *st* of an object *ob*, as a function *UpdateState* of *st*. With respect to the *Update* function or operation, $ob \mapsto st$ is a member of the pre-state *System*, while $ob \mapsto UpdateState \cdot st$ is a member of the post-state $System'$.

Neighbourhood Communications

The example in the previous section deliberately avoided the need to specify interactions between cells. In this section, a system will be examined that builds on the abstractions introduced earlier to address matters of communication and interconnection between cells.

Let the new iterative system consist of cells that again have state, but let each update itself based on its own state and the state of one other cell — its *partner*. The manner in which a cell chooses a partner is left unspecified, indeed it may interact with different partners over successive iterations, or even with itself.

Partnerships need not be mutual, so in the case where a cell *c*1 updates itself with reference to a cell *c*2, it does not necessarily follow that *c*2 updates with reference to *c*1. This freedom is allowed in order to postpone consideration of synchronization until the next section.

The sets *Object* and *State* are exactly as defined above, and *UpdateState* is defined to be a function from the product $State \times State$ to *State*.

In addition to the sets and functions mentioned earlier, a further set is required to describe the topology of the system. For the present, fully connected, case this will simply be the product set $Object \times Object$ which will be called *Topology*. In general, *Topology* will be defined as that subset of $Object \times Object$ whose elements represent neighbouring objects.

As before the cells iteratively alter their state; in the diagram shown in figure 5.3, the arrow *Update* is again used to indicate this fact.

However, the operation of updating is itself dependent upon the state of pairs of neighbouring cells. One may also say that topological information about neighbourhoods is used to decide whether a pair of cells will take part in the update operation. A 'meta-operation' is introduced, called *Constraint*, to limit pairings between objects to those that are members of *Topology*.

Note that the diagram does not reflect *Update* as taking a product for its domain.

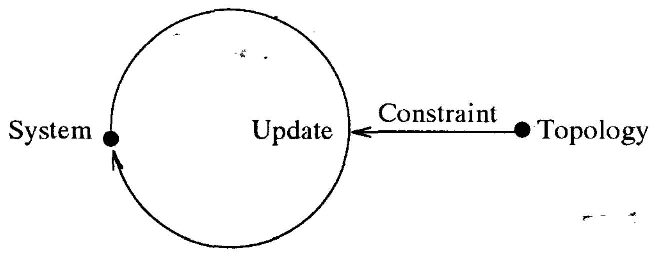

Figure 5.3: The *Update* operation shown to be determined by topological constraints, as indicated by the function *Constraint*.

Diagrammatically, the *Update* arrow is identical to that shown in the previous example (figure 5.2), and serves to indicate exactly that the post-state of *System* is dependent on the pre-state of *System* alone.

The evidence for communication between objects comes from the *Constraint* function. In the present example this is the primary role of *Constraint*, since any-to-any interactions are permitted and there is no occasion to prevent any particular partnership from forming.

The nature of *Update* is largely unchanged from the previous one:

$$\begin{aligned} Update: \quad & System \times System \rightarrow System': \\ & ((ob \mapsto st), (obn \mapsto stn)) \mapsto (ob \mapsto UpdateState \cdot (st, stn)) \end{aligned}$$

Take note that the state of the first member of the pair, *ob*, is being recomputed in the operation; *obn* is the partner of *ob*. *Update* does not affect the state of the second member. This is a convention for representing operations on object pairs.

Constraint maps *Topology* to the hom-set[1] $H((System \times System), System')$. Its range is further specified in the definition:

$$\begin{aligned} Constraint: Topology \rightarrow \{ \quad & \forall edge \in Topology, \exists self, neighb \in System | \\ & \pi_1 \cdot edge = \pi_1 \cdot self \wedge \\ & \pi_2 \cdot edge = \pi_1 \cdot neighb \bullet \\ & (self, neighb) \mapsto Update \cdot (self, neighb)\} \end{aligned}$$

[1]The set of all function from *System* × *System* to *System'*.

where π_1 and π_2 are projection functions (appendix A). This is to state that the range of *Constraint* is the set of all state-update operations performed on neighbouring pairs of objects.

An instance of *Constraint*, given objects *ob* and *obn* where (ob, obn) is a member of *Topology*, is:

$$(ob, obn) \mapsto ((self, neighb) \mapsto Update \cdot (self, neighb))$$

where $\pi_1 \cdot self = ob$ and $\pi_1 \cdot neighb = obn$; $self, neighb \in System$.

It may be noted in passing concerning figure 5.3 that it strongly resembles a relational diagram (see section 2.2.1), especially in possessing an arrow that denotes a higher-order relation. Similar arrows arise in the neural network specification of section 5.2. However, such higher-order relations are only used to indicate the natural dependence of binary operations upon system topology. In section 4.4, topological constraints are similarly accounted for, through operation instance declarations.

Specification of a Cellular Automaton

CA differ from the previous example in the following respects:

- Any regular topology is possible, not merely fully connected graphs.
- State update rules require knowing the state of all a cell's neighbours.
- Bulk-synchronous iterations are necessary.

The first two can be accounted for by modifying *Topology* and *Constraint*, respectively, as will be shown further below.

Collective synchronization can be specified as a property of the trace of the system. The use of traces here is in a simpler sense than in section 4.6, and is suited to describing the relatively simple dynamics of complex systems. The trace of the execution of the system, or the sequence of the update events, must exhibit the following pattern (stated in the notation given in appendix A):

$$Cycle_1 \hat{} Cycle_2 \hat{} \ldots$$

where $Cycle_i$ is a trace corresponding to the ith iteration of the system. Each $Cycle_i$ is allowed to be any permutation of the N events corresponding to the update of the N cells in the system. To express this in the notation of traces, first let $Event = \{updatecell_1, \ldots, updatecell_N\}$, where the event $updatecell_j$ is the state update operation performed on cell j. Then, for all i:

$$\begin{aligned} Cycle_i \in \{ \ & \forall c \in Event^*, \forall e \in Event \mid \\ & \#c = N \wedge \\ & c \uparrow \{e\} = < e > \bullet \\ & c\} \end{aligned}$$

Synchronized updates have an effect on the meaning of $System'$ in that cells, during $Cycle_i$, make use of their neighbours' states, as they were in $Cycle_{i-1}$, in order to update themselves correctly. Therefore $System'$ does not become $System$ until the completion of the current generation.

In CA, each event in a sub-sequence $Cycle_i$ generates a 'hidden' $System'$ that is not expressed immediately as $System$. The hidden post-states generated by events continue to accumulate until the final event of $Cycle_i$ has occurred, whereupon they are combined by simple set union and expressed as the pre-state of $Cycle_{i+1}$:

$$System = \bigcup_{j=1}^{N} System'_j$$

where N is the number of cells and $System'_j$ the hidden state produced by the update of cell j.

Turning to the state update function, given the degree of the interconnection graph, it will be of the following form:

$$UpdateState : State^{degree+1} \rightarrow State$$

The diagram of a CA system will be identical to that shown in figure 5.3, as again, the $Constraint$ arrow will specify how the topology of the system determines the update operations. The fact that updates are dependent on the state of all neighbours rather than on one 'partner' will be shown in the detailed specification of the $Update$ and $Constraint$ arrows.

$$\begin{aligned} Update: \quad & System^{degree+1} \rightarrow System': \\ & ((ob, st), (obn_1, stn_1), \ldots, (obn_{degree}, stn_{degree})) \mapsto \\ & (ob, UpdateState(st, stn_1, \ldots, stn_{degree})) \end{aligned}$$

$Topology$, in CA, is expected to be the edge-set of a regular graph.

$$\begin{aligned} Constraint: Topology \rightarrow \{ \; & \forall self \in System, \\ & \exists neighb_1, \ldots, neighb_{degree} \in System, \\ & \exists edge_1, \ldots, edge_{degree} \in Topology | \\ & \pi_1 \cdot edge_1 = \ldots = \pi_1 \cdot edge_{degree} \wedge \\ & \pi_1 \cdot edge_1 = \pi_1 \cdot self \wedge \\ & \pi_2 \cdot edge_1 = \pi_1 \cdot neighb_1 \wedge \ldots \wedge \\ & \pi_2 \cdot edge_{degree} = \pi_1 \cdot neighb_{degree} \bullet \\ & (self, neighb_1, \ldots, neighb_{degree}) \mapsto \\ & Update \cdot (self, neighb_1, \ldots, neighb_{degree})\} \end{aligned}$$

An instance of $Constraint$ is:

$$\begin{aligned} (ob_1, ob_2) \mapsto \; & ((self, neigb_1, \ldots, neighb_{degree}) \mapsto \\ & Update(self, neigb_1, \ldots, neighb_{degree})) \end{aligned}$$

(where $\exists e_1, \ldots, e_{degree} \in Topology$ such that for $i = 1 \ldots degree, \pi_1 \cdot e_i = ob_1 = \pi_1 \cdot self \wedge \pi_2 \cdot e_i = \pi_1 \cdot neighb_i$).

Note that ob_2 above is a free variable, meaning that regardless of the object that is the second component of the edge, that edge is mapped onto the state-update operation for the first component object. This representation is redundant in having $degree$ maps into every operation where just one would suffice. An interpretation of this might have been that the same update is to be performed more than once, with the same pre-state being in effect on each occasion. However, the earlier explanation of synchronization assured that each cell will be updated exactly once in a generation, so the interpretation permitting redundant updates is invalid.

To specify particular systems, all that is required is to state the function *UpdateState* and the edge-set *Topology*. At a lower level, *Object* and *State* need to be specified. *Object* may be taken to be a set of unique identifiers, e.g. the integers from 1 to the number of cells in the system. *State* is clearly problem-dependent. Essentially, these four abstractions would constitute a list of formal parameters to a group specification.

5.1.3 Some Examples

N-Body Systems

N-body computations involve all-to-all interactions between objects. An example from physics is the mutual gravitational force between bodies in space; another is the electrical interaction between charged particles. Simulations of such phenomena are known to be iterative, data parallel problems requiring quantities (forces, in the above examples) to be calculated for every pair in the set of objects.

N-body systems may be approximated to CA systems according to the following reasoning (using the gravitational case for illustration): associate each object in the system with a unique cell in a fully connected CA graph; let the state of a cell be the properties, including position, velocity and mass, of its object. Then the behaviour of the system is given by a CA update function that updates the cell's state by determining the resultant of all forces exerted upon its object by all other objects.

The required update operation, given N neighbours, is:

$$\begin{aligned} UpdateState : \; & State^{N+1} \rightarrow State : \\ & (st, stn_1, \ldots, stn_N) \mapsto \\ & f \cdot \sum_{i=1}^{N} (force \cdot (st, stn_i)) \end{aligned}$$

where st is the local state, stn_i is the state of the ith neighbour; $force \cdot (a, b)$ computes the force between two bodies whose states are given by a and b; f recomputes the local state as a function of the sum of forces.

CA, being discrete time systems, approximately model the behaviour of the continuous time physical systems. Obviously, the shorter the time interval simulated by each generation of the CA, the closer the approximation to the continuous time case.

Parallel Genetic Algorithms

Genetic algorithms (GA)[For93] can be represented as a kind of CA by considering the population of individuals to be structured. The structure may be a graph whose nodes represent the individuals and edges define possible pairings of individuals during the crossover phase. Where any two individuals may engage in crossover in a particular generation, the graph is fully connected.

The concept of a *population structure* is slightly different in placing sites, rather than individuals, at the graph's nodes. Each site contains an individual, and a crossover/replacement algorithm is performed at each site as follows: the local string is crossed over with an individual from a neighbouring site; one of the resulting children may then replace the local individual depending on the replacement policy. This is the approach adopted by ASPARAGOS[GS91], a highly parallel GA.

ASPARAGOS can be mapped onto a CA in a straightforward manner. The nodes and neighbourhoods of the population structure correspond directly to the cells and neighbourhoods of a CA. The state of a cell is the individual residing in the corresponding site (together with an extra component to be mentioned shortly). It only remains to state the update function, given a CA graph of degree N:

$$\begin{aligned} UpdateState : \quad & State^{N+1} \rightarrow State : \\ & (ind, nind_1, \ldots, nind_N) \mapsto \\ & replace \cdot (ind, crossover \cdot (ind, select \cdot (nind_1, \ldots, nind_N))) \end{aligned}$$

where *ind* is the local individual and $nind_1 \ldots nind_N$ are individuals in neighbouring sites; *select* picks a neighbour that *crossover* is to operate on with *ind*, to produce a pair of children; *replace* selects either *ind* or one of the children to be the updated local individual.

Generally, selection, crossover and replacement are stochastic operators that require a source of randomness (a random number generator). To avoid conflict with the functional definition of *UpdateState*, and in keeping with the localized nature of CA operations, such a source is taken as an extra component of the state of a site.

5.2 Hopfield Neural Networks

5.2.1 Introduction

The following is restricted to specifying the recall phase of the activity of the Hopfield neural network. The specification primitives introduced here can be applied to other neural network models, notably the popular Perceptron model[Lip87].

An artificial neural network (ANN)[HKP91] is a network of nodes called neurons that are connected together by weighted connections called synapses. Synapses are represented by weighted and directed arcs.

Operationally, synapses (hereafter called connectors) are unidirectional signal conductors whose output is computed by multiplying the input signal (represented, for example, by a real number in the range [0.0 ... 1.0]) by a weight that is specific to the connector. Neurons are switching units whose output is obtained by summing all input signals, subtracting a threshold value (specific to the neuron) from the sum and applying a (typically) nonlinear function (the transition function) to the result. A neuron is said to "fire" when it calculates the result.

It should be evident that connectors, being directed arcs in the network, take input from a single neuron and pass output to a single neuron. Neurons take input from an arbitrary number of connectors (perhaps none) and likewise, may output to an arbitrary number of connectors. The signal output by a neuron is propagated as input to all connectors that it projects. The signal output by a connector is fed into its destination neuron as an input.

The architecture of a Hopfield network is simply a fully connected collection of neurons (excluding any direct connections from a neuron back to itself). For any pair of neurons, the two connectors linking them (one in each direction) should possess the same weight value. Allowed neural output values are $+1$ and -1.

A Hopfield network is initialized by setting the neural outputs to an input pattern, following which it passes through an iterative phase where neural outputs are recomputed based on the current outputs of all other neurons. This phase may proceed indefinitely, although in practice it is terminated upon convergence to a stable state (when outputs remain static between consecutive iterations).

Different versions of the Hopfield network exhibit different dynamical modes that dictate when neurons recompute their state — referred to here as the dynamics of the network. Sequential dynamics have neurons firing one after another. On the other hand, parallel dynamics exist when neurons fire simultaneously. Essentially, sequential dynamics mean that the effect of a neuron firing (i.e. its new output) is propagated to all its neighbours before they in turn may fire, while with parallel dynamics this is not necessarily the case. Networks having parallel dynamics are prone to oscillate between states and thus not converge to a stable state, whereas sequential dynamics are guaranteed to converge.

Parallel dynamics may be divided into two forms: firstly, all neurons fire synchronously with an external clock such that at time t given by the clock, each neuron inputs the state of neighbouring neurons (via connectors) for time $t-1$ and computes its output accordingly; alternatively, each neuron may fire asynchronously, under the assumption that all neurons fire at the same rate, as judged by an external observer (fairness). Simultaneous firings are the cause of undesirable oscillation of states so the latter, asynchronous, variety is more likely to converge to a stable output pattern.

Sequential firing dynamics may be effected by repeatedly recomputing the neurons' state until convergence, either in a fixed order (cf. round-robin polling) or in an arbitrary order.

Figure 5.4 illustrates an example of each form of firing behaviour.

Connectors are assumed to recompute their output immediately upon the firing of their source neurons. Thus a specification of the dynamics of neurons is sufficient to specify the update dynamics of connectors in a Hopfield network.

5.2.2 The Method

The specification technique employed here is an extension of that presented in the specification of cellular automata (section 5.1). The method of identifying and clearly separating the data and operations in a complex system is preserved. Because of their complex temporal behaviour, the specification of dynamic aspects of neural networks require a more sophisticated approach than that used with respect to cellular automata above. The notation for specifying dynamic aspects is largely as in section 4.6.

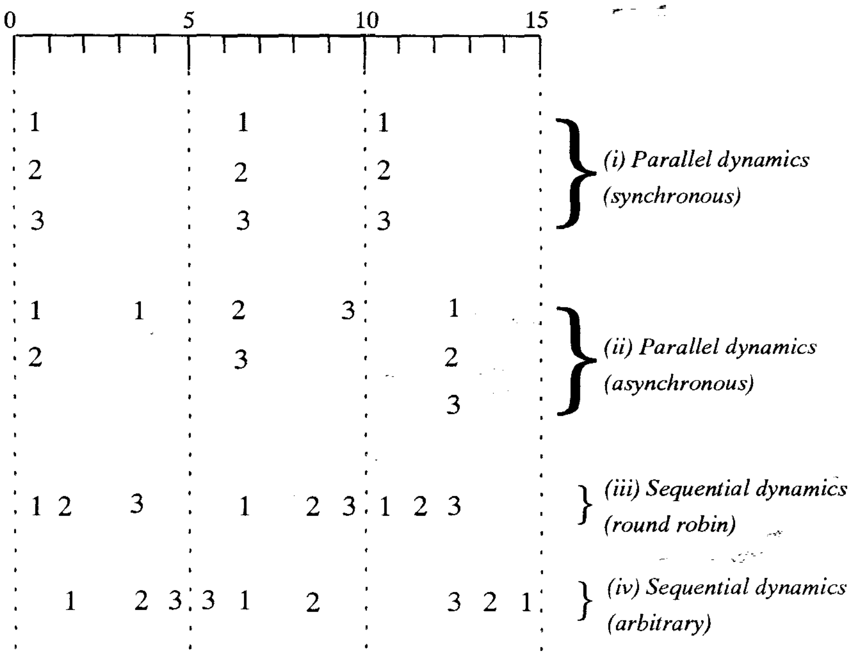
0
5
10
15
(i) Parallel dynamics (synchronous)
(ii) Parallel dynamics (asynchronous)
(iii) Sequential dynamics (round robin)
(iv) Sequential dynamics (arbitrary)

Figure 5.4: Comparative behaviour of four firing dynamics. The line at the top represents time as measured by some clock external to the Hopfield network. The network under consideration is composed of three neurons, labelled 1, 2 and 3. The event of one of them firing is recorded by writing the label of that neuron under the time at which it fired. Assume that the effect of a neuron firing at time t propagates to all other neurons by time $(t + 1)$. *(i)* shows parallel dynamics with synchronous updates; *(ii)* parallel dynamics with asynchronous updates; *(iii)* sequential dynamics with round robin updates, given a polling schedule of 1, 2 and then 3; and *(iv)* sequential dynamics with an arbitrary update order chosen for every cycle of 5 time units. Note that although *(ii)* and *(iv)* seemingly exhibit arbitrary patterns of firing, all neurons fire at the same rate in the given sample of 15 time units (i.e. 1 firing per 5 time units). This may be qualitatively referred to as fairness.

In a manner similar to section 5.1, the present specification will be developed through the following stages:

- Declarations: the sets and functions that characterize the problem. These sets and functions are primitives upon which the specification is based.
- Diagram: a pictorial representation of the state of the system and of the operations that alter state.
- Functions and Constraints: annotations to the diagram to define the operations upon state.

In addition it is necessary to specify the different firing modes mentioned above in the form of information about synchronization:

- Synchronization: specification of the points of synchronization between operations.

5.2.3 Declarations

The following sets are employed in specifying the recall phase of the Hopfield network:

- Sets of objects that constitute the system.
  - *Neuron*: set of neurons in the system. *Connect*: set of connectors in the system.
- Sets of states in which neurons may exist.
  - *NSumIn*: set of values allowed for the sum of inputs.
  - *NThresh*: set of threshold values.
  - *NOut*: set of allowed output values.
- Sets of states in which connectors may exist.
  - *CWeight*: set of weights.
  - *COut*: set of allowed output values.

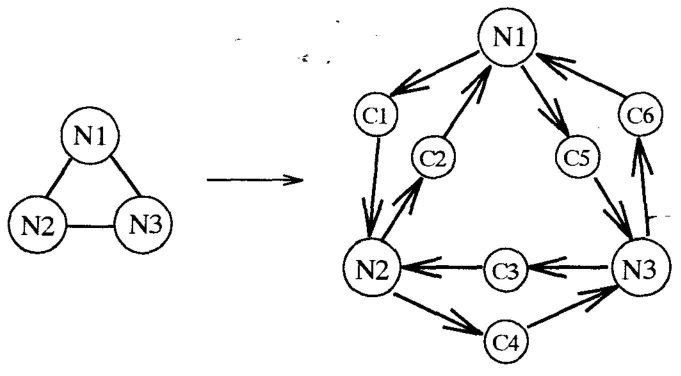

Figure 5.5: The Hopfield network composed of neurons N1, N2 and N3 and interconnected by three bidirectional connectors is represented by *Topology* as shown on the right. Each member of *Connect*, C1 ... C6, is a unidirectional connector. The arrows signify 'data flow' channels between neurons and connectors. *Topology* is the set of such channels, of which there are twelve in this case.

- *Topology*: set of edges corresponding to links between neurons and connectors.

The five state sets above are numerical sets. With the exception of *NOut*, they take values from the set of real numbers; *NOut*, as pointed out earlier, is $\{-1, +1\}$.

The implication of neurons having three state sets is that the instantaneous state of a neuron must be taken from the product $NSumIn \times NThresh \times NOut$. Similarly, the state of a connector must be taken from $CWeight \times COut$.

Topology is the set of connected pairs of objects in the system (see figure 5.5). *Topology* is not identical to the neural network since it has both neurons and connectors for nodes. It is therefore a directed, bipartite graph where connector nodes are linked with one source neural node and one destination neural node; neural nodes are linked with connector nodes as in the neural network.

Since the Hopfield network is fully-connected *Topology* may be given as a subset of $((Neuron \times Connect) \cup (Connect \times Neuron))$ where for every element co of *Connect*, there is exactly one elements of *Topology*, (co, ne_1), having co as the first member of the pair, and exactly one element, (ne_2, co), having co as the second member. This restricts connectors to be one-to-one links. It should be added to the above formulation that $ne_1 \neq ne_2$ so as to eliminate self-loops.

Turning to the declaration of functions that define Hopfield networks, the following need to be specified:

- $InitVal : Neuron \rightarrow NOut$
- $UpdateNOut : NSumIn \times NThresh \rightarrow NOut$
- $UpdateCOut : NOut \times CWeight \rightarrow COut$
- $UpdateSumIn : COut \times NSumIn \rightarrow NSumIn$

InitVal sets the initial output of each neuron to one of a given set of values; typically, such values would correspond to the input pattern that is to be classified by the neural network.

UpdateNOut corresponds to the firing of a neuron. It computes a simple step function that evaluates to -1 if the current sum of inputs is less than the threshold, otherwise to $+1$.

UpdateCOut models the flow of a signal through a connector. It multiplies a signal value by a weight value.

UpdateSumIn is a cumulative summing function. The need for this function arises when one attempts to specify the operation of summing the input signals to a neuron. The number of signal sources feeding into any neuron in a Hopfield network of N neurons is, of course, $N-1$; however for ANN in general, where networks are not required to be fully connected, the number of input connectors need not be fixed for all neurons in the network. In the general case, a single function to evaluate the sum of inputs cannot be found as the arity of such a function can not be fixed. The solution adopted here to support the summing of inputs in ANN generally, is to define an accumulator for each neuron that takes its value from *NSumIn*. The function *UpdateSumIn* is simply the addition function and is used to add incoming signal values, as they become available, to the current value of the accumulator.

5.2.4 Diagram

The diagram (figure 5.6) shows the state of objects in the Hopfield network (neurons and connectors) and how the objects alter each other's state. As in chapter 4 and the above specification of cellular automata, functions are defined to associate each object with its current state; in this case two functions are required, *NSystem* for neurons and *CSystem* for connectors:

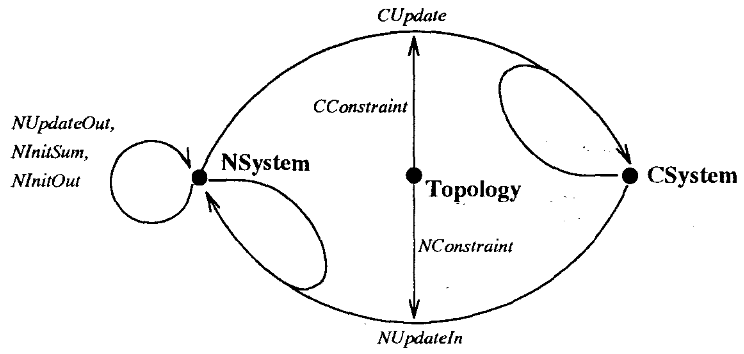

Figure 5.6: A diagram of state changes taking place in a Hopfield network. Nodes are labelled in bold font and arrows in italics. Two nodes represent the state of each component in the system at a particular time: *NSystem* denotes the state of each neuron and *CSystem* the state of each connector. The node *Topology* represents the structural state of the system, the channels of data flow between neurons and connectors. The arrows from node to node denote operations that transform the current state into the next one. The arrows stem from the type of objects whose current state determines the change, and point at the type of component undergoing change. Thus the operation *CUpdate* determines the next state of connectors with respect to the current states of connectors *and* neurons (the arrow has two stems). A full explanation is given in the text.

$$\begin{aligned} NSystem &: Neuron \rightarrow NSumIn \times NThresh \times NOut \\ CSystem &: Connect \rightarrow CWeight \times COut \end{aligned}$$

They have equivalent representations as sets of the form $(object, state)$, where *state* is a triple (for neurons) or a pair (for connectors). The association functions, in set form, are represented as nodes in the diagram. An additional node is the previously declared set *Toplogy*, which is discussed further below with reference to constrained operations.

The role of arrows in the diagram is to indicate the state-changing operations upon objects. The operations that are spoken of here are abstractions built upon the functions declared in section 5.2.3. Primitive functions were declared earlier to model various aspects of Hopfield networks, such as neurons summing inputs and connectors weighting inputs, without relating values (inputs, weights, etc.) to the state of objects (neurons and connectors). At the present level of abstraction, the aim is to specify how the operations, which are higher-order functions, apply primitive functions to objects in order to alter their state.

The diagram is such that arrowheads point to the affected set of objects while the arrow sources are those sets that determine the change. A traversal through the diagram will serve to illustrate. Taking a neural object from *NSystem*, and by following the arrow *NInitOut* (effectively applying *NInitOut* to that neuron), the neurons's state changes so that output is set to an initial value (see section 5.2.5 for formal definitions of *NInitOut* and other operations)[2]. Choosing next to follow the *NInitSum* arrow changes the neuron's state further so that its sum accumulator is initialized. Then following *NUpdateOut* will set the neuron's output according to its current sum of inputs and threshold value[3].

The abovementioned arrows loop back onto the *NSystem* node because corresponding operations determine the next state of neurons based solely on the current state of neurons. The other two arrows in the diagram represent operations that

[2]In terms of operation pre-states and post-states, if *NSystem* is the state of all neurons prior to applying *NInitOut* then the post-state, *NSystem'* differs from *NSystem* in that the output of a single neuron is now initialized (all other neurons are unchanged). It should be remembered that the post-state generated by the operation becomes the pre-state of the following operation.

[3]It will be assumed that the neural thresholds and connector weights have been correctly set beforehand during the learning phase.

require information from different classes of objects in order to effect state changes. *CUpdate* and *NUpdateIn* are operations that update the output of connectors and the input into neurons, respectively. Note that both arrows are pointed to by arrows from *Topology* and so represent *constrained operations.* An operation is said to be constrained by a set when elements of the set determine allowable instances of the operation[4]. An instance of an operation refers to the actual application of an operation to a specific object, so the total number of instances is equal to the size of the operation's domain. It is this number that is reduced by constraining the operation.

In the case of *CUpdate*, the post-state output of a connector is computed by multiplying its weight by the output of a neuron. There are as many instances of this operation as there are neurons in the network, but only one of them is appropriate since it is the output of the source neuron of the connector in question that is of interest. Thus *CUpdate* needs to be constrained by *Topology* to indicate that a neuron may output to a connector only if they are linked. The definition of *CConstraint*, the constraining function, makes this constraint explicit.

Now to resume the traversal of the diagram, pick an object from *NSystem* and follow the *CUpdate* arrow into *CSystem.* There are two novel aspects to this arrow. It has already been remarked how this arrow is constrained so that the operation instances that it represents are valid, implying that, for the neuron that has been picked, the number of allowed *CUpdate* instances is equal to the number of connectors that propagate away from it. However, the other noteworthy point is that the arrow has two sources, *NSystem* and *CSystem*, since an operation instance can not be uniquely defined when the identity of the connector that is to be updated is unknown. The operation requires information about both the neuron and the connector. As a result, *CUpdate* updates the state of a specific connector given its source neuron. So having picked a neuron from *NSystem*, it is necessary to pick a connector from *CSystem* in order to follow the *CUpdate* arrow. Doing so effects a state change on *CSystem* such that the output of the selected connector object is updated while the other objects are unchanged.

Finally, one may pick a connector from *CSystem* and its destination neuron from

[4]In the cellular automaton specification (section 5.1), *Update* was constrained by *Topology* so that cells were updated only with reference to neighbouring cells. Without the constraint, *Update* by itself would allow a cell to update with reference to an arbitrary collection of cells.

NSystem and follow the *NUpdateIn* arrow. The effect of this operation is to transform the state of the selected neuron such that its input sum accumulator is incremented by the output of the selected connector. *NupdateIn* is a constrained operation (by the arrow *NConstraint*) that is also determined by both neural and connector objects.

In summary, nodes in the diagram represent a set of objects of the same class having state, and arrows into nodes represent operations. Nodes at the origin of an arrow are taken as the pre-state of the corresponding operation while the node at an arrowhead is transformed by the operation into a post-state. The post-state of an operation becomes part of the pre-state of its subsequent operation. An instance of an operation is the operation together with arguments (objects) necessary to generate post-state. Constraints are arrows into other arrows that define legal instances of the operation corresponding to the destination arrow.

5.2.5 Functions and Constraints

In this section, the operations and constraining functions shown in the diagram are defined. All operations update the state of neurons by applying the primitive functions to current state values. Firstly, the three unconstrained operations are given as:

$$\begin{aligned}
NInitOut: \quad & NSystem \rightarrow NSystem':\\
& (ne \mapsto (sum, \theta, out)) \mapsto (ne \mapsto (sum, \theta, InitVal \cdot out))\\
NUpdateOut: \quad & NSystem \rightarrow NSystem':\\
& (ne \mapsto (sum, \theta, out)) \mapsto\\
& (ne \mapsto (sum, \theta, UpdateNOut \cdot (sum, \theta)))\\
NInitSum: \quad & NSystem \rightarrow NSystem':\\
& (ne \mapsto (sum, \theta, out)) \mapsto\\
& (ne \mapsto (0, \theta, out))
\end{aligned}$$

where sum, θ and *out* denote the current sum of inputs, threshold and output, respectively, of the neuron *ne*.

Constrained operations need to be defined in conjunction with their respective constraining functions. The following functions partly define the constrained operations:

$$\begin{aligned}
CUpdate:\ & NSystem \times CSystem \rightarrow CSystem' : \\
& ((ne \mapsto (sum, \theta, out)), (co \mapsto (w, cout))) \mapsto \\
& (co \mapsto (w, UpdateCOut \cdot (out, w)))
\end{aligned}$$

$$\begin{aligned}
NUpdateIn:\ & CSystem \times NSystem \rightarrow NSystem' : \\
& ((co \mapsto (w, cout)), (ne \mapsto (sum, \theta, out))) \mapsto \\
& (ne \mapsto (UpdateSumIn \cdot (cout, sum), \theta, out))
\end{aligned}$$

where w and $cout$ denote the current weight and output, respectively, of the connector co.

Clearly, these definitions allow arbitrary neuron and connector pairs to update each other's state, so the following constraining functions are required to complete $CUpdate$ and $NUpdateIn$:

$$\begin{aligned}
CConstraint:\ & \{\forall edge \in Topology | \\
& \pi_1 \cdot edge \in Neuron \wedge \\
& \pi_2 \cdot edge \in Connect \bullet \\
& edge\} \rightarrow \{\forall self \in CSystem, \\
& \exists edge \in Topology, source \in NSystem | \\
& \pi_1 \cdot edge = \pi_1 \cdot source \wedge \\
& \pi_2 \cdot edge = \pi_1 \cdot self \bullet \\
& (source, self) \mapsto CUpdate \cdot (source, self)\}
\end{aligned}$$

The domain of the function is the set of all links in $Topology$ that are directed from a neuron to a connector. The range is the set of operation instances of $CUpdate$ that update a connector with reference to its source neuron. The appropriate mapping rule is:

$$\begin{aligned} CConstraint: \ & (ne, co) \mapsto ((source, self) \mapsto \\ & CUpdate \cdot (source, self)) \end{aligned}$$

where there exists an element *edge* of *Topology* such that $\pi_1 \cdot edge$ is *ne*, an element of *Neuron*, and $\pi_2 \cdot edge$ is *co*, an element of *Connect*. Furthermore, *self* is a connector object with $\pi_1 \cdot self = co$ and *source* is a neural object with $\pi_1 \cdot source = ne$. The rule is to be interpreted as saying that an instance of *CUpdate* is allowable if and only if the corresponding neuron and connector pair are arranged in the neural network so that signals flow directly from the neuron to the connector.

The constraint function for *NUpdateIn* is very similar but in a reversed sense: the domain of the function is the set of all links in *Topology* that are directed from a connector to a neuron and its range is the set of operation instances of *NUpdateIn* that update a neuron with reference to a source connector:

$$\begin{aligned} NConstraint: \ & \{\forall edge \in Topology | \\ & \pi_1 \cdot edge \in Connect \wedge \\ & \pi_2 \cdot edge \in Neuron \bullet \\ & edge\} \to \{\forall self \in NSystem, \\ & \exists edge \in Topology, source \in CSystem | \\ & \pi_1 \cdot edge = \pi_1 \cdot source \wedge \\ & \pi_2 \cdot edge = \pi_1 \cdot self \bullet \\ & (source, self) \mapsto NUpdateIn \cdot (source, self)\} \end{aligned}$$

Similarly, mapping rule is:

$$\begin{aligned} NConstraint: \ & (co, ne) \mapsto ((source, self) \mapsto \\ & NUpdateIn \cdot (source, self)) \end{aligned}$$

The meanings of *source* and *self* are inverted from the rule for *CConstraint* (*source* is a connector object and *self* a neural object) and there is assumed to be an edge in *Topology* linking the connector to the neuron.

5.2.6 Synchronization

Discussion

In section 5.2.5, the operations that take place in a Hopfield network during recall were specified in terms of the state changes they cause. They have, however, been specified independently of each other so that there has thus far been no formal indication of the manner in which operations are coordinated to produce a correctly functioning system. In this section, matters of coordination between objects will be specified.

Given a set of operations, their coordinated action may be specified by an algorithm, a sequence in which those operations are to be carried out upon an initial system state in order to obtain the required behaviour (which is usually either the arrival at a desired final state or the evolution through a desired series of states). In systems composed of replicated objects, such as ANN, an object oriented specification of coordination may be applicable.

An object oriented model considers the state of the system to be distributed among its component objects. Objects may access external state by passing messages across object boundaries. A connector, for example, needs to retrieve the state (output value) of its source neuron in order to update its own state. It may do so by sending a message to the neuron, which in response returns the value. The unsuitability of this model lies in that objects are used as passive managers of state information, relying on some external source for direction, such as a special master object whose purpose is to instruct each object in turn to update its local state.

A more suitable model is a data-parallel one wherein objects are viewed as active units, or processes, that manipulate local state by following a local algorithm. Parallel activity of processes is the norm, whereas objects are usually associated with a sequential mode in which only one message appearing *in the entire system* can be responded to at a particular time. Parallelism, specifically data-parallelism, is a direct model of biological neural networks from a computational viewpoint. As such, it is a convenient model of both sophisticated (biologically realistic) and simple ANN where parallelism (as well as sequence) may be directly expressed. In sequential models, parallel activity can only be indirectly expressed by introducing the concept of interleaving and by simulating nondeterminism.

The objects in this specification, the elements of sets *Neuron* and *Connect*, are seen as processes in a data-parallel computation. The operations, on the other hand, are specified according to a broader model since they may refer to state across object boundaries, e.g. in section 5.2.5 the operation *CUpdate* examines the current state of a neuron and alters the state of a connector accordingly. Therefore at the level of objects, implicit in the performance of certain operations are lower level message passing operations.

Interprocess communication, however, is not treated in the specification because it is desired that all relevant information be derivable from higher level specifications. According to the CSP model, relevant information for defining a system consists of, firstly, its processes, and secondly, the unidirectional point-to-point communication channels between processes. The following observations are made with regard to the two requirements:

- The processes correspond to the objects given by the sets *Neuron* and *Connect*. It is argued below that the statements of a high-level system specification may be translated into a detailed process specification.

- If all operation instances (i.e. the operations together with the identity of the participating objects) are mentioned in the specification, then the required channels of communication can be found by associating a separate channel with each interacting pair of objects (or two channels, if the interaction is in both directions). In the case of Hopfield neural networks all operation instances are given in the set *Topology*: the constraining functions *CConstraint* and *NConstraint* dictate that instances of the operations *CUpdate* and *NUpdateIn* (the only ones involving inter-object communication) must correspond to edges in *Topology* in such a way that the connection pattern of the neural network is respected. There is an isomorphism between *Topology* and the required set of channels, mapping $(neuron, connector)$ and $(connector, neuron)$ pairs to neuron-to-connector and connector-to-neuron channels, respectively.

CSP uses the *trace* abstraction to specify processes. The *event* is the unit of system behaviour; a trace of a system's behaviour is a finite sequence that records the order in which events appear to an external observer. A process is said to satisfy a trace (its specification) when in all cases it engages in events in the order given by that trace. Both the duration of events and the possibility of two events occurring simultaneously are ignored by this abstraction.

High-level specification statements will be used to express the order of operations in the Hopfield network. What follows is an explanation of the meaning of these statement and a justification of their ability to translate into specifications of processes (traces). That such a translation is possible is a prerequisite for considering the objects presented here to be processes. It would enable specifications such as the present one to generate specifications of parallel programs (in CSP notation) automatically.

The behaviour of the Hopfield network will be considered at the level of operations. Operations will be considered as the events constituting the system, in the manner described below. More detailed considerations that are not included here are those at the level of parallel programming such as the nature and medium of communication and the implementation of the given functions of section 5.2.3.

The strategy adopted here is to specify explicitly the order in which operations are performed in the Hopfield network. To assist in this, the concept of the operation instance is extended into the time domain. Recall that an operation is defined by a function and that an operation instance is a function provided with all required arguments (which are objects). It is now necessary to index operation instances by a counter so that the instance, when first performed, is referred to by the index 1, and on the second occasion by 2, etc. Each indexed operation instance will henceforth be equated with an event occurring in the system.

Given that the *alphabet* (events of interest) of the Hopfield network consists of every occurrence of the operation instances defined in this specification, a complete account of the behaviour of the network may be given by stating its trace. Such an account would be unsatisfactory for two reasons: principally, it would be overconstrained because a trace defines a quasi-ordering of its events (every event may be strictly ordered with respect to every other), whereas in a data-parallel computation the ordering of any two events is not necessarily significant (e.g. it may be decided that two neurons may fire in an arbitrary order, and therefore that it is not important that either one of them fires before the other); another reason is that it would be tedious to specify the trace, a task requiring each event to be ordered with respect to all other events. The objection that a trace can not express arbitrary ordering suggests that the system may be specified by a *set of traces* where each element specifies a possible 'course of events' from the beginning to the end of the system's lifetime.

Rather than rendering specification an even more tedious task, specifying a set of traces should be much simpler than specifying a single trace. The approach taken is to assume initially that all events may occur in an arbitrary order. Thus the set of required traces is initially taken to be the set of traces that contain each event exactly once. The specification is then responsible for identifying those event-pairs that should be ordered. For each pair of events so ordered, traces that contradict the ordering are considered to be removed from the set of required traces. Consequently, a specification asserting that certain pairs of events are ordered defines a set of traces respecting those orderings.

The set of allowed traces so derived defines a specification of the of the system (viewed as a process). Note that a process is usually said to satisfy a specified trace if in all instances its behaviour adheres to the given trace. A process may be said to satisfy a set of traces if in all cases, it satisfies an element of the set.

The concept of arbitrary ordering allows concurrency to be expressed with trace sets, in the sense that events unordered with respect to each other might conceivably occur simultaneously. This equation of nondeterministic order with concurrency, however, does not hold in cases where the unordered events are actions that alter the *same data* (the mutual exclusion problem). Take for example the *NUpdateIn* operations that consecutively adds input values to an 'accumulator'. They need not be ordered because the eventual result of adding the inputs is independent of the order in which the inputs were taken. However, concurrency is certainly not implied by this form of nondeterminism as simultaneous updates would result in unpredictable values in the accumulator. The approach taken here is to assume that nondeterministic order does indeed permit concurrency and to enforce mutual exclusion, where required, by explicitly ordering the conflicting events (see the treatment of *NUpdateIn* below).

A systems specification given by a global trace set is easy to decompose into subspecifications of constituent objects. Given the set of allowed traces for the entire system, the objects making up the system are considered to be subprocesses whose specifications are derived from the system specification. For a particular object, its process is the system specification (set of traces) altered so that events not involving the object are removed from each trace.

Against this background it only remains to specify the necessary event-orderings for Hopfield networks. This takes the form of a list of statements asserting for each event the set of preceding events [5]. Though not necessary, it is desirable to include just those events that immediately precede the event in question. To include indirect precedents would not convey further information. The set of precedents may not include the event itself (self-reference) nor may the specification be contradictory (e.g. to state that an event x precedes an event y and elsewhere that y precedes x).

The system employed to label events is:

$operation_{(object_1, object_2, \ldots, object_M, timeIndex)}$

where *operation* is the name of the operation, $object_1, \ldots, object_M$ are the M objects participating in the operation and *timeIndex* is the occurrence-count of the particular operation instance, including the present. So the above event represents the *timeIndex*-th occurrence of the operation instance represented by the function:

$operation \cdot (object_1, object_2, \ldots, object_M)$

Parallel Dynamics

Initially, the case of synchronous parallel dynamics will be considered. Let N be a finite positive integer denoting the number of times each neuron should fire, and *systemTrace* the trace of a Hopfield network containing events labelled according to the abovementioned scheme.

Let *systemTrace* be constructed as follows (appendix A gives the meaning of the trace operators used):

$$\begin{aligned} systemTrace = \; & u\hat{\ } ev \\ & (\#ev = 1) \end{aligned}$$

systemTrace contains a record of the behaviour of the network up to, and including, the occurrence of some event of interest. The singleton trace *ev* denotes the event in

[5] An alternative, but equivalent, method is to depict causal dependencies between events in the form of a directed acyclic graph where nodes represent events and directed arcs represent binary precedence relations between events. These are known as *event structures*.

question and u the trace of its preceding events. Then the following statements define the necessary event precedences for parallel dynamics by giving the constraints upon u for all instances of ev. For an illustration of valid event precedences that satisfy all of the following statements, it may be helpful to consult figure 5.7.

1. $ev = \langle NInitOut_{(n,1)} \rangle \implies \langle \rangle$ **in** u

   $(\forall n \in Neuron)$

   The *NInitOut* operation takes places exactly once for each neuron, but there are no events that must necessarily precede it. The consequent of this statement is a tautology, indicating that the event may occur at any time (though when read in conjunction with subsequent statements, this freedom is restricted appropriately).

   An instance of *NInitOut* is defined for every element of *Neuron*, but there is only one event associated with each instance (i.e. the one having time index 1). This is to state that each neuron's output must be initialized once.

2. $ev = \langle CUpdate_{(c,n,1)} \rangle \implies \langle NInitOut_{(n,1)} \rangle$ **in** u

   $ev = \langle CUpdate_{(c,n,i)} \rangle \implies \langle NUpdateOut_{(n,i-1)} \rangle$ **in** u

   $(\forall c \in Connect, \forall n \in Neuron | (n,c) \in Topology)$

   $(1 < i \leq N)$

   CUpdate may be performed on a connector for the first time once the source neuron has been initialized. On following occasions, the source neuron must have updated its output a sufficient number of times to keep it 'in phase' with the connector (i.e. the connector updates for the i-th time after the source neuron has updated its output for the $(i-1)$-th time). The apparent lag between *NUpdateOut* and *CUpdate*, with respect to their time indices is to correct for the initial neural output update effected by *NInitOut*.

   CUpdate events are defined for all connectors together with their source neurons, and for all time indices from 1 up to N — the required number of neuron firings.

3. $ev = NInitSum_{(n,1)} \implies \langle \rangle$ **in** u

$ev = NInitSum_{(n,i)} \implies \langle NUpdateOut_{(n,i-1)} \rangle$ **in** u

$(\forall n \in Neuron, 1 < i \leq N)$

The sum of inputs of a neuron must be reset by *NInitSum* firstly when the neuron is initialized and then periodically upon the neural output being recomputed:

Each neuron's sum of inputs is reset N times, prior to each firing, so *NInitSum* is defined for all elements of *Neuron* and for time indices from 1 up to N.

4. *NUpdateIn* increments the current sum of inputs of a neuron by the output of an incoming connector. Thus for each input connector into a neuron, there is an associated instance of this operation. The following is an initial, but faulty, attempt to specify this operation:

   $ev = \langle NUpdateIn_{(n,c,i)} \rangle \implies \langle CUpdate_{(c,n_1,i)} \rangle$ **in** $u \wedge$

   $\langle NInitSum_{(n,i)} \rangle$ **in** u

   $(\forall n, n_1 \in Neuron, \forall c \in Connect | (c,n), (n_1,c) \in Topology)$

   $(1 \leq i \leq N)$

   Given a neuron n and one of its source connectors c this states that between successive updates of the sum of inputs of n by c, c is updated and n's sum of inputs is reset. Note that n_1 represents the source neuron of c.

   The difficulty arises, as pointed out earlier, in allowing unordered events to be regarded as concurrent actions. The statement above (considered along with the other conditions upon *systemTrace*) is seen not to impose any order between *NUpdateIn* events that update the sum of inputs of a neuron, say n. Thus to be explicit that an order does exist, instances of this operation should be forced to occur sequentially. To do this, a *sequence*[6] σ of all *NUpdateIn* events is introduced into the specification. The choice of a sequence is certainly

[6]Sequences will be taken to be linear structures containing a fixed, finite number of unique elements. Any sequence of N unique events, labelled $event_1, event_2, \ldots, event_N$, can be represented by the bijection:

$\sigma : \{1, 2, \ldots, N\} \rightarrow \{event_1, event_2, \ldots, event_N\}$

σ^{-1} is simply the inverse mapping from events to indices. The positional '$\langle$' and '$\rangle$' notation of traces applies to sequences as well.

an arbitrary one because all sequences containing *NUpdateIn* events for a neuron n are equally valid (because *NUpdateIn* performs the arithmetic summing function which is commutative). Therefore, it is convenient to define σ_n here as an arbitrary but fixed sequence for each neuron n, rather than introducing it in section 5.2.3 along with the sets and functions that uniquely characterize individual neural networks.

An arbitrary sequence of given events will be expressed by defining its range set. Define an arbitrary sequence of all *NUpdateIn* events upon neuron n with time index $1 \leq i \leq N$ as[7]:

$ran \cdot \sigma_{n,i} = \{\forall c \in Connect | (c,n) \in Topology \bullet NUpdateIn_{(n,c,i)}\})$

Then the revised synchronization specification for this operation is:

$ev = \langle NUpdateIn_{(n,c,i)} \rangle = \langle \sigma_{n,i} \cdot 1 \rangle \Longrightarrow$

$\quad \langle CUpdate_{(c,n_1,i)} \rangle$ **in** $u \wedge \langle NInitSum_{(n,i)} \rangle$ **in** u

$ev = \langle NUpdateIn_{(n,c,i)} \neq \langle \sigma_{n,i} \cdot 1 \rangle \Longrightarrow$

$\quad \langle \sigma_{n,i} \cdot (\sigma_{n,i}^{-1} \cdot NUpdateIn_{(n,c,i)} - 1) \rangle$ **in** $u \wedge$

$\quad \langle CUpdate_{(c,n_1,i)} \rangle$ **in** $u \wedge \langle NInitSum_{(n,i)} \rangle$ **in** u

$(\forall n, n_1 \in Neuron, \forall c \in Connect | (c,n), (n_1,c) \in Topology)$

$(1 \leq i \leq N)$

The first clause accounts for the first event in the sequence, which has to be preceded by a *NInitSum* event (to reset the accumulator). The second constrains subsequent events to respect the ordering given in the sequence.

5. $ev = \langle NUpdateOut_{(n,i)} \rangle \Longrightarrow$

   $(\forall c \in Connect, (c,n) \in Topology | \langle NUpdateIn_{(n,c,i)} \rangle$ **in** $u)$

   $(\forall n \in Neuron, 1 \leq i \leq N)$

   NUpdateOut events signify the firing of neurons. Their precedents are those *NUpdateIn* events that accumulate the current sum of inputs.

   NUpdateOut events are defined for all neurons in the network.

[7]Taken from the Z notation is the function *ran* which maps a function to its range set.

(i)

$$\begin{tikzcd} NIO_{(n,\ 1)} \arrow[r] & CU_{(c_1,\ n,\ i)} \end{tikzcd}$$

(ii)

$$\begin{tikzcd} & CU_{(c_1,\ n,\ i)} \\ NUO_{(n,\ i\text{-}1)} \arrow[ur] \arrow[dr] & \\ & NIS_{(n,\ i)} \end{tikzcd}$$

(iii)

$$\begin{tikzcd} CU_{(c2,\ n,\ i)} \arrow[dr] & & \\ & NUI_{(n,\ c_2,\ i)} \arrow[r] & NUO_{(n,\ i)} \\ NIS_{(n,\ i)} \arrow[ur] & & \end{tikzcd}$$

Figure 5.7: An alternative presentation of event precedence rules for the asynchronous form of parallel dynamics. Operation names have been abbreviated: *NInitOut* to NIO, *CUpdate* to CU, *NInitSum* to NIS, *NUpdateIn* to NUI and *NUpdateOut* to NUO. $n \in Neuron$ and $c_1, c_2 \in Connect$ are such that c_2 outputs to n and n outputs to c_1, i.e. $(c_2, n), (n, c_1) \in Topology$.

For convenience, the conjunction of all these constraints upon *systemTrace* may be visualized as directed acyclic graphs with events as nodes and precedence relations between event pairs as directed arcs. For example, see figure 5.7.

Such graphs resemble *event structures*[Win89], although here each node represents any member of a class of events rather than a unique event. In the style of event structures, each arrow in figure 5.7 points from an 'enabling' event to another event, where it is known that the latter may only occur after the former. Events at the root of a graph are initially enabled. An event having more than one enabling event can not occur until all enabling events have occurred. The graph labelled *(ii)* refers to any time index i greater than 1, while in *(iii)* i is any time index.

Note that the three graphs are meant to be consistent with each other, so for example in graph *(iii)* substituting 1 for i does not imply that $CUpdate_{(c_2,n,1)}$ is initially enabled, since that is contradictory to graph *(i)*.

The specification of synchronization constraints in this manner facilitates the construction of proofs, especially negative proofs, of temporal properties of the system.

In order to prove that a certain sequence of events is not possible, it is necessary to pick one of the latter events of the sequence, considered as a singleton trace and substituted for *ev* in the above formula for *systemTrace*, and deduce that one of the hypothesized predecessors of the event cannot occur in u.

The above rules state a number of important temporal characteristics of Hopfield nets, for example, that initialization events precede all other events; that periodically the sum of inputs is reset between successive output computations in all neurons; that in neurons the sum of inputs is computed from the current outputs of source connectors; and that connectors input the current output of their source neurons. These last two observations imply that any object, be it neuron or connector, will compute its next state by examining the current state of its source object(s). Neighbouring objects synchronize when one of them communicates its state (output value) to the other.

Apart from this form of synchronous communication between neighbours there has been no other synchronization constraint imposed thus far.

With the information already given concerning *systemTrace*, it is possible to prove that certain important properties. For example, the synchronous version of parallel dynamics can be said to be globally synchronized in that neurons fire once, then pause until all other neurons have fired before firing again. The property of neighbourhood synchronization just mentioned implies that the networks characterized by *systemTrace* are indeed globally synchronized, since Hopfield networks are fully connected. Formally, it is necessary to prove that, given $\{n_1, n_2\} \subseteq Neuron$ and $1 \leq i \leq N$, the following is not possible:

$$systemTrace \upharpoonright \{NUpdateOut_{(n_2,i+1)}, NUpdateOut_{(n_1,i)}\} = \langle NUpdateOut_{(n_2,i+1)}, NUpdateOut_{(n_1,i)}\rangle$$

which is equivalent to proving that n_2 may not fire for the $(i+1)$-th time until *after* n_1 has done so for the i-th time. Let $c \in Connect$ be the connector leading from n_1 to n_2 (i.e. $\{(n_1, c), (c, n_2)\} \subset Topology$). Then by statement 5 above, $NUpdateOut_{(n_2,i+1)}$ is preceded by $NUpdateIn_{(n_2,c,i+1)}$. This is in turn preceded by $CUpdate_{(c,i+1)}$ (statement 4). Finally, the latter event is preceded by $NUpdateOut_{(n_1,1)}$ (statement 2), resulting in the required contradiction that $NUpdateOut_{(n_1,1)}$ precedes $NUpdateOut_{(n_2,i+1)}$.

Another useful conclusion (proof not given) is that the sum of inputs is never reset in such a way that the computation of the current sum is interfered with. Given a neuron n, two successive *NInitSum* operations upon n will always be separated in *systemTrace* by the *NUpdateIn* operations required to compute the sum of inputs into n.

Finally, it can be shown that for synchronous parallel dynamics, the above rules constitute a sufficient definition. All *NUpdateOut* (i.e. firing) events with the same time index are permitted to occur in any sequence (i.e. they may proceed in parallel), and since *NUpdateOut* events with time index i compute neural outputs from outputs generated by *NUpdateOut* events with time index $(i-1)$ (i.e. firings do not interfere with each other). The latter may be demonstrated by choosing a neuron n_1 and one of its source connectors c, and the source of c, n_2; then $NUpdateOut_{(n_1,i)}$ for some $1 < i \leq N$ is preceded by $NUpdateIn_{(n_1,c,i)}$ (statement 5), which is preceded by $CUpdate_{(c,n_2,i)}$ (statement 4), which in turn is preceded by $NUpdateOut_{(n_2,i-1)}$ (statement 2). Thus n computes its activation at time i with reference to the output of other neurons at time $i-1$. There is no interference because a neuron firing at time i does not affect other neurons firing concurrently; it affects them at time $(i+1)$, as expected.

The asynchronous form of parallel dynamics requires a less demanding precedence rule for *NUpdateIn* events. Statement 4 above induces a synchronous form of communication between neurons where connectors act as synchronous message-passing channels. Where asynchronous firings are permitted, connectors act as buffers.

The required modification is essentially to reduce the dependence of *NUpdateIn* upon *CUpdate*. Again, given σ as in the synchronous case, statement 4 becomes:

$$
\begin{aligned}
&ev = \langle NUpdateIn_{(n,c,1)}\rangle = \langle \sigma_{n,1} \cdot 1\rangle \Longrightarrow \\
&\quad \langle CUpdate_{(c,n_1,1)}\rangle \textbf{ in } u \wedge \langle NInitSum_{(n,1)}\rangle \textbf{ in } u \\
&ev = \langle NUpdateIn_{(n,c,i)}\rangle = \langle \sigma_{n,i} \cdot 1\rangle \Longrightarrow \\
&\quad \langle NInitSum_{(n,i)}\rangle \textbf{ in } u \\
&ev = \langle NUpdateIn_{(n,c,i)} \neq \langle \sigma_{n,i} \cdot 1\rangle \Longrightarrow \\
&\quad \langle \sigma_{n,i} \cdot (\sigma_{n,i}^{-1} \cdot NUpdateIn_{(n,c,i)} - 1)\rangle \textbf{ in } u \wedge \\
&\quad \langle NInitSum_{(n,i)}\rangle \textbf{ in } u \\
&(\forall n, n_1 \in Neuron, \forall c \in Connect | (c,n), (n_1,c) \in Topology) \\
&(1 < i \leq N)
\end{aligned}
$$

Successive *NUpdateIn* events may take place with the interposition of an arbitrary number of *CUpdate* events (or none at all) rather than exactly one *CUpdate* event as was previously the case. The first occurrence of *NUpdateIn*, however, must be preceded by a *CUpdate* event, since the initial output of connectors is undefined.

Introducing asynchronicity into the system permits events to occur in less orderly patterns than in the synchronous case. For example, neurons may fire at different rates. Extreme cases where firing rates differ considerably are undesirable. These are matters of *fairness* and are not treated here in detail.

Sequential Dynamics

Sequential dynamics, like parallel dynamics, demand that firings take place in cycles where neurons fire once and wait until all other neurons have done so before firing again. They differ in that it is now necessary for neurons to 'interfere' (as the term is used above) with the 'concurrent' (i.e. occurring in the same cycle), firings of other neurons. A firing sequence is given that includes all neurons. The sequence may be fixed for the entire lifetime of the Hopfield network ('round-robin', see section 5.2.1) or vary from cycle to cycle ('arbitrary').

The two kinds of sequential dynamics differ only in the order of firing per cycle. Therefore, it is useful to formalize this distinction before giving the synchronizations for sequential dynamics. Given a network that is to undergo N firings by each of its neurons, or N cycles, the firing sequence for the entire lifetime of the network may be given by a *sequence of N subsequences*, that each give the order of firing for a particular cycle. Let the sequence be Σ, defined as:

$\Sigma = \sigma_1 \hat{} \sigma_2 \hat{} \ldots \hat{} \sigma_N$

where in the arbitrary sequential dynamics case, each σ_i is an arbitrary sequence of unique firing events in cycle number i. (Note that there is no relation between these σ and the σ used in statement 4 above to enforce an order among *NUpdateIn* events). Formally, for all $1 \leq i \leq N$:

$ran \cdot \sigma_i = \{\forall n \in Neuron \bullet NUpdateOut_{(n,i)}\}$

In the round robin case, all σ_i are identical, thus in addition to the above statement about the ranges of all σ_i, it is necessary to state that the order in which neurons fire in the first cycle is preserved in the following $N-1$ cycles. Formally, for all $n \in Neuron$ and $1 < i \leq N$:

$\sigma_1^{-1} \cdot NUpdateOut_{(n,1)} = \sigma_i^{-1} \cdot NUpdateOut_{(n,i)}$

The synchronizations for sequential dynamics will be similar to those of synchronous parallel dynamics, so only those previously given constraints that are in conflict with sequential dynamics will be pointed out and altered here. Firstly, the order of firing (in both forms of sequential dynamics), given as Σ, needs to be incorporated into the rule for *NUpdateOut* events. Thus statement 5 becomes:

$$\begin{array}{l}
ev = \langle NUpdateOut_{(n,1)}\rangle \wedge \Sigma^{-1}\cdot NUpdateOut_{(n,1)} = 1 \Longrightarrow \\
\quad (\forall c \in Connect, (c,n) \in Topology | \langle NUpdateIn_{(n,c,1)}\rangle \textbf{ in } u) \\
ev = \langle NUpdateOut_{(n,i)}\rangle \wedge \Sigma^{-1}\cdot NUpdateOut_{(n,i)} \neq 1 \Longrightarrow \\
\quad (\forall c \in Connect, (c,n) \in Topology | \langle NUpdateIn_{(n,c,i)}\rangle \textbf{ in } u) \wedge \\
\quad \langle \Sigma \cdot ((\Sigma^{-1}\cdot NUpdateOut_{(n,i)}) - 1)\rangle \textbf{ in } u \\
(\forall n \in Neuron, 1 \leq i \leq N)
\end{array}$$

The first clause concerns the very first firing to take place in the network, in which case the rule is exactly as in the original form of statement 5. Otherwise (second clause), that is for all firings excepting the first, it is additionally required that the immediately preceding firing event, as given by Σ, has already occurred.

These alterations will ensure that all firings will be strictly ordered with respect to each other. However, this is not sufficient for sequential dynamics. A form of interference between firings taking place in the same cycle is required. According to a proof given above for synchronous parallel dynamics, there cannot be interference in the sense that when a neuron fires in the present cycle, its output will only be input by other neurons and finally affect *their* outputs in the following cycle. With sequential dynamics, the output of the neuron firing at the present moment affects the output of the very next neuron to fire (usually both firings will be in the same cycle).

The chain of reasoning followed by that proof relied on connectors to buffer the outputs of their source neurons until the next cycle. This is evident from figure 5.7 where the upper branches of graphs *(ii)* and *(iii)*, taken together, form a path from firing event $NUpdateOut_{(n,i-1)}$ to firing event $NUpdateOut_{(n,i)}$ (the n's need not be identical) via a connector update event $CUpdate_{(c,n,i)}$. The significant observation is that the connector update event mediates the flow of information from cycle $i-1$ to cycle i. This is characteristic of parallel dynamics. There can be no connector update event to convey information between two firing events that occur in the same cycle.

Sequential dynamics, on the other hand, require connectors to convey information instantaneously between neuron firings. This is indicated by replacing the second clause of statement 2 by two new clauses. Given M, the number of neurons in the network and all σ_i as described earlier in this discussion of sequential dynamics, statement 2 becomes:

$$\begin{array}{l}
ev = \langle CUpdate_{(c,n,1)} \rangle \Longrightarrow \\
\quad \langle NInitOut_{(n,1)} \rangle \textbf{ in } u \\
ev = \langle CUpdate_{(c,n,i)} \rangle \wedge NUpdateOut_{(n,i-1)} = \sigma_{i-1} \cdot M \Longrightarrow \\
\quad \langle NUpdateOut_{(n,i-1)} \rangle \textbf{ in } u \\
ev = \langle CUpdate_{(c,n,i)} \rangle \wedge NUpdateOut_{(n,i-1)} \neq \sigma_{i-1} \cdot M \Longrightarrow \\
\quad \langle NUpdateOut_{(n,i)} \rangle \textbf{ in } u \\
(\forall c \in Connect, \forall n \in Neuron | (n, c) \in Topology) \\
(1 < i \leq N)
\end{array}$$

The second clause states that the updating of connectors that emanate from the neuron firing last in the previous cycle should be preceded by that firing. This case is identical to synchronous parallel dynamics as the connector conveys information from one cycle to the next. This is, however a boundary condition; the more general case is given by the third clause which states that a connector should update immediately (i.e. during the same cycle as) its source neuron fires.

Chapter 6

Translating Specifications into occam Code

Context

We have established a specification language for describing complex systems. The language has been shown to be suited to the task of specifying systems of interest, individually and in groups. In the previous chapter its application to the general problem of classifying complex systems was demonstrated.

In this chapter we make use of one of the principal design aims of the specification language, its implicit expressiveness of programmatic information in a generic fashion, in order to automate the programming of parallel simulations of complex systems.

The grouping of complex systems according to their similarities, which is enabled by the specification approach and notation, is made use of here to encapsulate the program induction procedures within an algorithmic skeleton environment.

6.1 Introduction

The motive here is to reduce the human programmer's effort of simulating complex systems. Special concern is directed at execution of simulations on parallel computers.

The effort referred to is that of manually designing and programming simulations, such that the resulting programs are *correct*, meaning that they faithfully reproduce the behaviour of the simulated systems, and *reusable*, in order that a simulation of a certain system, once written, may be used as a base from which to derive simulations of related systems, with considerably reduced effort when compared to programming 'from scratch.' With respect to the correctness criterion, this is a programming problem, or more accurately a class of problems because complex systems constitute a class of systems. The reusability criterion introduces a software engineering aspect into the general problem.

A further requirement of the simulations, subsidiary to the two above is that they make intelligent use of parallel processing resources, principally in that computations may be *manually distributed*, according to programmers' hints, across available processors. The problem of optimal resource usage in parallel machines involves a balance in the load placed upon communication, processing and memory. Achieving this balance is known to be very difficult even in highly specific parallel programming problems, in the general case it seems intractable. In anticipation of future developments in automatic mapping of parallel programs to parallel computers, it is adequate at present to develop simulations that are concurrent, yet dependent upon human intelligence to guide the placement of processes onto processors.

6.2 Executing Specifications

Parallel program induction from formal, application oriented specifications relies on the availability of information which is usually implicit in the semantics of the specification language and in the statements of a particular specification. The nature of this required information is dependent on the lower level language in which statements are to be generated. The lower the level of abstraction of the target language with respect to the specification language, the greater is the burden on the formal procedure concerned with translating from the higher to the lower level. This burden is in the form of numerous points at which decisions need to be taken in translating a high level declarative concept into one of many possible implementations.

In a previous chapter the problem of correctness of specification has already been treated. It was established that the mutual exclusion from shared data and the absence of deadlock may be deduced from instance specifications.

In the present case the target language will be occam, and specifically occam-2, since it affords a fairly close semantic approximation to CSP, which in earlier chapters was a foundation for the design of the specification language. However, occam programs are structured quite differently to formal specifications of the kind under investigation in this thesis. The most visible difference is the separation into static and dynamic portions. Radically, the two languages differ in their model of communication, occam offering two complementary primitives and specifications offering only one (a discussion of this is given in section 3.4.3). In addition, specification languages express algorithmic patterns in terms of an event ordering relation, whilst occam relies on block-structured programming constructs and programming idioms. The problems tackled by this chapter concentrate on reconciling these divergences between the languages.

It should be borne in mind that the specification language is aimed at program skeleton description. The specification of detailed computation is out of its scope. In its use of software modules to specify actual computations upon program state, it contains an interface to an external language. It would be desirable for such modules to be passed to specifications as occam procedures, as they may then be directly invoked by the generated occam code.

The approach taken here towards simulation program induction is in the form of information-adding transformations from an occam program containing minimal application-specific information into an occam program that contains all the application-specific information given in the specification. The specification is seen as a source of information required to guide such translations, instead of as an initial syntactic form upon which correctness preserving source-to-source transformations are performed in order finally to yield an equivalent programmatic form.

Prior to presenting the method in detail, the underlying reasoning and an explanation of terms follows. A structure known as an *occam skeleton* which is principally a subset of occam syntax and semantics is central to the scheme[Ikr95]. It is meant to be an alternative representation of the specified event ordering relation that, roughly speaking, constitutes a parse tree of the kind of occam program ideally suited to implementing or simulating complex systems.

That generic form of occam program is entirely constituted at the top level of replicated sequential processes, each simulating a single agent. Process replication in occam mirrors object instantiation in specifications. Therefore, while complex systems are viewed as assemblies of cooperating agents at the specification level, they are most naturally implemented analogously as communicating processes. Each agent is implemented by a sequential process that is likely to be iterative, as explained in the chapters on specification.

How is such a generic structure to be induced starting from the information given in a arbitrary instance specification? The problem of creating an occam skeleton is largely one of sorting events based on the ordering relation. The initial occam skeleton for any instance specification is the unsorted collection of its event alphabet. For events, the sorting sequence is identical to their specified temporal ordering. This initial form embodies none of the sequential information to be found in the specification.

The initial form is refined through the incorporation of each sequential ordering constraint in turn. In terms of sequential information, each refinement should be strictly information-increasing yet information-preserving. That is, the addition of new sequential information brought about by each refinement should not alter or contradict the existing sequential information in the occam skeleton.

The sorting process resembles a topological sort of the acyclic graph where nodes represent events and directed edges represent the event-precedence constraint.

The desired form of the final skeleton reflects the intended structure of the final occam program, that is, sequential component processes composed in parallel. While it is evident that such a structure will respect sequential constraints within processes representing individual agents, sequential constraints between agents need to be represented in occam skeletons as synchronous communication events between processes. At this level, communications are still one-sided and input-based, as explained in connection with the concurrent model of complex systems.

The unfolding of single communications into input-output pairs is one of the main concerns at the next step, which is the generation of valid occam code from occam skeleton structures. In contrast to the previous step, this is, with the exception of such interprocess communication considerations, largely a syntactic, source-to-source correctness preserving transformation. Events are translated directly into procedure calls. The procedures thus invoked are simply the modules supplied as parameters to the instance specification that have been textually embedded within 'wrapper' procedures.

6.3 Step 1: Constructing Occam Skeletons

6.3.1 Introduction

The following procedure is followed to construct the intermediate structure called the occam skeleton from the information specified in an instance specification which will be called I. The event alphabet of I may be generated as described in section 3.4.1 and is given by the set $E(I)$.

An occam skeleton is a tree whose nodes may be either one of the constructs SEQ, PAR or GUA, or a member of $E(I)$. Each member of $E(I)$ occurs at least once as a leaf node in the tree, implying that there are at least $\mid E(I) \mid$ leaf nodes. Members of $E(I)$ should not occur as non-terminal nodes. An arbitrary number of occurrences of constructs is allowed, but they should occur only as non-terminal nodes. Child nodes of constructs are given an order, that is, given that a non-terminal or construct node has a set of children C, there is a bijective labelling function $o : Z_{|C|} \longrightarrow C$, such that o(0) is always referred to as the first child of its parent, while o($\mid C \mid$-1) is the last child. As a notational convention, an occam skeleton may be written as a nested list using the prefix notation for trees.

The SEQ-skeleton [SEQ a b], where a and b denote events, is interpreted as a program that performs a and then b. It is a valid intermediary form of the ordering specification $a < b$. Where a and b represent subtrees, the skeleton is recursively interpreted as the interpretation of a, followed by the interpretation of b. For example, given that c, d, e and f are events, if a is [SEQ c d] and b is [SEQ e f] then the original specification can be implemented by any program that consecutively performs the operations denoted by the events c, d, e and f, in that order. After taking into account all such subsumptions of nested SEQ-skeletons, leaf nodes must be unique, with the exception of the rule for GUA-skeletons given below.

PAR-skeletons [PAR ...], likewise may be interpreted recursively. They mean that the operations corresponding to the subtrees of the PAR may be performed either sequentially without regard to children's order, or simultaneously. Again, the leaves of a PAR-skeleton must be unique, but this constraint is overridden for such leaves as are shared with a nested GUA-skeleton.

An entire GUA-skeleton [GUA ... z] corresponds to the occurrence of the event denoted by its last child, in this case labelled z. The significance of the children preceding z in the order, which must be events, is that of guards. That is, z is enabled only after the occurrence of all its guarding events. An event may not act as its own guard. Over an entire occam skeleton, guarding events are the only ones that may occur multiply, for the obvious reason that a single operation may be required as a prerequisite to several others. In allowing multiple occurrences in this special case, the semantics of events as unique instances of operations is preserved since guarding events represent pre-conditions to occurrences of guarded events and not actual occurrences.

The occam skeleton generation procedure can be implemented as an iterative refinement of an initial skeleton that is free of sequential information. This may be constructed simply by having a PAR root with all the events in the alphabet attached to it as leaf nodes in an arbitrary order. Meanwhile the event ordering relation $<$ is to be viewed as a store of all ordered pairs (x, y) such that $x < y$. Then the iterative procedure picks a remaining member of the store and alters the current occam skeleton to conform to the single piece of ordering information represented by that member, taking care to preserve the existing ordering information embodied in the skeleton. The manner in which transformations are performed will be described shortly on an case-by-case basis. This is repeated until the store is exhausted. At that point, the skeleton would contain all the information present in the event ordering portion of the dynamic specification.

In practice, the procedure outline above tends to produce skeletons whose structure is dependent upon the order in which the relations are selected from the store. Moreover, they rarely take on the structure that was identified above as being natural to occam implementations of complex systems, that is, as assemblies of replicated sequential processes corresponding directly to the structure of the topology given in the static specification.

An alternative method that avoids these difficulties, and furthermore exhibits a computational complexity of a lower order, is as follows. The initial skeleton is cast into a structure that is broadly similar to the required final structure. This requires, firstly, that events be discriminated into as many classes as there are objects in the system, and thereafter, that each class of events be constructed into an independent PAR-skeleton as was done for all events in the previous method, and finally, that the separate trees thus obtained be composed under a single PAR construct to form a two level tree. Semantically, this tree is equivalent to the initial skeleton of the previous method, since PAR constructs subsume those subtrees that are simple PAR-skeletons.

It is now possible to formulate a sequentialization procedure that is guaranteed to produce skeletons of the desired structure. The sequentialization may be performed in two consecutive phases, the first being parallel and the second sequential. In the first phase, the sequentialization of each subtree of the initial skeleton proceeds in the usual iterative manner, but only for those precedence relations that concern the local object alone. Thereafter, the remaining precedences, that is, those between events corresponding to operations across object boundaries, take effect.

The retention of the original structure of the skeleton is ensured by inserting a SEQ construct node between the top level PAR node and each object-skeleton, in the initial skeleton. Then, since the sequentialization rules under no circumstances either extract events from SEQ-skeletons or eliminate existing SEQ nodes, the integrity of object boundaries within the skeleton is assured throughout the sequentialization process.

It is further necessary to label the abovementioned inserted SEQ nodes with the class name of the corresponding object. This may be determined by examining the event labels, which by convention include the name of the target object. Then the class of the object may be determined from the static specification.

6.3.2 Sequentialization

The manner in which occam skeletons are altered to conform with precedence relations may be called sequentialization, since every alteration inevitably involves the addition of a sequential constraint between events and consequently, a restriction of the parallelism previously represented in the skeleton.

Its correct implementation is required to be information-preserving. This requires that transformations made to skeletons be sensitive to the context of the events concerned prior to the alteration, and preserve that context in such a way that existing relations are not affected by the alterations except where necessary. Such necessity would arise in cases where the relation to be imposed causes new dependencies indirectly, as a result of the transitivity of the precedence relation.

The following canonical transformation rules have been identified and incorporated into a prototype implementation. The rules fall into two major classes: the first contains one rule for the case when the lowest common ancestor (LCA) of the two events (leaf nodes) is a SEQ-skeleton; the second contains ten rules applicable when their LCA is a PAR-skeleton. The only rule falling under the first class is:

1) Condition: any. Action: none. Comment: the two events are already correctly ordered relative to one another, therefore the selection of this rule corresponds to a redundant sequentialization.

The rules falling under the second class always result in some alteration to the existing occam skeleton when faced with the ordering relation on events $e1 < e2$:

2) Condition: $e1$ and $e2$ have the same parent (PAR node.) Action: Delete $e2$; replace $e1$ with the SEQ-skeleton [SEQ e1 e2]. Comment: it would be equally valid to delete $e1$ and replace $e2$, since only the order of the PAR node's children would be changed thereby, which is semantically equivalent to the effect of the prescribed action. Note that the initial siblings of the pair remain unordered with respect to the pair.

3) Condition: $e1$ and $e2$ belong to different PAR parent nodes; up to the root of the their LCA, neither event node has a SEQ ancestor. Action: delete $e1$; replace $e2$ with [SEQ e1 e2]. Comment: absence of SEQ ancestors is required here to prevent existing orderings with respect ancestors of the 'moving' event $e1$ from being broken and to prevent formation of spurious orderings between $e1$ and the ancestors of $e2$.

4) Condition: $e1$ and $e2$ belong to different PAR parent nodes; up to the root of their LCA, only $e1$ has a SEQ ancestor. Action: delete $e2$; replace $e1$ with [SEQ e1 e2]. Comment: This is applicable when $e1$ possesses sequential relations that are not shared with $e2$, such that the set of the sequential relations involving $e1$ is a proper subset of the set of those involving $e2$. The action taken could quite possibly introduce the unspecified constraint that $e2$ should precede some of the events subsequent to $e1$. Difficulty results from the translation of the relation $e1 < e2$ asserting that $e2$ should occur at some point after $e1$, into the occam skeleton [SEQ e1 e2] which forces $e2$ to occur prior to all other events subsequent to $e1$. Two strictly correct alternative actions would be either (i) to replace $e2$ by [GUA e1 e2], which is the universally valid action, or (ii) to replace $e1$ with [SEQ e1 [PAR e2 [SEQ ...]]], where the PAR-skeleton maintains the concurrency between $e2$ and those other events (represented by the innermost SEQ-skeleton) which are subsequent to $e1$. This latter SEQ-skeleton would be constructed in a relatively non-local manner that requires analysis of the

highest SEQ-subtree, containing *e*1, of the LCA.

5) Condition: *e*1 and *e*2 belong to different PAR parent nodes; up to the root of their LCA, only *e*2 has a SEQ ancestor. Action: delete *e*1; replace *e*2 with [SEQ e1 e2]. Comment: This case is symmetric to 4).

6) Condition: *e*1 and *e*2 belong to different PAR parent nodes; up to the root of their LCA, both have SEQ ancestors. Action: replace *e*2 by [GUA e1 e2]. Comment: In general, no alternative action is possible without resorting to global analysis.

7) Condition: *e*1 has a PAR parent; the skeleton [SEQ pre+<e2>+post] exists (+ is the sequence concatenation operator), where pre and post are sequences of *e*2's siblings; up to the root of the events' LCA, *e*1 does not have a SEQ ancestor. Action: delete *e*1; replace post with the skeleton [PAR <e1>+post]. Comment: problem noted concerning action of 4) applies here too, since *e*1 may be needlessly brought into sequential relation with certain events in the wider sequential context of *e*2. Such events, if existing, would all lie below the highest SEQ-ancestor node of e2 within the LCA.

8) Condition: as 7), but up to the root of the two events' LCA, *e*1 has at least one SEQ ancestor. Action: replace *e*2 by [GUA e1 e2]. Comment: Unlike 7), no events are displaced in the skeleton, therefore extraneous sequential relations cannot arise.

9) Condition: *e*2 has a PAR parent; the skeleton [SEQ pre+<e1>+post] exists, where pre and post are sequences of e1's siblings; up to the root of the events' LCA, *e*2 does not have a SEQ ancestor. Action: delete *e*2; replace post with the skeleton [PAR <e2>+post]. Comment: symmetric to 7).

10) Condition: as 9), but up to the root of the two events' LCA, *e*2 has at least one SEQ ancestor. Action: replace *e*2 with [GUA e1 e2]. Comment: symmetric to 8)

11) Condition: *e*1 and *e*2 belong to SEQ parent nodes. Action: replace *e*2 with [GUA e1 e2]. Comment: in this and other rules which call for the guarding of *e*2 by e1, it should be noted that if *e*2 is already guarded, then no new GUA-skeleton is to be created, instead e1 should be added as a new child of the existing GUA node, at any position other than that of the last child (*e*2 would continue to be the last child.)

Sequentialization rules can be seen as skeleton-to-skeleton transformers that incorporate sequential information into an existing skeleton. During transformation, the sequentialization rules using only local information that is explicitly represented in occam skeletons. This information is localized in that the incorporation of one sequential relationship, say $e1 < e2$, requires only that the subtree rooted at the LCA of $e1$ and $e2$ be examined and altered. The LCA of the pair of ordered events is of vital importance in selecting of the correct rule for altering a particular occam skeleton. The subtree rooted at the LCA can be viewed as the smallest skeleton containing all information necessary to order the pair. All changes to the existing skeleton will be made within the LCA-subtree, and the resulting skeleton retains all the information that had been present prior to the alteration.

The danger in sequentializing beyond strictly the specified constraints is that certain event pairs may inadvertently fall under a SEQ LCA and be ordered arbitrarily as a result. This alone does not violate the specification of event orderings, since arbitrary ordering is one valid interpretation of unordered events in specifications. However, danger may arise in future if such an arbitrarily ordered pair is explicitly required to be ordered in the reverse sense. Then according to rule 1) the later ordering would be ignored thus leading to non-conformance with the specification. A conservative strategy that avoids over-sequentialization is to define just two rules for the second class: if neither $e1$ nor $e2$ possesses a SEQ ancestor node (up to their LCA), then this is equivalent to the conditions of rules 2) and 3) above, and the action of rule 2) becomes appropriate; otherwise, $e2$ should be guarded by $e1$, as in rule 8), for example.

The description of sequentialization in terms of sequential contexts is instructive. The term sequential context denotes a set constituted of all unguarded events lying under a particular reference node such that at least one ancestor of those events, up to but not including the distinguished node, is a SEQ node. Guarded events are not included in sequential contexts because GUA-skeletons are free to express sequential relationships with events lying beyond the subtree rooted at some distinguished node. The information represented by GUA-skeletons thus generally does not possess the locality of other skeleton forms.

The strategy of initially decomposing the system's alphabet into object-skeletons (which are also SEQ-skeletons), as described earlier, effectively creates as many sequential contexts, with respect to the root of the skeleton, as there are objects in the system. These initial sequential contexts are independent in two senses. Firstly, the intersection of any two contexts is always the empty set, and secondly, the object-skeletons are maximally concurrent relative to each other in that the LCA of any two nodes lying in separate object-skeletons is the PAR root node of the skeleton.

6.3.3 Treatment of Guards

At this point a skeletal form of the parse tree for the required simulation program is available. Before a transition can be made from the skeleton form to correct occam source code, it is necessary to find a mapping from the skeleton's one-sided communications model to occam's two-sided model. A descent from the high-level view of inter-object communication that has been maintained thus far, to the lower-level occam view of communications, is called for. This is done through the appropriate handling of GUA-skeletons. The transition is achieved by post-processing the occam skeleton generated into a form which explicitly represents communications as understood by the occam model.

In skeletons the mechanism used to indicate dependencies between concurrently operating sequential entities is the guard. The sequentialization rules make use of guards via GUA-skeletons to indicate sequential dependencies in preference SEQ-skeletons under certain circumstances. In order to order a pair of events $e1$ and $e2$ when it is known that $e1 < e2$, either the SEQ-skeleton [SEQ e1 e2] or the GUA-skeleton [GUA e1 e2] are created. In the former case, the removal of either $e1$ or $e2$ is also required, whichever one happens not to lie in a sequential context, relative to the pair's LCA. When both events lie in sequential contexts, removal of either one would break the information-preserving character of the sequentialization process. Therefore, under such conditions the creation of the GUA-skeleton is appropriate. A sequential context in occam terms is a sequential process, while dependencies between these contexts are identical to sequential relations between two such concurrent processes. Concurrent sequential contexts, that is, those lacking any inter-contextual sequential dependencies, are temporally decoupled and can be understood to run according to independent and unsynchronized clocks.

Any one of two kinds of dependency may be encoded in the form of a GUA-skeleton, either synchronization or data communication. In occam, synchronization is implicitly achieved by communication over channels. Therefore channel-mediated communications generally are the means of representing GUA-skeletons in occam code. In skeletons, however, communications the point in time at which communications are to occur are given in terms of the receiving object-skeleton. In occam terms, this method of describing communications is incomplete since reciprocal information concerning the transmitter is underspecified.

The precise identity of the data to be communicated is deduced from the event labels, given that the conventional labelling scheme has been followed. Through the same means the identity of the two communicating objects is also evident. However, due to the abovementioned underspecification of the transmitting side of communications, there is scope for making different interpretations as to when the output of the data is to take place. On the other hand, the guarded event should always be interpreted as occurring at the exact point that its parent GUA node appears in the occam skeleton.

The difficulty in positioning the outputting side stems from the lack of communications primitives, input and output, from the specification language. These primitives need to be inserted into the generated skeleton as 'hidden' events; hidden in the sense that they are invisible, and deliberately so, to the viewpoint of the specification language. In instance specifications, communications are implied by events belonging to binary operation instances. Implied communications are tied to state change in the receiving object because binary operations are always formulated as the setting of the receiving object's state to some function of its own attributes and the attributes of some neighbouring object, the outputting object. Thus events belonging to binary operation instances signify change in the receiving object, and furthermore, sequential orderings involving them relate to the ordering of the state change relative to other events.

The specified orderings therefore give no indication as to the ordering of hidden communication events that, at occam the implementation level, must be known. However, it should be recalled that binary operations are specified as mappings from the immediate pre-state of the class of the two objects involved, to the immediate post-state of the class of the target object. Therefore implementations should ensure, as far as is possible, that all communication of data takes place immediately before the data is to be used. This may be trivially ensured from the receiving side of the communication by inserting the relevant input event immediately prior to the binary operation instance event.

The corresponding output event should be placed subsequently to the relevant guarding event. Since this event and the guarded event lie in separate concurrent sequential contexts, it only possible to determine at run-time when a piece of data is required of a certain object by its neighbour. Such a demand-driven scheme may be implemented using occam's channel input guards and ALT construct. However, occam skeletons as defined here do not possess sufficient expressivity to capture these concepts. Within the current limits, therefore, a suitable alternative implementation strategy is needed.

The approach adopted by the prototype implementation is to treat outputs in exactly the opposite manner to the treatment of inputs; that is, to insert the output event immediately following the relevant guarding event. The hidden output (τ_o) and input (τ_i) events should be seen as being attached to the guarding (e1) and guarded (e2) events, respectively, such that if an arbitrary event a is related in the specification as $a < e1$ then it follows that $a < \tau_o$ and $a < \tau_i$; also if $a < e2$ then $a < \tau_i$. Conversely, if $e2 < a$ then $\tau_o < a$ and $\tau_i < a$; also if $e1 < a$ then $\tau_o < a$.

Given the foregoing, the occam skeleton is processed in the following manner in order to aid in occam code generation. Every GUA-skeleton is replaced by a SEQ-skeleton containing two children, the second child being the guarded event, and the first being a PAR-skeleton containing special input event nodes. There should be as many such nodes as there were guarding events, in a one-to-one correspondence. Each input event node is tagged with the identity of the outputting object as stated in the label of the guarding event.

In addition, guarding events are similarly processed. Every guarding event is replaced by a SEQ-skeleton containing two children, the first child being the guarding event, and the second being a PAR-skeleton containing special output event nodes which are in one-to-many correspondence with the events guarded by the event concerned. Each output event node is tagged with two items of information. Firstly, the identity of the inputting object as stated in the label of the corresponding guarded event, and secondly, the name of the attribute whose value is to be output. The latter may be found by determining the relevant operation name from the label of the guarded event, and then by inspecting the static specification to determine those attributes of the outputting object which are used to generate the post-state of the inputting object. For each such attribute, a separate output event node is created, thus giving rise to the abovementioned one-to-many relation. In some cases, it will be found that the operation is either one that does not input from the current object, or is an unary operation. This means that the guarding relationship is to be interpreted as simply an inter-object synchronization without regard to the data being transferred. Then the output event node should be tagged with an invalid attribute name.

As a result of this post-processing the skeleton will be changed into a structure consisting solely of SEQ and PAR constructs, the GUA constructs having been eliminated, together with unique nodes for the entire event alphabet of the specified system, in addition to the special communication events.

However, it should be borne in mind that the hidden events added to the skeleton do alter implicitly the event ordering relation given in the dynamic specification. Moreover, the alterations made are such that the sequential relations embedded in the skeleton have increased in number. Thus a verified specification may become incorrect by the addition of hidden events. In particular, deadlock may arise. At this stage it is appropriate to repeat the verification of the instance specification. Assuming occam's synchronous model of communications, two new relations may need to be added to the specified event orderings for every input-output pair. The addition will reflect the blocking character of synchronous communications.

If an output happens to occur prior to the complementary input, this prevents a sequential process from proceeding after a channel output is committed until the complementary channel input has occurred. Thus hidden input events necessarily precede the event immediately following their complementary hidden output event. The former event can effectively be taken to be the guarded event. The latter is the first event successive to the guard yet prior to the guarded event; should there be no such event, there is no successor to the guard within the sequential contexts of the guard and the guarded event (relative to their LCA), and therefore no new ordering relation is to be added.

By applying the same reasoning to the case where an input occurs prior to the complementary output, another sequential relation may also arise. Though the two cases are mutually exclusive, the relative timing of inputs and outputs cannot be established until run-time. For this reason, the conservative approach taking both possibilities into account should be taken.

6.4 Step 2: Generating Occam Programs

6.4.1 Introduction

Having obtained an occam skeleton representation of the dynamic instance specification, the next step is to generate an occam program conforming to the structure of the skeleton. The skeleton is a high-level parse tree of an occam program which will simulate the specified complex system. The occam skeleton encoded as a tree is translated into a textual program skeleton by traversing the tree in preorder, while in most cases outputting an appropriate occam keyword or statement for each node encountered. The level of indentation for any generated string of program text is obvious from the depth of the relevant node. The process demonstrated in this section requires only a single-pass traversal through the occam skeleton. Processing of nodes is deliberately a localized one in the sense that the code to be emitted for most nodes is dependent upon the contents of that node, regardless of its relation to other nodes.

In the present step the contents of the static specification are put to greatest use. This data provides the declarative content of the generated occam program.

The semantics of the occam skeleton entities SEQ-skeletons, PAR-skeletons and events, have direct correspondents in occam programming, in the form of SEQ constructs, PAR constructs and procedure calls, respectively.

6.4.2 Declarations

The required declarations are communication protocols, the names of variables and channels, their types and protocols, respectively, and also the names, parameters and implementations of procedures. In the textual placement of these declarations, attention should be paid to occam's scope rules.

Each object will be implemented as a separate process. In the previous step, the top level structure of occam skeletons has been deliberately cast into a form that ensures that this program design policy will be followed. The variables that will be in scope for each process will correspond directly to the attributes of the object concerned. Each process therefore, will have local variables that are in scope over the entire process body. All this assumes that the class of each object is known. However, that information would have been recorded in the initial skeleton in specially labelled

SEQ nodes as mentioned earlier, and remains in the final skeleton. Upon encountering such a node whilst traversing the skeleton, the text for the relevant local variable declarations should be emitted. The name and type of these variables are taken from the static specification.

Communications over channels are required to implement both attribute access between neighbouring objects and synchronization. Inter-process channels need to be declared globally since the generated skeleton structures are invariably 'flat', that is, there is no organization of object-skeletons based on communication neighbourhoods. It is therefore the responsibility of the code generation mechanism to uphold the occam usage rules regarding unidirectionality and two-sidedness of communications. The maximum number of channels required is equal to the number of children of all GUA-skeletons, discounting the last children of GUA nodes. The number of distinct channels actually required will constitute a smaller number, and this will be discussed in a later section. For each inter-object guard pair, the direction of the corresponding channel should match that given by an edge of the topology as set out in the static specification.

According to the present scheme, code generated for inter-process communication requires that all attributes of the source object should be output sequentially. Therefore, sequential protocols for all inter-process communication instances should be declared. In the system's topology given in the static specification, every unique instance of an edge leading from an instance of a particular class requires a unique protocol. Thus in general, in a system composed of instances of N classes, at most N protocols are needed, for example when the topology is a fully connected graph. The lower limit for such protocols would be 0, in cases where no interaction occurs between objects of distinct classes. The protocols should, naturally, be declared having unique names. A scheme which ensures unique naming is to append the word 'Protocol' to the name of the transmitting object's class. The sequence of types declared within these protocol declarations is the same as the types of the given sequence of attributes in the static specification.

Intra-process communications do not serve to transmit data, they are required instead to implement synchronization between operations taking place within a single process. This distinction between inter-process and intra-process communications is discussed later in this chapter in connection with the translation of input events into occam statements. As such a simple protocol for conveying values of a primitive type, such as an INT protocol, is satisfactory.

Assuming that the maximum number of channels will be declared, some unique labelling scheme is required. A suitable scheme is to examine each tagged output event node to determine the identity of both the guard and guarded events. The two event labels, when combined, will always yield a unique name. Channel protocols may be deduced by examining the tag of the input event node, which reveals the identity of the relevant outputting object. From this, the appropriate communications protocol to be employed is evident since there is a unique protocol for every class of outputting object. However, where the outputting and inputting objects are identical, in other words, when the channel does not cross an object boundary, it should be declared as following a simple INT protocol.

The generated occam procedures will contain the main computational code for the simulation. Essentially, the role of procedures will be to provide a superstructure for the external software modules provided in the specification. It is assumed that these modules are occam functions. Each operation declared in the static specification is used to determine the format of a corresponding procedure of the same name. Since an operation effects state changes to one or more attributes of a single object, in its procedural form it may be declared locally to each class. In occam terms, a class would be implemented via replicated PAR constructs and therefore the appropriate scope of declaration of procedures would be restricted to such constructs. In the present code generation strategy, however, instances of a certain class are not implemented generically but through differentiated textual forms corresponding to object-skeletons. An equivalent scoping effect may be achieved by reproducing procedure declarations for every process corresponding to a separate object. That, though, will swell the amount of code generated considerably.

It may be more appropriate to declare procedures singly at the highest scope, if it is possible to ensure that they are named uniquely, to prevent name clashes, and have access to the variables representing relevant attributes through procedural arguments. The first requirement is automatically satisfied by the specification model which dictates that operations be named uniquely. Thus operation names may be adopted by their corresponding occam procedures. The second requirement, concerning access to attributes, would have been automatically catered for during the specification verification stage by checks preventing any single attribute from being written to by several operations concurrently and from being written to and read from simultaneously.

An unary operation will require access to a single object's attributes. Therefore the formal parameters declared for its procedure will correspond to the attributes of that object. A binary operation will access the attributes of two objects, which are therefore required as parameters. In both cases, all required attributes, with the exception of those attributes altered by the operation, need to be declared as VAL parameters, that is, the relevant actual parameters will be passed by value.

Strictly speaking, only those attributes that are used by an operation need be given as parameters. This may be determined by examining the right-hand side of the specification's operation definition. Attributes that are to be updated are identified as having values in their respective slots that differ between the left and right hand sides of the operation definition (slots are positions within a state tuple, as given in section 4.2). On the other hand, attributes that need to be read are known by their appearance in the form of arguments to modules or their appearance in slots other than their own. It is possible for attributes to fill both roles.

6.4.3 Statements

The declarative framework having been erected, it is now possible to insert the actual computational statements. These statements divide into two forms. On one hand, the statements comprising the body of procedures, and on the other, those comprising the body of the overall system process. The code generation strategy is such that the latter group of statements will simply consist of calls to the operation-procedures arranged under SEQ or PAR constructs. The former group of statements will define the sequential process appropriate to each procedure.

The composition of procedures will be addressed first. The body of a procedure will consist of a series of statements under a single SEQ construct. By inspecting the right-hand side of corresponding specified operation, in particular the content of each slot in turn, each member of the series is generated as follows. If the slot encountered contains the same symbol as the corresponding slot in the left-hand side, then no alterations are called for and attention moves to the next slot. If the slot contains a symbol that, in the left-hand side, was found in a different slot, then the copying of one attribute to another is required. The final alternative is chosen should the slot contain the name of a module function. If so, the relevant module function is invoked along with arguments corresponding to attribute values given in the current slot.

In specifying operations, when the right-hand side slots refer to left-hand side slots, the intention is to make use of pre-state values to compute the post-state. Thus during the computation of the post-state, side-effects should not be allowed to corrupt the record of pre-state. For this reason, all assignments to attributes should be made to temporary variables, and only upon completion of the procedure are the new values to be committed to attributes. Temporary variables of the appropriate types should be declared local to the procedure for each parameter representing a mutable attribute, or in other words, for each parameter not declared as being a value parameter.

The first statement generated in a procedure is a multiple assignment of all mutable attributes to corresponding temporary variables. Thereafter, code for each slot changed by the operation is generated sequentially. Since separate attribute updates do not interfere with each other, and each attribute is altered once by each operation, it is natural to compose statements making up this second stage of code generation under a single PAR construct. As outlined above there are two possibilities as to the mode of attribute update, either through copying attribute values or through generation of new values based on application of functions to attribute values. In the first case, the generated statement is an assignment of the formal parameter corresponding to the attribute to be copied, to the temporary variable shadowing the formal parameter corresponding to the current slot. The second case would generate an assignment to the same temporary variable, but the source of the assignment would be found from a function call. This function call would be derived simply by writing the module name and, enclosed within brackets, the formal parameters corresponding to arguments given in the operation specification, in identical sequence. The final stage again consists of a single multiple assignment statement, but on this occasion the sense of the assignment would be opposite to the multiple assignment written in the first stage. This last statement represents the commitment of the newly-computed values to their respective attributes.

It is now possible to generate the code for the process that will simulate the specified system. The computations required to perform the simulation are achieved by performing the operations given in the static specification, in the correct order as dictated by the dynamic specification. The occam skeleton constructed thus far represents the information given in the dynamic specification in a tree structure. This structure is really an abstract form of a parse tree of the required program. The translation of the skeleton into source code, then, may be achieved by a simplified reversal of the normal parsing process.

It is clear that SEQ and PAR nodes translate directly into identical keywords in occam. These nodes represent occam constructs whose subordinate constructs and statements are found in the skeletons in the form of descendant non-terminal and leaf nodes. Skeletons are constructed in such a manner that non-terminals translate to SEQ or PAR constructs, while leaf nodes translate to one of three kinds of statement, either procedure calls, input statements or output statements.

Were all the non-terminals simply SEQ nodes, a correct program would consist of a SEQ construct enclosing translations of all leaf nodes sequentially arranged according to any left-to-right tree traversal, such as preorder. The other extreme skeleton form would be one that contains only PAR non-terminals. Here, a correct program would be a PAR construct enclosing an arbitrary sequence of all translated leaf nodes. However, due to the abstract structure into which skeletons are cast, neither one of these forms will result from the sequentialization phase. In general, skeletons will have PAR roots, while nodes at the next layer will be SEQ nodes corresponding to the the roots of individual object-skeletons. The object-skeletons themselves, typically, are mixtures of SEQ and PAR non-terminal nodes.

The order of child nodes is only significant for SEQ non-terminals. For consistency, however, the code generation strategy of the prototype implementation retains the child-node ordering found in the skeleton. This permits a simple strategy that relies upon a preorder traversal of the skeleton. During traversal, the translation of every node encountered is emitted as a line of text. The depth of nesting of the generated text is equivalent to the depth of the encountered node within the skeleton of the corresponding node. By this means, the intended nesting of constructs is preserved in the generated source code, as is the intended ordering of constructs subordinate to SEQ constructs.

In the previous section, the declaration of channels and procedures was given as being at the top level of the program, and consequently they are always global in scope. Variables representing objects' attributes would be declared in the next level so as to have scope over processes simulating the objects. Emitting code for these declarations is triggered by encountering either root node or any of the nodes at the next layer.

One other, very localized, form of variable declaration is also required in the case of temporary input variables. These declarations can be explained in the context of translating input events. Recalling that all input events occur in SEQ-skeletons of the form [SEQ [PAR input1 input2 ...] guarded], where $input_i$ and *guarded* are events, it is clear that when an input is performed in order for the received data to be used by the guarded operation, it is necessary for the data to be stored locally in variables upon input, prior to being put to use. The most restricted scope of these variables is the SEQ construct corresponding to the relevant SEQ-skeleton. The SEQ construct that fulfills this role will be known as the input block.

Input events require the most care during the translation process. In addition to requiring the extra declarations above, it should be noted that they may occur in SEQ-skeletons prior to unary operations, in which case they serve only the purpose of synchronization, as well as prior to binary operations, in which case they may serve a data transmission role in addition to synchronization.

The majority of leaf nodes encountered are not hidden events, that is, they are neither input nor output events, but rather events explicitly specified in the dynamic specification's definition of the system's alphabet. The translation of these events into textual occam is performed in a uniform way, with one exception as will be noted concerning input events. In general these events translate to procedure calls. The identity and arguments of the procedure call are derived by taking advantage of the convention for event labelling. As stated earlier, procedure names are identical to the names of corresponding operations, thus the name of the procedure is found trivially.

As far as unary operations are concerned, the argument list will simply consist of all attributes of the object. Due to the nature of unary operations, this object will be identical to the object of the current object-skeleton. This may be trivially found by inspecting the event label for the name of the object participating in the operation and then

The construction of argument lists when translating binary operations events should take into account the fact that these events are always preceded by input events. Input events are responsible for receiving attributes data from neighbouring objects into local variables. Such local variables, along with the current object's attributes, need to be passed as actual parameters to the procedure implementing the binary operation. It is assumed that the required temporary variables would have been declared upon encountering the enclosing input block. Then the binary operation event would be translated into the required occam procedure call format, that is, the name of the procedure which is identical to the first component of the event's label, followed by the argument list giving first the names of the local variables just described and then the names of the local object's attributes.

Each inter-object input event is translated into a single input statement. The channel concerned is the unique one devoted to the current communication, which would have been declared globally, of an appropriate sequential protocol, as stated in the previous section. The order in which attributes are passed through a channel should, naturally, respect the declared protocol, but in addition both the concerned input and output statements should agree as to the attribute connected to each member of the sequential protocol. The convention is to follow the arbitrary order in which attributes are given in the specification.

Intra-object communications serve only to synchronize operations and thus do not require attribute values to be passed. For such input nodes in a skeleton, a single value of some primitive type may be input and discarded immediately.

The treatment of output events for inter-object and intra-object communications should be the reverse of the treatment of their input event counterparts. The same channels and protocols should be used. Values output in the inter-object case should be all the local object's attributes, while for the intra-object case only a single value of some primitive type is to be output.

6.5 Improvements

Possibilities exist to improve the scheme set out thus far so as to reduce code size and run-time memory consumption, to introduce distributed processing, to support a more expressive set of programming abstractions and to incorporate deadlock-free programming idioms. Certain improvements may be made by post-processing the generated code, but most rely upon additions and alterations being made to the overall scheme.

An obvious refinement that generated occam code is open to is distributed processing. Since the decomposition of any system into independent objects is a prime requirement during specification, and since this decomposition is maintained throughout the generation process, the most direct process distribution strategy is to consider the top-level processes (corresponding to the objects of the system) to be the units of distribution. These processes do not share variables and access to data across their boundaries is implemented through inter-process communication.

Recalling from section 1.1 that typical complex systems have simple agents, and from section 2.2.2 that high communication traffic between agents is largely responsible for the complexity of these systems, it is to be expected that a single agent-process would be more efficiently executed on processors exhibiting relatively low computation-to-communication ratios. Given coarse grained parallel computers, however, achieving efficient execution of generated programs would require the aggregation of neighbouring agent-processes into coarser grained units. It is with coarse grained processors in mind that the program generation procedure has been designed to maximize parallelism down to the level of the individual agent-process. On the other hand, modifying the given procedure to maximize parallelism at the sub-agent level may improve execution performance only on very fine grained processors.

Should the program execution environment provide support for virtual channels, then processes may be arbitrary assigned to available processors. Otherwise additional software components would be needed, for example in order to multiplex several logical channels through a single physical link. In both cases, an adjustment to the generated code may prove to be necessary for certain occam configuration languages. This is to restrict the scope of channels mediating intra-object communications to become local to the process representing the object itself, since channels declared globally or at the topmost level my be misinterpreted by the process configurer as being inter-process channels.

Having established that process distribution is possible, the problem of optimal allocation of processes to processors remains an open one. Other skeleton programming environments have used cost models and trial executions to choose good distributions (for example [BFU93], [Bra93]). By its very nature, the distribution problem is normally tackled with the help of implementation platform specific information from users, relating to the availability of resources.

Code size and memory allocated to channels may be reduced by not declaring redundant channels. Such channel declarations are made due to the policy of allocating a separate channel to every complementary guard/guarded event pair. The number of required channels may be pruned by 'collapsing' the role of a set of channels into a single channel wherever it is possible to show that no two members of the set are used concurrently. Checking may be performed during the stage where a skeleton's guards are converted into hidden events, following the sequentialization stage. Formally, the requirements for two channels, $c1$ and $c2$, to be collapsed as follows. Let $e1$ and $e2$ be the guard event and complementary guarded event, respectively, to which $c1$ is allocated. Let $e3$ and $e4$ be similarly related to $c2$. Then all four of the following relations should be defined according to the dynamic specification: $e1 < e4$; $e2 < e4$; $e1 < e3$; $e2 < e3$.

The main contributing factor towards code size in the present strategy is the repetition of almost identical blocks of code. In occam programming, most instances of this kind of code repetition are eliminated through replicated SEQ and PAR constructs, and through the WHILE construct. Interestingly, what is desired here is the exact opposite of the common code optimization of 'unrolling' loops. To be able to 'roll' similar blocks of code into a single construct would reduce code size significantly. This is a problem in pattern detection that would preferably be tackled at the occam skeleton level rather than at the more verbose occam source code level.

The characteristic of the sequentialization process which preserves SEQ-skeletons has already been exploited to cast the topmost level of the generated occam programs into a fixed format. The same characteristic may be put to use, in certain cases, in order to enforce a desired format for code generated at the level of the individual object. In particular, provably deadlock free structures such as IO-PAR may be enforced in certain cases. It is known that a system composed exclusively of IO-PAR processes cannot deadlock. These processes have the characteristic structure of two sequential phases being iterated. All inter-process communications are performed concurrently during the first phase. That is, inputs and outputs take place in parallel, therefore the name IO-PAR. During the second phase no communications take place, just computations upon the previously communicated values. Then the first phase resumes prompting communications to take place again, and so on iteratively. However, in order for generated code to be cast into such a structure, it is necessary to show that communications and computations may be separated into consecutive phases in such a manner and still conform to the system's dynamic specification.

6.6 Algorithmic Skeleton Framework

6.6.1 Algorithmic Skeletons

An open problem in parallel programming is the nature of abstractions which may be used to program generics. Generics may exist at several levels. A certain algorithm or operation may be generic in that it is applicable to various data structures. At a higher level, a whole module of program code may be generic to a class of similar applications. Another case would be a communications infrastructure such as a processor farm harness that is suitable not only to a class of related programs but to a class of related processor networks too.

In all these example classes, the bulk of the information relating to computing structures is common to all members. This shared information may be referred to as the skeleton of the class. On the other hand, information specific to individual members is less voluminous. Should the skeleton have been implemented satisfactorily, it is preferable to reuse the implementation rather than reimplementing it. It may be assumed that programmers who are familiar with the domain of the class concerned would also be familiar with the terms of reference used to describe the specific, extra skeletal information. This may be exploited to construct menu-based or form-based programming interfaces for entire classes.

Cole proposed the term *algorithmic skeletons* to describe a generic form of parallel programming[Col89]. Subsequent investigations have proved the usefulness of the technique to expressing the generality present among a variety of instances of parallel code. the field has widened to such an extent that certain the concepts integral to algorithmic skeletons are no longer seen as such. New terms such as paradigm-oriented programming and skeleton-oriented programming are in use to capture the wider applicability of the algorithmic skeletons concept to parallel programming.

A skeleton offers a high-level domain-oriented programming interface. Indeed, it is desirable for the interface to hide the parallel nature of implementation completely. The concern of the user would ideally be limited to declaring the problem to be solved on the domain-dependent terminology of the skeleton chosen. The internal mechanisms of the skeleton are responsible for using the given declarations to implement a parallel program either in the form of code in an intermediate language or as executable code. These mechanisms exploit knowledge about the restricted domain of the skeleton concerned in order to generate efficient yet portable programs.

The users input would be in the form of data and code parameters, such as the number of slave processes to be spawned and the function to be computed upon each portion of data, respectively. In practice, the user would also be required to state information regarding the target execution hardware such as the number and nature of processors. Functional languages are popular programming interfaces to skeleton programming systems, due to their support for higher-order functions.

Of crucial importance to the skeletons approach is that parallel programs may be classified into groups whose essential implementation techniques are similar. A library of such skeletons may be assembled, leaving to prospective users the choice of the appropriate skeleton for their programming problem. While this approach is natural when the basis of classification is algorithmic, it is less so for application-oriented classifications. It is quite possible for similar applications to be best implemented in different ways. This renders the construction of application-class oriented skeletons difficult.

In the class of complex systems, commonalities exist to a lesser extent in algorithmic terms than in application oriented terms. The development of a unified skeleton oriented programming system for this class would therefore more profitably be concerned with generating correct programs in a suitable existing programming language, occam. Questions of efficiency and portability of programs are therefore absorbed into the general research into occam. The programming system is thus responsible for all level higher than occam programming, and will be a front end to an occam compiler.

6.6.2 Programming Environment for Simulating Complex Systems

Grouping was integral to the approach towards specifying complex systems. Since simulation programs are derived on the basis of these specifications, an algorithmic skeleton environment for simulating complex systems becomes feasible.

One method to implement algorithmic skeletons is to write the program framework for a class of applications and provide a simple interface to aid in embedding application-specific program code into such a framework. In the present instance, however, it is not the generated code that is reusable, but rather the group specification. We have not endeavoured to translate group specifications directly into code. Group specifications are a step removed, in a manner of speaking, from the stage of program inducibility, since they need firstly to be instantiated through full parameterization. By delaying the code generation phase until instantiation, the benefits gained include that correctness-checking is enabled upon specifications rather than upon bare occam code.

A programming system may be constructed around the given code generation procedure, encapsulating it in such a way that the users of the system need only be concerned with specifications, not with the occam code or the method of its generation and optimization[ICW96]. The system would be equipped with a range of group specifications suitable for implementing various complex system classes. The concern of the system's user is to select the appropriate group specification and to parameterize it suitably in order to obtain the desired instance specification, which would then be submitted to those modules of the system responsible for the verification and implementation of specifications.

Algorithmic skeletons, in general, are characterized by their ability to support the implementation of a well-defined group of applications. Moreover, by design, they resist adaptation to application areas that they are not intended to support. The projected framework for complex systems would possess these characteristics by offering restricted choices for instantiating specifications. This restriction arises as a result of the inability of parameters to alter fundamentally the structural properties of any group specification. For example, by definition parameters may not substitute for such structural entities as class definitions, operation definitions, in the static domain, or for precedence rules, in the dynamic domain. The power of parameterization lies instead in the ability, on one hand, to state quantitatively the required dimensions of such abstractions as a system's topology and the number of its interactions. On the other hand, parameterization can specify qualitatively the types of a class' attributes and, via modules, the nature of state-change caused by operations.

This form of restricted expression must be combined with adequate documentation in order to guide users to select appropriate group specifications for their problems, and thereafter to guide the instantiation of specifications. The distinction should be made between the authors of the libraries of group specifications, who may be called specification writers, and the users or clients who make use of these specifications for the purpose of writing simulations. Formulation, refinement and documentation of group specifications are the responsibility of specifications writers. Their task requires in-depth analysis of the application area in order to determine the common static and dynamic characteristics, and also to test that common instance specifications are verified. However, the results of this task, once completed, may be stored and reused by clients of specification libraries.

Group specifications offer users, through the possibilities for parameterized instantiation, a system-specific programming interface. Should it be desired to insulate users from the notational and conceptual details of the specification language, then the library browsing mechanism should extract and display only the informal portions of group specification documents. In order to select and make use of a certain group specification, users need only be aware of the kinds of applications that the specification was designed to cater for, as well as the role of each formal parameter.

The simplest architecture for such a programming environment would be one wherein the data flows are simple. For example, having composed by some means the required instance specification, the user presents the result to an front-end module that performs an initial verification. If successful, the specification is passed on to a second module that constructs a conforming occam skeleton. The result, together with the original specification are then input by a post-processing module that introduces hidden events into the skeleton. Before proceeding, a revised specification containing an updated dynamic specification that incorporates hidden events needs to be presented to the verification module. Should the second verification be a success, the specification and skeleton are processed by the final module which performs the actual generation of code, through, for instance, the single-pass procedure described earlier.

There is scope for improving such a simple system by introducing interaction with the user in order to aid the user in writing correct instance specifications and to aid the environment in generating correct and more efficient code. This requires the verification module to report errors to the user, pointing out areas of the specification which are faulty and suggesting appropriate corrective measures. For example, users may be allowed to provide hints to the remaining modules regarding the course of action to be adopted at various decision-points.

6.7 Conclusion

The procedure given is specifically aimed at implementing simulations of complex systems insofar as it is designed to convert complex systems specifications into correct occam code. However it is applicable to any system — not only those commonly understood to be complex — that may be specified in the given notation.

The essence of the procedure is the construction of an abstract parse tree, or skeleton, for an occam program from the information given in the formal specification, followed by conversion from the skeleton representation to an occam representation. The latter step involves a reconciliation of between the models of message passing present in the complex system specification language and in the occam paradigm.

The proposed simulation programming environment is based on the algorithmic skeletons concept, however the environment is seen by users as supporting a set of skeletons, each one implementing a distinct class of complex systems simulations, rather than as a single algorithmic skeleton. A specialized notation for specifying each class of complex systems is defined by group specifications.

Chapter 7

Conclusion

Here the fulfilment of the objectives of the thesis, as stated in the introduction, is assessed in the light of the findings of each chapter. It should be recalled that the objectives were classed under the headings of modelling, classification and parallel simulation.

Chapter 2 explained in detail the requirements for fulfilling the modelling objective. The reliance of the remaining two objectives of the thesis upon satisfactory modelling was noted. Thereafter models formulated elsewhere were described. The main contribution of this chapter was an outline of a concurrent model of complex systems which constitutes a synthesis of object oriented modelling with the communicating process architecture philosophy of the occam language.

Chapter 3 further contributed to the modelling objective by detailing the proposal for abstracting real complex systems into the formal domain. Consideration was given to the abstraction of groups of complex systems, in anticipation of having to classify systems. The need to reconcile an ideal model of complex systems with the computational model of occam also arose in this chapter. The main result of this chapter was a set of guidelines for performing abstraction in such a way that:

1. individual complex systems as well as groups of systems may be formalized; and
2. information for creating occam simulation programs (e.g. network of processes and flow of control within each process) may be deduced.

In particular, investigation of the first point ensured that the objective of classification would be met, through the use of parameterized specifications. Considerations of the second point, which effectively means compatibility of the model with occam, did limit the power of the model to capture dynamic interconnections and non-local communications. In these aspects the expressivity of the model is more restricted than that of the Swarm system, for example.

Chapter 4 completed the definition of the model by providing both the notation for specifications and the set of guidelines for interpreting them. Correctness was defined as the consistency of specifications with respect to the model. With a view to program induction, formal verification procedures were given to test, at the specification level, for the existence of undesirable conditions in generated programs.

Chapter 5 applied the model to formalizing two example complex systems, asynchronous cellular automata and Hopfield neural networks. In its treatment of these two classes of systems, the ability to write group specifications and to formulate classifications was exercised. This chapter served to reinforce, by example, the findings of previous chapters.

Chapter 6 defined a procedure to perform the specification-driven induction of simulation programs in occam. The existence of this procedure enables specification writers to work with the specification language instead of occam. The procedure is open to improvement in many respects, especially as fundamental occam features such as alternation and construct replication have not been exploited. The present lack of external communications in the generated programs is a significant omission. External communications are necessary in order to enable interaction by users with running simulations.

This chapter also proposed a programming environment, inspired by the skeleton-oriented programming paradigm, and directed towards exploiting group specifications as units of specification reuse. The suggested environment would also offer specialized programming interfaces for each group specification.

Appendix A

Symbols Used

Sets and Functions

Set Comprehension: the set of squares of the first 10 positive integers, for example, may be written as:

$$\{\forall i \in \mathcal{Z} \mid 1 \leq i \leq 10 \bullet i^2\}$$

Function Definition: *name: signature: mapping-rule.* E.g., the function *sq* from integers to integers may be defined as:

$$sq : \mathcal{Z} \rightarrow \mathcal{Z} : i \mapsto i^2$$

Function Application: the application of function f to x is written:

$$f.(x)$$

Projections: the projection functions π_1 and π_2 map ordered pairs to their first and second components, respectively. E.g.:

$$\pi_2.(1,0) = 0$$

Traces

The following trace notation has been adopted from [Hoa85]:

Inclusion: s **in** t iff the subtrace s is included in t. E.g.:

$$\langle lunch, tea \rangle \ \textbf{in} \ \langle breakfast, lunch, tea, supper \rangle$$

Concatenation: denoted by the symbol $\hat{}$:

$$\langle breakfast, lunch \rangle \hat{} \langle supper \rangle = \langle breakfast, lunch, supper \rangle$$

Length: denoted by the symbol #:

$$\#\langle a, b, c \rangle = 3$$

Restriction: denoted by the symbol $\upharpoonleft$. Eliminates from a trace those events not belonging to the restricting set. E.g.:

$$\langle a, b, a, c, d, c, a \rangle \upharpoonleft \{a, b\} = \langle a, b, a, a \rangle$$

Star: A^*, where A is a set of events, is the set of all traces taking events from A.

Bibliography

[BB92] U. Bhalla and J.M. Bower. Genesis: a neuronal simulation system. In F. H. Eeckman, editor, *Neural Systems: Analysis and Modeling*, pages 95–102. Kluwer Academic Publishers, 1992.

[BCG82] Elwyn Berlekamp, John Conway, and Richard Guy. *Winning Ways for Your Mathematical Plays*, volume 2, chapter 25. Academic Press, 1982.

[BFU93] Alexander Biriukov, Airat Fatychov, and Denis Ulyanov. Dynamo: A processes-processors mapper for occam2 programs. In Reinhard Grebe, Jens Hektor, Susan C. Hilton, Mike R. Jane, and Peter H. Welch, editors, *Transputer Applications and Systems '93*, volume 1, pages 312–323, Amsterdam, 1993. IOS Press.

[Bra93] Tore A. Bratvold. A skeleton-based parallelising compiler for ml. In *Proceedings of the Fifth International Workshop on Implementation of Functional Languages*, pages 23–34, September 1993.

[Bur88] Alan Burns. *Programming in occam 2.* Addison Wesley, 1988.

[Bur94] Roger Burkhart. The Swarm multi-agent simulation system. Position Paper for OOPSLA '94 Workshop, September 1994.

[BV91] Hugues Bersini and Francisco J. Varela. Hints for adaptive problem solving gleaned from immune networks. In Schwefel and Manner [SM91].

[Col89] Murray Cole. *Algorithmic Skeletons: a Structured Approach to the Management of Parallel Computation.* Pitman, 1989.

[FJ94] Stephanie Forrest and Terry Jones. Modeling complex adaptive systems with Echo. In R.J. Stonier and X.H. Yu, editors, *Complex Systems: Mechanisms of Adaptation*, pages 3–21. IOS Press, 1994.

[For93] Stephanie Forrest. Genetic algorithms: Principles of natural selection applied to computation. *Science*, 261:872–878, August 1993.

[Fox92] G.C. Fox. Parallel computers and complex systems. In David G. Green and Terry Bossomaier, editors, *Complex Systems: From Biology to Computation*, Amsterdam, 1992. IOS Press.

[GS91] Martina Gorges-Schleuter. Explicit parallelism of genetic algorithms through population structures. In Schwefel and Manner [SM91], pages 150–159.

[Han93] Per Brinch Hansen. Parallel cellular automata: A model program for computational science. *Concurrency: Practice and Experience*, 5(5):425–448, August 1993.

[HKP91] J.A. Hertz, A.S. Krogh, and R.G. Palmer. *Introduction to the Theory of Neural Computation.* Addison Wesley, 1991.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[ICW96] I.M. Ikram, P.G. Clayton, and E.P. Wentworth. A parallel programming environment for simulating complex systems. In Hamid R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1097–1105. CSREA, 1996.

[Ikr95] I.M. Ikram. A method to generate occam skeletons from formal specifications. In A.L. Steenkamp, editor, *Proceedings of SAICSIT Research and Development Symposium*, pages 115–125. University of South Africa, 1995.

[INM88a] INMOS. *Communicating Process Architecture.* Prentice Hall, 1988.

[INM88b] INMOS. *occam 2 Reference Manual.* Prentice Hall, 1988.

[Lip87] Richard P. Lippmann. An introduction to computing with neural nets. *IEEE Acoustics, Speech and Signal Processing Magazine*, pages 4–22, April 1987.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[PNS+91] Ray C. Paton, Hyacinth S. Nwana, Michael J.R. Shave, Trevor J.M. Bench-Capon, and Sheila Hughes. Transfer of natural metaphors to parallel problem solving applications. In Schwefel and Manner [SM91].

[Pre94] Lutz Prechelt. CuPit — a parallel language for neural algorithms: Language reference and tutorial. Technical Report 4/94, Universitat Karlsruhe, January 1994.

[Ros72] Robert Rosen. Some relational cell models: The metabolism-repair systems. In Robert Rosen, editor, *Foundations of Mathematical Biology*, volume 2, chapter 4, pages 217–253. Academic Press, 1972.

[Ros91] Robert Rosen. *Life Itself.* Complexity in Ecological Systems. Columbia University Press, New York, 1991.

[SM91] Hans-Paul Schwefel and R. Manner, editors. *Parallel Problem Solving from Nature*, volume 496 of *Lecture Notes in Computer Science*, Berlin, 1991. Springer-Verlag.

[Spi89] J.M. Spivey. *The Z Notation.* Prentice Hall, 1989.

[Wei91] Gérard Weisbuch. *Complex Systems Dynamics.* Addison-Wesley, 1991.

[Win89] Glynn Winskel. An introduction to event structures. In J.W. Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 364–397. Springer-Verlag, 1989.

[Wol84] Stephen Wolfram. Cellular automata as models of complexity. *Nature*, 311:419–424, October 1984.