Specifying and Reasoning about Concurrent Systems in Logic

David Roger Gilbert

A thesis submitted to the University of London
for the degree of Doctor of Philosphy

Department of Computing
Imperial College of Science, Technology and Medicine

June 1990

Abstract

This thesis reports research which attempts to unite the specification, implementation and conformance testing of concurrent systems by the use of a common formalism, first order logic.

A technique employing the formalism has been developed for the specification of the intended behaviour of basic instances of concurrent systems. Communication in a concurrent system is viewed as occurring through bindings made by atomic unification to shared logic variables. The behaviour is conceptualised as atomic observations which belong to partially ordered sets of the states of logic variables. It is possible to describe these sets by sentences in first order logic which can then be transformed into Horn clause programs and interpreted mechanically. Such computations bind variables in the query to data structures which represent the sets of observations of the specified system.

The partially ordered sets of observations can also be characterized by Horn clause descriptions which relate the members of a set of observations to its maximal elements. These secondary descriptions form the basis of logic programs which when executed on an interpreter employing a suitable computation rule exhibit the behaviour described by the specification from which they were derived.

A logic programming language SILCS has been developed whose operational semantics encapsulate the computational rule required for the correct interpretation of these Horn clause programs. SILCS is expressive enough to describe many of the behaviours associated with concurrent systems including concurrency, parallelism and sequentiality. The logical basis of SILCS facilitates reasoning about the behaviour of SILCS programs. Equivalence relations are considered for SILCS programs which permit the comparison of data-structure and process-network stores.

Methods are described for the translation of SILCS programs into programs in concurrent logic programming languages that are capable of implementing concurrent systems. A theory of conformance and equivalence is presented which permits comparisons to be made between specifications and implementations in logic of concurrent systems. The construction of a sequential interpreter for SILCS programs and its use in comparing the behaviour of these programs with logic specifications is reported.

Acknowledgements

Firstly I would like to thank Chris Hogger, my supervisor at Imperial College. Chris has a deep understanding of the theory and practice of logic programming and a special interest in the relationship between logic specifications and programs. His dedication to the use of classical logic as a framework for reasoning and computation kept me firmly on the classical path. Chris has always been willing to make time, whatever the hour, to discuss the progress of my research and to comment on the progress reports that I submitted to him.

My thanks also go to many other people in the Department of Computing at Imperial College who provided me with the stimulating environment and support needed for my research. The members past and present of the Parlog group and logic programming section have been especially supportive as well as being good friends. Steve Gregory deserves my special thanks for his initial suggestions regarding the direction of my research. I should also like to include the administrative and secretarial staff and the untiring members of Technical Support in my thanks.

During the course of my studies I moved from my research post at Imperial College to take up a position as a lecturer in the Department of Computer Studies, Loughborough University of Technology. My sincere thanks are due to the Head of Department, Professor Ernest Edmonds, who provided me with the time and resources which enabled me to complete the thesis. Without this support I would undoubtedly have left the task unfinished. I would also like to express my thanks to John Cooke at Loughborough with whom I had many fruitful discussions, and all the technical staff who supported my hardware and software requirements.

Finally my thanks are due to my father, Leslie Gilbert, for much useful advice, the rest of my family for putting up with me and Mary Medyckyj who understood and encouraged me.

My research has been partially funded by the Science and Engineering Council, ESPRIT and the Department of Computer Studies at Loughborough University of Technology.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces the domain of the study and recounts the motivation and objectives of the investigation.

1.1 Aims

The research reported in this thesis was undertaken with the aim of exploring the feasibility and viability of using first order predicate logic to specify and reason about concurrent systems. The specific aims of the research were:

(1) To develop a method for expressing specifications of concurrent systems in first order predicate logic.

(2) To investigate the issues involved in transforming the specifications into logic programs, including testing for conformance and equivalence.

1.2 Specifying and implementing concurrent systems

The engineering of any system comprises three stages: specification, implementation and conformance testing. Ideally the methodologies used should be mutually compatible so that the relationship between specifications and programs can be formally described. Un-

fortunately this not always the case, even for simple systems. As systems increase in complexity so do the methodologies used for their specification and implementation, heightening the danger of incompatible descriptions and implementations. Conformance testing is often seen as an attempt to bridge the gap between specifications and code.

Physical constraints dictated that early computer systems were built on the sequential model attributed to von Neumann. The design of such systems was relatively simple, and the programming languages used in their implementation were imperative in nature; statements in these languages consisted of direct instructions to the underlying machine.

Concurrent systems evolved in response to the requirement for an increase in processing power coupled with a fall in real terms in the cost of hardware. These systems are more complex than their sequential predecessors and there is presently a growing profusion of techniques used for their design and implementation, many of which have been adapted from those developed for sequential systems. However the design and analysis of concurrent systems present problems which do not exist in the construction of sequential systems. Imperative languages are not well suited to task of programming concurrent systems whose architectures do not conform to a unique model and are radically different from sequential systems. An increasingly popular requirement is for open distributed systems which can comprise components conforming to a variety of model architectures. The design and implementation of such systems is one of the major challenges that software engineers currently face.

Much effort has been devoted to both the specification and the implementation of concurrent systems, but these activities have rarely been coordinated. Recent advances in specification techniques and the design of concurrent programming languages have not yet brought the two activities together within the same framework.

Methods for *specifying* concurrent systems have been devised which do not refer to any particular machine architecture, and some have been accepted as international standards for the specification of open systems. Such formal description techniques are rapidly gaining acceptance and software tools have been constructed to aid their use. The technologies with the most coherent frameworks are those based on the algebraic approach to the specification of concurrency.

Declarative languages promise to free the programmer from the association of programming languages with specific architectures. Many such languages are in the process of maturing as usable alternatives to imperative languages. There are a number of formalisms which serve as foundations for such languages — one group, the 'logic programming languages', is based on first order logic. Recent advances in language design and implementation have enabled non-trivial concurrent systems to be implemented using these languages.

However the central problem facing the designer of distributed systems remains that of ensuring that a system correctly implements its specification. This task is facilitated if specification techniques and programming languages refer to a common model rather than a physical architecture. The work reported in this thesis is an attempt to close the gap that currently exists between specifications and implementations. We intend to use a common technology, namely first order logic, for both the specification and implementation of concurrent systems. The perceived benefits of such a coherent approach are the mechanisation of both the process of producing an implementation from a specification and that of conformance testing.

1.3 Contributions

This work makes the following contributions to the body of knowledge associated with the specification of concurrent systems:

Methods for

- expressing natural language specifications of concurrent systems as sentences in first order predicate logic which describe the observations of such systems as partially ordered sets,
- converting the first order logic specifications into logic programs in which the sets are represented by explicit data structures,
- deriving concurrent logic programs from the first order logic specifications whose execution results in the behaviour described by the specifications,

- transforming certain classes of the concurrent logic programs into committed choice concurrent logic programs,
- generating tests for conformance between specifications and concurrent logic programs.

Also developed during the research were:

- A concurrent logic programming language SILCS, characterized by concurrency, synchronisation, atomic unification and don't-know non-determinism. The operational semantics of SILCS is given with reference to a pure Horn-clause interpreter. SILCS programs can be regarded as an intermediate form between descriptions in logic of the predicted sets of observations and programs in committed choice logic programming languages.
- A theory for reasoning about various equivalences between SILCS programs.
- An interpreter for SILCS, written in Prolog, enhanced with the following facilities: deadlock detection, spy, trace and snapshot of computations.
- A theory for the comparison of programs written in concurrent logic programming languages and pure Horn clause specifications of concurrent systems. The theory provides for dealing with conformance testing and test generation.
- A prototype conformance tester, permitting comparison between the behaviour of SILCS programs and that predicted by specifications written in pure Horn clauses.

1.4 Overview of chapters

Chapter 2 introduces the class of systems within the domain of this study, defining them and describing their distinguishing characteristics. Examples of complete systems are introduced in an informal manner and mini-systems are described which illustrate the characteristics of concurrent systems. We examine issues relating to the specification of concurrent systems, and review and compare existing methodologies designed for this purpose.

Chapter 3 describes a method for specifying concurrent systems using first order logic. Communication between concurrent processes is regarded as taking place via bindings to shared variables. The method takes an extrinsic view of such systems and is based on reasoning about the sets of observations that can be made about the bindings made to variables. Natural language specifications of some small illustrative systems are related to their observational descriptions and sentences in first order predicate logic are derived from them. The specifications in logic are amenable to formal analysis and serve as a basis for the derivation of two classes of logic programs. The members of the first class '$\mathcal{SET}$' reason about data structures which represent the sets of observations while members of the second class '$\mathcal{PROG}$' form the basis of concurrent logic programs whose execution produces the behaviour predicted by the specifications.

Chapter 4 introduces the concurrent logic language SILCS. Programs in SILCS can be derived from programs of the class '$\mathcal{PROG}$'. The operational semantics of SILCS is defined in terms of an abstract logic metainterpreter. The design decisions taken regarding SILCS are explained and justified and its computational model defined with reference to an idealised metainterpreter. Examples of specifications written in SILCS are presented. Equivalences are developed for SILCS programs which permit comparison between certain classes of SILCS programs. The computational model of SILCS preclude it from being a suitable language for the implementation of concurrent systems

Chapter 5 explores the relationship between SILCS and concurrent logic programming languages which have been developed for implementing concurrent systems. The relationship between SILCS and these concurrent logic programming languages is explored in this chapter and rules are given for mappings between SILCS programs and such languages.

Chapter 6 discusses the relationship between logic specifications and implementations of concurrent systems. The relation described is one of conformance between specifications and implementations and the theory provides a basis for the design of a conformance tester.

Chapter 7 summarises the research, describes related work, draws conclusions and discusses future directions that can be explored resulting from this thesis.

Appendix A describes the construction of a portable interpreter for SILCS, written in

Prolog. The interpreter was enhanced with deadlock detection, spy, trace and snapshot of computations.

Appendix B reports the construction of a prototype conformance tester, permitting comparison between the behaviour of SILCS programs and that predicted by specifications written in pure Horn clauses.

Appendices C and D contain technical summaries of work by other authors which is related to our own. We compare our work with related work in Chapter 7.

Appendix E contains a glossary of abbreviations used in the thesis.

Chapter 2

Concurrent systems and their specifications

This chapter introduces the class of systems within the domain of this study, defining them and describing their distinguishing characteristics. Examples of complete systems are introduced in an informal manner and mini-systems are described which illustrate the characteristics of concurrent systems. We examine issues relating to the specification of concurrent systems and review and compare existing methodologies designed for this purpose.

2.1 Fundamental aspects of concurrent systems

Many systems, both natural and artificial, are characterized by concurrent behaviour. Examples of such artificial systems include computer networks and distributed systems. This work takes simple concurrent systems as basic exemplars for more complex systems to which the methods developed herein may be applied.

The general properties of concurrent systems can be divided into two categories:

(1) dynamic properties (behaviour),

(2) static properties (e.g. the data types of objects).

Both of these categories are of importance, but we will concentrate on dynamic properties since they are characteristic of concurrent systems, whilst static properties are not. The dynamic properties of a concurrent system constitute its 'behaviour', comprising discrete events. In the case where the system is artificial and computation-based then its behaviour is the manifestation of the computations it comprises.

We take the view that events can occur *sequentially* or *simultaneously* — thus an observer of a system will be able to differentiate between those events which take place one after another and those which occur 'at the same time'. We require such an observer to be able to distinguish between different but simultaneous events, hence being capable of making simultaneous observations. Different observers may have different interpretations of the time-slice of an observation; our ideal observer's time slice is the minimum required to observe an event.

We make a fundamental distinction between *concurrency* and *parallelism* when discussing systems in which events can occur simultaneously:

- Two or more events are said to occur *concurrently* iff they are simultaneous and communicate with each other.
- Two or more events are said to occur in *parallel* iff they are simultaneous and do not communicate with each other.

In practice, most systems in which events occur simultaneously exhibit a mixture of parallelism and concurrency in that communication may not take place between all simultaneous events. We shall loosely refer to such systems as 'concurrent systems'.

2.1.1 Concurrent computation-based systems

The focus of this research is on concurrent computation-based systems. At the lowest level of behaviour in such systems are atomic computational steps, or atomic events. We abbreviate 'atomic computational step' to *atomic computation* when convenient and not ambiguous.

Definition 2.1 Each atomic computation is an operation on data and its execution is instantaneous. ■

We may represent an atomic computation diagrammatically by a rectangle and optionally associate a unique symbolic name with the computation. Figure 2.1 illustrates an atomic computation labelled with the symbol a.

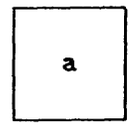

Figure 2.1: One atomic computation

Simultaneously occurring atomic computations are categorised as occurring either in parallel or concurrently. If two or more simultaneous atomic computations communicate with one another, then we say that they are *constrained* by one another. This communication may be merely agreement on a common value, or may involve exchange of data between the computations. We represent communication between atomic computations diagrammatically by linking them with a line "—" (agreement) or an arrow "⟶" (exchange of data). Atomic computations occurring in parallel are not connected by any link in our representation. Figure 2.2 illustrates a set of three simultaneous atomic computations. Computations a and b communicate by an agreement on a data value and therefore occur concurrently. Computation c occurs in parallel to a and b.

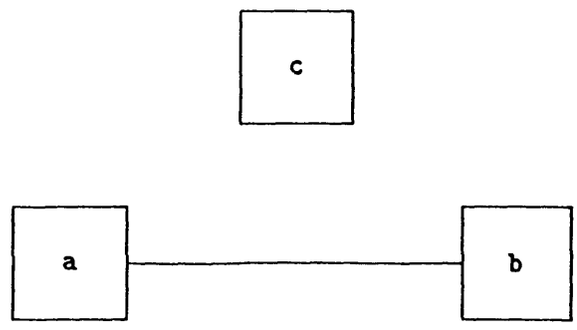

Figure 2.2: Simultaneous atomic computations

All communication in our scheme is synchronous:

Definition 2.2 In a synchronous system every member of a set of concurrent atomic computations is constrained by every other member of that set and no atomic computation may terminate while any computation with which it shares a constraint has not yet terminated. ■

We associate with each terminated computation a value drawn from a domain, hence permitting termination to be many-valued. One such domain contains only the values {SUCCESS, FAIL}, in which case we deem a terminated computation to have either *succeeded* or *failed*. It is this bi-valued domain of truth values that we consider in this research.

Definition 2.3 If any member of a set of simultaneously occurring atomic computations fails, then all the computations in that set fail. ■

Definition 2.4 The overall truth value associated with a set of simultaneously occurring computations is SUCCESS iff all the computations succeed, and FAIL iff any one of the computations fails. ■

Definition 2.5 A *sequence* of atomic computations is a totally ordered set of computations no two of which occur simultaneously. Topologically, the set is a chain with the bottom element $\perp$ being the first computation in the sequence and the top element $\top$ being the last in the sequence. The ordering relation is over time; each member of the set apart from one (the first) is constrained by an immediate predecessor in that it cannot be initiated until the predecessor has terminated. ■

In our informal portrayal we represent a sequence of atomic computations by connecting them with "$\Longrightarrow$" (Figure 2.3).

We further restrict the definition of sequence by requiring that a computation in a sequence can be initiated iff the preceding computation in the sequence succeeds. The overall truth value associated with a sequence of computations is SUCCESS iff every member of the sequence succeeds and FAIL iff any one of the computations fails. A sequence terminates either when every member of the sequence has succeeded or as soon as any member fails.

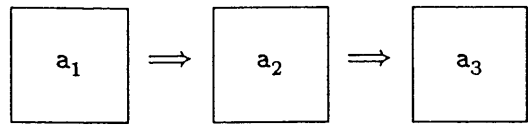

Figure 2.3: Sequential atomic computations

2.1.2 Processes and systems

We can, for convenience, describe a set of atomic computations as a *process* or a *system*.

Definition 2.6 A *process* comprises the executions of one or more atomic computations. ■

Definition 2.7 A *system* comprises one or more processes. ■

The atomic computations which constitute a process or system can be either simultaneous or sequential or both. A process can comprise other processes but cannot comprise systems whereas a system can comprise other systems or processes. We will represent processes and systems by *dashed rectangles* and, optionally, an associated symbolic name. Figure 2.4 depicts two processes A and B grouped together as a system C. Process A comprises two concurrent atomic computations a_1 and a_2 while B comprises the sequential atomic computations b_1, b_2 and b_3.

Definition 2.8 A process/system is said to *communicate* with another process/system iff communication takes place between atomic computations common to both processes/systems. ■

Definition 2.9 A process/system *completely synchronises* with another process/system iff every atomic computation in the first synchronises with one or more atomic computations in the second. ■

Definition 2.10 A system is *closed* if there is no communication between the computations it comprises and those of any other system. ■

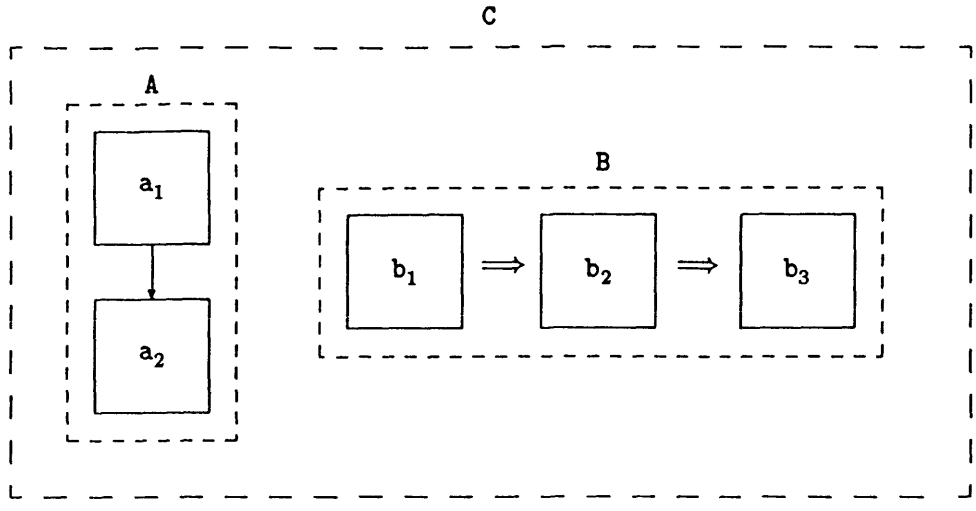

Figure 2.4: Processes and systems

Definition 2.11 A process terminates with SUCCESS iff every atomic computation that it comprises succeeds. ■

Definition 2.12 A system terminates with SUCCESS iff every process that it comprises terminates with SUCCESS. ■

2.1.3 Suspension, deadlock and livelock.

We recall that an atomic computation is an operation on data. The ability to perform such an operation may be dependent on the availability of data, subject to the definition of the operation. If the data is of the incorrect type for the operation to be performed, then the operation *fails*.

Definition 2.13 An atomic computation is in a *suspended state* iff there is insufficient data in that state for the computation to be performed. ■

Definition 2.14 An atomic computation is *executable* iff there is sufficient data for the computation to be performed. ■

The rules governing the suspension of sets of atomic computations are as follows:

Definition 2.15 A set of concurrent atomic computations is suspended iff any member of the set is suspended. ■

Since atomic computations which occur in parallel are not constrained by one another, then the suspension of a member of a set of parallel atomic computations does not affect the execution of any other members of that set.

Definition 2.16 A process is *suspended* iff all of its constituent atomic computations are suspended. ■

Definition 2.17 A closed system is *deadlocked* iff all of its constituent processes are suspended. ■

Definition 2.18 A closed system comprising n processes is *livelocked* iff m of its processes are suspended forever where $(0 < m < n)$, and at least one of its processes can be executed. ■

The following definitions are given informally. A process or system is *executing* if at least one atomic computation which it comprises is executable. A *non-terminating* process or system is one which comprises one or more infinite sequences of computations. A process or system which has not yet terminated may either be non-terminating or comprise one or more computations which are suspended.

2.1.4 Communication patterns

An important aspect of any model of communication is the classes of communication patterns that it admits. These classes can be broadly classified as:

- communication possible only on a one-to-one basis,
- communication on a one-to-many basis,
- communication on a many-to-many basis.

The inclusion of data transfer in a model of communication implies that there are producers (writers) and consumers (readers) in the model. We say that a consumer receives data from a producer in a communication which involves data transfer. The possible communication patterns are:

- **Pairwise:** one producer to one consumer,
- **Broadcast:** one producer to many consumers, or many producers of identical messages to one consumer,
- **Multiway:** N producers to M consumers, or M producers to N consumers.

2.1.5 Synchronous and asynchronous communication

'Real' concurrent systems may be based on totally synchronous, totally asynchronous or bounded asynchronous communication. We have previously defined a synchronous system as one in which all partners in a communication are mutually constrained (Definition 2.2). In a totally asynchronous system *consumers* are constrained by communication with producers but producers are unconstrained. In a bounded asynchronous system there is some buffering of data between producers and consumers; producers are unconstrained by consumers until the buffer is full. A programming language used to implement any of the three systems has to employ a blocking receive to implement consumer processes. However the language primitive required to implement a producer in a totally asynchronous system send is a non-blocking send whilst a blocking send is required to implement totally synchronous systems. There is a well-founded correspondence between the models of the three types of systems:

- asynchronous systems may be represented and implemented in totally synchronous systems by the interposition of unbounded buffers (queues) between producers and consumers,
- totally synchronous systems may be represented and implemented in asynchronous systems by a 'rendez-vous' technique, for example, using multiple channels or multiplex channels,

- a bounded asynchronous system may be represented and implemented in a totally synchronous system by the interposition of bounded buffers between producer and consumer.

Concurrent systems which are totally synchronous are more amenable to specification and analysis than asynchronous systems. This is because communication in an asynchronous system is effectively via unbounded buffers (queues) which are not part of the system specification. In synchronous systems *all* communication can be explicitly reasoned about. Asynchronous communication can be represented by synchronous communication if communicating partners are explicitly connected by queues (see above) permitting reasoning about the communication medium. The disadvantages of the added complexity thus introduced is outweighed by the advantage of being able to reason explicitly about system states, including any buffers between communicating partners.

2.1.6 Observational equivalence

Two or more processes are observationally equivalent iff the least quantum of their behaviour cannot be distinguished by an 'idealised' observer of the processes. Different interpretations of the terms 'least quantum' and 'distinguish' result in proposals for different types of observational equivalence (see [92]). The notion of equivalence for concurrent systems is discussed in detail in Chapters 4 and 6.

2.2 Illustrative mini-systems

In order to investigate the ways in which concurrent systems may be specified some illustrations of such systems are now presented. The examples are well-documented ones from the area of computer science and have been chosen both for their simplicity and the dynamic properties of concurrent systems which they illustrate. All the example systems are founded on synchronous communication in order to facilitate reasoning about states (see Section 2.1.5).

2.2.1 Synchronous communication

The most generic example of a mini system is that of two processes which synchronise in some way. In the simplest case each process consists of a single atomic computational step (Figure 2.5). A more complex computation occurs when each process is a sequence

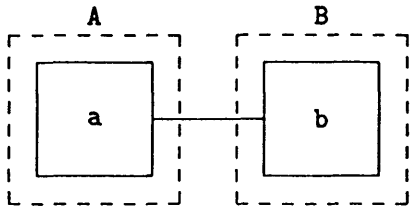

Figure 2.5: Two-way communication

of atomic computations (Figure 2.6), the computation unfolding with both processes progressing in lock-step synchronisation.

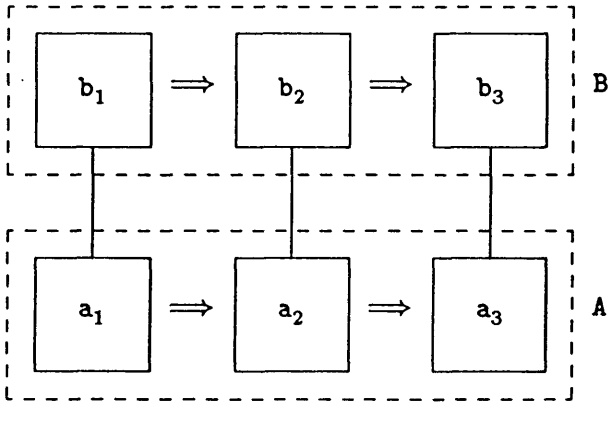

Figure 2.6: Lock-step synchronisation

The example may be extended by considering more than two processes, each of which synchronises with all the others, for example a set of three concurrent atomic computations a, b and c (Figure 2.7).

Since all communication is synchronous, the system in Figure 2.7 is equivalent to that in Figure 2.8 in which the communication path between a and c is not represented explicitly.

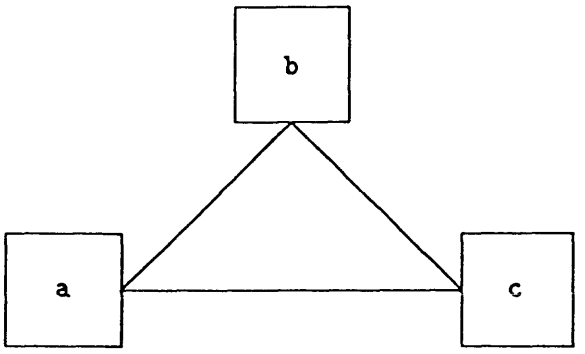

Figure 2.7: Three-way communication (i)

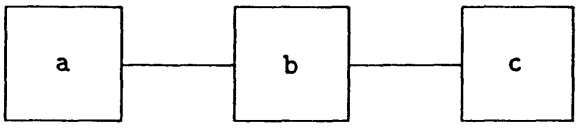

Figure 2.8: Three-way communication (ii)

2.2.2 Concurrent value passing: producers and consumers

Perhaps a more meaningful and familiar example is that of exchange of data between the communicating partners. The simplest case is that of two atomic computations a and b where, for example, a transmits a value v to b. In this case we identify a as the *producer* and b as the *consumer* in the communication, representing this by "a $\xrightarrow{v}$ b" (Figure 2.9). In the more complex case where both A and B are processes, each comprising a sequence $1 \ldots n$ of atomic computations where $\mathtt{A}_k$ transmits v_k to $\mathtt{B}_k$ $(1 \leq k \leq n)$, the behaviour of the two processes is such that A and B are in lock-step (Figure 2.10).

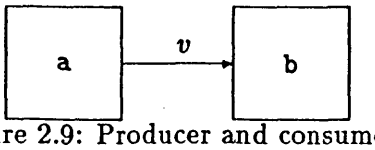

Figure 2.9: Producer and consumer

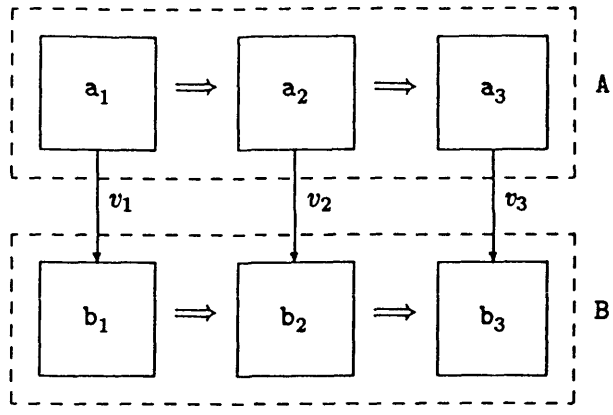

Figure 2.10: Producer and consumer (lock-step)

2.2.3 Sequential value passing

If two members a_k and a_{k+1} of a sequence of atomic computations have access to the same data store Γ in state S and the successful execution of A_k alters Γ by value v to give Γ' in a new state S', then we say that $\Gamma \overset{v}{\mapsto} \Gamma'$ and the execution of a_{k+1} utilises the new state of the store. Effectively the value v is passed sequentially from a_k to a_{k+1}. We represent this diagrammatically by "$a_k \overset{v}{\Longrightarrow} a_{k+1}$" (Figure 2.11).

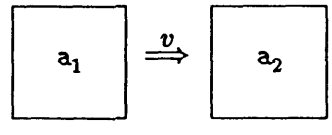

Figure 2.11: Sequential atomic computations (value passing)

2.2.4 Buffers and pipes

Figure 2.12 depicts a system in which atomic action a passes value v *indirectly* to atomic action c via process B which comprises the sequence of atomic computations b_1 and b_2. The effect of B is to impose a sequential ordering on the executions of a and c. If b_1 and b_2 were to be concurrent, then so would a and c. However if b_1 and b_2 were to execute in parallel, then a and c would not communicate with each other.

Process B is a *one-place buffer* which receives a message from a and transmits it to c:

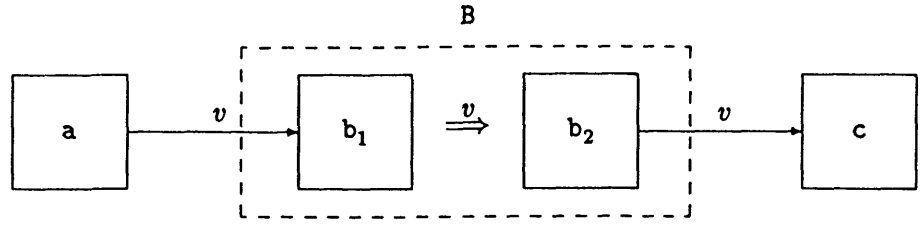

Figure 2.12: One-place buffer

- The two component computations b_1 and b_2 are composed sequentially,
- b_1 receives a data value from a and on termination passes that value to b_2,
- b_2 transmits the data value to c.

It is our intention to demonstrate in Chapter 4 that a buffer with a capacity of greater than one item can be constructed either by using a data-store which can be accessed by both b_1 and b_2, or by composing several one-place buffers in a linear process structure.

2.2.5 Queues

A *first-in-first-out* (FIFO) queue is effectively a buffer of unbounded capacity. Items are inserted at one end and removed from the other. The actions of insertion and removal can be performed simultaneously or in an interleaved manner, unless the queue is empty. Specification techniques which assume that all communication is synchronous use queues to describe asynchronous communication. Communication in networks and distributed systems is usually asynchronous, and can be represented in this manner. A *last-in-first-out* (LIFO) queue is an unbounded stack and access to such a structure comprises *interleaved* actions to add or remove items. A *double-ended queue* (DEQ) permits addition and removal of items at either end of the queue.

2.2.6 Transformers

Transformers are a class of systems which transform data whilst attempting to relay it. The transformation may alter the order of transmission of messages, corrupt them or lose them.

Ordering Interesting mini-systems can be constructed from queues and buffers which are neither fully FIFO nor LIFO. Transformers, unlike buffers, are active in that they may alter the data that they relay from producer to consumer. An example is an *expedited data queue*: a producer and a consumer are connected via a communication medium, represented by a queue. Messages are either ordinary or expedited. An expedited message always arrives before any ordinary messages that are still in the queue (i.e. not yet received) when the expedited message is sent. As a refinement there can be several levels of priority of expedited message.

Message content A *corrupting transmission medium* is an example of a system which transmits messages in a (possibly) corrupted form.

Message Loss A *lossy medium* does not guarantee that all messages which enter the medium are transmitted. Such a system can be modeled by a variation of the buffer system.

Filters are instances of transformers. They are required in a distributed computing environment where there are servers which provide resources and clients which use these resources. Message protocols govern the communication between clients and servers. A server may have access to sensitive information, accessible only to cleared users (clients). Protection for the server from access by unauthorised clients can be provided by a filter. To be effective, a message filter must satisfy four requirements:

(1) every message between client and server must be inspected by the filter,

(2) no sensitive information is allowed to pass out through the filter from a server to an uncleared client,

(3) no ill-formed message is allowed to pass between a client and the server,

(4) all legitimate messages must pass unimpeded between the client and server.

The first three are *access* requirements, whilst the last is a *service* requirement. Validation here means the inspection of a message by the filter. Filters must exhibit functional and performance transparency.

2.3 Issues in the specification of concurrent systems.

In this section we review and compare existing methodologies designed for the specification of concurrent systems.

A *specification language* is a notation for describing the set of observations that an idealised observer can make of the system specified. Different specification languages have different degrees of appropriateness to different specification tasks, as well as different properties in themselves. The terms *Formal Specification Language* (FSL) and *Formal Description Technique* (FDT) can be considered synonymous. An FDT has a precise mathematical basis, and its use:

- avoids ambiguity, while leaving room for implementation freedom,
- allows complex descriptions to be written succinctly without problems of misrepresentation,
- provides a basis for reasoning about specifications which can be analysed and checked for desirable properties (e.g. freedom from deadlock).

Consistency between specifications can be verified and the equivalence of specifications can be determined. This is important if the technique of stepwise refinement is to be adopted as an implementation methodology.

A basic issue in the specification of concurrent systems is how concurrency is represented in the formalism. One generic approach in reasoning about sets of concurrently occurring events is to treat these sets as sequences of interleaved atomic events. We review below some of the techniques which employ this approach, along with those which attempt to reason about concurrency directly.

2.4 Primitive process constructions – Milner's principles

The work of Robin Milner has been very influential in the field of the specification of concurrent systems. In [94] he proposes nine principles which should guide the choice of a

set of primitive constructions These constructions should be sufficiently rich to form the basis of useful practical tools, for example calculi, specification languages and programming languages. He does not directly define the concepts of process and event, but prefers to articulate the meaning of these terms by formulation of the principles. We reproduce below the principles in shortened form:

(1) An *interaction* among processes consists in their participation in a single atomic event. This precludes communication via shared memory, buffers, or by use of rendez-vous. It does not limit the number of processes which may participate in an interaction.

(2) Every event is an interaction among processes. This implies that the observable behaviour of a system consists entirely in its interaction with its environment which may be a human observer who – being also a process – may only inspect or observe the system by interacting with it.

The following seven principles describe constructors which yield complex processes from simple ones.

(3) The behaviour of any process constructor depends only upon the behaviours of the processes which it composes.

(4) *Conjunction* is a construction needed to impose synchronisation on a designated set of actions between all the members of a set of processes, permitting all other actions to occur freely.

(5) *Encapsulation* is needed to render a designated set of actions unobservable.

(6) *Disjunction* is a construction needed to force each of a number of processes to synchronise with any one of the others on a designated set of actions.

(7) *Renaming* is a construction needed in order to change the action names of a process.

(8) Constructions are needed for *sequential control*, rich enough to express a wide range of distributed processes.

(9) *Simultaneity* – the simultaneous occurrence of two actions is also an action.

2.5 Existing formalisms and languages

There are several categories of specification methods which are widely accepted for the description of concurrent systems, for example:

(1) Finite State Automata

(2) Abstract Machine Models

(3) Formal Languages

(4) Sequencing Expressions

(5) Petri Nets

(6) Buffer Histories

(7) Abstract Data Types

(8) Programs and Program Assertions

Such methods have traditionally developed from two viewpoints — programs and state machines. However, there is no fundamental difference between these and many specification techniques incorporate ideas from both.

The finite state machine model is motivated by the observation that protocols may be viewed as rules specifying the responses or outputs of a protocol 'machine' to each command or input. A closely related set of techniques is based on finite graph representations. Examples include communicating finite state machines, Communicating Sequential Processes (CSP) [58], a Calculus of Communicating Systems (CCS) [92], CIRCAL [91] and Petri Nets [102].

The program model is based on the premise that protocols and services are a class of information processing procedures, and that programming languages provide a means for describing such procedures. Examples of such languages include SDL [7] Estelle [67] and LOTOS [66].

The plethora of approaches reflects the fact that many of the languages have been developed for specific domains and also that there has been no consensus as to which simple

rules in a mathematical notation are sufficient to give precise meaning to specifications of concurrent systems.

We now take a more detailed look at some of the specification techniques mentioned above.

Communicating Sequential Processes - CSP

Developed by Hoare, and defined in [58], CSP is a mathematical theory of concurrency based on traces and refusals. A process is regarded as being a mathematical abstraction of the interactions between a system and its environment. General processes are described from this point of view, as are processes exhibiting concurrency, nondeterminism, communication and sequencing. All communication is treated as being synchronous in CSP, and explicit buffers are used if asynchronous systems need to be described. There may be more than two partners in any communication, and choice may be nondeterministic or controlled by the environment. The programming language occam [65] is based on CSP.

A Calculus of Communicating Systems - CCS

CCS was developed by Robin Milner [92], and is an algebraic technique based on the analysis of the interleaving of atomic events using observations and equivalences. The technique does not admit simultaneous events and is restrictive in ways which include permitting communication between two parties only. Several extensions of CCS have been proposed, for example ECCS [39], SCCS [93] and CHOCS [128]. Milner himself has proposed general behavioural laws governing concurrent systems (see [94] above) which can be used to extend CCS.

CIRCAL

CIRCAL, developed by George Milne [91] is a mathematical calculus designed to describe and analyse concurrent systems, whether hardware or software. Unlike CCS and CSP, the calculus permits descriptions of both asynchronous and simultaneous behaviour, including multiple synchronisation using abstraction to allow modeling of a system at different levels of behaviour. An operational semantics, called *acceptance* semantics, is given to CIRCAL

syntax where meaning is conferred in terms of active experimentation.

Petri Nets

Petri nets [102] permit the clear representation of concurrency without having to resort to interleaving. Net theory also permits the description of non-determinism, but the arbitrary order of occurrence of causally independent events does not count as non-determinism, and representation of data flow as opposed to pure synchronisation is poorly handled. Milner [94] suggests that petri net theory may permit the categorisation of computable processes as "exactly those recursively acyclic nets which obey certain natural conditions, for example that the in-degree and out-degree of nodes should be finite."

Estelle

Estelle [67] is based on Pascal with extensions to describe finite state machines. Together with LOTOS (see below), ESTELLE is one of the two formal description techniques developed under the auspices of the International Standards Organisation (ISO) for the formal specification of open distributed systems, and in particular those related to the the Open Systems Interconnection (OSI) computer network architecture. An Estelle specification defines a distributed system as a hierarchy of state machines which communicate by exchanging messages through bi-directional channels connecting their communication ports. Messages are queued at either end of a channel. The incorporation of unbounded FIFO queues into the language permits the modeling of synchronous or asynchronous communication.

LOTOS

The Language of Temporal Ordering Specification (LOTOS) [66] is a description technique based on process algebras, specifically CCS and to a lesser extent CSP. It incorporates the Abstract Data Type (ADT) ACT-ONE [37] for the description of data types (sorts).

Being based closely on CCS, a specification in LOTOS permits a hierarchy of process definitions where a process is an entity capable of performing internal and unobservable

actions as well as being able to interact with its environment, which is itself composed of other processes. Complex interactions between processes are built up out of elementary units of synchronisation, atomic events. Every event implies process synchronisation and may involve an exchange of data. Communication patterns may be multi-partner, not just one-to-one, and an event occurs at an interaction point termed a gate.

Observation, itself regarded as an interaction, is central to the LOTOS model of communicating systems, and thus when a process performs an observable action there is an assumed interaction between the process and the observer. LOTOS specifications are analysed in terms of equivalence theory whose foundation is inherited from CCS.

SDL

SDL [7] is based on the extended finite state machine model, with two concrete syntaxes, one graphical and one textual. It is supplemented by the ADT ACT-ONE. This combination is supported by a well-defined formal semantics. SDL has constructs to represent structures, behaviours, interfaces and communication links, as well as abstraction, module encapsulation and refinement.

2.6 Logics for specification of concurrent systems

2.6.1 Standard first order logic

Kowalski and Sergot [77] have developed the *event calculus* to reason about events and time within a logic programming framework based on standard logic. The intended applications of the event calculus are the updating of databases and narrative understanding. Its use avoids the frame problem inherent in the situational calculus [90] which deals with global states. The notion of event is taken to be more primitive than that of time and both are represented explicitly by means of Horn clauses augmented with negation by failure; however the calculus is neutral with respect to whether events have a duration or not. The formalism can represent events with unknown times as well as events which are partially ordered and simultaneous, but its use to reason about concurrent systems has yet to be

demonstrated.

Although little work has been undertaken using first order predicate logic to *describe* the dynamic behaviour of concurrent systems, logic languages exist which enable the *implementation* of such systems. Examples of concurrent logic programming languages are Parlog [51], Concurrent Prolog [116] and Guarded Horn Clauses [131]. However we have shown in previous research [49] that these languages are not generally suited to the task of the *specification* of such systems. The operational semantics of these languages do not permit guarded output, and lack synchronisation of producers and consumers as a language primitive. Recent proposals have been made by Saraswat [114] and Shapiro [120] for the design of concurrent logic languages which incorporate some or all of these desirable features, but problems with their efficient implementation have prevented their use.

2.6.2 Temporal Logic

Temporal logics are logics which permit reasoning about the ordering of events and their use in the specification of concurrent systems has been described by Gabbay [43, 44] and Manna and Pnueli [87]. Some work has been done on the use of algebraic ordering in specifications in temporal logic and Allen has explored the possibility of dealing with multiple agents in such specifications [2, 3]. However, standard first order logic has a greater flexibility of expression than the temporal logics, and there are proof procedures which have been developed for standard first order logic and for logic programming in particular. Expressive power is gained by treating time and events explicitly rather than implicitly through the use of natural, but weak modal operators for notions such as 'future', 'since' and 'while'. The advantages of modal and temporal logics over standard logic lie in their greater conciseness and the representation level that they offer to the user.

2.7 Summary

Processes and systems consist of sets of atomic computations; 'concurrent systems' comprise atomic computations whose behaviour can occur simultaneously. A concurrent sys-

tem can also, but not exclusively, exhibit sequential behaviour. We make a further distinction between two types of simultaneous computations — those which execute independently of, i.e. in parallel to, other computations and those whose execution is dependent on, i.e. concurrent with, the execution of other computations. Concurrency is characterized by suspension as well as success or failure. The example systems described in this chapter exhibit synchronous communication; we elect to describe asynchronous communication as synchronous communication which takes place via an unbounded buffer. A number of formalisms developed for the description of concurrent systems have been reviewed in this chapter. We believe that first order logic is a suitable formalism for the specification of concurrent systems and that it should be possible to derive executable concurrent logic programs from such descriptions.

Chapter 3

Specifications in logic of concurrent systems

3.1 Introduction

This chapter describes a method for specifying concurrent systems, using first order logic. The method takes an extrinsic view of such systems and is based on reasoning about the sets of observations that can be made about a system. We assume that bindings are made to write-once variables during the execution of a system and that shared variables are the means of communication between concurrent processes. The write-once nature of these variables enables us to reason about them as variables within a theory of first order logic, treating variable assignment as an instance of unification. The structures to which the variables are bound are logical terms and we assume that an observer of such a system is capable of observing the binding states of these variables. The observations are members of a partially ordered set with a bottom element representing the initial unbound state of the observable variables.

The sets of observations can also be described as directed acyclic graphs.

Each path through the graph from the minimum vertex to a maximum vertex comprises the observations of one computation. The representation of the observations of a system as a graph facilitates reasoning which distinguishes between sequential and

concurrent events.

Natural language specifications of some small illustrative systems are related to descriptions in first order logic of the observations of the systems. The first order logic sentences can be transformed into Horn clauses whose execution on a logic interpreter results in the binding of variables in the query to data structures which represent the sets of observations of the specified system. Such sets can also be characterized by Horn clause descriptions which relate the members of the set to its maximal elements. These secondary descriptions form the basis of logic programs which when executed on an interpreter employing a suitable computational rule exhibit the behaviour described by the specification from which they were derived. The specifications in logic are amenable to formal analysis and can be used to generate conformance test suites. The design of a logic programming language and the description of a logic interpreter for it is the subject of the next chapter.

In this chapter we refer to example systems characterized by behaviour which is *reactive* (responsive to an environment), *concurrent* and *dynamic*. These systems are simple enough to permit easy understanding on an informal basis and yet illustrate the concepts introduced.

3.3 First Order Logic

Our proposed method for specifying and reasoning about concurrent systems is based on first order logic. We refer the reader to [20, 122] for thorough treatments of symbolic logic. In this section we present the concepts that are required for an understanding of our approach to the specification of concurrent systems in logic.

There are two aspects to first order logic, syntax and semantics. We first consider syntax. A first-order language is a language in which the symbols and formulae are defined using an alphabet consisting of variables, constants, function symbols, predicate symbols, connectives, quantifiers and punctuation symbols. We employ the following syntax for representing statements in logic:

(1) variable symbols: strings starting with an *uppercase* alphabetic character.

(2) constant symbols: strings starting with a *lowercase* alphabetic character, or a numeric character.

(3) function symbols: strings starting with a *lowercase* alphabetic character, or the empty string (see page 55).

(4) predicate symbols: strings starting with a *lowercase* alphabetic character

(5) the connectives $\wedge$ $\vee$ $\leftarrow$ $\leftrightarrow$ $\rightarrow$ $\neg$

(6) the quantifiers $\exists$ $\forall$

(7) punctuation symbols ",", "(" ")"

Function symbols may be of various arities and may be written prefix, infix or postfix where appropriate. Constants are function symbols of arity zero. Predicate symbols may be of various arities and may similarly be written prefix, infix or postfix. We adopt the following precedence hierarchy to avoid the heavy use of brackets in formulae:

$$\neg \; \forall \; \exists$$
$$\vee$$
$$\wedge$$
$$\leftarrow \; \rightarrow \; \leftrightarrow$$

We informally describe the meanings of the connectives as:

$\neg$	negation
$\wedge$	conjunction (and)
$\vee$	disjunction (or)
$\leftarrow$	implication as in "q $\leftarrow$ p" means '*if* p *then* q'
$\rightarrow$	implication, as in "p $\rightarrow$ q" means '*if* p *then* q'
$\leftrightarrow$	equivalence

"$\exists$ " is the existential quantifier so that "$\exists$ X" means 'there exists an X'. The universal quantifier is "$\forall$ " so that "$\forall$ X" means 'for all X'.

Definition 3.16 *Terms* are defined recursively as follows:

(1) a constant is a term

(2) a variable is a term

(3) If f is an n-place function symbol, and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

(4) All terms are generated by the above rules

■

We may for convenience represent the term $f(t_1, \ldots, t_n)$ where f is the empty string by the n-tuple $\langle t_1, \ldots, t_n \rangle$.

Definition 3.17 An *atom* (atomic formula) is an expression of the form $p(t_1, \ldots, t_n)$ where p is an n-place predicate symbol and $t_1 \ldots t_n$ are terms. ■

Definition 3.18 A *literal* is an atom or the negation of an atom. A *positive literal* is an atom and a *negative literal* is the negation of an atom. ■

Definition 3.19 A *well-formed formula*, or *formula* for short, is defined as follows:

(1) An atom is a formula.

(2) If F and G are formulae then so are ($\neg$F), (F $\wedge$ G), (F $\leftarrow$ G), (F $\rightarrow$ G), (F $\leftrightarrow$ G).

(3) If F is a formula and X is a variable then ($\forall$X F) and ($\exists$X F) are formulae.

■

The scope of a quantifier occurring in a formula is the formula to which the quantifier applies. We may use brackets to clarify the formula in question. E.g. the scope of the universal quantifier in the formula ($\forall$X)(p(X) $\rightarrow$ s(X)) is (p(X) $\rightarrow$ s(X)). We may omit the brackets round quantifiers when no ambiguity would so arise.

Definition 3.20 An occurrence of a variable is *bound* iff the occurrence is within the scope of a quantifier employing that variable, or is the occurrence in that quantifier. An occurrence of a variable in a formula is *free* iff this occurrence of the variable is not bound.

■

Definition 3.21 A variable is bound in a formula if at least one occurrence of it is bound. A variable is free in a formula if at least one occurrence of it is free in the formula. ■

Definition 3.22 A formula is closed iff it contains no free variables. Such a formula is called a *sentence*. ■

Definition 3.23 The first order language given by an alphabet consists of the set of all sentences which can be constructed from the symbols of the alphabet. ■

We briefly introduce some more terminology which will be useful regarding the form of logic sentences.

Definition 3.24 A *clause* is a sentence of the form

$$\forall X_1 \ldots \forall X_n (L_1 \vee \ldots \vee L_m)$$

where $X_1,\ldots,X_n$ are all the variables occurring in the disjunction of the literals $L_1 \vee \ldots \vee L_m$.

■

We may write a clause

$$\forall X_1 \ldots \forall X_n \, (P_1 \vee \ldots \vee P_k \vee \neg Q_1 \vee \ldots \vee \neg Q_j)$$

where $P_1 \ldots P_k$, $Q_1 \ldots Q_j$ are atoms as the equivalent form

$$\forall X_1 \ldots \forall X_n \, (P_1 \vee \ldots \vee P_k \leftarrow Q_1 \wedge \ldots \wedge Q_j)$$

using the following equivalences:

$$(\neg Q_1 \vee \ldots \vee \neg Q_j) \equiv \neg(Q_1 \wedge \ldots \wedge Q_j)$$
$$A \vee \neg B \equiv A \leftarrow B$$

and then by convention omit the universal quantifiers:

$$P_1 \vee \ldots \vee P_k \leftarrow Q_1 \wedge \ldots \wedge Q_j$$

Definition 3.25 A *Horn clause* is a clause containing at most one positive literal. ■

The following clauses are Horn clauses:

$$P \leftarrow Q_1 \wedge \ldots \wedge Q_j$$
$$P \leftarrow$$
$$\leftarrow Q_1 \wedge \ldots \wedge Q_j$$

By convention we may omit the "$\leftarrow$" in the unit clause $P \leftarrow$.

Definition 3.26 A *definite program clause* is a clause of the form

$$P \leftarrow Q_1 \wedge \ldots \wedge Q_j \qquad (j \geq 0)$$

where Q_1 , ..., Q_j are positive literals.

A *definite program* is a finite set of definite program clauses. ■

Definition 3.27 A definite goal is a clause of the form

$$\leftarrow Q_1 \wedge \ldots \wedge Q_j \qquad (j \geq 1)$$

where $Q_1, \ldots, Q_j$ are positive literals. ■

Thus a Horn clause is either a definite program clause or a definite goal.

We further extend our definitions to permit negative literals in programs and goals:

Definition 3.28 A *normal program clause* is a clause of the form

$$P \leftarrow Q_1 \wedge \ldots \wedge Q_j \qquad (j \geq 0)$$

where $Q_1, \ldots, Q_j$ are literals.

A *normal program* is a finite set of normal program clauses. ■

Definition 3.29 A normal goal is a clause of the form

$$\leftarrow Q_1 \wedge \ldots \wedge Q_j \qquad (j \geq 1)$$

where $Q_1, \ldots, Q_j$ are literals. ■

Interpretations and models

We have previously defined the syntax of the first order language. In order to be able to discuss the truth or falsity of a formula we need to attach meanings to each of the symbols in the formula. An *interpretation* specifies the meaning for each symbol in a formula and consists of a non-empty domain of discourse over which the variables range, the assignment to each constant symbol an element of the domain, the assignment to each function symbol of a mapping on the domain and the assignment to each predicate symbol of a relation on the domain. An interpretation in which a formula expresses a true statement is called a *model.* The intended interpretation, which should also be a model, gives the meaning of the symbols in a formula.

The formulae which are true in every interpretation of each of the axioms of a theory are the theorems of that theory. Theorems are logical consequences of the axioms. *Resolution* is an inference rule which can be used to demonstrate that a particular formula is a logical consequence of a set of axioms. We discuss resolution theorem proving with respect to

Horn clauses in Chapter 4.

3.4 Observations of logic programming systems

Although the demonstration of logical consequence is the goal of theorem proving, from the computational point of view we are more interested in the bindings made to the logical variables of the theorem during the construction of the proof of the validity of that theorem. If we regard the axioms and theorem as a logic program which is executed during the proof process then these bindings constitute the output of the program.

A logic programming system can be viewed as a black box for computing bindings. The internal workings of such a system should be invisible to an observer whose only interest in such system is its input-output behaviour. We have chosen to model concurrent systems as logic programming systems during whose execution variables are bound. Our assumption is that an observer can detect the *incremental* bindings that are made to variables as the system executes. Communication in such a system occurs via bindings made to *shared* variables. Bindings may be made to internal variables during the execution of a system, but these are not detectable by the observer.

The representation of communication by incremental bindings made to shared variables was first suggested in a logic programming context by van Emden and de Filho [135] and was based on ideas first discussed by Gilles Kahn and David MacQueen [69]. Concurrent logic programming languages implement communication in this way.

Other models of communication

The specification techniques reviewed in Chapter 2 represent communication in a variety of ways. Milner [94] has proposed that 'an interaction among processes consists of their participation in a single atomic event' and claims that communication which occurs via shared memory, buffers or a rendez-vous does not satisfy this description. CCS [92] and CSP [58] are both based on this abstract notion of communication, as is the formal description technique LOTOS [66] in which events are atomic in that they occur instantaneously, and occur at an action point, or *gate*. The abstract nature of events and gates, linked with the

absence of a computational model for LOTOS makes the derivation of implementations from specifications in LOTOS very difficult, as we have reported in previous work [48].

3.5 Unification

The approach that we take to the description of a concurrent system is to reason about the set of observations that can be made of the system during its execution. An observation is regarded as being the binding state of one or more variables. We now discuss in greater detail the nature of such variables, the manner in which bindings are described and the structure of the sets of observations.

3.5.1 The unification algorithm

We initially consider the basic atomic computation in concurrent systems to be *unification* over terms. This is an instance of constraint evaluation, a more general computational scheme, discussed by Maher in [86] and reviewed by Clark in [25]. Our method of specifying systems can be extended to encompass this. Unification combines checking and non-destructive assignment in one action. The unification algorithm was proposed by Robinson in [105] (see Lassez [80] for a detailed discussion of unification).

We regard a unifier σ for a set $\{E_1, \ldots, E_k\}$ of expressions to be a *most general unifier* iff for each unifier θ for the set there is a substitution λ such that $\theta = \sigma \circ \lambda$. The unification algorithm below for finding a most general unifier for a finite set of nonempty expressions is adapted from that given in Chang and Lee [20]. We first define the *disagreement set* for expressions:

Definition 3.30 The disagreement set of a nonempty set W of expressions is the set obtained by locating the first symbol (from the left) at which not all the expressions in W have the exactly the same symbol and extracting from each member of W the subexpression that begins at that position. ■

Definition 3.31 The unification algorithm is defined as follows:

Step 1 Set $k = 0$, $W_k = W$ and $\sigma_k = \varepsilon$.

Step 2 If W_k is a singleton then stop: σ_k is a most general unifier for W. Otherwise, find the disagreement set D_k of W_k.

Step 3 If there exist elements V_k and t_k in D_k such that V_k is a variable that does not occur in t_k then go to Step 4. Otherwise stop; W is not unifiable.

Step 4 Let $\sigma_{k+1} = \sigma_k\{V_k/t_k\}$ and $W_{k+1} = W_k\{V_k/t_k\}$.

Step 5 Set $k = k + 1$ and go to Step 2.

■

A most general unifier is thus a set of replacements of the form $\{V_1/t_1, \ldots, V_n/t_n\}$ where each V_k is a distinct variable and t_k a term which may contain variables, with the proviso that the replacements do not determine directly or indirectly the assignments to a variable of a term that strictly contains that variable (the 'occurs check').

In our model of concurrent systems we are interested in the *binding history* of variables, that is the set of all those assignments generated by unification which contribute to the binding of terms to variables during the execution of a system.

Assumption 3.1 Unification is *atomic* in a concurrent environment.

All calls to unification of a common (shared) variable must be treated as one atomic action and thus all partners in a unification to a common variable are mutually constrained.

We may write *unification* for *atomic unification* for brevity. In our portrayal of concurrent systems using logic we use the infix predicate symbol $=/2$ as the name for the unification relation. Thus we may write $t_1 = t_2$ as an atomic computation where t_1 and t_2 are terms. Additionally we assume the existence of a relation unify/3 s.t. if unify(t_1, t_2, S) holds then S stands for the mgu of t_1 and t_2, and is a data structure representing the set $\{V_1/t_1, \ldots, V_n/t_n\}$ where $V_1 \ldots V_n$ are the variables in t_1 and t_2.

In order to reason in some meta-language about the binding states of variables in some object system, we assume that distinct system variables are represented by distinct tuples

in the meta-language. For example the variable V in a system S will be represented by the tuple $(V'), written $V', where V' is a constant representing the variable V and $ is a unique prefix function symbol. However when no ambiguity arises we will represent a variable by its name alone.

3.6 Communication and unification

We now discuss the behaviour of concurrent systems within a framework of first order logic.

Assumption 3.2 Communication in a concurrent system can be represented as bindings made to shared logic variables.

We regard communication as taking place during *unification*, which we hold to be atomic (Assumption 3.1). The effect of this is to prohibit any partner in a communication from 'running ahead' of any of the other participants in the unification. If the unification fails, then all the participating processes fail. We extend our pictorial representation of atomic computations in the following manner: the bindings that are made as a result of an atomic computation are associated with each rectangle representing that computation (Figure 3.4).

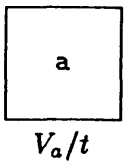

Figure 3.4: One atomic computation (unification)

Simultaneous atomic computations are represented *textually* in our description by their composition using the "$\wedge$" operator. For example if atomic actions a and b are the unifications $V_a = t_a$ and $V_b = t_b$ respectively where V_a and V_b are variables and t_a and t_b are terms, then the simultaneous occurrence of a and b is $(V_a = t_a \wedge V_b = t_b)$.

Definition 3.32 The *variable set* of a term/atom is the set of distinct variables which the term/atom contains. ■

We denote the variable set of a term t by $v(t)$, and define this set by:

(1) if the term is a variable V, then $v(V) = \{V\}$.

(2) if the term is a constant c, then $v(c) = \emptyset$.

(3) if the term is $f(t_1, \ldots, t_n)$, then $v(f(t_1, \ldots, t_n)) = \bigcup_{k=1}^{k=n} v(t_k)$.

Similarly, the variable set of an atom $a(t_1, \ldots, t_n)$ is $\bigcup_{k=1}^{k=n} v(t_k)$, denoted by $v(a(t_1, \ldots, t_n))$. The variable set of a *proposition* (an atom with no arguments) is the empty set.

Definition 3.33 Two unifications a and b occur in *parallel* iff they do not share any variables. i.e. $v(a) \cap v(b) = \emptyset$

■

Consider the simultaneous computations $V_a = t_a \wedge V_b = t_b$. If V_a and V_b are distinct variables and t_a and t_b do not have any variables in common, i.e. $v(t_a) \cap v(t_b) = \emptyset$, then the above composition is that of two *parallel* atomic actions, with the resultant binding set: $\{V_a/t_a, V_b/t_b\}$. (Figure 3.5).

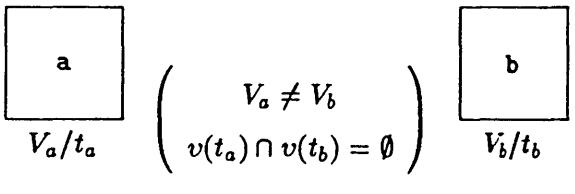

Figure 3.5: Parallel unifications

Definition 3.34 Two unifications a and b occur *concurrently* if they share one or more variables. i.e. $v(a) \cap v(b) \neq \emptyset$ ■

For example, $V = t_a \wedge V = t_b$ are two concurrent unifications which if successful result in V being bound to the mgu of t_a and t_b i.e. $V/\{mgu(t_a, t_b)\}$. Variables in the terms t_a and t_b may also be bound as a result. An instance of concurrent atomic unification is:

$$(V = f(p, Y) \wedge V = f(X, q))$$

(illustrated in Figure 3.6) resulting in the binding set

$$\{V/f(p,q), X/p, Y/q\}$$

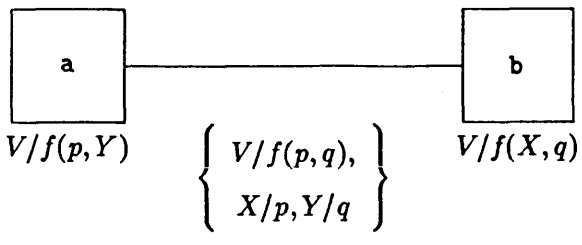

Figure 3.6: Concurrent unifications

Assumption 3.3 The simultaneous occurrence of more than one instance of the same unification is indistinguishable from one occurrence of that unification. ■

Thus $(V = t \wedge V = t)$ is equivalent to $V = t$, and in general $(\bigwedge_1^n t_1 = t_2)$ is equivalent to $(t_1 = t_2)$ where $\bigwedge_1^n t_1 = t_2$ is the conjunctions of n identical instances of the call $t_1 = t_2$.

The observations that can be made of a system are represented by the *bindings* made to variables as a result of unifications. In our model all communication is synchronous and is represented by bindings made to shared logic variables. While a process is executing incremental instantiations may be made to these shared logic variables until the computation terminates. The variables will not necessarily be ground in their final states.

We have previously defined a *sequence* of atomic computations (Definition 2.5), and we informally introduce the operator "&" to describe sequencing of atomic computations. Two sequential atomic computations a and b are represented by a&b where a occurs before b. "&" represents logical conjunction as does "$\wedge$". An operational semantics for both operators is given in Chapter 4.

3.6.1 Computations, paths and states

The *execution* of a system or process comprises one or more computations. Graph theory provides a useful basis for describing these sets of computations. In this section we describe terminating systems (finite computations) since our reasoning is based on *finite* graphs.

Definition 3.36a A *state* S is a tuple $\langle G, \theta \rangle$ where G is a goal and θ is a substitution. θ is empty in the initial state of a system. ■

Definition 3.36b A computation is any derivation beginning from an initial state which can be generated during the execution of a system. There is only one initial state of a system. ■

A computation can be represented as a sequence of state changes

$$S_0 \longrightarrow S_1 \longrightarrow S_2 \longrightarrow \ldots$$

where S_0 is the initial state. A *terminating* computation can be represented by

$$S_0 \longrightarrow S_1 \longrightarrow S_2 \longrightarrow \ldots \longrightarrow S_n$$

where S_0 is the initial state and S_n the final state. We may also represent the terminating computation above as the graph $G = (X, U)$ where

$X = \{S_0, S_1, S_2, \ldots, S_n\}$

$U = \{(S_0, S_1), (S_1, S_2), \ldots, (S_{n-1}, S_n)\}$

Such a graph comprises just one path

$$[(S_0, S_1), (S_1, S_2), \ldots, (S_{n-1}, S_n)]$$

The order relation on the graph is defined by U, since for every (S_i, S_j) in U we may write $S_i \prec S_j$.

A system may terminate in different ways, or may reach the same state in a variety of ways, resulting in more than one path through the graph. Each path represents one computation. The graph is

directed since $(S_i, S_j) \in U$ means that state S_j is the next state after state S_i,

acyclic since once a state has been reached, it can never be returned to.

3.6.2 Channels

We now look more closely at the data structures to which variables can be bound during a computation. The bindings may occur incrementally and the binding states of a variable can be depicted as a directed acyclic graph. In this section we introduce the use of first order logic to describe such graphs.

Definition 3.37 A *channel* is a collection of states of a variable that occur during one or more computations, i.e. the set of instantiations that can be made to that variable. ■

We will refer to such a variable as a *channel variable*, although all variables may be regarded as such. The set is partially ordered by a prefix relation over the data structure to which the channel variable becomes bound. This relation which we shall denote by $\leq$ is reflexive, antisymmetric and transitive. X $\leq$ Y means that Y is an instance of X, or X is equal to Y. The instantiation of a channel variable is effected by a process, and thus the channel can be described as a *graph* ordered by $\leq$, through which there are one or more paths, each representing a distinct computation (see Section 3.6.1).

Variables are bound during unification to logic terms. The topology of a complex term is a *tree*. Subtrees in a data structure can be instantiated in parallel or sequentially. If a channel variable is bound to a complex term, an element in the channel poset can have more than one successor. Thus the channel {V, f(A,B), f(p,B), f(A,q), f(p,q)} describes the possible binding states of channel variable V with final binding state V/f(p,q). The channel can be more comprehensively described by the directed acylic graph $G = (X, U)$ where

$X = \{ V , f(A,B) , f(p,B) , f(A,q) , f(p,q) \}$

$U = \{ (V, f(A,B)) , (f(A,B) , f(p,B)) , (f(A,B), f(A,q)), (f(A,B), f(p,q)) ,$
$(f(p,B), f(p,q)) , (f(A,q), f(p,q)) \}$

The predecessor relation $\prec$ on X is described by:

$\{ V \prec f(A,B) , f(A,B) \prec f(p,B) , f(A,B) \prec f(A,q) , f(A,B) \prec f(p,q) ,$
$f(p,B) \prec f(p,q) , f(A,q) \prec f(p,q) \}$

Figure 3.7 is a graph diagram of the channel. The nodes are labelled with the state of the channel variable, and the arcs are labelled with the bindings made during the transition from one state to the next.

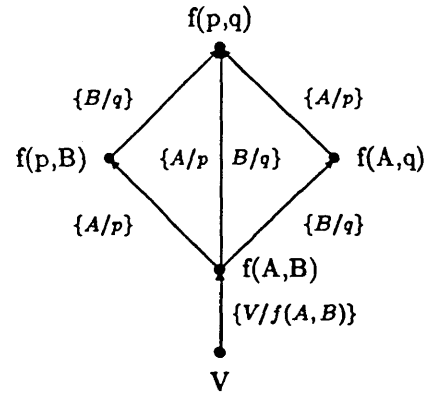

Figure 3.7: Representation of a channel, final state V/f(p,q)

We define $\leq$ using $\prec$ using first order logic:

$$X \leq Y \leftrightarrow (X = Y) \vee (X \prec Y) \vee \exists Z\, (X \prec Z \wedge Z \leq Y)$$

The path between two nodes A and B in a graph $G = (X, U)$ can be described as a partial list using the infix binary functor "." (lists are discussed in more detail in Section 3.6.3).

path(A,B,U,P) $\leftrightarrow$
((A,B) $\in$ U $\wedge$ P=A.B) $\vee$
$\exists$ C,P′ ((A,C) $\in$ U $\wedge$ P=A.P′ $\wedge$ path(C,B,U,P′))

Definition 3.37b If X and X' are terms each of which represents a state of a channel variable, then $X \prec X'$ iff X' is derived from X either by instantiating at least one variable in the variable set of X to a term $f(v_1, \ldots, v_n)$ where $v_1 \ldots v_n$ are variables, or by binding X to a constant. ■

Since we can characterize a graph by its order relation we may also define a path from A to B in a graph of $\prec$ by

path(A,B,P) $\leftrightarrow$

(A $\prec$ B $\wedge$ P=A.B) $\vee$

$\exists$ C,P'(A $\prec$ $\wedge$ C P=A.P' $\wedge$ path(C,B,P'))

where $\prec/2$ is defined by:

X $\prec$ Y $\leftrightarrow$ (X=\$V $\wedge$ Y=f(\$A,\$B)) $\vee$ ... $\vee$ (X=f(p,\$B) $\wedge$ Y=f(p,q))

Note that we use \$V to represent the name of the variable V.

Any channel can be described by a directed acyclic graph.

The set of states described by such a graph can be partitioned using node ranks. The definition of a node rank set for an acyclic directed graph $G = (X, U)$ is given below, where $\Gamma^-(x_i)$ is the set of predecessors of node x_i, N_p is a set of nodes with no predecessors, of rank p.

$$N_p = \left\{ x_i \in X - \bigcup_{k=0}^{p-1} N_k \;\middle|\; \Gamma^-(x_i) \subseteq \bigcup_{k=0}^{p-1} N_k \right\}$$

where q is the smallest integer such that

$$X = \bigcup_{k=0}^{q} N_k \quad , \quad \bigcap_{k=0}^{q} N_k = \emptyset$$

Our translation into first order logic of the partitioning is:

Channel (1)

channel(X,K) $\leftrightarrow$

$\forall$ N,X' ((nopreds(N,X) $\wedge$ $\neg$(seteq(N, X)) $\wedge$ union(N,X',X) $\wedge$

channel(X',K') $\wedge$ K=s(K')) $\vee$

nopreds(N,X) $\wedge$ seteq(N,X) $\wedge$ K=0)

nopreds(N,X) $\leftrightarrow \forall$ A,B((A $\in$ X $\wedge$ (B $\prec$ A $\rightarrow$ B $\notin$ X)) $\leftrightarrow$ A $\in$ N)

union(X,Y,Z) $\leftrightarrow \forall$ U (U $\in$ Z $\leftrightarrow$ U $\in$ X $\vee$ U $\in$ Y)

seteq(A, B) $\leftrightarrow \forall$ X (X $\in$ A $\leftrightarrow$ X $\in$ B)

In the above first order description X is a channel, N is a set rank K of nodes without predecessors (i.e. N_K). K is a structure s(s(...s(0) ...)) representing a natural number k by $k = s^k(0)$, and the final value of k is the smallest integer q such that

$$X - \bigcup_{k=0}^{q} N_k = \emptyset$$

In the following, we are not interested in the ranks themselves, but in the composition of the rank set. We rewrite the above definition of a channel, explicitly reasoning about the rank sets, the bottom element void of the channel and the set of final states[1] of the channel variable.

Channel (2)

channel(Ts,X) $\leftrightarrow$

$\exists$ X',Bs (nopreds(Bs,X) $\wedge$ $\neg$(seteq(Ts, X)) $\wedge$ union(Bs,X',X) $\wedge$ channel(Ts,X')) $\vee$
seteq(Ts, X)

Another way of expressing the channel in logic is to relate the set of states with no predecessors in one node rank with the corresponding set in the next higher node rank.

Channel (3)

channel(Ts,X) $\leftrightarrow$ channel(Ts,X,{void})

channel(Ts,X,Bs) $\leftrightarrow$

$\exists$ X', Bs' ($\neg$(seteq(Ts, Bs)) $\wedge$ next-rankset(Bs, Bs')
$\wedge$ union(Bs,X',X) $\wedge$ channel(Ts,X,Bs')) $\vee$
(seteq(Ts, X) $\wedge$ seteq(Ts, Bs))

next-rankset(P,Q) $\leftrightarrow$

$\forall$ A,B,C (A $\in$ P $\wedge$ A $\prec$ B $\wedge$ $\neg$(A $\prec$ C $\wedge$ C $\prec$ B) $\leftrightarrow$ B $\in$ Q)

[1]The maximum elements of the channel, i.e the set of all the data structures to which the channel variable can be bound.

Transformation techniques for the derivation of logic programs from sentences in full first order form have been described by Clark [24], Hogger [59, 60] and Kowalski [75]. Using appropriate definitions of $\in$ we can derive by standard transformations the recursive definition of union/3, seteq/2, nopreds/2 and next-rankset/2.

We define $\in$ by the following:

$X \in Y.Ys \leftrightarrow X=Y \vee X \in Ys$

$X \in nil \leftrightarrow \mathbf{false}$

and transform union/3 into a recursive form by taking the original definition of union/3 and treating the first occurrence of $\leftrightarrow$ as $\leftarrow$.

$union(X,Y,Z) \leftarrow \forall\, U\ (U \in Z \leftrightarrow U \in X \vee U \in Y)$

Substituting the definition of $\in$:

$union(X,Y,A.Z) \leftarrow \forall\, U\ ((U=A \vee U \in Z) \leftrightarrow U \in X \vee U \in Y)$

and expanding

$union(X,Y,A.Z) \leftarrow$

$\quad A \in X \vee A \in Y \wedge$

$\quad \forall\, U\ (U \in Z \leftrightarrow U \in X \vee U \in Y)$

i.e.

$union(X,Y,A.Z) \leftarrow$

$\quad A \in X \wedge union(X,Y,Z)$

$union(X,Y,A.Z) \leftarrow$

$\quad A \in Y \wedge union(X,Y,Z)$

The base case of the recursion is derived by taking X, Y and Z to be nil, and using

$$(X \in Y \leftrightarrow \mathbf{false})$$

$union(nil,nil,nil) \leftarrow \mathbf{false} \leftrightarrow (\mathbf{false} \vee \mathbf{false})$

Since (**false** $\leftrightarrow$ (**false** $\vee$ **false**)) is equivalent to **true**, we write the base case as

union(nil,nil,nil) $\leftrightarrow$ **true**

i.e.

union(nil,nil,nil)

Finally, the Horn clause definition of $\in$ is

$X \in X.Ys$

$X \in Y.Ys \leftarrow X \in Ys$

Similarly we can define set equality recursively:

seteq(X, Y) $\leftrightarrow$

$\forall A(A \in X \rightarrow A \in Y) \wedge \forall B(B \in X \leftarrow B \in Y)$

i.e.

seteq(X, Y) $\leftrightarrow$ subset(X,Y) $\wedge$ subset(Y,X)

subset(X,Y) $\leftrightarrow \forall A\ (A \in X \rightarrow A \in Y)$

and by transformations outlined in Hogger [59, 60] and Kowalski [75]. we can derive

seteq(X,Y) $\leftarrow$ subset(X,Y) $\wedge$ subset(Y,X)

subset(nil,Y)

subset(X.Xs,Ys) $\leftarrow X \in Ys \wedge$ subset(Xs,Ys)

Similarly we derive a recursive program for nopreds/2

nopreds(nil,nil)

nopreds(N.Ns,X) $\leftarrow$

$N \in X \wedge$ nopreds'(N,X)

nopreds'(X,nil)

nopreds'(N,A.X) $\leftarrow N \not\in A \wedge$ nopreds'(N,X)

$X \not\prec Y \leftarrow X=Y$

$X \not\prec Y \leftarrow Y \prec X$

We also can derive a recursive form of next-rankset/2, but this requires the use of a predicate which describes the set of items to one predecessor:

next-rankset(nil,nil)

next-rankset(B.Bs,C) ←

set-pred(B, SetX) ∧ nopreds(Cs,SetX) ∧

union(Cs,Cs′,C) ∧ next-rankset(Bs,Cs′)

This definition of set-pred requires the specific enumeration of all the immediate successors of all the members of a channel. Thus for the channel

{X, X=f(A,B), X=f(a,B), X=f(A,b), X=f(a,b)}

set-pred/2 is defined by

set-pred(P,Q) ↔

(P=$X ∧ Q={f($A,$B)}) ∨

(P=f($A,$B) ∧ Q={f(a,$B), f($A,b), P=f(a,b)}) ∨

(P=f(a,$B) ∧ Q={f(a,b)}) ∨

(P=f($A,b) ∧ Q={f(a,b)}) ∨

(P=f(a,b) ∧ Q={})

Definition 3.38 A *channel instance* is a collection of bindings states made to a variable during one computation. The set of states is a path through the graph of the channel. ■

A channel instance of the channel represented in Figure 3.7 is {V, f($A,$B), X=f(a,b)} (Figure 3.8).

We define a channel instance Ci for a set of observations with a set of maximal elements Ts by:

channel-instance(Ts,Ci) ↔ channel-instance(Ts,Ci,void)

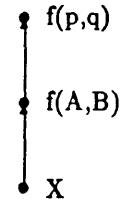

Figure 3.8: Representation of a channel instance, final state V/f(p,q)

channel-instance(Ts,Ci, El) $\leftrightarrow$

$\quad$ El $\in$ Ts $\wedge$ Ci={El} $\vee$

$\quad$ $\exists$ Ci', El' (El $\notin$ X $\wedge$ El $\prec$ El' $\wedge$

$\quad$ channel-instance(Ts,Ci',El') $\wedge$ union({El}, Ci', Ci))

3.6.3 Streams

One data structure to which a channel variable can eventually become bound during a computation is a *list*. In the following we denote a list by a pair prefixed with the infix function symbol "." whose first argument is any term and whose second argument is a list. The empty list is denoted by the constant `nil`. A list is thus a data structure in the form of a binary tree with the tree name being "." where data is stored at the leaves. A list can be *complete*, *incomplete*, or *partial*:

Definition 3.39 A *complete list* is a list whose rightmost leaf is the constant `nil`. ■

Definition 3.40 An *incomplete list* is a list whose rightmost leaf is a variable. ■

Definition 3.41 A *partial list* is a list whose rightmost leaf is not the constant value `nil`. ■

For example, a complete list is `a.b.c.nil`, an incomplete list is `a.b.c.X` and a partial list is `a.b.c`. An incomplete list is also a partial list.

Definition 3.42 A *stream* is a poset of instantiations of a stream variable, partially ordered by the prefix relation over lists. The poset is a chain and can be represented by a directed acyclic graph through which there is only one path from the initial unbound state of the stream variable to its final binding state (a list). ■

Each member of the set of bindings is a *partial list*, except for the final state which *may* be a complete list (i.e. terminating in `nil`).

We assume that each binding state of a variable V can be observed (see Section 3.5), and choose to represent the tail variable of partial lists by the unique constant `tail` to distinguish partial lists from complete lists. We give the type definition of *list* and *partial-list* below:

list(Xs) $\leftrightarrow$ (Xs=nil) $\vee$ (Xs=X.Ys $\wedge$ list(Ys))

partial-list(Xs) $\leftrightarrow$ Xs=tail $\vee$ (Xs=X.Ys $\wedge$ partial-list(Ys))

For simplicity we assume that if a variable is incrementally instantiated to a list then the instantiation of the leaves proceeds *sequentially*. Thus in the list $a_1.a_2.\ldots.a_n$ each a_i is instantiated before a_{i+1} $(1 \leq i < n)$.

In such a set every member of the chain uniquely covers another element, except for the bottom element. An element X covers another element Y iff Y $\prec$ X. This assumption is not an undue restriction, and facilitates the description of systems in which communication occurs by sequential atomic computations. A stream S of bindings to variable V where the top element represents the binding to a complete list L of length N maps onto the subset of natural numbers from 1 to N+2 inclusive; the number of elements of the chain is N+2. For example, if the top element is V/a.nil then the chain is {tail, a.tail, a.nil}.

Definition 3.43 We define the order relation for streams as follows, where prefix(X,Y) denotes $X \leq Y$ and pred(X,Y) denotes $X \prec Y$:

prefix(X,Y) $\leftrightarrow$

X=Y $\vee$ $\exists$ Z (pred(X,Z) $\wedge$ prefix(Z,Y))

pred(X,Y) $\leftrightarrow \exists$ D pred(X,Y,D)

pred(X,Y,D) $\leftrightarrow$
(append(X,D.tail,Y)) $\vee$
$\exists$ Z (append(Z,tail,X) $\wedge$ append(Z,nil,Y) $\wedge$ D=nil)

append(X,Y,Z) $\leftrightarrow$
((X=nil $\vee$ X=tail) $\wedge$ Y=Z) $\vee$
$\exists$ X',Z',A (X=A.X' $\wedge$ Z=A.Z' $\wedge$ append(X',Y,Z'))

■

The definition of a stream is simpler than that of a channel and is related to that of a *channel instance* since there is only one path through the graph representing the set of observations of a stream . We describe a stream S recursively where T is the top element of S, i.e. the unique final binding state of the stream variable and B is bottom element of S i.e. the initial unbound state of the variable.

Stream (1)

stream(T,S) $\leftrightarrow$ stream(T,S,tail)

stream(T,S,B) $\leftrightarrow$
$\exists$ B' S',C (pred(B,B') $\wedge$ singleton(B,C) $\wedge$ union(C,S',S) $\wedge$ stream(T,S',B')) $\vee$
$\exists$ C (T=B $\wedge$ singleton(T,C) $\wedge$ seteq(C, S))

singleton(X,Y) $\leftrightarrow \forall$ U (U $\in$ Y $\leftrightarrow$ U=X)

Note that in the above definition pred/2 is a simplification of set-pred/2 for channels due to the chain structure of streams, and pred(X,Y) expresses X $\prec$ Y. The relation singleton(X,Y) expresses Y = {X}. The sentence about singleton can be transformed into a Horn clause:

singleton(X, X.nil)

3.6.4 Complexity of systems

Communication in concurrent systems can be represented by incremental bindings to shared variables. We can distinguish between systems according to the *complexity* of the systems.

Definition 3.44 The *complexity* of the communication on a particular channel is a function of the ordering relation on that channel poset, and is expressed as the number of distinct paths from $\perp$ to the maximal elements in the graph of the poset. The complexity of a system is the maximum of the complexities of all the channels within the system. ■

If a channel has a complexity of 1, then all communication over that channel is sequential. A stream has a complexity of 1 and hence imposes a sequential order on the production and transmission of messages. Stream based systems are chosen as examples in this chapter due to their low complexity.

3.7 Observations

3.7.1 Observable variables

As a refinement of the model of communication presented so far we assume that some variables in a system are observable or public and that others are hidden or private.

Definition 3.45 An *observable* variable of a process is one which can be shared with other processes. Likewise a system may have observable variables, the state of which can be inspected by an observer of that system. A process or system which has one or more observable variables is *open*. A process or system with no observable variables is *closed*. A process or system which has both observable and hidden variables is *partially-open*. Unification is an open computation. ■

Definition 3.46 A *hidden* variable of a process is not accessible to any other processes. A hidden variable of a system cannot be observed by an observer. ■

Remark 3.1 Two or more processes can be concurrent only if they share common observable variables. However the sharing of such variables does not imply that these variables will be used for communication between these processes, so that such sharing is only an indication of possible concurrent behaviour. However, if two or more simultaneous processes do not have any observable variables in common then they occur in parallel.

We can now annotate concurrent computations and parallel computations with reference to observable variables. Atomic computations are labelled by the unification relation $=/2$ whose arguments contain the observable variables of that computation. Atomic computations are always open (Definition 3.45) — the observable variables of an atomic computation (or a process) can be hidden by including that computation in a closed process. (Figure 3.9).

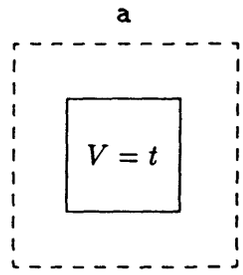

Figure 3.9: Simple closed system

The symbolic names of processes and systems are annotated with arguments standing for observable variables. For example we denote a process with name c and observable variables X and Y by c(X,Y). Figure 3.10 illustrates the partially open system c(V) whose observable variable is V and hidden variables are X and Y. Thus the only observation that can be made of the system c is the binding {V/f(p,q)}.

3.7.2 Observable states

We can now formalise what we mean by the state of a system, and how states relate to observations.

Definition 3.47

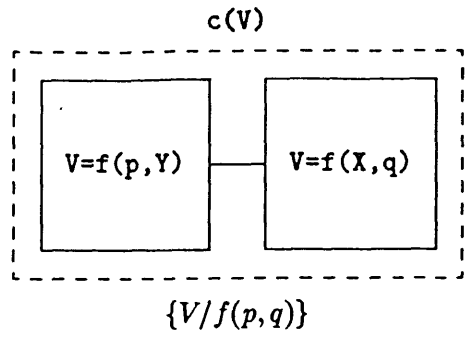

Figure 3.10: Partially open system

(1) A *state* S_n of a system is represented by the n-tuple $\{B_{V_1}, \ldots, B_{V_n}\}$ of the bindings of the n variables in that system, where B_{V_n} represents the binding state of variable V_n.

(2) The *observable state* of a system is represented by the tuple of the states of the observable variables of the system.

■

Definition 3.48

(1) The *initial state* S_0 of a system is an n-tuple $\{V_1, \ldots, V_n\}$ of the initial states of the n variables of the system.

(2) The *initial observable state* of a system is the initial state of the observable variables of the system.

■

3.7.3 Observations of stream based systems

We now focus our attention on systems where all communication occurs via the incremental binding of shared variables to lists. We call such communication *stream based*, and refer to the shared variables as *stream variables*. The set of states of such a variable is called a stream (Definition 3.42). The members of a stream are partial lists except for the top

element which is either a partial list or a complete list. The binding of the tail variable to `nil` represents an *end-of-transmission* message, which can occur only once on any one stream variable.

In Definition 3.43 we defined the order relation on streams using logic. We now also define the length of the list to which a stream variable is bound (i.e a state of the stream variable) by:

Definition 3.49

length(X,L) $\leftrightarrow$

((X=nil $\vee$ X=tail) $\wedge$ L=0) $\vee$

$\exists$ X1,Xs,L'(X=X1.Xs $\wedge$ L= s(L') $\wedge$ length(Xs, L')

■

When we partition the graph of a stream into rank node sets, each set contains a single state since there is only one path through the graph (at most one successor and predecessor for each node). If the rank of node x_i is i, then we say that x_i is the ith state of the stream variable V, and write this as B^i_V. The length of B^i_V is denoted by $|B^i_V|$; note that $|B^0_V| = 0$. The binding history of a stream variable V is the set $\mathcal{S} : \{B^i_V \mid 0 \leq i \leq n\}$ where n is the rank of the final state of V.

In a system comprising k stream variables $V_1 \ldots V_k$, the predecessor relation, denoted by pred_s, which relates a state S_n and a state S_{n+1} is defined by:

$\text{pred}_s(\{B_{V_1}, \ldots, B_{V_k}\}, \{B'_{V_1}, \ldots, B'_{V_k}\}) \leftrightarrow$

$\text{pred}(B_{V_1}, B'_{V_1}) \vee \ldots \vee \text{pred}(B_{V_k}, B'_{V_k})$

where state S_n comprises the tuple $\{B_{V_1}, \ldots, B_{V_k}\}$ of stream variables $V_1 \ldots V_k$. For brevity of notation, we represent a state of variable V_j by B_{V_j}, and its possible successor state by B'_{V_j}. A *possible successor state* of a variable V_j in state n is either the nth or the $n+1$th state of the variable.

The order relation $\leq_s$ on a stream based system can then be defined in terms of pred_s

$\leq_s$(X,Y) $\leftrightarrow$

$$X{=}Y \vee \text{pred}_s(X,Y) \vee \exists\, Z\ (\text{pred}_s(X,Y) \wedge \leq_s(Z,Y))$$

3.8 Mapping natural language descriptions to observable states

In this section we describe some *stream-based* abstract concurrent systems in natural language and provide mappings from natural language descriptions to observable states for systems based on stream communication. We flag natural language specifications by "•", and observable states by "o".

Most of the systems that we describe in this section are *stream transformers* which process an *input* to produce an *output*. We can represent such a system as a graph $G = (X, U)$ where X is a set of pairs of states of the input / output variables and U is a set of pairs of such pairs:

$$X = \{(B_I^j, B_O^k) \mid 0 \leq j \leq s\ ,\ 0 \leq k \leq t\}$$

$$U = \{((B_I^j, B_O^k), (B_I^{j'}, B_O^{k'})) \mid j \leq j' \wedge k \leq k' \wedge 0 \leq j \leq s \wedge 0 \leq j' \leq s \wedge 0 \leq k \leq t \wedge 0 \leq k' \leq t\}$$

where s and t are the ranks of the final states of I and O respectively.

We use the notation *faithfully* denoted by $\mathcal{F}$, possibly subscripted, to indicate the relation between the input and the output. Possible subscripts are c (indicates corruption of items), l (loss), r (repetition) and s (input sequence preserved). An overline on a subscript letter denotes the opposite behaviour to that denoted by the letter alone. Thus $\mathcal{F}_{\bar{c},\bar{l},\bar{r},s}$ denotes a system in which every item is copied from the input to the output, with no corruption, loss or repetitions, and order preserved. We present below mappings from the *faithfulness* terminology into that of stream variable binding histories. All of the systems are assumed to be *fair* for both input and output:

Definition 3.50 We say that a system processes input *fairly* iff it never arbitrarily refuses input. Likewise a system which processes output fairly never refuses to output items if items are available for output. ■

3.8.1 Faithfulness and stored items

Observation 3.1 [Prefixes and faithfulness]

- A system faithfully $\mathcal{F}_{\bar{c},\bar{I},\bar{r},s}$ copies its input I to its output O, i.e. without corrupting, losing or repeating items, and preserving their order of input.

  - The system has two observable variables I and O. In all possible states of the system S_k where $S_k \in \mathcal{S}$, the binding to the output variable stream O is a prefix of the binding to the input variable V, or the binding to O is the same as the binding to I, i.e. prefix(B_O,B_I) $\vee$ $B_O = B_I$.

Observation 3.2 [Storing items up to a maximum]

- A system faithfully $\mathcal{F}_{\bar{c},\bar{I},\bar{r},s}$ copies its input I to its output O, and can store a maximum of Max items s.t. $Max \geq 0$. In a particular state S_k, n items are in the system's internal store, and they are $X_1 \ldots X_n$

  - The system has two observable variables I and O. For all possible states $S_k \in \mathcal{S}$ of the system, the input is a prefix of the output, or the input and the output states are the same, i.e. prefix(B_O, B_I) $\vee$ $B_O = B_I$.

  - In the particular state S_k, the number of items in the store is indicated by the difference n between the length of the states of I and O (see Definition 3.49). i.e. $n = |B_I| - |B_O|$.

  - The number n of items in the store can never exceed the stated maximum, i.e.

$$\forall n : n \leq Max$$

  - The items in the list $Items_k$ stored are those items in B_I but not in B_O. Since B_O is a prefix of B_I, then append(B_O,$Items_k$, B_I) holds. (See Definition 3.43 for the definition of append/3).

Observation 3.3 [Storing items — no upper limit]

- A system faithfully $\mathcal{F}_{\bar{c},\bar{I},\bar{r},s}$ copies its input I to its output O, and in a particular state S_k has stored n input items. There is no limit on the items that it may store. Its internal store contains n items and they are $X_1 \ldots X_n$

  - The description of the observable states is as for Observation 3.2 but *without* the restriction that $\forall\, n : n \leq Max$.

3.8.2 Determining the state of the store

Observation 3.4 [An empty store]

- A system faithfully $\mathcal{F}_{\bar{c},\bar{I},\bar{r},s}$ copies its input I to its output O, and in a particular state has no input items stored.

  - The system has two observable variables I and O. In all possible states of the system $S_k \in \mathcal{S}$, (prefix(B_O, B_I) $\vee$ $B_O = B_I$).

  - In the particular state S_k when the store is empty, the number of items in the store is zero, i.e. $n = 0$, $B_I = B_O$, and $|B_I| = |B_O|$.

Observation 3.5 [A store neither full nor empty]

- A system faithfully $\mathcal{F}_{\bar{c},\bar{I},\bar{r},s}$ copies its input I to its output O, and in a particular state its store is neither full nor empty.

  - The system has two observable variables I and O, and an associated maximum size to its store, Max. In all possible states of the system $S_k \in \mathcal{S}$,
    (prefix(B_O, B_I) $\vee$ $B_O = B_I$).

  - In the particular state S_k when the store is neither full nor empty, the number of items in the store is n, i.e. $n \leq Max$, and $n = |B_I| - |B_O|$.

Observation 3.6 [A full store]

- A system faithfully $\mathcal{F}_{\bar{c},\bar{I},\bar{r},s}$ copies its input I to its output O, and in a particular state its store is full.

  - The system has two observable variables I and O, and an associated maximum size to its store, Max. In all possible states of the system $S_k \in \mathcal{S}$, $(\text{prefix}(B_O, B_I) \vee B_O = B_I)$.

  - In the particular state S_k when the store is full, the number of items in the store is n, i.e. $n = Max$, and $n = |B_I| - |B_O|$.

3.8.3 The state of the store determines the next observable state

Observation 3.7 [Behaviour when the store is empty]

- A system can store n items ($n > 0$) and when empty can only input an item.

  - A system is empty in a state S_n if the length of list to which the input variable is bound equals the length of the list to which the output variable is bound.

  - In the successor state S_{n+1}, the length of the list to which the input variable is bound exceeds by one the length of the list to which the input variable is bound in state S_n.

  - The length of the list to which the output variable is bound is the same in both state S_{n+1} and state S_n.

  - The length of the list to which the input variable is bound in state S_{n+1} exceeds by one the length of the list to which the output variable is bound in state S_{n+1}.

Observation 3.8 [Behaviour when the store is full]

- A system can store a maximum of Max items ($Max > 0$), and when full can only output an item.

- The difference between the length of the list to which the input variable is bound and the length of the list to which the output variable is bound cannot be more than Max in any state. $|B_I| - |B_O| \leq Max$.

- The system is full in state S_n if the length of the list to which the input variable is bound is equal to the length of the list to which the output variable is bound. $|B_I^n| = |B_O^n|$.

- and in the next state S_{n+1}, the length of the list to which the input variable is bound is unchanged while the length of the list to which the output variable is bound is incremented by one. $|B_I^{n+1}| = |B_I^n|$ and $|B_O^{n+1}| = |B_O^n| + 1$.

Observation 3.9 [Behaviour when the store is neither full nor empty]

- A system can store a maximum of Max items and, when neither full nor empty, can only output items, or only input an item, or input and output two distinct items simultaneously.

- The difference between the lengths of the lists to which the input variable and the output variable are bound cannot be more than Max in any state. $|B_I| - |B_O| \leq Max$.

- In any state S_n, if the difference between the length of the lists to which the input variable and the output variable are bound is less than Max and greater than zero, the system is neither full nor empty. $0 < (|B_I| - |B_O|) < Max$.

- and in the next state S_{n+1} :

inputting an item: the length of the list to which the input variable is bound is incremented by one while the length of the list to which the output variable is bound is unchanged.

outputting an item: the length of the list to which the output variable is bound is incremented by one while the length of the list to which the input variable is bound is unchanged.

inputting and outputting two different items: the lengths of the lists to which both the input and output variables are bound are incremented by one, and the last item of each list is different.

3.9 Descriptions in natural language of illustrative systems

We describe illustrative stream based systems using natural language. One example of such systems is a producer (or consumer) of partial lists which comprise the observations of the system. Other exemplars can be regarded as variants of transformers which transform input partial lists into output partial lists where each pair $(Input, Output)$ describes a discrete state of the system. Such systems can be specified by:

(1) a statement of the relationship that is to exist between members of the pair in each state,

(2) a statement of the rule describing the relationship between each pair and its successor.

The complete history of any given system is represented by the set of all the observations that can be made of the system. It is the intention of our treatment that the complete history can be described in first order logic.

3.9.1 A producer

A *sequential* producer produces messages in linear order on a stream variable, incrementally binding the variable to a list of messages. We assume that a *terminating* producer flags its termination with an end-of-transmission (*eot*) message, indicated by binding the tail of the stream variable to `nil`, thus making the bindings to that variable a complete list (see Definition 3.39).

The message stream is a strict poset of partial lists ordered by the prefix_1 relation whose bottom element is the partial list `tail` and top element is a complete list. There is only one complete list in the set.

3.9.2 An N-bounded buffer.

An N-bounded buffer inputs items and reproduces them faithfully $\mathcal{F}_{\bar{c},\bar{l},\bar{r},s}$ on its output, eventually outputting all items, and being able to store up to N items during the interim

period. In all possible states of the buffer the output is a prefix of the input (Observation 3.2).

The buffer can be either full or empty or neither empty nor full. In a particular state the number of items in the store is indicated by the difference between the length of the states of the input and output variables.

(1) When full it cannot accept (input) an item, but must output an item fairly:

- the system is full in state S_n if the difference between the lengths of the lists to which the input and output variables are bound is equal to Max (Observation 3.6).
- in the next state S_{n+1}, the length of the list to which the input variable is bound remains unchanged while the length of the list to which the output variable is bound incremented by one (Observation 3.8).

(2) When empty it cannot output an item, but must input an item fairly.

- The buffer is empty when the length of the list to which the input variable is bound equals the length of the list to which the output variable is bound. (Observation 3.4).
- In the next state length of the list to which the input variable is bound exceeds by one the length of the list to which the input variable is bound in the empty state, and the length of the list to which the output variable is bound is the same in both states (Observation 3.7).

(3) While it is neither empty nor full the buffer can input and output two (different) items simultaneously, or can input and output items fairly in some interleaved sequence.

- The buffer is neither empty nor full when the difference between the length of the lists to which the input and output variables are bound is within the range $r : 1 \leq r < Max$ (Observation 3.5).

  From Observation 3.9 the next state is arrived at by one of the following three actions:

(a) **only inputting an item:**
the length of the list to which the input variable is bound is incremented by one while the length of the list to which the output variable is bound is unchanged.

(b) **only outputting an item:**
the length of the list to which the output variable is bound is incremented by one while the length of the list to which the input variable is bound is unchanged.

(c) **both inputting and outputting two different items:**
the lengths of the lists to which both the input and output variables are bound are incremented by one, and the last items are different.

3.9.3 One slot buffer

A one slot buffer is a special case of an N-buffer, in which the maximum number of items that can be stored is 1. It inputs items and reproduces them $\mathcal{F}_{\bar{c},\bar{I},\bar{r},s}$ on its output. An item input will be the next item output, and the store can be either full or empty, but never part-full. When full it cannot accept (input) an item, but must output an item fairly. When empty it cannot output an item, but must input an item fairly. It inputs/outputs in an alternating sequence: *in, out, in, out, ...* Later in this chapter we show that an N-place buffer can be constructed from N 1-place buffers.

3.9.4 An unbounded buffer (FIFO queue).

A queue is a special case of a buffer for which there is no maximum number of items that can be stored. It inputs items and reproduces them $\mathcal{F}_{\bar{c},\bar{I},\bar{r},s}$ on its output. An item that is input is eventually output. If the environment cannot accept an item offered by the buffer for output then the buffer can always accept items offered by the environment. Its store can be either empty or part-full — when empty it cannot output an item, but must input an item fairly. While it is non-empty it may input and output two (different) items simultaneously, or input/output items fairly in some interleaved sequence.

3.9.5 Expedited data queue.

An expedited data queue (xdq) accepts two types of items: normal and expedited. An xdq is faithful $\mathcal{F}_{\bar{c},\bar{I},\bar{r},\bar{s}}$ for normal items and faithful $\mathcal{F}_{\bar{c},\bar{I},\bar{r},\bar{s}}$ for expedited items respectively, but is faithful $\mathcal{F}_{\bar{c},\bar{I},\bar{r},\bar{s}}$ over all items.

The xdq is unbounded like an ordinary queue, can never refuse to input an item and can refuse to output an item only when it is empty. When the xdq is non-empty it can accept and output items either simultaneously or in an interleaved manner.

If an expedited item is input, then it will be output before any normal item which has been input but not yet output. If a normal item is input, it will not be output before an expedited item that has been previously input and not yet output. Note that if an expedited item is input and an item is output simultaneously, then the output item may be a normal item or an expedited item (but different from that simultaneously input).

3.10 Descriptions in logic of illustrative systems

In this section we describe some example systems using logic. We concentrate on stream based systems due to their low complexity (see Definition 3.44). We can describe each example system in two ways.

$\mathcal{SET}$ A Horn clause description of the set of observations that can be made of the system.

$\mathcal{PROG}$ A Horn clause description of the relationship between each member of the set and the maximal elements of the set.

$\mathcal{SET}$ and $\mathcal{PROG}$ can be regarded as logic programs in the logic languages $L_{\mathcal{S}}$ and $L_{\mathcal{P}}$ respectively. A logic interpreter can be defined for each language, based on the operational semantics of that language.

When we execute a logic program written in language $L_{\mathcal{S}}$ on an interpreter $I_{\mathcal{S}}$ we are only interested in the final binding states of the variables in the query. Thus the operational semantics for $L_{\mathcal{S}}$ need not embody concurrency and we can use any sequential logic programming language, for example Prolog, as the basis for $I_{\mathcal{S}}$. In fact a program $\mathcal{SET}$ can

be regarded as a Prolog program if suitable care is taken over the ordering of clauses and calls.

The execution of $\mathcal{PROG}$ should result in the behaviour described by $\mathcal{SET}$. Thus we are interested in the binding states created *during* the execution of a logic program written in $L_{\mathcal{P}}$. The operational semantics of $L_{\mathcal{P}}$ define a logic interpreter $I_{\mathcal{P}}$ whose operation produces or simulates concurrent behaviour. We define the semantics of $L_{\mathcal{P}}$ in the next chapter.

3.10.1 Stream Producers

The producer of a channel is a process which incrementally binds a channel variable. Similarly a consumer of a channel incrementally checks that a variable is bound to an expected data structure. Messages on a channel correspond to leaves in the data structure to which the channel variable is incrementally bound. In the case that the channel is a stream the messages correspond to the items in the list to which the stream variable becomes bound. Messages can be complex data structures, but the restriction that each member of the stream poset except for the bottom element uniquely covers its immediate predecessor means that messages on a stream variable are produced sequentially. The observations that can be made of a stream producer are just the observations that can be made of the stream variable which the producer incrementally instantiates.

3.10.1.1 Set description of a producer

We repeat below the definition in logic of a stream given in Section 3.6.3, page 75 and the definition for the predecessor relation:

stream(T,S) $\leftrightarrow$ stream(T,S,tail)

stream(T,S,B) $\leftrightarrow$

$\exists$ B′ S′,C (pred(B,B′) $\wedge$ singleton(B,C) $\wedge$ union(C,S′,S) $\wedge$ stream(T,S′,B′)) $\vee$

$\exists$ C (T=B $\wedge$ singleton(T,C) $\wedge$ seteq(C, S))

singleton(X,Y) $\leftrightarrow \forall$ U (U $\in$ Y $\leftrightarrow$ U=X)

prefix(X,Y) $\leftrightarrow$
    X=Y $\vee \exists$ Z (pred(X,Z) $\wedge$ prefix(Z,Y))

pred(X,Y) $\leftrightarrow \exists$ D pred(X,Y,D)

pred(X,Y,D) $\leftrightarrow$
    (append(X,D.tail,Y)) $\vee$
    $\exists$ Z (append(Z,tail,X) $\wedge$ append(Z,nil,Y) $\wedge$ D=nil)

append(X,Y,Z) $\leftrightarrow$
    ((X=nil $\vee$ X=tail) $\wedge$ Y=Z) $\vee$
    $\exists$ X',Z',A (X=A.X' $\wedge$ Z=A.Z' $\wedge$ append(X',Y,Z'))

We transform the sentences into Horn clauses by:

(1) rewriting $\leftrightarrow$ by $\leftarrow$,

(2) omitting existential quantifiers in the body of a clause,

(3) rewriting (a $\leftarrow$ b $\vee$ c) as the two clauses (a $\leftarrow$ b) and (a $\leftarrow$ c),

(4) using a suitable data structure, for example a list, to represent a set.

The above description of a stream can thus be represented as the following Horn clauses. We have chosen to represent sets in Horn clauses by lists. Given the top element of the stream, i.e. the final binding state of the stream variable, the relation stream(Top,S) represents the binding states of the stream variable as S, a list of lists.

stream(T,S) $\leftarrow$ stream(T,S,tail)

stream(T,S,B) $\leftarrow$ pred(B,B') $\wedge$ union(B.nil,S',S) $\wedge$ stream(T,S',B')
stream(T,S,B) $\leftarrow$
    (T=B $\wedge$ T.nil=S)

union(nil,nil,nil)
union(X,Y,A.Z) ←
    A ∈ X ∧ union(X,Y,Z)
union(X,Y,A.Z) ←
    A ∈ Y ∧ union(X,Y,Z)

X ∈ X.Ys
X ∈ Y.Ys ← X ∈ Ys

prefix(X,X)
prefix(X,Y) ↔
    pred(X,Z) ∧ prefix(Z,Y)

pred(X,Y) ← pred(X,Y,D)

pred(X,Y,D) ←
    append(X,D.tail,Y))
pred(X,Y,D) ←
    append(Z,tail,X) ∧ append(Z,nil,Y) ∧ D=nil

append(nil,Y,Y)
append(tail,Y,Y)
append(X,Y,Z) ←
    X=A.X′ ∧ Z=A.Z′ ∧ append(X′,Y,Z′)

The Horn clauses above can be regarded as a logic program of type $\mathcal{SET}$ in language $L_{\mathcal{S}}$. The execution of a query of the form

← stream(Top, Stream)

where Top is initially bound to a complete list on an interpreter for $L_{\mathcal{S}}$ results in Stream being bound to a data structure representing the set of binding states of a stream variable. This data structure is a list items each of which is a list representing one binding state of

the stream variable.

3.10.1.2 Logic program for a producer

We now describe in first order logic the relationship between each of the members of a stream and the final state of the stream variable. A stream producer incrementally instantiates a stream variable to a list and the order relation on the set of bindings is based on list prefixes. We repeat the top level definition for prefix/2 on streams given on page 90:

prefix(X,Y) $\leftrightarrow$

$\quad$ X=Y $\vee$ $\exists$ Z (pred(X,Z) $\wedge$ prefix(Z,Y))

Partially evaluating prefix/2 w.r.t. the definition of pred/2 and append/3, we write:

prefix(X,Y) $\leftrightarrow$ $\exists$ Diff (append(X,Diff,Y))

We can describe this incremental instantiation by a predicate relating each member of the stream poset to the top element. Given T, the final state of the stream variable, then the relation approximates(X, Diff, Stream,Top) relates X, a member of the stream S, to the top element Top where Diff is the difference between X and Top.

approximates(X, Diff, Stream,Top) $\leftrightarrow$ (X $\in$ Stream $\leftrightarrow$ append(X,Diff,Top))

We rewrite this as a recursive relation where Diff is the difference between element X and the top element of the poset. X is initialized to the bottom element of the set in order to find the differences between the top element of the set and the other members of the set.

approximates(Diff, Top, X) $\leftrightarrow$

$\quad$ pred(X,Top,Diff) $\vee$

$\quad$ $\exists$ X', D, Diff' (pred(X,X',D) $\wedge$ Top $\neq$ X' $\wedge$ X $\leq$ Top $\wedge$ Diff=D.Diff'

$\quad\quad$ $\wedge$ approximates(Diff', Top, X'))

pred(X,Y,D) $\leftrightarrow$

$\quad$ append(X,D.tail,Y) $\vee$

$\quad$ $\exists$ Z (append(Z,tail,X) $\wedge$ append(Z,nil,Y) $\wedge$ D=nil)

We can replace the two arguments Top and X by the pair ⟨Top,X⟩ which denotes a difference-list. For example, if Top=a.b.c.nil and X=a.b.tail then the tuple is ⟨a.b.c.nil,a.b.tail⟩ which represents the difference-list c.nil-tail.

approximates(Diff, ListPair) ↔
    (empty(ListPair) ∧ Diff=nil) ∨
    ∃ First,ListPair′ (decompose(ListPair, First, ListPair′) ∧
        Diff=Head.Diff′ ∧ approximates(Diff′, ListPair′))

decompose(⟨Top,X⟩ ,Head,⟨Top,X′⟩) ↔ append(X, Head.X′,Top)
empty(⟨Top,X⟩) ↔ ∃ Z (append(Z,nil,Top) ∧ append(Z,tail,X))

We now replace the difference-list ⟨Top,X⟩ by the list L s.t. append(A,L,B) holds:

approximates(Diff, List) ↔
    (empty(List) ∧ Diff=nil) ∨
    ∃ First,ListPair′ (decompose(List, First, List′) ∧
        Diff=Head.Diff′ ∧ approximates(Diff′, List′))

decompose(X,Y,Z) ↔ X=Y.Z

empty(X) ↔ X=nil

Partially evaluating approximates/2 w.r.t. empty/1 and decompose/3 we derive the relation produces/2 where `produces(X,Y)` denotes that X is a stream variable incrementally bound to the list Y:

produces(Diff, List) ↔ (List=nil ∧ Diff=List) ∨
    ∃ Z, Diff′, List′ (Diff=Z.Diff′ ∧ List=Z.List′ ∧ produces(Diff′, List′))

The above sentence can be rewritten as Horn clauses where produces(StreamVar, FinalState) describes the incremental binding of the stream variable StreamVar to its final binding state.

produces(StreamVar, List) ←

StreamVar=nil ∧ List=nil

produces(StreamVar, List) ←

StreamVar = X.StreamVar′ ∧ List=X.List′ ∧ produces(StreamVar′, List′)

The definition of produces/2 can be regarded as a logic program of a general type $\mathcal{PROG}$ in a language $L_{\mathcal{P}}$. For each program P of type $\mathcal{PROG}$ derived from a first order description F of a system there is a corresponding program S of type $\mathcal{SET}$ derived from F. The execution of P and a suitable query on an interpreter which embodies the computational rule of $L_{\mathcal{P}}$ results in the behaviour predicted by S. For example, a query that can be made w.r.t. produces/2 is

← produces(StreamVar, List)

where List is initially bound to a complete list. The execution of this query will incrementally bind StreamVar in the manner described by the $\mathcal{SET}$ program stream/2 defined in Section 3.10.1.1.

The computational rule employed by an interpreter for the language $L_{\mathcal{P}}$ is described in the next chapter and forms the basis of the definition of the operational semantics of the logic programming language SILCS.

3.10.2 Buffers

A buffer is a relation between two streams, Input and Output. These streams can be considered together as the poset of pairs ⟨Input,Output⟩ with ⟨tail,tail⟩ as the bottom element. A buffer should eventually copy all items input to the output, preserving their order. Thus the poset can be described by a directed acyclic graph with maximal element ⟨X,X⟩ where X is a complete list. An N-bounded buffer can store up to N items and thus the ordering relation $\leq$ on the poset is related to N, the state of the store and prefix on lists. We give below the definition of the predecessor $\prec_N$ (pred) relation and the ordering relation $\leq_N$ (order) for an N-buffer. The definition of $\prec_N$ is based on the earlier descriptions of the expected behaviour of a bounded buffer given in Section 3.9.2.

order(N, X, Y) ↔

X=Y ∨ pred(N, X,Y) ∨ ∃ Z (pred(N,X,Z) ∧ order(N,Z,Y))

pred(N, ⟨I,O⟩, ⟨I′, O′⟩) ↔

∃ Z(append(Z,nil,I) ∧ append(Z,tail,O) ∧ I=I′ ∧ I=O′) ∨

(empty(I,O) ∧ O′=O ∧ input(I,I′)) ∨

(full(N,I,O) ∧ I′=I ∧ output(I,O,O′)) ∨

(part-full(N,I,O) ∧ O′=O ∧ input(I,I′)) ∨

(part-full(N,I,O) ∧ I′=I ∧ output(I,O,O′)) ∨

(part-full(N,I,O) ∧ input(I,I′) ∧ output(I,O,O′))

empty(I,O) ↔ I=O

full(N,I,O) ↔ ∃ L1, L2(length(I,L1) ∧ length(O,L2) ∧ N=L1-L2)

part-full(N,I,O) ↔ ∃ L1, L2(length(I,L1) ∧ length(O,L2) ∧ N > L1-L2)

length(X,L) ↔

((X=tail ∨ X=nil) ∧ L=0) ∨

∃ Z,X′,L′ (X=Y.X′ ∧ L=s(L′) ∧ length(X′,L′))

input(I,I′) ↔

∃ X (append(I,X.tail,I′)) ∨ ∃ Y (append(Z,tail,I) ∧ append(Z,nil,I′))

output(I, O, O′) ↔ ∃ X,Y (append(O, X.Y, I) ∧ append(O, X.tail, O′))

Each path through the graph represents one possible way in which the buffer behaves, i.e. one computation. A definition of the top level of the N-buffer, based on Channel (3) is:

n-buffer(N,Top,Set) ↔ n-buffer(N,Top,Set,{⟨tail,tail⟩})

n-buffer(N, Top, Set, Mins) ↔

∃ Set′, Mins′ (next-rankset(N,Mins,Mins′) ∧ union(Mins, Set′, Set)

$\wedge$ n-buffer(N, Top, Set′, Mins′)) $\vee$
seteq({Top}, Mins) $\wedge$ seteq(Set, Mins)

The definition of next-rankset/3 is not recursive:

next-rankset(N, Mins, Mins′) $\leftrightarrow$
$\forall$ A,B,C (A $\in$ P $\wedge$ pred(N,A,B) $\wedge$ $\neg$(pred(N,A,,C) $\wedge$ pred(N,C,,B)) $\leftrightarrow$ B $\in$ Q)

3.10.2.1 Set description of a buffer

We transform the recursive first order logic sentences describing an N-buffer into Horn clauses using the same techniques as we did for producers in Section 3.10.1.1, i.e. by:

(1) rewriting $\leftrightarrow$ by $\leftarrow$,

(2) omitting existential quantifiers in the body of a clause,

(3) rewriting (a $\leftarrow$ b $\vee$ c) as the two clauses (a $\leftarrow$ b) and (a $\leftarrow$ c),

(4) using a suitable data structure, for example a list, to represent a set.

Our initial first order description is based on that for the N-buffer given above.

n-buffer(N,Top,Set) $\leftarrow$ n-buffer(N,Top,Set, ⟨tail.tail⟩.nil)

n-buffer(N, Top, Set, Mins) $\leftarrow$
next-rankset(N,Mins,Mins′) $\wedge$ union(Mins, Set′, Set)
$\wedge$ n-buffer(N, Top, Set′, Mins′)
n-buffer(N, Top, Poset, Mins) $\leftarrow$
Top.nil = Mins $\wedge$ Set = Mins

next-rankset(N,nil,nil)
next-rankset(N,B.Bs,C) $\leftarrow$
set-pred(N,B,SetX) $\wedge$ nopreds(Cs,SetX) $\wedge$
union(Cs,Cs′,C) $\wedge$ next-rankset(N,Bs,Cs′)

nopreds(nil,nil)

nopreds(N.Ns,X) ←

$\quad$ N ∈ X ∧ nopreds′(N,X)

nopreds′(X,nil)

nopreds′(N,A.X) ← N ∉ A ∧ nopreds′(N,X)

union(nil,nil,nil)

union(X,Y,A.Z) ←

$\quad$ A ∈ X ∧ union(X,Y,Z)

union(X,Y,A.Z) ←

$\quad$ A ∈ Y ∧ union(X,Y,Z)

X ∈ X.Ys

X ∈ Y.Ys ← X ∈ Ys

The definition of set-pred/3 can be achieved by specific enumeration of all the possible successors of a state, or by the use of a set predicate. The general first order definition of the setof-solutions relation was given by Clark [30] as:

Setof (1)

setof-solutions(T,G,S) ↔

$\quad$ ∀ X (∃ $L_1, \ldots, L_n$(G ∧ X=T) ↔ X ∈ S)

where G is a goal and S a list containing an instance of term T for each solution to G. $L_1, \ldots, L_n$ is the list of local variables in G. Naish [99] further requires that the list S is sorted with respect to some arbitrary total order over terms and duplicate elements are removed:

Setof (2)

setof-solutions(T,G,S) ↔

$\quad$ ∀ X (∃ $L_1, \ldots, L_n$(G ∧ X=T) ↔ X ∈ S) ∧ sorted(S)

Given an implementation of setof-solutions/3 we can then define set-pred/3 by:

set-pred(N,B,SetX) ← setof-solutions(X, pred(N,B,X), SetX)

The Horn clause definition of pred/3 can be directly derived from the first order description given in the previous section, the first clause of which describes the case when the buffer receives an end-of-transmission (`nil`) on the input stream.

pred(N, ⟨I,O⟩, ⟨I′, O′⟩) ← append(Z,nil,I) ∧ append(Z,tail,O) ∧ I=I′ ∧ I=O′
pred(N, ⟨I,O⟩, ⟨I′, O′⟩) ← empty(I,O) ∧ O′=O ∧ input(I,I′)
pred(N, ⟨I,O⟩, ⟨I′, O′⟩) ← full(N,I,O) ∧ I′=I ∧ output(I,O,O′)
pred(N, ⟨I,O⟩, ⟨I′, O′⟩) ← part-full(N,I,O) ∧ O′=O ∧ input(I,I′)
pred(N, ⟨I,O⟩, ⟨I′, O′⟩) ← part-full(N,I,O) ∧ I′=I ∧ output(I,O,O′)
pred(N, ⟨I,O⟩, ⟨I′, O′⟩) ← part-full(N,I,O) ∧ input(I,I′) ∧ output(I,O,O′)

empty(I,O) ← I=O
full(N,I,O) ← length(I,L1) ∧ length(O,L2) ∧ subtract(L1,L2,L) ∧ N=L

part-full(N,I,O) ←
    length(I,L1) ∧ length(O,L2) ∧ subtract(L1,L2,L) ∧ N > L

subtract(X,0,X)
subtract(s(X),s(Y),Z) ← subtract(X,Y,Z)

s(X) > 0
s(X) > s(Y) ← X > Y

length(tail,0)
length(nil,0)
length(X,L) ← X=Y.X′ ∧ L=s(L′) ∧ length(X′,L′)

input(I,I′) ← append(I,X.tail,I′)
input(I,I′) ← append(Z,tail,I) ∧ append(Z,nil,I′)

output(I, O, O′) ← append(O, X.Y, I) ∧ append(O, X.tail, O′)

The Horn clause description n-buffer/3 of the set of observations of an N-place buffer can be regarded as a program of type $\mathcal{SET}$ in the logic language $L_{\mathcal{S}}$. An N-place buffer is described by the call

$$\leftarrow \text{n-buffer(N,Top,Set)}$$

3.10.2.2 Set description of a one-place buffer

In order to describe a one place buffer, we can modify the description n-buffer/3 to the following, since a one-place buffer is either empty or full, but never part-full.

1-buffer(N,Top,Set) $\leftarrow$ 1-buffer(Top,Set, $\langle$tail.tail$\rangle$)

1-buffer(Top, Set, Min) $\leftarrow$
    pred(Min,Min′) $\wedge$ union(Min.nil, Set′, Set) $\wedge$ 1-buffer(Top, Set′, Min′)
1-buffer(Top, Poset, Min) $\leftarrow$
    Top = Min $\wedge$ Set = Min.nil

pred($\langle$I,O$\rangle$, $\langle$I′, O′$\rangle$) $\leftarrow$ append(Z,nil,I) $\wedge$ append(Z,tail,O) $\wedge$ I=I′ $\wedge$ I=O′
pred($\langle$I,O$\rangle$, $\langle$I′, O′$\rangle$) $\leftarrow$ I=O $\wedge$ O′=O $\wedge$ input(I,I′)
pred($\langle$I,O$\rangle$, $\langle$I′, O′$\rangle$) $\leftarrow$
    I $\neq$ O $\wedge$ I′=I $\wedge$ output(I,O,O′)

3.10.2.3 Induction on buffer descriptions

We have shown above how N-place buffers can be described as posets ordered on a relation $\leq_N$. We can also inductively define the poset of observations for an N-place buffer by composing states for empty and full buffers using $\prec_0$ and recursion. The descriptions of *empties* and *fulls* for an N-place buffer are required in this definition. The intention is later to relate this form of $\mathcal{SET}$ to a similar technique for the $\mathcal{PROG}$ form of the buffer descriptions, where we compose N 1-place buffers to form one N-place buffer.

The relation empties(T,Set,B) defines Set as the set of all the empty states of an N-place buffer where B is the initial state, i.e. a pair $\langle$tail,tail$\rangle$ and T is a pair $\langle$X,X$\rangle$ denoting

(N=0 ∧ empties(T,Set,B))

buffer(N,T,Set,B) ←

N=s(N′) ∧ buffer(N′,T,Set′,B) ∧ fulls(N,T,F,B) ∧ union(F,Set′,Set)

union(nil,nil,nil)

union(X,Y,A.Z) ←

A ∈ X ∧ union(X,Y,Z)

union(X,Y,A.Z) ←

A ∈ Y ∧ union(X,Y,Z)

X ∈ X.Ys

X ∈ Y.Ys ← X ∈ Ys

empties(T,E,B) ←

(B=T ∧ E=T.nil)

empties(T,E,B) ←

pred(0,B,B′) ∧ B′ ≠ T ∧ E=B.E′ ∧ empties(T,E′,B′)

fulls(N,T,F,B) ←

(B=T ∧ F=T.nil)

fulls(N,T,F,B) ←

pred(0,B,B′) ∧ B′ ≠ T ∧ full(N,T,X,B)

∧ F=X.F′ ∧ fulls(N,T,F′,B′)

full(N,T,X,B) ← T=⟨T′,T′⟩ ∧ B=⟨B′,B′⟩ ∧ X=⟨T′,B′⟩

3.10.2.4 Logic programs for buffers

A bounded buffer incrementally copies the bindings on one stream variable to another. The description of a buffer as a poset of bindings forms the basis of our derivations. We can describe this incremental copying by a predicate relating each member of the poset to the top element, as we did for the producer relation (Section 3.10.1.2). We use the order

relation for buffers defined on page 95. Partially evaluating order/3 w.r.t. pred/3 and append/3, we obtain

order(N,X,Y) $\leftrightarrow \exists$ Diff (pair-append(X,Diff,Y))

pair-append($\langle X_i,X_o\rangle$,$\langle D_i,D_o\rangle$, $\langle Y_i,Y_o\rangle$) $\leftrightarrow$
append(X_i,D_i,Y_i) $\wedge$ append(X_o,D_o,Y_o)

We relate each member of the set of observations to the terminal state of the buffer:

approximates(X, Diff, BufferSet ,Top) $\leftrightarrow$
X $\in$ BufferSet $\rightarrow$ pair-append(X,Diff,Top)

We rewrite this as a recursive relation where Diff is the difference pair between element X and the top element of the poset for one chain (computation). In order to find the difference between the top element of the set and all the other elements of the set, we initialise X to the bottom element of the set $\langle$tail,tail$\rangle$.

approximates(Diff, Top, X, N) $\leftrightarrow$
(pred(N, X, Top) $\wedge$ Diff=$\langle$nil,nil$\rangle$)$\vee$
$\exists$ X', Diff' (pred(N,X,X') $\wedge$ Top $\neq$ X'
$\wedge$ X $\leq$ Top $\wedge$ convert(Diff, X, X', Diff') $\wedge$ approximates(Diff', Top, X', N))

convert($\langle$I,O$\rangle$, $\langle X_i,X_o\rangle$, $\langle X_i',X_o'\rangle$, $\langle$I',O'$\rangle$) $\leftrightarrow$
$\exists$ A(X_o=X_o' $\wedge$ diff(X_i,X_i',A) $\wedge$ update(I,A,I') $\wedge$ O'=O) $\vee$
$\exists$ B(X_i=X_i' $\wedge$ diff(X_o,X_o',B) $\wedge$ I=I' $\wedge$ update(O,B,O')) $\vee$
$\exists$ A,B(diff(X_i,X_i',A) $\wedge$ diff(X_o,X_o',B) $\wedge$ update(I,A,I') $\wedge$ update(O,B,O'))

diff(X,Y,D) $\leftrightarrow$
append(X,D.tail,Y) $\vee$
$\exists$ Z(append(Z,tail,X) $\wedge$ append(Z,nil,Y) $\wedge$ D=nil)
update(A,B,C) $\leftrightarrow$ (B=nil $\wedge$ A=nil) $\vee$ (B $\neq$ nil $\wedge$ A=B.C)

We dispense with the arguments Top and X, replacing them with an argument representing the state of the buffer's store. This argument is a pair $\langle$Input,Output$\rangle$ where Input and

Output are the states of the input and output stream variables in a particular state. The store is then represented as a list derived from the difference of the input and output bindings. For example, if in one state the input-output pair is ⟨a.b.c.tail, a.b.tail⟩, then the store can be represented by ⟨c.tail,tail⟩. This representation simplifies the definition of pred/3 which can be rewritten without altering its logical meaning:

approximates(Diff, Store, N) ↔
    ∃ X,Y,Z (Store=⟨X,Y⟩ ∧ append(Z,nil,X) ∧ append(Z,tail,Y) ∧ Diff=⟨nil,nil⟩) ∨
    ∃ X′, Diff′ (pred(N,Store, Store′) ∧ convert(Diff, Store, Store′, Diff′) ∧
        approximates(Diff′, Store′, N))

pred(N, ⟨I,tail⟩, ⟨I′, tail⟩) ↔
    (empty(I,tail) ∧ input(I,I′)) ∨
    (full(N,I,tail) ∧ output(I,I′)) ∨
    (part-full(N,I,tail) ∧ input(I,I′)) ∨
    (part-full(N,I,tail) ∧ output(I,I′)) ∨
    (part-full(N,I,tail) ∧ input(I,I′) ∧ output(I,I′))

empty(I,tail) ↔ I=tail

full(N,I,tail) ↔ ∃ L1(length(I,L1) ∧ N=L1)

part-full(N,I,tail) ↔ ∃ L1(length(I,L1) ∧ N > L1)

length(X,L) ↔ ((X=tail ∨ X=nil) ∧ L=0) ∨
    ∃ Z,X′,L′ (X=Y.X′ ∧ L=s(L′) ∧ length(X′,L′))

input(I,I′) ↔
    ∃ X (append(I,X.tail,I′)) ∨ ∃ Y (append(Z,tail,I) ∧ append(Z,nil,I′))
output(I, X.I)

The definition of convert/4 is simplified:

convert(⟨I,O⟩, ⟨X_i,X_o⟩, ⟨X_i',X_o'⟩, ⟨I′,O′⟩) ↔

$\exists$ A(append(X_i',A.tail,X_i) $\wedge$ update(I,A,I') $\wedge$ O'=O) $\vee$

$\exists$ B(X_i = B.X_i' $\wedge$ update(O,B,O') $\wedge$ I=I')$\vee$

$\exists$ A,B,Tmp (X_i = B.Tmp $\wedge$ append(X_i', A.tail, Tmp) $\wedge$

$\wedge$ update(I,A,I') $\wedge$ update(O,B,O'))

We now combine pred/3 and convert/4:

approximates(Diff, Store, N) $\leftrightarrow$

$\exists$ X,Y,Z (Store=⟨nil,tail⟩ $\wedge$ Diff=⟨nil,nil⟩) $\vee$

$\exists$ X', Diff' (pred(N,Store, Store',Diff,Diff')

$\wedge$ approximates(Diff', Store', N))

pred(N, Store, Store', Diff, Diff') $\leftrightarrow$

$\exists$ X (empty(Store) $\wedge$ input(Diff,Diff',X) $\wedge$ add(X,Store,Store')) $\vee$

$\exists$ Y (full(N,Store) $\wedge$ output(Diff,Diff',Y) $\wedge$ remove(Y,Store,Store')) $\vee$

$\exists$ X (part-full(N,Store) $\wedge$ input(Diff,Diff',X) $\wedge$ add(X,Store,Store')) $\vee$

$\exists$ Y (part-full(N,Store) $\wedge$ output(Diff,Diff',Y) $\wedge$ remove(Y,Store,Store')) $\vee$

$\exists$ X,Y, TmpD, TmpS (part-full(N,Store) $\wedge$ input(Diff,TmpD,X) $\wedge$

output(TmpD,Diff',Y) $\wedge$ add(X,Store,TmpS) $\wedge$ remove(Y,TmpS,Store'))

input(⟨X.I,O⟩, ⟨I,O⟩, X)

output(⟨I,Y.O⟩, ⟨I,O⟩, Y)

add(X, ⟨A,tail⟩, ⟨B,tail⟩) $\leftrightarrow$ append(A, X.tail, B)

remove(Y, ⟨Y.A,tail⟩, ⟨A,tail⟩)

The description of the N-buffer can be rewritten as Horn clauses, replacing the argument representing the pair of input and output streams by two separate arguments. Again, we note that this form can be regarded as a logic program of the type $\mathcal{PROG}$ in language $L_{\mathcal{P}}$.

bufferN(nil,nil,Store,N) $\leftarrow$

empty(Store)

bufferN(In.Ins, Outs, Store, N) ←

empty(Store) ∧ add(In,Store,Store′) ∧ bufferN(Ins, Outs, Store′, N)

bufferN(Ins, Out.Outs, Store, N) ←

full(N,Store) ∧ remove(Out,Store,Store′) ∧ bufferN(Ins, Outs, Store′, N)

bufferN(In.Ins,Outs,Store,N) ←

part-full(N,Store) ∧ add(In,Store,Store′) ∧ bufferN(Ins, Outs, Store′, N)

bufferN(Ins, Out.Outs, Store, N) ←

part-full(N,Store) ∧ remove(Out,Store,Store′) ∧ bufferN(Ins, Outs, Store′, N)

bufferN(In.Ins, Out.Outs, Store, N) ←

part-full(N,Store)

∧ add(In,Store,Tmp) ∧ remove(Out,Tmp,Store′) ∧ bufferN(Ins, Outs, Store′, N)

input(⟨X.I,O⟩, ⟨I,O⟩, X)

output(⟨I,Y.O⟩, ⟨I,O⟩, Y)

add(X, ⟨A,tail⟩, ⟨B,tail⟩) ← append(A, X.tail, B)

remove(Y, ⟨Y.A,tail⟩, ⟨A,tail⟩)

3.10.2.5 A one-place buffer

A one-place buffer is an instance of bounded buffers and is a special case since the buffer store can only be full or empty, never part-full. Thus we can rewrite the Horn clause description for a one-place buffer:

buffer1(nil,nil,⟨nil,tail⟩) (i)

buffer1(In.Ins, Outs, Store) ← (ii)

empty(Store) ∧ add(In,Store,Store′) ∧ buffers1(Ins, Outs, Store′)

buffer1(Ins, Out.Outs, Store) ← (iii)
full(s(0),Store) ∧ remove(Out,Store,Store′) ∧ buffer1(Ins,Outs,Store′)

Rewriting the above, since a full store is ⟨X.tail,tail⟩ and an empty store is ⟨tail,tail⟩

buffer1(nil,nil) (i)

buffer1(In.Ins, Outs) ← (ii)
buffer1′(Ins, Outs, In)

buffer1′(Ins, Out.Outs, In) ← (iii)
Out=In ∧ buffer1(Ins,Outs)

Partially evaluating (ii) and (iii), and normalizing, we derive:

buffer1(Ins,Outs) ← (i)
Ins = nil ∧ Outs=nil

buffer1(Ins, Outs) ← (ii)
Ins = In.Ins′ ∧ Outs=In.Outs′ ∧ buffer1(Ins′,Outs′)

3.10.2.6 Composing buffer specifications

We showed in Section 3.10.2.4 how an N-place buffer 'program' could be derived from the set description of an N-place buffer. This 'program' uses a data structure (a difference-list in fact) to represent the stored items. In this section we propose that an N-place buffer can be described by the composition of N 1-place buffers. We claim that with a *suitable computation rule* the following goal behaves as an N-place buffer:

← buffer1(Ins, $Mids_1$) ∧ buffer1($Mids_1$, $Mids_2$) ∧ ... ∧ buffer1($Mids_{N-1}$,Outs)

The structure described is *static* in the sense that we require the N 1-place buffers to be composed at the initialization of the N-place buffer. A computation rule which enables

such a composition to behave in the desired way is discussed in the next chapter.

A two-place buffer can be described as follows:

buffer2(Ins, Mids, Outs) ←
    buffer1(Ins,Mids) ∧ buffer1(Mids,Outs)

In the above description the shared variable Mids is a public variable of the two-place buffer. This is not what is required, since another process composed with the two-place buffer could communicate with the two one place buffers via the variable Mids. The two-place buffer is better described as follows, where the variable Mids is hidden by being made local to the body of the clause for buffer2:

buffer2(Ins, Outs) ←
    buffer1(Ins,Mids) ∧ buffer1(Mids,Outs)

In general, an N-place buffer can be described by:

bufferN(N,Ins,Outs) ←
    N=0 ∧ Ins=Outs
bufferN(N,Ins,Outs) ←
    N=s(N′) ∧ buffer1(Ins,Mids) ∧ bufferN(N′, Mids, Outs)

3.10.3 Queues

3.10.3.1 Set description of an unbounded buffer

An unbounded buffer (LIFO queue) can be described as a buffer which is never full:

queue(Top,Set) ← queue(Top,Set, ⟨tail.tail⟩ .nil)

queue(Top, Set, Mins) ←
    next-rankset(Mins,Mins′) ∧ union(Mins, Set′, Set)
        ∧ queue(Top, Set′, Mins′)
queue(Top, Poset, Mins) ←
    Top.nil = Mins ∧ Set = Mins

next-rankset(nil,nil)

next-rankset(B.Bs,C) ←

set-pred(B,SetX) ∧ nopreds(Cs,SetX) ∧

union(Cs,Cs′,C) ∧ next-rankset(Bs,Cs′)

set-pred(B,SetX) ← setof-solutions(X, pred(B,X), SetX)

pred(⟨I,O⟩, ⟨I′, O′⟩) ←

append(Z,nil,I) ∧ append(Z,tail,O) ∧ I=I′ ∧ I=O′

pred(⟨I,O⟩, ⟨I′, O′⟩) ←

I=O ∧ O′=O ∧ input(I,I′)

pred(⟨I,O⟩, ⟨I′, O′⟩) ←

I ≠ O ∧ O′=O ∧ input(I,I′)

pred(⟨I,O⟩, ⟨I′, O′⟩) ←

I ≠ O ∧ I′=I ∧ output(I,O,O′)

pred(⟨I,O⟩, ⟨I′, O′⟩) ←

I ≠ O ∧ input(I,I′) ∧ output(I,O,O′)

3.10.3.2 Set description of an expedited data queue

An expedited data queue is similar to a normal queue, except that items queued can be either *normal* or *expedited*. The ordering of normal items is always preserved as is that of expedited items, but an expedited item in the store is always output before a normal item in the store. The definition of pred/2 and output/3 used in the ordinary queue definition must therefore be altered:

pred(⟨I,O⟩, ⟨I′, O′⟩) ←

samelength(I,O) ∧ O′=O ∧ input(I,I′)

pred(⟨I,O⟩, ⟨I′, O′⟩) ←
    not-samelength(I,O) ∧ O′=O ∧ input(I,I′)

pred(⟨I,O⟩, ⟨I′, O′⟩) ←
    not-samelength(I,O) ∧ I′=I ∧ output(I,O,O′)

pred(⟨I,O⟩, ⟨I′, O′⟩) ←
    not-samelength(I,O) ∧ input(I,I′) ∧ output(I,O,O′)

samelength(I,O) ← length(I,L) ∧ length(O,L)

not-samelength(I,O) ← length(I,L1) ∧ length(O,L2) ∧ L1 ≠ L2

output(I,O,O′) ←
    sortx(I,Ixs) ∧ sortx(O,Oxs) ∧ Ixs ≠ Oxs ∧
    append(Ixs′, Ix.tail,Ixs) ∧ append(Os,Ix.tail,O′)

output(I,O,O′) ←
    sortx(I,Xs,Ins) ∧ sortx(O,Xs,Ons) ∧
    append(O, X.Y, I) ∧ append(O, X.tail, O′)

sortx(Y,Xs) ←
    partition(Y,X,N) ∧ sort(X,Xs)

with suitable definitions for partition/3 (partitions a list of items into expedited and normal items), and sort/2.

3.10.3.3 Logic programs for queues

A queue is effectively an unbounded buffer. We propose that queues can be represented in logic programs of type $\mathcal{PROG}$ in two ways:

(1) As 'data structure' queues, based on the program derived in Section 3.10.2.4,

(2) As 'process network' queues, based on the ideas developed in Section 3.10.2.6.

Data structure queues

We can adapt the definition of an N-place buffer given on page 104 of Section 3.10.2.4 to define a queue using an unbounded list to represent the store of data items. Note that the routine to remove an item from the store fails if the store is empty.

queue(nil,nil,Store) ←
    empty(Store)

queue(In.Ins, Outs, Store) ←
    add(In,Store,Store′) ∧ queue(Ins, Outs, Store′)

queue(Ins, Out.Outs, Store) ←
    remove(Out,Store,Store′) ∧ queue(Ins, Outs, Store′)

queue(In.Ins, Out.Outs, Store) ←
    add(In,Store,Tmp) ∧ remove(Out,Tmp,Store′) ∧ queue(Ins, Outs, Store′)

Process network queues

In Section 3.10.2.4 we suggested that buffers programs written in language $L_{\mathcal{P}}$ can be composed, and in Chapter 4 we will show that this is the case for programs in SILCS, a language of type $L_{\mathcal{P}}$. In this section we explore the way in which a queue can be built from composed buffer programs.

The method described above for buffers builds an N-place buffer from the composition of N 1-place buffers. However, we cannot define a queue using this method since the following Horn clause does not describe the output of items from the queue.

queue(Ins,Outs) ← buffer1(Ins,Mids) ∧ queue(Mids,Outs)

What is required is a chain of one place buffers which can grow dynamically in size as

inputs are made, but permitting output at any time. In this version of the queue program, the number of 1-place buffers is incremented every time a message is received on the input, but removal of an item from the queue does not decrement the number of buffers:

queue(Ins,Outs) ←
    Ins=I.Ins′ ∧ queue(Ins′,Mids) ∧ buffer1′(Mids,Outs,I)

buffer1′(Ins, Out.Outs, In) ←
    Out=In ∧ buffer1(Ins,Outs)

buffer1(In.Ins, Outs) ←
    buffer1′(Ins, Outs, In)

In the next chapter we will show that such a program does behave as an unbounded buffer given a suitable computational rule for the language $L_{\mathcal{P}}$.

3.11 Summary

In this chapter we have described concurrent systems by observations which can be made of the public variables of the system. We consider these variables to be *logic variables* which have a write-once property. The set of observations of a concurrent system are partially ordered by a relation which defines that system.

The sets can be described as directed acyclic graphs comprising pairs of observations linked by the immediate predecessor relation. Descriptions of these sets in first order logic can be made based on the order relation and the predecessor pairs. Two types of Horn clauses, $\mathcal{SET}$ and $\mathcal{PROG}$, can be derived from the first order descriptions. These forms can be regarded as logic programs in the logic languages $L_{\mathcal{S}}$ and $L_{\mathcal{P}}$ respectively. The successful execution of $\mathcal{SET}$ on a suitable logic interpreter results in a variable in the query being bound to a data structure which represents the set of observations that can be made of the system described. The operational semantics of $L_{\mathcal{S}}$ do not have to incorporate concurrency and Prolog is a suitable model of such a language. The *behaviour* of $\mathcal{PROG}$ when executed

on a suitable interpreter is that described by $\mathcal{SET}$ and the operational semantics of $L_{\mathcal{P}}$ must include rules governing the concurrent execution of $\mathcal{PROG}$. In the next chapter we describe SILCS, a concurrent logic language of the type $L_{\mathcal{P}}$ and define its operational semantics.

Chapter 4

The logic language SILCS

4.1 Introduction

In the previous chapter we have shown how concurrent systems can be specified in full first order logic by sentences describing the graphs of the partially ordered sets of observations that can be of such systems. The two Horn clause forms $\mathcal{SET}$ and $\mathcal{PROG}$ which can be derived from these sentences can be regarded as logic programs in logic languages $L_{\mathcal{S}}$ and $L_{\mathcal{P}}$ respectively. The $\mathcal{SET}$ form explicitly describes sets of observations while the $\mathcal{PROG}$ form relates each observation to a final system state. The semantics required of $L_{\mathcal{S}}$ is indifferent with respect to concurrency; however, the definition of the semantics of $L_{\mathcal{P}}$ must incorporate concurrency.

This chapter proposes an operational semantics for $L_{\mathcal{P}}$ and describes the logic language SILCS as an example of such a language. The operational semantics of SILCS describes concurrent, parallel and sequential processes and employs all-solutions nondeterminism. Unification is an atomic action in SILCS. The design decisions taken regarding SILCS are explained and justified, and its computational model defined with reference to an idealised metainterpreter. Examples of specifications written in SILCS are presented.

SILCS is based on the Horn clause subset of first order predicate logic, with constraint evaluation. We describe in this thesis that subset of SILCS, which we will call $SILCS_U$, for which the only constraint permitted is equality, i.e. unification over terms. When no

ambiguity arises, we refer to $SILCS_U$ as SILCS.

4.2 Specifications, implementations and SILCS

The process that we have described so far in this thesis for the design and construction of concurrent systems comprises the following stages:

(1) formulating initial specifications in a restricted form of natural language,

(2) describing the systems in first order logic using the natural language specifications as a guide,

(3) deriving executable Horn clause descriptions in a language $L_{\mathcal{S}}$ of the sets of observations from the first order logic sentences,

(4) deriving Horn clause 'programs' in a language $L_{\mathcal{P}}$ from the first order logic sentences, whose *behaviour* is that described by the first order descriptions.

We now propose that the following stages will be required to achieve the implementation of the systems:

(5) deriving SILCS programs from the 'programs' of (4),

(6) deriving programs in committed choice concurrent logic programming languages from the SILCS programs.

We have sketched a methodology in Chapter 3 for relating the sentences produced by the activity in (2) to those in (1), and have shown how standard transformations can be used to derive the Horn clauses of (3) and (4) from (2).

In this chapter we propose a methodology for stage (5), the derivation of SILCS programs from the Horn clause programs in $L_{\mathcal{P}}$. However we do not regard SILCS as a language suitable for the *implementation* of concurrent systems due to

- the overheads associated with implementing synchronous communication,

- the all-solutions non-determinism of the language,

and instead we outline a method for stage (6) in Chapter 5. Neither do we propose that SILCS is used for the purpose of initially specifying systems in logic since full first order descriptions are often more concise and in some ways more expressive than Horn clause programs [59]. SILCS is a bridge between the first order descriptions of the behaviours of systems as posets of observations and programs written in committed choice concurrent logic programming languages.

Finally, there are testing stages which we have omitted from the above list. The methodology proposed in Chapter 6 forms the basis of conformance tests relating set descriptions (3), SILCS programs (5) and implementations (6).

4.3 SILCS as a specification language

SILCS can be regarded as a *constructive* specification language since it permits specifications to be constructed which can be interpreted mechanically. In this respect it is similar to algebraic languages like CCS [92], CSP [58] and LOTOS [66]. Its design has been influenced by the following general requirements:

- The specification technique should be capable of adequate expressibility for the problem domain.
- The technique should possess clearly defined semantics.
- Specifications made in the technique should be amenable to formal analysis.
- There should be clearly defined rules for transforming specifications into implementations in one or more target languages.

4.4 Expressiveness

The characteristics of the systems to be described dictate to a large extent the expressiveness of the description technique. The concurrent systems we have in mind are process-

based and characterized by the following dynamic behaviours:

- Creation (initiation)
- Termination
- Indeterminism[1] (nondeterminism) and choice.
- Synchronisation and communication
- Suspension
- Deadlock
- Livelock
- Dependence
- Progression

Such process based systems may be constructed from networks of processes. We view these networks as *trees*, or more properly *graphs* which may or may not be cyclic — such topologies may not necessarily admit the use of 'hierarchies' as a descriptive term. The specification technique should provide for the encapsulation of dynamic behaviour and allow for the determination of the possible observational equivalence of given systems.

4.5 Syntax of SILCS

The alphabet of SILCS consists of:

(1) variables, denoted by strings whose initial character is an uppercase letter,

(2) constants, denoted by strings whose initial character is a lower case letter,

(3) function symbols denoted by strings,

[1]Indeterminism has been equated with committed-choice nondeterminism and don't-care nondeterminism by Shapiro [119]. In [103] Ringwood distinguishes between indeterminism as a choice of one of several possible alternatives, and nondeterminism as a choice of all possible alternatives in the context of an automaton faced with a possibly branching computation.

(4) predicate symbols denoted by strings,

(5) the three connectives $\wedge$ & $\leftarrow$

(6) punctuation symbols "(" ")" ","

Predicate and functor names are strings whose initial character is a lower case letter. Predicates and functors are distinguished by context, in that a functor may appear only as an argument to a predicate, and predicates may not be arguments to predicates.

We repeat in a concise form some of the definitions of the syntax of first order logic which were given in Section 3.3.

- An *atom* (atomic formula) is an expression of the form $p(t_1,\ldots,t_n)$ where p is an n-place predicate symbol and $t_1 \ldots t_n$ are terms.
- *Terms* are variables, constants or expressions of the form $f(t_1,\ldots,t_n)$, where f is an n-place function symbol and $t_1 \ldots t_n$ are terms.
- A *literal* is an atom or the negation of an atom (respectively a positive literal or negative literal).

The syntax of SILCS is that of *definite clauses* and *definite goals*, i.e. Horn clauses. There is no negation in SILCS, but we assume the existence of the inequality check $\neq$. There are two conjunction operators in SILCS, "$\wedge$" and "&" which respectively flag simultaneous or sequential evaluation. A SILCS clause is an expression of the form

$$P \leftarrow C_1 \ <and\text{-}op> \ \ldots \ <and\text{-}op> \ C_n$$

and a goal is an expression of the form

$$\leftarrow C_1 \ <and\text{-}op> \ \ldots \ <and\text{-}op> \ C_n$$

where P and $C_1 \ldots C_n$ are positive literals. P is termed the *head* of the clause, and C_1 $<and\text{-}op>$ ... $<and\text{-}op>$ C_n the *body* comprising the conditions $C_1 \ldots C_n$. Each $<and\text{-}op>$ is either "$\wedge$" or "&". There are no guard or cut constructs in SILCS.

Clauses are assumed to be universally quantified and quantifiers are omitted. $C_1 \ldots C_n$ are the joint conditions and P the conclusion of such a conditional clause. Informally we

can say that 'for each assignment of each variable, if $C_1 \ldots C_n$ are all true, then P is true'. Brackets may be used to group conditions in the clause body to avoid ambiguity. A unit clause is a clause of the form

$$P$$

standing for "$P \leftarrow$" and is an unconditional clause for which we informally say 'for each assignment of each variable, P is true'.

Operators are permitted, and are functor symbols, being either pre-, post- or infix. There is one predefined operator, "."/2, which is infix and used to construct lists. The constant `nil` represents an empty list. Thus `a.b.c.nil` represents the complete list with three elements `a`, `b` and `c`.

The basic types allowed in SILCS_U in which the basic unit of computation is unification, are logic terms which represent data structures. Term unification is an atomic computation and is represented by the predicate =/2 written infix. In full SILCS we take *constraint* evaluation over reals, integers, strings, etc. as the basic unit of computation and the mechanism of communication rather than term unification.

Clauses of a system description in SILCS can be regarded as specifications of behaviour.

Definition 4.1 A *process specification* for P is the set of SILCS clauses with the same predicate symbol P. ■

Definition 4.2 A *system specification* is a finite set of process specifications, one of which is about the initial system state. ■

We adopt the convention that the initial process state is defined by a process specification comprising just *one* clause in normalised form. The predicate name of this process specification is `init` and each argument represents an *observable variable* (see Section 3.7.1). The execution of a system is initiated by the SILCS goal

$$\leftarrow init$$

where *init* is an atom of the form $\texttt{init}(V_1, V_2, \ldots, V_k, \ldots, V_n)$, $0 \leq n$, each V_k being an observable variable of the system. If $n = 0$ then the system is *closed*.

A note on explicit simultaneous and sequential operators

There are explicit simultaneous and sequential AND operators in SILCS, but only one OR operator (parallel-OR). Explicit simultaneous and sequential AND operators are required in a language designed for the description of concurrent systems (see Milner [94]). Some concurrent logic programming languages do not have an explicit sequential AND operator, but rely on suspension to achieve the ordering of operations. For this reason, we regard these languages as being insufficiently expressive for specification purposes. Examples of such languages are Concurrent Prolog [116] and Guarded Horn Clauses [131].

Some committed choice logic programming languages have an explicit sequential OR operator, for example the ";" of Parlog [51] and the "otherwise" of Concurrent Prolog. However, these constructs are useful for *programmers* who wish to implement default cases with the minimum of coding. The same result can be achieved in SILCS (in a more verbose manner) by the use of the inequality relation "$\neq$".

4.6 Semantics of SILCS

The semantics of SILCS can be described by the three kinds of semantics for logic programs first proposed by van Emden and Kowalski [136]. The *model-theoretic semantics* of SILCS is that of the standard minimal model semantics of definite clauses, and the fixpoint semantics of the language is a special case of this in that the latter only deals with sets of ground atomic formulae which are procedure declarations. The *operational semantics* of SILCS are given with reference to an idealised logic metainterpreter which employs the SLD-resolution[2] rule described by Kowalski [72], a refinement of the original procedure of

[2]Resolution is an inference rule which can be used to prove indirectly that a formula F is a logical consequence of a specification S. The negation of F is added to the axioms in S, and if a contradiction (the empty clause $\Box$) is derived, then $S \models F$, and the derivation is successful. Non-successful derivations may be failed or infinite. The standard propositional rule *modus tollens*

Robinson [105]. In this section we describe in detail the operational semantics of SILCS, since it forms the basis for the construction of an interpreter for the language.

The operation of an idealised logic interpreter has been discussed by Kowalski [74], and more recently by Hill and Lloyd [56]. Given any two languages, it may be possible to simulate the proof procedure of one language L_1 within the other L_2. This is accomplished by defining a binary relationship Pr in L_2 which holds when a conclusion can be derived from assumptions in L_1. We then refer to L_2 as the meta-language $\mathcal{ML}$ for the object language $\mathcal{OL}$ L_1. In general, a simple problem solver can behave like a more sophisticated one by acting as the meta-language for a more sophisticated object language.

In the following we assume that the object language (L_1) is SILCS , and the meta-language (L_2) is Pure Definite Clauses (PDC). Programs in PDC are Horn clauses and no computational strategy is defined for the language. Clauses in SILCS are named by terms in PDC. The binary relationship Pr defined in PDC takes a list of terms representing SILCS goals and attempts to demonstrate their provability with respect to a given SILCS program represented by a term in PDC.

Definition 4.3 [adapted from Kowalski [74]]. Given a representation of clauses by means of terms, a definition Pr in PDC correctly represents the provability relation, named 'demonstrate', of SILCS iff whenever X and Y are clauses of SILCS named by terms X' and Y' of PDC respectively, conclusion Y can be derived from assumption X in SILCS iff conclusion *demonstrate(X',Y')* can be derived from assumptions Pr in PDC. ■

If the relation Pr, expressed in the $\mathcal{ML}$, correctly represents the provability relation of the $\mathcal{OL}$, then the direct execution of the $\mathcal{OL}$ and its simulation in the $\mathcal{ML}$ are equivalent and interchangeable. This equivalence is identical to the *reflection principle* investigated by Weyhrauch [140].

We outline below the rules that define the top level of `demonstrate/2`: they are based on the standard interpreter presented in Kowalski's book [74].

demonstrate(Program, Goals) ←

is effectively employed in SLD-resolution. See [85] for a detailed discussion of SLD-resolution.

empty(Goals)

demonstrate(Program, Goals) ←
select(Goals, Goal, RestGoals) ∧
member(Procedure,Program) ∧
renamevars(Procedure, Goals, Procedure′) ∧
parts(Procedure′, Head, Body) ∧
match(Goal, Head, Substitutions) ∧
add(Body, RestGoals, TempGoals) ∧
apply(Substitution, TempGoals, NewGoals) ∧
demonstrate(Program, NewGoals)

The $\mathcal{ML}$ program interprets goals of the $\mathcal{OL}$ named by Goals above with respect to a program of the $\mathcal{OL}$ named by Program. Both Goals and Program refer to $\mathcal{ML}$ data structures, for example lists. The relations `select/3`, `member/2` and `add/3` are operations on these data structures. The initial goal is

← demonstrate(Program,Goals)

and the computation terminates with success when Goals is bound to a term standing for an empty list of $\mathcal{OL}$ goals. This binding represents the *empty clause* which has an empty head and empty body; it is denoted by □ and is interpreted as a contradiction.

We adapt the interpreter to accord with our intended semantics of SILCS_{U}:

(1) The atomic unit of computation is term unification.

(2) Atomic computational steps occur *simultaneously* iff they are conjoined by the "∧" operator, in which case:

   (a) Computations which do not share variables (and are thus not constrained by one another) may proceed independently, i.e. in parallel.

   (b) Computations which share variables are constrained by one another and must proceed concurrently. The communication of constraints (bindings to variables) synchronises all computations which share those variables.

(3) Atomic computational steps proceed *sequentially* iff they are conjoined by the " &" operator.

(4) A computation *suspends* iff

(a) either there is not enough data for it to execute,

(b) or it is constrained by a member of another sequence group (see Section 4.10.5).

(5) An atomic computation $A = B$ *succeeds* iff A and B can be unified, otherwise it *fails*.

SILCS primitives are defined whose execution suspends until their arguments are sufficiently instantiated for execution to occur. Included in the set of SILCS_U primitives are arithmetic evaluation and comparison predicates `is/2`, `>/2` and `</2`.

The initial goal to be proved by the interpreter about any system is the initial system state `init`, and the set of process specifications about system S constitute the SILCS program.

Clauses in SILCS are represented in the interpreter in *normalised form*.

Definition 4.4 A clause is in *normalised form* iff every there is a distinct variable in each argument place of the head. ■

A clause is converted to this form by replacing each non-variable (or non-distinct variable) argument in the head by a new variable not yet used in the clause and by the addition to the body of the clause of a call which unifies the old and new arguments. The new call is conjoined to the body with the "$\wedge$" operator. Thus the normalised form of:

$p(X,a,X) \leftarrow C_1 \wedge \ldots \wedge C_n$

is

$p(X,Y,Z) \leftarrow C_1 \wedge \ldots \wedge C_n \wedge Y{=}a \wedge Z{=}X$

We assume the existence of a predefined binary unification relation "=" written infix, whose operation is that defined by Robinson in [106], and incorporates the *occurs check*. We also assume the existence of a predefined binary inequality relation "$\neq$" written infix.

Success, failure and suspension

Algebraic specification techniques such as CCS and LOTOS do not have the notion of failure with respect to communication: an attempted communication can either succeed, or not yet happen (suspends). Standard proof procedures in first order logic have the notions of success or failure, or neither (ie infinite computation). Suspension can be described in terms of infinite (and useless) computations at the metalevel. We permit the specification of suspension using SILCS primitives, and hence the specification of deadlock.

4.7 Representation of processes in SILCS

We follow Shapiro [120] in the representation of process structures in the logic programming paradigm (Table 4.1). Additionally we represent atomic actions by unifications. In this interpretation each goal atom $g(T_1,\ldots,T_n)$ is regarded as a process whose name is the atom name g/n and whose state is represented by the n arguments to the atom. An entire goal is viewed as a network (possibly acyclic) of processes where interconnection patterns are specified by shared logical variables. Communication is by instantiation of these variables, the atomic action of a system being represented by a call to unification.

Process model	Horn clause form
atomic action	unification
process	goal
process network	resolvent (conjunctive goal)
process action	goal reduction
instruction for process action	clause
communication channel	shared logical variable
communication	instantiation of shared logical variables

Table 4.1: Processes and logic programs

4.8 Communication in SILCS

Communication in SILCS is represented by the bindings made to shared variables which can be instantiated to rich data structures. Such a variable is called a *channel* variable, or a *stream* variable in the case that its final state is a list (see Definition 3.37 on page 66, and Definition 3.42 on page 73). In the case of stream variables, messages correspond to the terms to which the leaves of the list are bound. The operational semantics of the interpreter which incorporates atomic unification ensures that the communication specified in SILCS is synchronous and that shared variables cannot act as unbounded buffers.

There is no primitive assignment operation in SILCS and neither are there data-flow annotations in SILCS programs. Hence there is no distinction between *input* and *output* in the program text; the functions of each are subsumed by unification. Channel names may be passed between processes as messages on other channels in order to permit dynamic reconfiguration of process networks.

4.9 Observational equivalence

A scheme for observational equivalence is a central part of any specification scheme based on observations: the behaviour of a process is categorised by how it appears to an external observer. Two processes are equivalent if no observations can distinguish between them, and two subprograms are congruent if the result of placing each of them in any program context yields two equivalent programs (see Hennessy [55]).

We define observational equivalence on processes to be a relation between their histories, represented by bindings made to variables, ordered by the type of the data structure to which the variables become bound. Thus channel histories are observations in SILCS. We ignore unobservable or 'hidden' actions, i.e. those which bind variables local to a clause. A variable is local to a clause iff that variable occurs in the body of a clause but does not occur in the head of that clause. Even though the execution mechanism for such programs might in reality substitute these calls into one run queue, from the observer's point of view they are invisible.

4.10 The operation of the idealised SILCS interpreter

In this section we detail the operation of the idealised PDC interpreter for SILCS a simplified version of which was outlined in Section 4.6. The interpreter is more complex due to the way in which concurrency and sequentiality are represented operationally. The SILCS interpreter repeatedly collects individual calls into *atomic* or *sequence* groups and processes them according to the rules which define the operational semantics of concurrency and sequentiality.

Definition 4.5 A SILCS *goal* is a negative clause of the form

$$\leftarrow G_1 \wedge \ldots \wedge G_n \qquad (n > 0)$$

where each of $G_1,\ldots,G_n$ is either a call or a sequence group. ■

Definition 4.6 A *call* is an atomic formula (atom). ■

Definition 4.7 A *sequence group* is of the form A&B where A and B are goals. The *head* of the sequence group is A and the *tail* of the group is B. ■

4.10.1 Atomic groups and constraints

Atomic groups are treated as the unit of atomic reduction in the interpreter — no results of reductions of the members of an atomic group are made available until all of its members have been reduced successfully. The reduction of an atomic group suspends iff at least one member of the atomic group suspends. Distinct atomic groups are reduced in parallel by the interpreter.

Definition 4.8 An atomic group comprises calls and sequence groups which are constrained by one or more variables. ■

Constraint in general can be *direct* or *indirect*. Membership of an atomic group in the idealised SILCS interpreter is determined on the basis of indirect constraint.

Definition 4.9 Two or more calls are *directly constrained* iff they share the same variable. ■

For example the atomic computations (X=Y ∧ X=Z) share the variable X on which they are directly constrained and are members of the same atomic group. The same is true of the calls (p(X) ∧ q(X,Y)). However the evaluation of non-atomic calls which do not directly share may still result in constraints being generated; consider the following goal and SILCS program:

← a(X) ∧ b(X,Y) ∧ c(Y)

a(X)
b(X,Y) ← X=Y
b(X,Y)
c(Y)

The result of the call to `b/2` for one computation when selecting the first clause of `b/2` is to unify the variables X and Y; hence the calls a(X) and c(Y) are *indirectly* constrained for that computation. However X and Y are not constrained when the second clause of `b/2` is selected.

Definition 4.10 Two or more calls are *indirectly constrained* during a computation iff during the course of the computation one or more variables in the arguments of one are unified with variables in arguments of the other. ■

Since membership of atomic groups in SILCS is determined on the basis of indirect constraint, the same calls may be grouped differently for different computations. We assume that static analysis of a SILCS specification is employed to determine the possible indirect constraints for each computation. In the worst case this analysis is equivalent in computational complexity to interpreting the specification.

An alternative strategy for the determination of membership of atomic groups is *dynamic* analysis during the operation of the interpreter, involving the analysis and possible regrouping of all calls in the goals queue after each atomic group has been reduced. This

method inhibits possible parallel execution of the interpreter.

A computationally less expensive (but less accurate) method is to determine membership on the basis of *possible constraints*. For example the calls a(X) and c(Y) are possibly constrained in the goal (a(X) ∧ b(X,Y) ∧ c(Y)) since during the computation the evaluation of b(X,Y) may unify its arguments.

Definition 4.11 Two calls are *possibly constrained* iff each one shares one or more variables with another common call. ■

This method of analysis is employed in the implementation of the SILCS interpreter (Appendix A).

4.10.2 Sequence groups

Sequence groups have been defined in Definition 4.7. A sequence group is included in an atomic group if any of the calls that constitute sequence group contains a variable on which the atomic group is founded. Since the *head* of a sequence group consists of the goal (or goals) in that group which form the first argument to the first & operator in the group, the head can comprise one or more sequence groups. For example, the following calls constitute one sequence group: `(a∧b)&(c∧d)`. The head of the group is `(a∧b)`.

Only the head of the sequence group is reduced during in the processing of an atomic group; the sequence group is then returned to the goals queue with the head replaced by its reduction (see below). This requires the dynamic analysis of goals in the atomic group and the possible regrouping of them into new atomic groups.

4.10.3 Reduction strategy of the SILCS interpreter

The SILCS interpreter reduces calls by grouping them into atomic groups and then reducing each atomic group as a unit. The following strategy is used for interpreting SILCS programs:

(1) Initialise the suspension queue to empty and *exitstatus* to the empty string `''`.

(2) Group the calls in the goal into distinct atomic groups to form the run queue.

(3) While the run queue is not empty do

select an atomic group for reduction and attempt to reduce it according to the strategy for reduction of atomic groups

- if the result is success, then add the reduced form (one or more atomic groups) to the run queue
- if the result is failure then set *exitstatus* to `fail` and exit.
- if the result is suspension, then add the atomic group to the suspension queue.

end-while.

(4) If *exitstatus* $\neq$ `fail` then

- if the suspension queue is empty then set *exitstatus* to success
- else if the suspension queue is not empty then set *exitstatus* to `deadlock`.

(5) exit with *exitstatus*.

4.10.4 Reduction of members of an atomic group

(1) Initialise *local-exitstatus* to the empty string ''.

(2) Divide the members of the atomic group into suspended goals and runnable goals according to the rules of suspension (see below).

(3) Initialise the local suspension queue to the suspended goals.

(4) Initialise the local run queue to the runnable goals.

(5) Initialise the local reduced queue to `empty`.

(6) While the local run queue is not empty do

select a goal for reduction

- if the goal is a call C then

(a) find a clause C'← B such that C and C' can be unified (if no such clause exists then exit with *exitstatus* set to `fail`)

(b) rename the variables in the clause so that they are distinct from the variables in the call

(c) unify C and C' goal with substitution S

(d) add the body B of the clause to the local reduced queue

(e) apply the substitutions to the local run queue, local reduced queue and the local suspension queue.

- if the goal is a sequence group A&B then attempt to reduce A using the reduction strategy for atomic groups
  - if the reduction suspends then add the sequence group to the local suspension queue
  - if the reduction fails then exit with *exitstatus* set to `fail`
  - if the reduction succeeds, returning A' (the set of calls representing the reduced form of A) then
    - if A' is empty then add B to the local reduced queue
    - else add A'&B to the local reduced queue

end-while.

(7) (a) If *exitstatus*=`fail` then return *exitstatus* else

(b) if the local reduced queue is empty and the local suspension queue is not empty then *exitstatus*:=`suspension` and return ⟨*exitstatus* , suspension queue⟩

(c) else regroup the local reduced queue and the local suspension queues as one or more atomic groups, set *exitstatus* to `success` and return ⟨*exitstatus* , new atomic group(s)⟩.

4.10.5 Suspension

Suspension within an atomic group is determined as follows:

Definition 4.12 A goal *G* is suspended on a sequence group *SG* iff

(1) G is a call and shares one or more variables with the tail of SG but does not share variables with the head of SG.

(2) G is a sequence group and a call in its head shares one or more variables with the tail of SG but does not share variables with the head of SG.

■

Note that an atomic group can be suspended on a sequence group which is itself suspended.

Examples

(1) Given the atomic group {f(A), (g(B)&f'(A))}, the call f(A) is suspended on the variable A which it shares with the call f'(A) in the second position in the sequence group (g(B)&f'(A)).

(2) Given the atomic group {f(A), (g(B)&f'(A)), (h(C)&g'(B))}, the call f(A) is suspended on the variable A shared with the call f'(A) in the sequence group (g(B)&f'(A)), and the sequence group (g(B)&f'(A)) is suspended on the variable B shared with the call g'(B) in the sequence group (h(C)&g'(B))

(3) Given the atomic group {f(A), ((g(B)∧ (h(C)&f'(A)))&j(D))}, the call f(A) is suspended on the variable A which it shares with f'(A) in the sequence group (h(C)&f'(A)).

Definition 4.13 An atomic group is suspended iff no calls are runnable and there are suspended calls. ■

Definition 4.14 A goal is deadlocked iff all the atomic groups which it comprises are suspended. ■

For example the following goal is deadlocked:

← (g(B)&f'(A)) ∧ (f(A)&g'(B))

since the goal comprises one atomic group {(g(B)&f'(A)), (f(A)&g'(B))} with two members which are mutually suspended:

(1) (g(B)&f'(A)) is suspended on the variable B in the sequence group (f(A)&g'(B))

(2) (f(A)&g'(B)) is suspended on the variable A in the sequence group (g(B)&f'(A))

4.10.6 Output of the interpreter

The interpreter halts with *success* iff the goals list is empty, with *fail* iff any goal fails or with a *deadlock* indication iff the attempted reduction of each atomic group in the goals list is suspended.

The output of the interpreter consists of the bindings made to the variables of the initial goal state. These bindings are made available to an observer and constitute the *complete traces* of the specification. Each *complete trace* represents one path through the graph of the poset of observations. The set of all of these these traces thus comprises all the paths through the graph.

The observer is permitted to observe bindings made simultaneously to different variables. Computations which do not produce bindings on a variable visible to the observer are invisible or *internal* computations.

4.11 Transition rules describing the semantics of SILCS

In this section we present the semantics of SILCS as a labelled transition system, as discussed in [70].

Definition 4.15 A *labelled transition system* TS is a tuple $TS =< S, Act, T, s_0 >$ where:

- S is a *non-empty* set of *states*,
- Act is a *non-empty* set of *atomic actions*,
- $T = \{—\mu \rightarrow \subseteq S \times S \mid \mu \subseteq Act\}$ is a set of *transition relations*,
- $s_0 \in S$ is the *initial state*.

■

We only study labelled transition systems with a countable set of states, since we describe finite (terminating) systems in our descriptions.

A *state* comprises a goal G (i.e. a process or system) and a substitution θ, denoted by the pair $< G, \theta >$. The initial state is the initial goal about a system and the empty substitution ε. The empty goal (success) is represented by $\square$. Failure is represented by $\blacksquare$.

An atomic action is a unification $X = Y$ where X and Y are terms. We represent the most general unifier associated with an atomic action by the set of bindings $\{V_1/t_1, \ldots, V_n/t_n\}$ where $V_1 \ldots V_n$ are the variables in X and Y which are bound to the terms $t_1 \ldots t_n$.

Given a state $< S_n, \theta >$, a labelled transition is of the form

$$< S_n, \theta > \text{—}\mu\text{→} < S_{n+1}, \theta \circ \mu' >$$

indicating that S_n performs the action set μ and transforms into S_{n+1} where

(i) $< S_n, \theta >$ is the initial state in the transition,

(ii) S_n moves to S_{n+1} by action set μ,

(iii) $< S_{n+1}, \theta \circ \mu' >$ is the final state in the transition,

(iv) $\theta \circ \mu'$ denotes the substitution whose application has the effect of applying θ and then applying the most general unifier μ' of the action set μ.

The use of a labelled transition on its own in a definition is an *axiom*.

Note that we denote the *variable set* of a term or set of terms μ by $v(\mu)$ (see Definition 3.32, page 62).

A *rule* is expressed by:

$$\frac{\mathrm{B} \text{—x→} \mathrm{B}'}{\mathrm{B}_1 \text{—x→} \mathrm{B}_1'} \qquad (\textit{Condition})$$

where $\mathrm{B} \text{—x→} \mathrm{B}'$ is a precondition, B_1 composes B with other goals, and $\mathrm{B}_1 \text{—x→} \mathrm{B}_1'$ if all the conditions in the set *Condition* are true.

4.11.1 Axioms

The axioms in our system are:

$$< \mu, \theta > \text{—}\mu\rightarrow < \Box, \theta \circ \mu' > \qquad \text{iff TRUE}(\mu)$$

$$< \mu, \theta > \longrightarrow < \blacksquare, \theta > \qquad \text{iff FALSE}(\mu)$$

$$< \Box \wedge B, \theta > \longrightarrow < B, \theta >$$

$$< A \wedge \Box, \theta > \longrightarrow < A, \theta >$$

4.11.2 Rules

The rules of sequential composition are:

$$\frac{< A, \theta_a > \text{—}\mu\rightarrow < A', \theta_a \circ \mu' >}{< A\&B, \theta > \text{—}\mu\rightarrow < A'\&B, \theta \circ \mu' >} \qquad A' \notin \{ \Box , \blacksquare \}$$

$$\frac{< A, \theta_a > \text{—}\mu\rightarrow < \Box, \theta_a \circ \mu' >}{< A\&B, \theta > \text{—}\mu\rightarrow < B, \theta \circ \mu' >}$$

$$\frac{< A, \theta_a > \longrightarrow < \blacksquare, \theta_a >}{< A\&B, \theta > \longrightarrow < \blacksquare, \theta >}$$

The rules of parallel composition are:

$$\frac{< A, \theta_a > \longrightarrow < \blacksquare, \theta_a >}{< A \wedge B, \theta > \longrightarrow < \blacksquare, \theta >}$$

$$\frac{< B, \theta_b > \longrightarrow < \blacksquare, \theta_b >}{< A \wedge B, \theta > \longrightarrow < \blacksquare, \theta >}$$

$$\frac{< A, \theta_a > \text{—}\mu_a\rightarrow < A', \theta_a \circ \mu'_a >}{< A \wedge B, \theta > \text{—}\mu_a\rightarrow < A' \wedge B, \theta \circ \mu'_a >} \quad \begin{pmatrix} A' \neq \blacksquare \\ v(\mu_a) \cap v(B) = \emptyset \end{pmatrix}$$

$$\frac{< B, \theta_b > \text{—}\mu_b\rightarrow < B', \theta_b \circ \mu'_b >}{< A \wedge B, \theta > \text{—}\mu_b\rightarrow < A \wedge B', \theta \circ \mu'_b >} \quad \begin{pmatrix} B' \neq \blacksquare \\ v(\mu_b) \cap v(A) = \emptyset \end{pmatrix}$$

$$\frac{\begin{array}{l} < A, \theta_a > \text{—}\mu_a\rightarrow < A', \theta_a \circ \mu'_a > \text{ and} \\ < B, \theta_b > \text{—}\mu_b\rightarrow < B', \theta_b \circ \mu'_b > \end{array}}{< A \wedge B, \theta > \text{—}\mu_a \cup \mu_b\rightarrow < A' \wedge B', \theta \circ \mu'_a \circ \mu'_b >} \quad \begin{pmatrix} A' \neq \blacksquare, B' \neq \blacksquare \\ v(\mu_a) \cap v(B) = \emptyset \\ v(\mu_b) \cap v(A) = \emptyset \end{pmatrix}$$

$$\frac{\begin{array}{l} < A, \theta_a > \text{—}\mu_a\rightarrow < A', \theta_a \circ \mu'_a > \text{ and} \\ < B, \theta_b > \text{—}\mu_b\rightarrow < B', \theta_b \circ \mu'_b > \end{array}}{< A \wedge B, \theta > \text{—}\mu_a \cup \mu_b\rightarrow < A' \wedge B', \theta \circ \mu'_a \circ \mu'_b >} \quad \begin{pmatrix} A' \neq \blacksquare, B' \neq \blacksquare \\ v(\mu_a) = v(\mu_b) \end{pmatrix}$$

$$\frac{\begin{array}{l} < A, \theta_a > \text{—}\mu_{a_1} \cup \mu_{a_2}\rightarrow < A', \theta_a \circ \mu'_a > \text{ and} \\ < B, \theta_b > \text{—}\mu_{b_1} \cup \mu_{b_2}\rightarrow < B', \theta_b \circ \mu'_b > \end{array}}{< A \wedge B, \theta > \text{—}\mu_{a_1} \cup \mu_{b_1} \cup \mu_{a_2} \cup \mu_{b_2}\rightarrow < A' \wedge B', \theta \circ \mu'_{a_1} \circ \mu'_{a_2} \circ \mu'_{b_1} \circ \mu'_{b_2} >}$$

$$\begin{pmatrix} A' \neq \blacksquare, B' \neq \blacksquare \\ v(\mu_{a_1}) \cap v(B) = \emptyset \\ v(\mu_{b_1}) \cap v(A) = \emptyset \\ v(\mu_{a_2}) = v(\mu_{b_2}) \end{pmatrix}$$

where

(1) $v(X)$ is the set of observable variables of X

(2) if X and Y are substitution sets, then $mgu(X, Y)$ is the most general unifier of these substitutions.

Note that a deadlock situation arises for the following composition:

$$< A, \theta_a > -\mu_a\rightarrow < A', \theta_a \circ \mu'_a > \text{ and } < B, \theta_b > -\mu_b\rightarrow < B', \theta_b \circ \mu'_b >$$

$$\begin{pmatrix} A' \neq \blacksquare,\ B' \neq \blacksquare \\ v(\mu_a) \cap v(\mu_b) = \emptyset \\ v(\mu_a) \subseteq v(B) \\ v(\mu_b) \subseteq v(A) \end{pmatrix}$$

4.12 Metalevel facilities in SILCS

SILCS has a construct `call/1` similar to the 'call' in Prolog. The argument of `call/1` is a term representing a single SILCS goal, or a conjunction of such goals. If the term is insufficiently instantiated then the invocation of `call/1` suspends. For example the goal

$$\leftarrow \text{call(X) \& X= (0 < s(0))}$$

deadlocks (suspends forever) but

$$\leftarrow \text{call(X) } \wedge \text{ X= (0 < s(0))}$$

succeeds with {X/(0 < s(0))}.

The metalevel facilities of SILCS permit the writing of specification interpreters in SILCS itself and the construction of tools to aid the specifiers. The basic SILCS-in-SILCS interpreter has an operational semantics based on the algorithm given in Section 4.10.3. Processes can be treated as first class objects by the use of the metacall since process names can be passed in messages.

4.13 SILCS programs

In this section we illustrate the derivation of SILCS programs from Horn clause specifications for the producers and buffers, and show that such programs can be compared using an equivalence relation.

SILCS has sequential and simultaneous AND-conjunction operators, "&" and "$\wedge$" respectively. Any translation scheme from the Horn clause form of descriptions into SILCS

programs has to take into account the mapping of the logical $\wedge$ operator into SILCS AND-conjunction operators.

4.13.1 Stream producers

We take as an example the translation of the produces/2 description into a SILCS program. The Horn clause description given in Section 3.10.1.2 on page 93 is reproduced below:

produces(StreamVar, List) $\leftarrow$

StreamVar=nil $\wedge$ List=nil

produces(StreamVar, List) $\leftarrow$

StreamVar = X.StreamVar' $\wedge$ List=X.List' $\wedge$ produces(StreamVar', List')

The semantics of SILCS regarding the composition and reduction of atomic groups means that all the goals within each clause (clauses being considered separately) belong to one atomic group and are reduced together. As an example, consider the following goal w.r.t. the above program.

$\leftarrow$ produces(V, a.b.c.nil)

We rewrite the goal by unfolding:

$\leftarrow$ V=a.V' $\wedge$ V'=b.V'' $\wedge$ V''=c.V''' $\wedge$ V'''=nil

The goal comprises one atomic group of concurrent unifications:

{V=a.V' , V'=b.V'' , V''=c.V''' , V'''=nil }

The operational semantics dictate that the unifications in the atomic group are performed concurrently so that the variable V is unified with the list `a.b.c.nil` in one atomic reduction. The effect of this is that messages a, b and c are produced simultaneously. The poset of bindings to the stream variable V is { tail, a.b.c.nil } where `tail` stands for the initial state of the stream variable.

In order to force the sequential production of messages the Horn clause description has to be transformed into SILCS with sequential AND operators between the unification on the stream variable and the recursive call to the produces relation:

produces(StreamVar, List) ← (i)

StreamVar=List & List=nil

produces(StreamVar, List) ← (ii)

(StreamVar=X.StreamVar′ ∧ List=X.List′) & produces(StreamVar′, List′)

The unfolding of the call

← produces(V, a.b.c.nil)

now becomes

← V=a.V′ & V′=b.V″ & V″=c.V‴ & V‴=nil

The goal thus comprises *one* atomic group containing one member, a sequence group of four elements each of which is a unification:

{(V=a.V′ & V′=b.V″ & V″=c.V‴ & V‴=nil)}

The operational semantics of SILCS dictate that the elements of the sequence group are reduced sequentially. This causes the unifications to be performed on the stream variable sequentially and the messages are produced in the sequence `a ; b ; c`. The poset of instantiations of the stream variable V is { tail, a.tail, a.b.tail, a.b.c.tail, a.b.c.nil } where `tail` stands both for the initial state of the stream variable and for the tail variable of each of the incomplete lists in the set.

4.13.2 Bounded Buffers

The Horn clause description of bounded buffers needs to be translated into SILCS with care. We take as our first example the one-place buffer described in Section 3.10.2.5 on page 106:

buffer1(Ins,Outs) ←

Ins = nil ∧ Outs=nil

buffer1(Ins, Outs) ←

Ins = In.Ins′ ∧ Outs=In.Outs′ ∧ buffer1(Ins′,Outs′)

Consider the above program and the pure Horn clause form of the stream producer (page 136) together with the following goal

← produces(V, a.b.c.nil) ∧ buffer1(V,O)

This query unfolds to:

$$
\begin{aligned}
\leftarrow \quad & V=a.V' \wedge V=X.V_1' \wedge O=X.O' \wedge \\
& V'=b.V'' \wedge V_1'=Y.V_1'' \wedge O'=Y.O'' \wedge \\
& V''=c.V''' \wedge V_1''=Z.V_1''' \wedge O''=X.O''' \wedge \\
& V'''=nil \wedge V_1'''=Z1 \wedge O'''=Z1
\end{aligned}
$$

As in the case of the stream producer the calls in the query comprise one atomic group of concurrent unifications, and the effect is that no buffering takes place — the variables V, O and List are unified in one atomic reduction. The poset of instantiation states of the buffer variable pair ⟨V,O⟩ is { ⟨tail,tail⟩, ⟨a.b.c.nil,a.b.c.nil⟩ } where `tail` stands for the initial state of the stream variables *V* and *O*.

Using the version of producers/2 with the sequential operators (page 137) results in the following unfolding of the query:

$$
\begin{aligned}
\leftarrow \quad & (V=a.V' \;\&\; V'=b.V'' \;\&\; V''=c.V''' \;\&\; V'''=nil) \\
& \wedge O=X.O' \wedge O'=Y.O'' \wedge O''=Z.O''' \wedge O'''=Z1 \\
& \wedge V=X.V_1' \wedge V_1'=Y.V_1'' \wedge V_1''=Z.V_1''' \wedge V_1'''=Z1
\end{aligned}
$$

The atomic group is

{ (V=a.V′ & V′=b.V″ & V″=c.V‴ & V‴=nil), O=X.O′ , O′=Y.O″ , O″=Z.O‴ , O‴=Z1 , $V=X.V_1'$, $V_1'=Y.V_1''$, $V_1''=Z.V_1'''$, $V_1'''=Z1$ }

All the unifications on the variables V and O are performed concurrently with the first element in the sequence group of computations associated with the producer. Thus even though the producer is sequenced, the variables V and O are bound to the list a.Y.Z.Z1 on the first reduction. The messages a,b,c are not buffered, but merely transmitted in the sequence `a ; b ; c`. The message slots represented by the variables Y and Z are created

eagerly by bindings to O at the start of the computation. In the following representation of the poset of bindings we denote variables (other than the tail of a list) by prefixing the variable name with "$" (see Section 3.5).

{ ⟨tail,tail⟩, ⟨a.$Y.$Z.tail,a.$Y.$Z.tail⟩, ⟨a.b.$Z.tail,a.b.$Z.tail⟩, ⟨a.b.c.tail,a.b.c.tail⟩, ⟨a.b.c.nil,a.b.c.nil⟩ }

In order to overcome the problem of 'eager output' from the buffer and to make it buffer the items correctly, we introduce sequencing into the buffer:

buffer1(Ins,Outs) ←
    Ins = nil & Outs=nil

buffer1(Ins, Outs) ←
    Ins = In.Ins′ & Outs=In.Outs′ & buffer1(Ins′,Outs′)

The unfolded query now is:

← (V=a.V′ & V′=b.V″ & V″=c.V‴ & V‴=nil) ∧
(V=X.V_1' ∧ O=X.O′ ∧ V_1'=Y.V_1'' ∧ O′=Y.O″ ∧
V_1''=Z.V_1''' ∧ O″=Z.O‴ V_1'''=Z1 ∧ O‴=Z1)

which forms an atomic group comprising two members each of which is a sequence group.

Sequencing is enforced on the instantiations of the variables V and O by the semantics of SILCS. The first reduction performed is the pair of atomic computations (V=a.V′ ∧ V=X.V_1') resulting in the binding set {X/a, V′/V_1'}. The poset of observations of the buffer is:

{⟨tail,tail⟩, ⟨a.tail,tail⟩, ⟨a.tail,a.tail⟩, ⟨a.b.tail,a.tail⟩, ⟨a.b.tail,a.b.tail⟩, ⟨a.b.c.tail,a.b.tail⟩, ⟨a.b.c.tail,a.b.c.tail⟩, ⟨a.b.c.nil,a.b.c.tail⟩, ⟨a.b.c.nil,a.b.c.nil⟩ }

and is a path through the graph of the poset, the items strictly ordered by pred(s(0),X,Y), depicted in Figure 4.1.

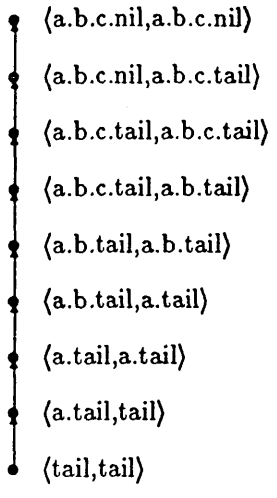

Figure 4.1: Graph of bindings for a one-place buffer

4.14 Equivalences

The characterization of processes through the description of the poset of bindings to their observable variables permits the comparison of programs which implement these descriptions. Using N-place buffers as an example we show how process descriptions can be derived by induction and relate process-network and data-store buffers to these descriptions.

4.14.1 Data store buffers

SILCS programs for N-place buffers using a list to store the buffered items can be derived directly from the poset descriptions of N-place buffers. We base the SILCS program on the Horn clause definition of the bufferN relation derived previously in Section 3.10.2.4, page 104. Note that `tail` is a constant in the SILCS program, and that we write (A,B) for the tuple ⟨A,B⟩.

bufferN(nil,nil,Store,N) ←

    empty(Store)

bufferN(In.Ins, Outs, Store, N) ←

    empty(Store) ∧ add(In,Store,Store′) ∧ bufferN(Ins, Outs, Store′, N)

bufferN(Ins, Out.Outs, Store, N) ←
full(N,Store) ∧ remove(Out,Store,Store′) ∧ bufferN(Ins, Outs, Store′, N)

bufferN(In.Ins,Outs,Store,N) ←
part-full(N,Store) ∧ add(In,Store,Store′) ∧ bufferN(Ins, Outs, Store′, N)

bufferN(Ins, Out.Outs, Store, N) ←
part-full(N,Store) ∧ remove(Out,Store,Store′) ∧ bufferN(Ins, Outs, Store′, N)

bufferN(In.Ins, Out.Outs, Store, N) ←
part-full(N,Store) ∧ add(In,Store,Tmp) ∧ remove(Out,Tmp,Store′) ∧
bufferN(Ins, Outs, Store′, N)

add(X, (A,tail), (B,tail)) ← append(A, X.tail, B)

remove(Y, (Y.A,tail), (A,tail))

empty(I,tail) ← I=tail

full(N,I,tail) ← length(I,L1) ∧ N=L1

part-full(N,I,tail) ← length(I,L1) ∧ N > L1

length(tail,0)
length(X.Y,s(L)) ← length(Y,L)

We can represent the store as a difference-list with a variable replacing the constant `nil`. Thus the store (a.b.c.tail,tail) will now be represented as (a.b.c.X,X). This means that addition to the store can now be performed in constant time and that addition and removal of items can be concurrent. Checking the state of the store requires the use of ==/2, which does not bind its arguments, to avoid incorrect bindings to the tail variable of the difference-list, for example:

empty(X,Y) ← X == Y

A neater solution, proposed below, is to add to the buffers program an extra argument representing the number of items in the store, and to use it to reason about the state of the store. The value of the argument is initialized to 0 when the buffer is invoked.

```
bufferN(nil,nil,Store,N,Size) ←
        empty(Size)
bufferN(In.Ins, Outs, Store, Size) ←
        empty(Size) ∧ add(In,Store,Store′) ∧ bufferN(Ins, Outs, Store′, N, s(0))

bufferN(Ins, Out.Outs, Store, N, Size) ←
        full(N,Size) ∧ remove(Out,Store,Store′) ∧ bufferN(Ins, Outs, Store′, N)

bufferN(In.Ins,Outs,Store,N, Size) ←
        part-full(N,Size) ∧ add(In,Store,Store′) ∧ bufferN(Ins, Outs, Store′, N, s(Size))

bufferN(Ins, Out.Outs, Store, N, Size) ←
        part-full(N,Size) ∧ remove(Out,Store,Store′) ∧
        Size=s(Size′) ∧ bufferN(Ins, Outs, Store′, N, Size′)

bufferN(In.Ins, Out.Outs, Store, N, Size) ←
        part-full(N, Size) ∧ add-remove(In,Out,Store,Store′) ∧
        bufferN(Ins, Outs, Store′, N, Size)

add(X, (A,X.Tail), (A,Tail))
remove(Y, (Y.A,Tail), (A,Tail))

add-remove(X,Y, (Y.A,X.Tail), (A,Tail))

empty(0)

full(N,Size) ← Size=N
```

part-full(N,Size) ← Size > 0 ∧ N > Size

s(X) > 0

s(X) > s(Y) ← X > Y

The behaviour of the above SILCS program does not conform to that predicted by the poset description of a bounded buffer due to the semantics of unification in SILCS. As an example, we consider a two-place buffer as an instance of N-place buffers. We compose the buffer with the sequenced stream producer described earlier:

← bufferN(Ins,Outs, (X,X), s(s(0)), 0) ∧ producer(Ins, a.b.c.nil)

The expected poset of the observations of ⟨Ins,Outs⟩ for this goal is:

{⟨tail,tail⟩, ⟨a.tail,tail⟩, ⟨a.b.tail,tail⟩, ⟨a.tail,a.tail⟩, ⟨a.b.tail,a.tail⟩, ⟨a.b.tail,a.b.tail⟩, ⟨a.b.c.tail,a.tail⟩, ⟨a.b.c.tail,a.b.tail⟩, ⟨a.b.c.tail,a.b.c.tail⟩, ⟨a.b.c.nil,a.b.tail⟩, ⟨a.b.c.nil,a.b.c.tail⟩, ⟨a.b.c.nil,a.b.c.nil⟩ }

We represent this poset diagrammatically in Fig 4.2. The diagram has been annotated with indications of the transitions of the buffer in terms of inputs and outputs. Thus I_a denotes the input of item a, O_b denotes the output of item b and $I_b + O_a$ denotes the simultaneous input and output respectively of the two items a and b. We denote the end of transmission (*eot*) by the binding of the tail of the list to `nil`.

In order to analyse this goal we first normalize the SILCS buffer program, make the maximum store size 2 and represent the store by a term, either a constant `empty`, a pair `full(A,B)` or a singleton `part-full(A)`.

buffer2(Ins,Outs,Store) ←

Store=empty ∧ Ins=nil ∧ Outs=nil

buffer2(Ins, Outs, Store) ←

Store=empty ∧ Ins=In.Ins′ ∧ buffer2(Ins′, Outs, part-full(In))

buffer2(Ins, Outs, Store) ←

Store=full(Out,X) ∧ Outs=Out.Outs′ ∧ buffer2(Ins, Outs′, part-full(X))

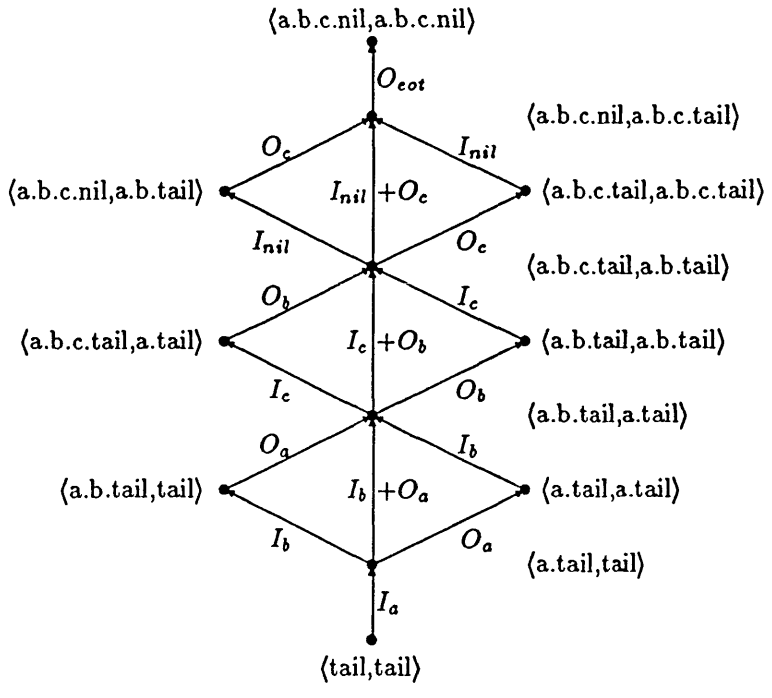

Figure 4.2: Graph of bindings for a two-place buffer

buffer2(Ins,Outs,Store) ←

Store=part-full(X) ∧ Ins=In.Ins′ ∧ buffer2(Ins, Outs, full(X,Ins))

buffer2(Ins, Outs, Store) ←

Store=part-full(Out) ∧ Outs=Out.Outs′ ∧ buffer2(Ins, Outs′, empty)

buffer2(Ins, Outs, Store) ←

Store=part-full(Out) ∧ Ins=In.Ins′ ∧ Outs=Out.Outs′ ∧

buffer2(Ins, Outs, part-full(In))

One unfolding of the goal

← buffer2(Ins,Outs,empty) ∧ producers(Ins,a.b.c.nil)

where producers/2 is the non-sequenced producer is:

← Ins=X.Ins′ ∧ Ins′=Y.Ins″ ∧ Outs=X.Outs′ ∧
Ins″=Z.Ins‴ ∧ Outs′=Y.Outs″ ∧ Outs″=Z.Outs‴ ∧
Ins‴=nil ∧ O‴=nil ∧
Ins=a.Ins′ ∧ Ins′=b.Ins″ ∧ Ins″=c.Ins‴ ∧ Ins‴=nil

All the unifications form one atomic group and are executed concurrently. This is the case whatever clause selection method is employed by the SILCS interpreter. The poset of all the possible observations of the buffer comprises two members, ⊥ and ⊤, i.e. {⟨tail,tail⟩, ⟨a.b.c.nil,a.b.c.nil⟩}. There is only one path through the graph, and hence only one possible computation.

If the producer is sequenced then the goal unfolds to:

← Ins=X.Ins′ ∧ Ins′=Y.Ins″ ∧ Outs=X.Outs′ ∧
Ins″=Z.Ins‴ ∧ Outs′=Y.Outs″ ∧ Outs″=Z.Outs‴ ∧
Ins‴=nil ∧ O‴=nil ∧
(Ins=a.Ins′ & Ins′=b.Ins″ & Ins″=c.Ins‴ & Ins‴=nil)

The result is identical to that for the one-place buffer discussed above. All the unifications on the variables Ins and Outs are performed concurrently with the first element in the sequence group of computations associated with the producer. The messages `a`, `b` and `c` are thus not buffered, but transmitted in the sequence `a ; b ; c`. The message slots, represented by the variables X,Y,Z are created eagerly by binding Ins and Outs to the list a.Y.Z.Z1 at the start of the computation. As before, we represent the unbound tail of the incomplete list to which a stream variable is bound by `tail`. The poset of states of the buffer variable pair ⟨Ins,Outs⟩ for the computation described above is thus:

{ ⟨tail,tail⟩, ⟨a.$Y.$Z.tail,a.$Y.$Z.tail⟩, ⟨a.b.$Z.tail,a.b.$Z.tail⟩, ⟨a.b.c.tail,a.b.c.tail⟩, ⟨a.b.c.nil,a.b.c.nil⟩ }

The SILCS program whose execution will result in the predicted behaviour *must* sequence the recursive calls to the buffer:

```
buffer2(Ins,Outs,Store) ←
    (Store=empty ∧ Ins=nil) & Outs=nil
```

buffer2(Ins, Outs, Store) ←

(Store=empty ∧ Ins=In.Ins′) & buffer2(Ins′, Outs, part-full(In))

buffer2(Ins, Outs, Store) ←

(Store=full(Out,X) ∧ Outs=Out.Outs′) & buffer2(Ins, Outs′, part-full(X))

buffer2(Ins,Outs,Store) ←

(Store=part-full(X) ∧ Ins=In.Ins′) & buffer2(Ins, Outs, full(X,Ins))

buffer2(Ins, Outs, Store) ←

(Store=part-full(Out) ∧ Outs=Out.Outs′) & buffer2(Ins, Outs′, empty)

buffer2(Ins, Outs, Store) ←

(Store=part-full(Out) ∧ Ins=In.Ins′ ∧ Outs=Out.Outs′) &
buffer2(Ins, Outs, part-full(In))

There are then several possible unfoldings of the goal each one corresponding to a different path through the graph of the observations of the buffer. For example, one unfolding is:

← (Ins=X.Ins′ & Ins′=Y.Ins″ & (Outs=X.Outs′ ∧ Ins″=Z.Ins‴) &
(Outs′=Y.Outs″ ∧ Ins‴=nil) & Outs″=Z.Outs‴ & O‴=nil)
∧ (Ins=a.Ins′ & Ins′=b.Ins″ & Ins″=c.Ins‴ & Ins‴=nil)

These goals form one atomic group with two members, each of which is a sequence group. The set of observations for one possible computation is:

{⟨tail,tail⟩, ⟨a.tail,tail⟩, ⟨a.b.tail,tail⟩, ⟨a.b.c.tail,a.tail⟩, ⟨a.b.c.nil,a.b.tail⟩,
⟨a.b.c.nil,a.b.c.tail⟩, ⟨a.b.c.nil,a.b.c.nil⟩ }

The set of the observations of all the possible computations form the poset is that required by the definition of the buffer.

The general form of a SILCS program for data store N-place buffer is described by the relation bufferN(Ins,Outs,Store,N,Size) where Ins and Outs are the input and output variable respectively, Store is a difference-list holding the stored items, N is the maximum number of items that can be stored and Size is the number of items in the store. The SILCS program is:

bufferN(Ins,Outs,Store,Size,N) ←

(Size=0 ∧ Ins=nil) & Outs=nil

bufferN(Ins, Outs, Store, Size,N) ←

(Size ≥ 0 ∧ Size < N ∧ Ins=In.Ins′ ∧ Store=X/Y) &

bufferN(Ins′, Outs, In.X/Y, s(Size), N)

bufferN(Ins, Outs, Store, Size, N) ←

(Size ≤ N ∧ Size > 0 ∧ Outs=Out.Outs′ ∧ Store=X/Out.Y) &

Size=s(Size′) ∧ bufferN(Ins, Outs′, X/Y,Size′,N)

bufferN(Ins, Outs, Store, Size, N) ←

(Size < N ∧ Size > 0 ∧ Outs=Out.Outs′ ∧ Ins=In.Ins′ ∧ Store=X/Out.Y) &

bufferN(Ins, Outs′, In.X/Y,Size, N)

4.14.2 Process network buffers

In Section 3.10.2.6 we introduced the idea that the composition of N 1-place buffers as a linear network would when executed on a suitable interpreter behave as an N-place buffer. In this section we show that the operational semantics of SILCS defines such an interpreter.

The general form of such an N-place buffer is

buffer(N,Ins,Outs) ←

buffer1(Ins,Mid_1) ∧ buffer1(Mid_1,Mid_2) ∧ ... ∧ buffer1(Mid_N,Outs)

We take as our example a two-place buffer comprising two one-place buffers (see page 107).

buffer2(Ins,Outs) ← buffer1(Ins,Mid) ∧ buffer1(Mid,Outs)

and use the definition of buffer1/2 from Section 4.13.2, page 139

buffer1(Ins,Outs) ←

    Ins = nil & Outs=nil

buffer1(Ins, Outs) ←

    Ins = In.Ins′ & Outs=In.Outs′ & buffer1(Ins′,Outs′)

The goal

← buffer2(Ins,Outs) ∧ producers(Ins,a.b.c.nil)

can be rewritten as:

← buffer1(Ins,Mids) ∧ buffer1(Mids,Outs) ∧ producers(Ins,a.b.c.nil)

The computation of each one-place buffer considered alone can be represented by a graph which comprises just one chain (Fig 4.1). We represent the synchronised composition of the two buffers in Fig 4.3 below by abuse of the diagrammatic representation of a graph. The synchronisation of the two processes is indicated by "A•– – $\overset{x}{-}$ – →•B" where the processes A and B are forced to synchronise and the value x is passed from A to B during the synchronisation. In the example process A is buffer1(Ins,Mids) and process B is buffer1(Mids,Outs). The annotation I_a denotes the input of item a on buffer1(Ins,Mids) and O_a denotes the output of item a from buffer1(Mids,Outs).

The graph for the two-place buffer process network buffer is depicted in Fig 4.4 below. Internal transitions have been added in the form of M_x which denotes the acceptance by buffer1(Mids,Outs) of item x from buffer(Ins,Mids) on the channel variable Mids.

We can derive the graph depicted in Fig 4.4 from Fig 4.3 by combining the paths for the computations of each one-place buffer. The input of b on Ins, denoted by I_b, and the output of a on Outs, denoted by O_a, occur in *parallel*. The semantics of SILCS describes the independent occurrence of a and b as being equivalent to a *then* b or b *then* a or a *simultaneous* b. We replace the parallel lines for input on Ins and output on Outs

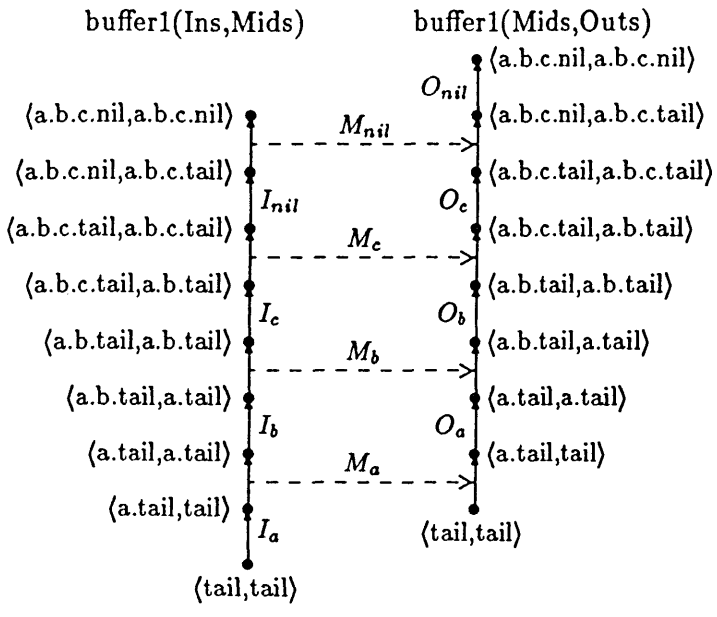

Figure 4.3: Synchronising two one-place buffers

(Fig 4.5) by the structure in Fig 4.6 indicating these alternative computations.

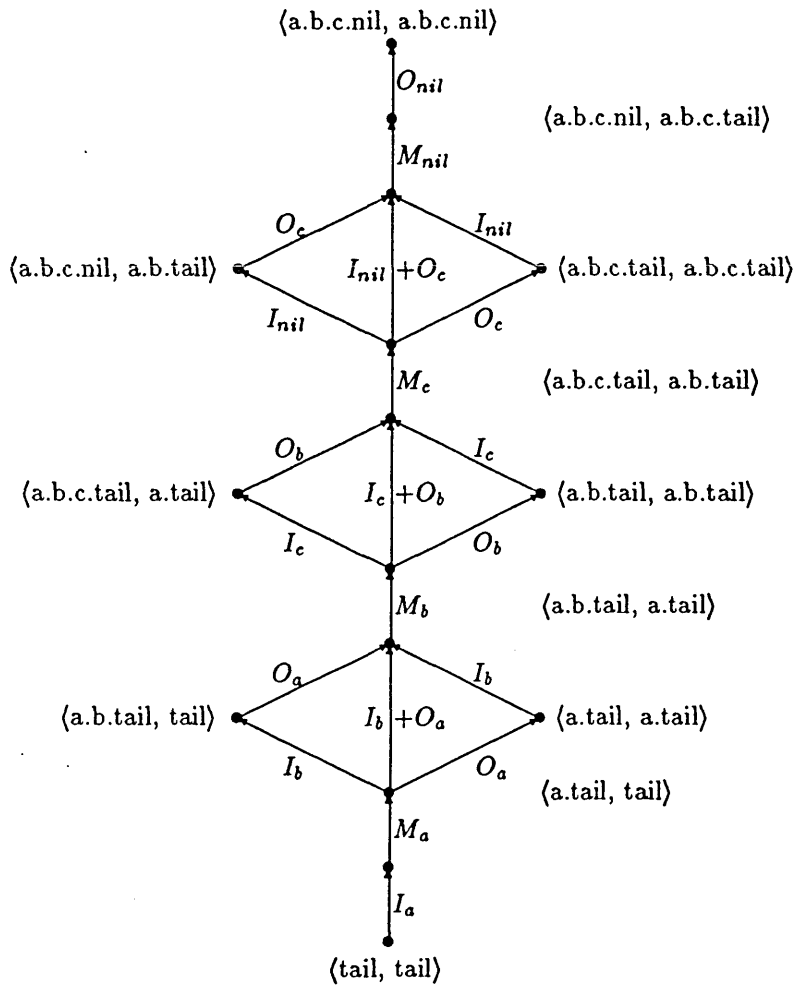

Figure 4.4: Graph for a two-place process buffer (with internal transitions)

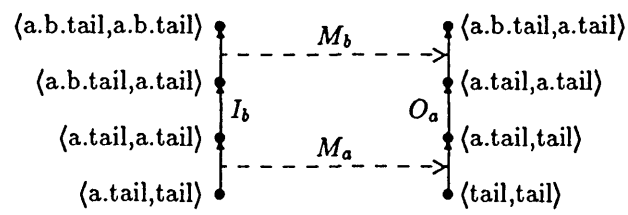

Figure 4.5: Portion of individual graphs of two one-place buffers

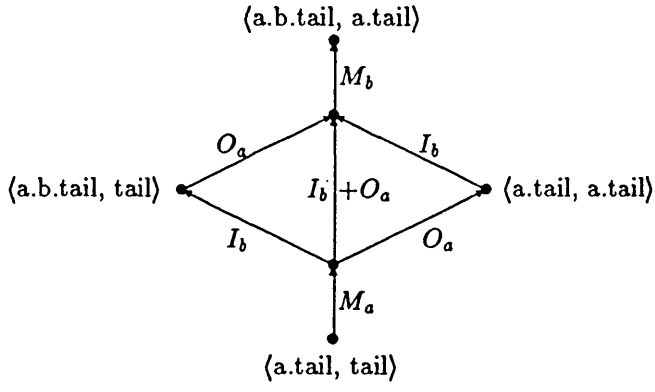

Figure 4.6: Portion of graph of a two-place process based buffer

4.15 Queues

The descriptions of queues developed in Section 3.10.3 can be transformed into SILCS programs by the addition of the sequencing operator. For example, the data structure queue from Section 3.10.3.3 on page 110 forms the basis for the following SILCS program:

```
queue(Ins,Outs,Store,Size) ←
    (Size=0 ∧ Ins=nil) & Outs=nil

queue(Ins, Outs, Store,Size) ←
    (Size ≥ 0 ∧ Ins=In.Ins' ∧ Store=X/Y) &
    queue(Ins', Outs, In.X/Y, s(Size) )

queue(Ins, Outs, Store,Size) ←
    (Size > 0 ∧ Outs=Out.Outs' ∧ Store=X/Out.Y) &
    (Size=s(Size') ∧ queue(Ins, Outs', X/Y,Size' ))

queue(Ins, Outs, Store, Size) ←
    (Size > 0 ∧ Outs=Out.Outs' ∧ Ins=In.Ins' ∧ Store=X/Out.Y) &
    queue(Ins, Outs', In.X/Y,Size )
```

The process-network queue described on page 111 forms the basis of the following SILCS program:

```
queue(Ins,Outs) ←
    Ins=I.Ins' & (queue(Ins',Mids) ∧ buffer1'(Mids,Outs,I))

buffer1'(Ins, Outs, In) ←
Outs=In.Outs' & buffer1(Ins,Outs')

buffer1(Ins, Outs) ←
    Ins=In.Ins' & buffer1'(Ins', Outs, In)
```

In turn, this can be used as the basis of the following SILCS description of an expedited data queue. We assume that an expedited data item I is a tuple of the form $x(I)$ and that a normal data item I is a tuple of the form $n(I)$.

```
xdq(Ins,Outs) ←
    Ins=Item.Ins' & (xdq(Ins,Mids) ∧ cell(Mids, Item, Outs) )

cell(Ins, Item, Outs) ←
    Item=x(I) & Outs = (x(I),n(I')).Outs' & cell(Ins,n(I'),Outs')
cell(Ins, Item, Outs) ←
    Item=x(I) & Outs = x(I).Outs' & cell(Ins,Outs')
cell(Ins, Item, Outs) ←
    Item=n(I) & Ins = n(I').Ins' & Outs=n(I).Outs' & cell(Ins,n(I'),Outs')
cell(Ins, Item, Outs) ←
    Item=n(I) & Ins = (x(I'),n(I)).Ins' & cell(Ins,x(I'),Outs')

cell(Ins,Outs) ←
    Ins=Item.Ins' & cell(Ins', Item, Outs)
```

A more efficient form of the queue is one which can grow and shrink in size:

```
queue(Ins,Outs) ←
    Ins=In.Ins' & (queue(Ins,Mids) ∧ queue-cell(Mids,In,Outs))

queue-cell(Ins,In,Outs) ←
    Ins=In'.Ins' & (queue-cell(Ins',In',Mids) ∧ queue-cell(Mids,In,Outs))
queue-cell(Ins,In,Outs) ←
    Outs=In.Outs' & Ins=Outs'
```

We can then describe an expedited data queue which also can grow and shrink in size:

```
xdq(Ins,Outs) ←
    Ins=Item.Ins' & (xdq(Ins,Mids) ∧ cell(Mids, Item, Outs) )
```

cell(Ins, Item, Outs) ←

Item=x(I) & Ins=Item.Ins′ & (cell(Ins′,Item,Mids) ∧ cell(Mids,x(I),Outs))

cell(Ins, Item, Outs) ←

Item=n(I) & Ins=n(I′).Ins′ & (cell(Ins′,n(I′),Mids) ∧ cell(Mids,n(I),Outs))

cell(Ins, Item, Outs) ←

Item=n(I) & Ins=x(I′).Ins′ & (cell(Ins′,n(I),Mids) ∧ cell(Mids,x(I′),Outs))

cell(Ins, Item, Outs) ←

Outs=Item.Outs′ & Ins=Outs′

4.16 Summary

Descriptions in first order of the set of observations of a concurrent system form the basis of a Horn clause program in language $L_{\mathcal{P}}$ whose execution results in concurrent behaviour conforming to a description in language $L_{\mathcal{S}}$. This chapter has described the syntax and semantics of the logic programming language SILCS which is an instance of $L_{\mathcal{P}}$. The operational semantics of SILCS are defined by an abstract logic interpreter for the language. Programs in SILCS have been derived from the examples developed in Chapter 3 and their computations depicted as graphs. Equivalences have been developed for SILCS programs which permit the comparison of data-structure and process-structure systems. However, SILCS has not been designed for the *implementation* of concurrent systems — its semantics dictate expensive computational mechanisms, for example atomic unification and synchronous communication. The 'all-solutions' model of non-determinism employed by SILCS also makes it unsuitable for the implementation of systems exhibit *committed-choice* behaviour.

This chapter has raised the following issues which are investigated in the remainder of the thesis:

(1) How programs in concurrent languages designed for the *implementation* of concurrent systems can be derived from SILCS programs.

(2) The definition of the conformance relation between specifications and programs.

(3) The construction of an interpreter for SILCS and its use in the investigation of conformance between SILCS and the set descriptions of systems

Chapter 5

Implementing SILCS programs

5.1 Introduction

The logic programming language SILCS which was described in Chapter 4 is not a candidate for the implementation of concurrent systems. Its computational model based on synchronous communication and atomic unification hinders the efficient construction of such systems using the language, and the all-solutions form of non-determinism employed by SILCS make the language unsuitable for constructing systems which admit only one computational path. Logic programming languages which are more suited for the task of implementing concurrent systems have been designed by other researchers, and we propose that SILCS programs are used as the basis for the development of programs in such languages. The relationship between SILCS and other concurrent logic programming languages, and mappings from SILCS programs into programs in these languages are described in this chapter.

5.2 Specifications, implementations and simulations

We distinguish between a *specification* and an *implementation* of a system in that a specification is a description of the set of observations that can be made of the system, whereas an implementation of a specification is machine code which when executed results in be-

haviour predicted by the specification. The term 'machine code' denotes instructions for a machine which may be realised in hardware or software. In the latter case, we refer to such a software realisation as an emulator for an abstract machine, i.e. an implementation of a specification for the abstract machine. It should be noted that the distinction between 'hardware' and 'software' is blurred by the practice of micro-coding instructions in processors, and that abstract machines may be implemented by special-purpose hardware rather than by software emulation.

Similarly a *specification language* is a notation for describing the set of observations that it is possible to make of the system specified, whereas a *programming language* is a notation for the construction of computer programs. However the distinction between specification languages and implementation languages is blurred by the possibility of building *interpreters* for languages, as opposed to *compilers*.

Definition 5.1 An *interpreter* is a computer program written in a meta language which translates symbols in an object language into machine code instructions, and executes them during the live operation of the interpreter. The machine code instructions produced by the interpreter are not usually stored in non-volatile memory. ■

Definition 5.2 A *compiler* is a computer program which translates symbols written in a source language into machine code instructions[1] which can be later directly executed without reference to the source. ■

Programming languages may be either interpreted or compiled. Compiled programs usually execute more rapidly than interpreted programs, although the compiled object code may be larger in size than the source code.

Specification languages are not usually compiled due to the complex nature of the systems which they describe, but they may be interpreted. A *constructive* specification language is one which permits specifications to be constructed which can be interpreted mechanically. The instructions generated during the operation of the interpreter can be regarded as an implementation of the specification, but they are not usually stored in long-term memory.

[1]The machine code instructions produced by a compiler from the source code are termed *object code*, not to be confused with the 'object language' of Definition 5.1.

Non-constructive specification languages do not give an explicit model of the system being specified but instead describe invariant properties of the system. A specification in such a language usually has no mechanically-derivable implementation.

A *simulation* of a system is a representation of the desired behaviour of the system. For example, parallel behaviour may be simulated by interleaving. A simulation is a *partial implementation* of a system if under certain circumstances the simulation truly implements the system. As an illustration, if a simulation is run on m processors and the system simulated comprises n processes, n varying dynamically, then if there is always at least one processor available to run each process the simulation is a complete implementation of the system. However if during the execution of the simulation some processes have to share processors then the simulation is a *partial implementation* of the system.

5.3 Programming languages for implementing concurrent systems

The design of programming languages used for system construction is dictated primarily by the characteristics exhibited by the target system, and also to some extent by the hardware on which the system is to be implemented. In the case of *concurrent* systems, the behaviour of the desired system, including communication and synchronisation, is a paramount influence on the programming language and the ideal target hardware is unlikely to be based on the traditional von Neumann model. While it is true that concurrent systems have been implemented on that traditional sequential model, for example the construction of multi-user operating systems running on mono-processors by the use of time-sharing, the trend in hardware design is now towards both multi-processors and distributed systems.

Early programming languages were oriented towards the control of the von Neumann machine and were *imperative* — programs in such languages possess an operational semantics which can only be understood with reference to their effect on the state of the machine. Languages developed for programming these sequential machines are not suited to the control of parallel systems unless these systems are just a conservative extension of the

von Neumann model. Thus in a shared-memory parallel computer the basic von Neumann instruction set can be preserved, extending the semantics of the instructions, for example concurrent read and writes, or adding new instructions. However, communication and synchronisation between processes which reside on different processors in a loosely coupled system are not readily expressed in this extended instruction set.

A *declarative* programming language is based on an abstract formalism, and statements in such a language do not contain features which make sense in machine-level terms, such as side-effects. The lambda calculus [21] forms the basis for functional programming languages, the first of which was LISP [89]. Logic programming is based on first-order predicate logic, in practice on the Horn clause subset, the procedural interpretation of which was given by Kowalski [72]. Prolog, the first logic programming language, was designed and implemented by Colmerauer and his team at Marseilles in 1972 [107]. More recent logic programming languages include those based on constraint evaluation, for example Prolog III [33], CLP($\mathcal{R}$) [68], cc [114] and those based on parallel evaluation strategies [120].

A central tenet of this thesis is that declarative languages are suitable as programming languages for the construction of concurrent systems. These languages are independent of machine architecture and are based on formalisms which make them amenable to rigorous analysis. A programming language based on first order logic is potentially well suited to this role since its computational rule can include concurrent evaluation. The technology of concurrent logic programming has matured to the point that languages in this family have been used to implement practical systems, for example the work reported by Foster [40, 41] using Parlog, and the use of Concurrent Prolog to construct the Logix system [57, 123]. Also there are robust implementations of several of these languages on a variety of multiprocessor configurations systems, for example Parlog [34], Strand [42], KL1 [101] and Flat Concurrent Prolog [127].

5.4 Why SILCS is not an implementation language for concurrent systems

SILCS is a 'half-way house' between a logic specification language and a concurrent logic programming language. Clauses written in SILCS may be understood as procedures in the manner described by Kowalski [72], with a control strategy defined by the operational semantics of SILCS. Thus SILCS specifications are *logic algorithms* (see Kowalski [73] or Hogger [62]), i.e. triples $\langle G, L, C \rangle$, where G is the goal $\leftarrow \texttt{init}(V_1, \ldots, V_n)$, L the set of clauses or *process specifications*, and C the *control* embodied in the rules of the SILCS interpreter. However, the concurrent systems that are the targets of SILCS descriptions are characterized by *committed-choice non-determinism*, i.e. if at some point in the computation there is a non-deterministic choice between several goals the system will arbitrarily commit to one of the alternatives. For example, consider an implementation of the producer–bounded-buffer–consumer system. If the buffer is neither full nor empty, the producer is prepared to offer an item and the consumer is prepared to accept an item then the buffer will perform only one of the following possible actions:

(1) accept an input item

(2) offer an item for output

(3) accept an input item, and offer a different item for output

The semantics of choice in SILCS is that of all-solutions non-determinism, and all alternatives in a choice are explored in parallel by an idealised SILCS interpreter. Committed choice concurrent logic programming languages are well suited for implementing the systems which are described by SILCS programs, but are ill suited for the task of *specifying* concurrent systems, a topic discussed by Gilbert in [49]. In the following sections we take a closer look at the characteristics of these languages and describe ways in which committed choice concurrent logic programs can be derived from SILCS programs.

5.4.1 Non-determinism and a lack of guards

The most general kind of non-determinism is 'don't know' non-determinism: 'don't care' non-determinism is just a restricted case of this. Schemes are possible where the number of choices made by the system are less than the maximum presented to it, but more than one. In general, given a choice of N possibilities, the system may choose to explore M paths where $1 \leq M \leq N$.

Committed choice (don't care) non-determinism is suitable for the *implementation* of concurrent systems, but not for the *specification* of such systems since in general we wish to reason about all the possible behaviours that a system may exhibit. Given a specification formalism incorporating don't-know non-determinism, the specifier can ensure that the system makes only one choice by the use of appropriate predicates in the body of the relevant clause in the specification. From the implementor's point of view, a refinement of a specification that eliminates redundant choices will be part of the implementation process. For the purpose of verification, don't-know non-determinism is required in the verifier to provide generate and test facilities. This does *not* preclude a parallel or concurrent language for the construction of a verifier.

Don't know non-determinism is required if automated systems are to be constructed which attempt to compare implementations by deriving common specifications from them. This non-determinism might however be required in transformational and analytical tools rather than in the 'language' itself — but it is always a good test of a technique if tools for manipulating formulae written in it can be constructed using the same language, an approach taken by van Eijk [132].

The use of guards enables the choice of clause commitment, or *selection* if we regard guards as being part of the clause selection process, to be made more deterministic. The question as to whether there should be more than one part to a guard is open and recent proposals by Saraswat [113] have been for distinct input and output ('Ask' and 'Tell') components of a guard.

The characteristics required of a choice operator for algebraic specification languages have been discussed by Milner [95] and Hoare [58]. Choice in these formalisms is *committed*,

and does not directly correspond with the semantics of the logical choice operator. From the point of view of expressibility, a choice operator which can respond to its environment is desirable. This implies that for a language with guards, output can be made in the guard, and that some testing evaluation be performed in the system when choice occurs. This sort of choice mechanism is used with effect in LOTOS specification of buffers and expedited data queues described by Brinksma [11]. CCLPs do not have this mechanism, but there exist techniques to program around this, for example mode reversal and the use of external monitor processes. We have shown in [49] that the former technique does not permit composition of system components; in [48] we have demonstrated that external monitor processes are clumsy and awkward to program, difficult to reason about and result in inefficient execution.

5.5 Committed choice concurrent logic programming languages

Concurrent logic programming languages are characterized by an evaluation strategy including and-parallelism, committed-choice non-determinism and some form of synchronisation. The reader is referred to [120, 114, 126] for detailed introductions to these languages. The evaluation of constraints has been explored as an alternative to term unification in parallel logic languages and is described in [114, 25, 86]. We present here a short summary of concurrent languages which employ unification — implementations of concurrent constraint logic languages have yet to be made available to system builders.

5.5.1 The Relational Language

Stream-processing in a logic programming context was initially discussed by van Emden and de Lucena [135], while an early version of Prolog which employed co-routining to achieve stream-parallelism was IC-Prolog [30]. However, the first logic programming language having an operational model which permitted concurrency was the Relational Language [26], designed by Clark and Gregory. A full interpreter for this language was never implemented, but its design was very influential on Concurrent Prolog, Guarded

Horn Clauses and Parlog (see below).

The Relational Language (RL) introduced committed choice non-determinism into logic programming by the use of guards, a construct borrowed from Dijkstra's Guarded Commands [36]. The model of synchronisation employed by the RL for a producer of data is that of unbounded buffer communication, similar to that proposed by Kahn and MacQueen [69]. Shared logic variables indicated the communication channels and a producer is never affected by its inability to send data down a channel. A consumer of data is lazy in the model, since its evaluation path is affected by the availability of data on an input channel. The RL model permits the size of the buffer between producer and consumer(s) to be specified. In the special case that the size was is, completely synchronous communication can be achieved as in CCS [92, 95] and CSP [58].

Explicit sequential and parallel AND and OR operators are part of the RL. The mode annotations "?" and "↑" are employed to indicate the input and output modes respectively of arguments to relations. Modes are *strict* in that all the arguments to a data structure in the head of a clause take the same mode as the argument of the relation in which that data structure is located. Suspension is effected by the satisfaction of the read-only constraint that no variables in an input position in a call can be bound by unification with the head of a candidate clause during a reduction.

The RL was very influential on the development of many concurrent logic programming languages which are currently in use. We present below a summary of the differences between these offspring and their common ancestor; a more detailed comparison may be found in [120]. These languages can be defined to have a common base syntax; programs are a finite set of guarded clauses of the form:

$$H \leftarrow G_1, \ldots, G_n \mid B_1, \ldots, B_m. \qquad n,m \geq 0$$

"|" is the *commit* operator[2] and $G_1 \ldots G_n$, $B_1 \ldots B_m$ are atoms. The guard *comprises* $G_1, \ldots, G_n$ and the body comprises $B_1, \ldots, B_m$. The "," operator is the parallel-AND connective. H is the head of the clause. A goal statement is a conjunction of goals of the form:

[2]The commit operator in Parlog is ":".

$\leftarrow P_1, \ldots, P_n.$ $\quad n > 0$

A note on guards

One issue in the design of these languages which has been motivated by reasons of efficiency is that of *flat guards*. A flat guard contains only calls to system predicates and a language which permits only flat guards is called a *flat language*. Current implementations of concurrent logic programming languages often impose such a restriction for efficiency reasons, thus prohibiting calls to user defined relations which may be recursively. We do not discuss flat languages further in this thesis, since there are algorithms for translating between flat and deep guarded programs [97].

However there has been a debate in the concurrent logic programming community regarding atomic input and output. The first flat language to combine input matching and atomic test unification was Saraswat's CP[↓,|,&] [109, 112]. This idea was first generalised by Saraswat in [113] where Ask and Tell parts of clauses were proposed which gave rise to the language cc(↓,|) described in his thesis [114] which contains a detailed discussion of this question in the context of a concurrent constraint programming paradigm. Similar ideas for programming languages have been proposed by Ringwood [104] and Shapiro [120] (see below).

5.5.2 Parlog

The design of Parlog by Clark and Gregory, first reported in [27] and developed by Gregory in [51] was a direct outcome of their work on the RL. Major differences between the two languages are that Parlog does not retain the annotations about the capacity of channel variables capacity, and modes are allowed to be 'weak'. The effect of the former is that shared variables act as unbounded buffers so that communication is synchronous only regarding the consumer and thus producers are completely asynchronous ('eager'). Weak modes free relations from the restriction of being either input or output (but not both) on any particular argument, effectively negating the usefulness of mode annotations as indicators of desired behaviour. Unification in Parlog is not atomic.

Parlog retains the RL's explicit sequential and parallel operators, and introduces two unification relations in addition to standard unification (=/2), none of which employ the occurs-check. Test unification, ==/2, has mode (?,?) and tests the syntactic identity of two terms up to variable names; it suspends if it could proceed only by binding a variable in one its arguments. One-way unification, $\Leftarrow$/2, likewise has mode (?,?) but the left argument is *weak*; the call $t_1 \Leftarrow t_2$ unifies t_1 and t_2 by binding variables in t_1 to make t_1 and t_2 syntactically identical. If it could proceed only by binding variables in t_2 the call suspends.

No output is possible in a guard, and a guard which can (directly or indirectly) unify variables in input positions is *unsafe*. A Parlog compiler should enforce the guard safety check.

There is a *kernel*, or standard form for Parlog, and programs written in the moded form can be translated into the kernel form. The algorithm employed replaces head arguments by distinct variables and introduces a call to $\Leftarrow$ in the guard for each variable in an input position — identical variables in input positions are replaced by distinct variables in the guard and are compared using test unification. Output variables are unified with data structures in the body of the clause by calls to full unification[3]. For example, the Parlog clause

mode merge(?,?,↑).
merge([X|Xs],Ys,[X|Zs]) ← merge(Xs,Ys,Zs).

would be represented in Kernel Parlog as:

merge(Xs,Ys,Zs) ← [X|Xs'] $\Leftarrow$ Xs | Zs = [X|Zs'] , merge(Xs', Ys, Z').

Parlog also has metalevel processing facilities: `call/1` with mode (?), which acts like Prolog's primitive of the same name, and `call/3`. This latter call has the mode pattern *call(Goal?, Status↑, Control?)*, where *Control* is a stream of commands of the form {`suspend, continue, fail`}, and *Status* a stream of messages of the form {`suspended, continued, succeeded, failed`}. This primitive, introduced in [51], permits the con-

[3]In Gregory's thesis [51] output was achieved by *assignment*. This mechanism was subsequently changed to unification in [28] and [52].

struction of fail-safe processes. Work by Foster [40] has explored in detail the use of augmented metacalls to implement an operating system in Parlog.

Negation is implemented as *negation-as-failure* [22], and is subject to the same restrictions as negation in Prolog [99]. An interface is provided to an all-solutions (don't-know non-deterministic) language via the `set/3` and `subset/3` primitives. The former has the mode declaration *set(Solutions↑,Term?,Conjunction?)*, and corresponds to the Prolog primitive `bagof/3`. The mode declaration *subset(Solutions?,Term?,Conjunction?)* ensures that `subset/3` is lazy and only returns solutions for those variable-slots in the list *Solutions* which have been provided by a producer. The semantics of the all-solutions language invoked by calls to `set/3` and `subset/3` is ill-defined.

5.5.3 Parlog86 and Guarded Definite Clauses

A syntactic variant of Parlog has been proposed by Ringwood [103] and is called Parlog86. Guard declarations are omitted and all modes default to input as in GHC (see below). Output is achieved by calls to unification in the body of a clause. For example, the Parlog86 version of the Parlog clause about merge/3 is:

merge([X|Xs],Ys,Zs) ← Zs = [X|Zs'] , merge(Xs', Ys, Z').

In [104] the same author has proposed an extension of Parlog86 called Guarded Definite Clauses (GDC). The unifiability test in the guard and the unifiability output in the body are an atomic step. For a clause to be a candidate for committal the guard must not only ascertain that the bindings prescribed by the unifiability primitive are possible but also obtain exclusive access to those variables which would be bound if the unification were to be performed. Hence output is atomic and guarded, providing a blocking send in addition to the blocking receive of Parlog. These proposals are similar to those made by Saraswat [113] for Ask-and-Tell concurrent logic programming languages. Since atomic guarded output is difficult to implement, a variant of GDC is proposed by Ringwood in which all guarded unification is replaced by guarded assignment.

5.5.4 Pandora

An extension of Parlog, called Pandora, has been proposed by Bahgat and Gregory [5] based on original ideas put forward by Clark and Gregory in [29]. Stream-and parallelism is combined with don't-know non-determinism in a unified logic programming language. Pandora extends Parlog with a deadlock handling mechanism and a simple non-deterministic fork primitive. The operational semantics of Pandora is a generalisation of Warren's Andorra model [53] and provides a programming paradigm of don't know non-deterministic concurrent communicating processes.

There are two basic kinds of relation in a Pandora program: *and-parallel* relations and *deadlock* relations. An and-parallel relation is defined by a normal Parlog procedure while a deadlock relation is defined by both an and-parallel procedure and a deadlock procedure. Both kinds of relations may call each other freely, except that a deadlock relation may not be called from a guard. A "don't-know" relation is a further type of relation whose definition can be compiled into a committed-choice and-parallel procedure and a deadlock procedure.

The non-deterministic fork is of the form $\{conj_1;\ldots;conj_n\}$ where each $conj_i$ is a Parlog conjunction. This fork may only appear as the sole member of a body of a clause of a deadlock relation. On execution, n computations are spawned where the ith or-branch of the fork goal is replaced by $conj_i$.

A Pandora query may contain goals for both committed choice relations and don't-know relations. The computation starts in the and-parallel phase during which committed-choice goals are reduced according to the semantics of Parlog. Don't-know goals are also reduced if they are deterministic. If a computation deadlocks and the deadlocked resolvent contains at least one goal for a deadlock relation, *one* such goal is reduced using its deadlock procedure instead of its and-parallel procedure.

5.5.5 Concurrent Prolog

Shapiro's initial proposal for Concurrent Prolog (CP) [116], was strongly influenced by the design of the Relational Language. However, there are significant differences between

CP and the RL. Concurrent Prolog employs the read-only variable rather than mode declarations as its synchronisation mechanism. The read-only annotation is indicated by "?" which can be attached to any variable, restricting it to read mode only. Any attempt to instantiate a read-only annotated unbound variable X? to a non-variable term is forced to suspend until the corresponding writable variable X is instantiated. Saraswat [110] has demonstrated that the original definition of the read-only mechanism presented problems, and the language has undergone several revisions, described in [119, 120]. CP can perform general unification, including both input matching and output assignment prior to commitment, and unification is *atomic*, so that safe algorithms can be written for multiple writers onto one stream [130], or efficient stream merges [111].

There is no sequential-AND connective in CP due to the desire to keep the number of language constructs to a minimum and the wish to encourage programmers to use dataflow synchronisation. Also the implementation of the sequential-AND construct on a parallel machine requires solving the problem of distributed termination detection. To run `A&B`, if & is the sequential-AND construct, the run-time system has to detect that `A` and all child processes that it has spawned have terminated before initiating `B`. The *otherwise* declaration in CP implements a default case similar to that which can be achieved using Parlog's sequential-OR operator.

CP does not have the three-argument meta-call construct of Parlog. Fail-safe systems in CP are constructed using extra control arguments to relations which are executed in enhanced meta-interpreters, as reported by Safra [108]. In [57] Hirsch has shown that partial evaluation can be employed to eliminate the overhead of meta-interpreters.

Shapiro [120] has proposed several variants of the original CP model. One of these is FCP(|), a simple concurrent logic programming language closely related to Flat GHC. Guards comprise guard test predicates, one of which can be a *special* atomic test unification predicate. An extension of this language is FCP(:) in which *all* unification attempted is atomic. An FCP(:) clause has the form

$$H \leftarrow \text{Ask} : \text{Tell} \mid \text{Body} .$$

where Ask and Tell are possibly empty conjunctions of atoms. Ask atoms have guard test predicates and Tell contains only equality atoms. On clause try the goal/head input

matching and guard checking are performed. If they fail or suspend then the clause try fails or suspends respectively. If they succeed then the unification specified by the Tell is performed, which can either succeed or fail, but not suspend. If it succeeds the rest of the clause try is the substitution combining the ask and the tell substitutions. FCP(:) can be extended to FCP(:,?) by the addition of read-only variables.

5.5.6 Guarded Horn Clauses

Guarded Horn Clauses, proposed by Ueda [131], possesses the simplest syntax and semantics of all the languages derived from the RL. It does not have mode declarations, read-only variables, or sequential operators (although the *otherwise* construct is part of the language). Its semantics incorporates committed-choice non-determinism, with the synchronisation rule that a computation invoked by the head or guard of a clause cannot bind variables in a call to that clause. Variable bindings are performed by the unification process "=" in the body of the clause, and as in Parlog unification is not atomic.

An implementation of GHC requires the run-time checking of guard safety, as opposed to Parlog's static (compile-time) guard safety check. A subset of the language, known as *safe* GHC, has been identified which guarantees that for any goal the evaluation of head unification and the guard part never instantiates a variable appearing in the goal to a non-variable term.

KL1 [71] is a stream AND-parallel logic programming language based on Flat GHC. It possesses a meta-call facility named shoēn, similar to Parlog's *call/3*, permitting the construction of operating systems in the language The shoēn is a meta-logical unit with a pair of streams, named the *control stream* and the *report stream*. The control stream is used to start, stop or abort the goals from outside the shoēn. Termination of all goals or events that occurred inside a shoēn, such as failure or exception, are reported on the report stream inside the shoēn. Shoēn can be nested to form a shoēn tree whose leaves are KL1 goals [101].

5.5.7 Strand

Strand [42] is a flat concurrent logic programming language combining features from Parlog, GHC and CP. The guard contains a *sequence* of predefined guard kernel predicates, and the body contains a collection of user defined procedures, assignment operations and/or body kernel predicates. A clause head defines match operations which may bind clause variables, and its guard defines test operations which may not. Mode declarations are optional, and are are identical in form to those of Parlog — if omitted, all arguments are assumed to be input. However, output in Strand is achieved by *assignment*, not by unification. Other features of Strand, such as modules and remote procedure calls are not relevant to the subject of this thesis.

5.6 Concurrent constraint logic programming languages

Influential work in the area of constraints and concurrency has been done by Maher [86] whose has described the ALPS class of logic programming languages, and Saraswat [114] whose research has defined cc, a family of concurrent constraint logic programming languages. Both these groups of languages use constraint propagation as a basis for communication. Saraswat also discusses the notion of atomic constraint operations within the context of guarded input and guarded output.

5.6.1 ALPS

Maher investigated a class of flat committed-choice languages called ALPS which are algorithmic and declarative. The members of the class vary in their domain of computation and the expressiveness allowed to appear in their guards. The commit laws of ALPS differ from Parlog, CP and GHC, but are closest to GHC — no annotations or modes are used. An ALPS programs is a finite collection of rules of the form

H ← G [] B

where H is an atom, G a conjunction of constraints (the guard) and B a conjunction of atoms and constraints (the body). The "[]" symbol is the commit operator. A goal is

a multiset of atoms and constraints. ALPS can be seen as subsuming pure Prolog since the latter is a degenerate case with no allowed constraints. Constraints in ALPS are not solved, but globally propagated. A successful derivation of an initial goal consisting of a set of atoms and a set of constraints consists of an empty set of atoms and a final set of constraints which may be simplified into a more concise form. Maher extends this scheme to permit parallel evaluation of goals and incorporates the notion of committed choice and concept of suspension. A goal consists of four sets, A_i and Z_i containing atoms and C_i and W_i containing constraints. A_i and C_i are atoms and constraints respectively which can write to global memory, and similarly Z_i and W_i can only read from global memory. The commit law of ALPS states that an atom A in the presence of a set of constraints C commits to a rule if the rule is validated, or it is the only one satisfied. Validation corresponds to an input of constraints and bindings from the global store into the guard part of the rule and satisfaction corresponds to propagating constraints and bindings to the global store from the guard part of the rule. Maher uses an elaborate scheme for publication of constraints, but Saraswat [114] claims that it does not matter whether publication is atomic or eventual.

5.6.2 cc

Saraswat's scheme [113, 114] is essentially very similar to that of ALPS in that the cc languages have an ask and tell component in their guard. He proposes a framework which encompasses the main features of the flat concurrent programming languages. Saraswat proposes that clauses are guarded by two sets of constraints, Ask and Tell constraints which may communicate with the set of constraints imposed so far in a computation. Ask constraints are blocking in that commitment cannot be made to a clause until the store has enough information to satisfy the constraint. Tell operations may be Eventual or Atomic — the first kind denotes a system which guarantees only that the constraints will be communicated to the store eventually, whilst the second kind denotes a system which guarantees that constraints will be communicated to the store atomically. A cc($\downarrow$,|) clause is of the form

$$H \leftarrow C_1 : C_2 \mid B$$

where C_1 is a set of Ask constraints and C_2 a set of Tell constraints. A goal may commit to such a clause only if it is possible to ask the constraint $((A{=}H) \wedge C_1)$ from the store and to Atomically Tell the constraint $((A{=}H) \wedge C_1 \wedge C_2)$ to the store. The variant, $cc(\downarrow, |_e)$ indicates the eventual nature of publication of the constraints by replacing the "|" operator by "$|_e$". Saraswat claims in [114] that there is a very close relationship between ALPS programs and $cc(\downarrow, |)$ [4] programs and that the former can be embedded in the latter.

5.6.3 Andorra Prolog

Another concurrent logic programming language, based on a different evaluation model to those languages descended from RL, is Andorra Prolog [53, 54]; its origins are both in traditional Prolog and P-Prolog which was developed by Yang [141]. The model does not require any special annotation to control parallelism, and does not restrict the language by requiring procedures to generate only one solution. Determinancy is used as the key to control and-parallelism, giving the model many of the characteristics of variants of traditional Prolog, for example Nu-Prolog [100] and PrologII [32].

In the pure Andorra model a program is a set of definite clauses each clause being of the form:

H :- G_t , G_b.

H is the head of the clause, and the body is a list of atoms that can be partitioned into the prefix list G_t – the *guard*, and a suffix list G_b – the *body*. G_t is a list of simple constraints, for example test equality and arithmetic comparison; G_b is a list of atoms.

Andorra Prolog incorporates features intended to provide for user-directed control which are lacking in the pure model. These include a *commit* operator "|" so that an Andorra Prolog clause can be of the form

H :- G_t , | , G_b.

where commit is part of the guard and always terminates it. Any constraints occurring

[4] Sarawsat has developed a complex notation for the cc family of languages, and actually uses $cc(\downarrow, \rightarrow)$ at this point in his thesis.

after the commit are part of the body.

A clause is *candidate-commit-enabled* of an atom A if A unifies with the head H in a binding environment B, and the guard G_t is solvable in the context of B∪ B_{AH}, where B_{AH} are the bindings resulting from unifying A and H. Commitment only takes place if the goals in G_t can be solved uniquely. A goal (A,C) where C is the list of candidate clauses of A w.r.t. some binding list B is *commit-enabled* if there is at least one commit-enabled clause in C. A goal is *reducible* if it is either determinate or commit-enabled; a single arbitrary commit-enabled clause is chosen for reduction.

One difference between Andorra Prolog and committed choice logic languages is that Andorra computations avoid potential deadlock by or-forking (*or-parallel extension*). Andorra Prolog permits the delaying of a goal until it is reducible (i.e. determinate or commit-enabled) by the use of the control declaration `:- delay p/n` for a predicate with functor p of arity n. Computations with delay declarations may deadlock.

Andorra Prolog systems allow the programmer to annotate programs with information providing the compiler with directives about clause selection in the case of non-deterministic programs. Other strategies proposed to avoid the necessity of run-time analysis include static compile-time analysis of programs to detect non-determinism.

5.7 Unification schemes

Unification over terms is the 'traditional' means of expressing the unit of computation in many logic programming schemes and is an instance of equality reasoning. The unification algorithm was proposed by Robinson in [105], and has been defined in Section 3.5.1 of this thesis. In this section we summarize the unification schemes employed by concurrent logic programming languages and compare them with that of SILCS. Communication and synchronisation are expressed by this simple yet powerful mechanism in logic programming languages.

Full term unification with the occurs check is employed in an idealised logic programming language based on pure Horn clauses, but implementations of most logic programming languages omit this check for efficiency reasons. An exception is Prolog II [32, 137] which

removes the occur check from unification in order to utilise an equality theory over infinite rational trees. Synchronisation is employed in the unification mechanism of variants of Prolog which permit parallel evaluation, often as part of an explicit delay mechanism for example MU-Prolog [99], NU-Prolog [100] and Prolog II. The delay mechanism forms part of a resolution scheme proposed by Naish in [98]. This is a primitive co-routining scheme in that the assumption of any particular computational rule is overridden if data is not currently available in a call, but may be made available later from another call.

Concurrent Prolog (CP) and its flat form FCP [116, 118] employ read-only variables and unification in an analogue of the freeze mechanism employed by PrologII, although recent developments have downgraded the role of the read-only variable [120]. An alternative to variable annotation is the use of mode declarations employed in Parlog [51] to effect suspension and control the input of values into a clause. This scheme is similar to that of MU-Prolog. GHC [131] also employs a similar scheme in which all modes are assumed to be input, and output is achieved by unification in the body of the clause. STRAND [42] has combined *assignment* with a data test used for suspension and input of values into a clause.

Supersets of term unification

Unification can be viewed as a special case of a more general process of checking whether or not a set of equations has a solution for some other equality theory [25]. Checking the satisfiability of constraint formulae forms the basis of constraint logic programming, discussed by Catherine Lassez in [78], and in more technical detail by Jean-Louis Lassez in [81, 79]. The CLP($\mathcal{R}$) [68] and the cc family of concurrent constraint languages [114] are based on the evaluation of constraints.

5.8 Suspension and concurrency

Concurrency involves communication and synchronisation. Communication in SILCS is totally synchronous and occurs through bindings made to shared variables; the atomic unit of computation in SILCS is unification which is an atomic process. In this section

we compare the suspension mechanism of SILCS with those of the concurrent logic programming languages discussed above. For the purposes of the following discussion we will denote a language by the letter L followed by symbols denoting basic concepts employed in the language definition which express suspension. The possible subscripts are listed in Table 5.1.

Symbol	Meaning
$=$	atomic unification
$\Leftarrow$	one-way unification
$\wedge$	simultaneous conjunction
&	sequential conjunction
$\vee$	suspension on choice

Table 5.1: Language annotations for suspension

In a language $L(=, \wedge)$ concurrent unifications occur atomically. In an *implementation* of this scheme, members of a concurrent group of unifications suspend until all the members of the group have successfully executed, but an observer will not be able to detect this. This scheme does not permit the explicit description of suspension in a concurrent system.

$L(\Leftarrow, \wedge)$ describes the class of concurrent logic programming languages which combine a data check and unification within a context local to one clause to force the suspension of computations. We include Parlog [51], Concurrent Prolog [116] and GHC [131] in this class even though their suspension mechanisms are ostensibly different. The mechanisms employed by Parlog and GHC are more or less the same and are based on suspending one-way unification, while Concurrent Prolog (in some of its definitions) uses annotated read-only variables. Both of these methods can be implemented by a suspending data-check. However this scheme alone is not sufficient to prevent a producer from running ahead of a consumer in a communication unless explicit back communication techniques are employed (see below).

P-Prolog [141] and Andorra Prolog [53] belong to the $L(\vee)$ class of languages. Suspension occurs when a non-deterministic choice is to be made, until bindings cause the choice to become deterministic. Both P-Prolog and Andorra use program annotations to restrict the circumstances when such suspension occurs. Pandora [5] employs both suspension

on bindings to variables in a clause and suspension on choice, and can be described by $L(\Leftarrow, \vee)$.

A language $L(=, \wedge, \&)$ involves the use of sequencing to define suspension when combined with unification and the simultaneous operator. Thus a call

$$\leftarrow (\text{X=a}) \wedge (\text{Y=b \& X=a})$$

causes both calls X=a to synchronise if the operational semantics of the language are appropriately defined. SILCS is a language of the $L(=, \wedge, \&)$ class. The definition of the sequential operator of Parlog does not involve concept of suspension in a combination with the parallel operator. For example, in the Parlog goal

$$\leftarrow (\text{X}\Leftarrow\text{a}) \text{ , } (\text{Y=b \& X=a})$$

the call (X$\Leftarrow$a) suspends until the second call to X=a is executed in the sequence (Y=b & X=a). However in the goal

$$\leftarrow (\text{X=a}) \text{ , } (\text{Y=b \& X}\Leftarrow\text{a})$$

the two calls involving the variable X are not synchronised. The use of one-way unification alone will not enable the synchronisation to take place.

5.9 Implementing SILCS programs using concurrent programming languages

We have previously introduced the concepts of interpretation, compilation and simulation (Sections 5.2 and 5.2). SILCS programs could be implemented, as opposed to simulated, by either interpretation in a meta-language or transformation into a target language. Both such a meta-language and a target language would have to be able to support concurrent execution. The most likely candidates for these languages are concurrent logic programming languages since their syntax and semantics are close to that of SILCS. An interpreted program is likely to execute less efficiently than a transformed program, since transformation in this case can be regarded as the compilation of a SILCS program into a target language.

Interpretation

Interpretation requires the construction of an interpreter in meta-language with parallel programming constructs, which will treat SILCS programs as the object language. The operational semantics of SILCS dictates the construction of such a program which will be based on the idealised Horn clause interpreter for SILCS — hence the choice of a logic programming language as the meta-language. The best candidates for the meta-language are those whose semantics include atomic unification for both input and output, otherwise interpretation will be very inefficient due to the need to directly implement such unification algorithms in the interpreter itself.

If the meta language chosen is a committed-choice language then problems arise in the representation of the don't-know non-deterministic behaviour of SILCS. Those concurrent logic programming languages which can compute all-solutions are Pandora and Andorra. However neither of these languages has inbuilt atomic output unification, leading to the inefficiencies described in the previous paragraph. Implementations of these languages were not available at the time that this research was performed.

Transformation

Transformation of SILCS into a concurrent programming language is a form of compilation which treats a SILCS program as the source code and the concurrent programming language as the object code. It is inevitable that heuristics will be required to effect such transformations due to the differences between SILCS and possible target languages. We explore the possible transformation techniques in the next section.

5.10 Transforming SILCS programs into programs in a CLPL

In this section we assume that SILCS programs will be mechanically transformed into programs in concurrent logic programming languages, aided by human guidance in some cases. The mechanical translator can be written in any programming language, but a logic programming language is preferable due to the term reader and syntactic analyser which

is part of such a language.

Mapping SILCS into a committed choice concurrent logic programming language requires

(1) Syntactic transformations.

(2) Preserving the semantics of SILCS programs regarding synchronisation and atomic unification.

(3) Converting the don't know non-determinism of SILCS into the don't-care non-determinism of the target language, which may involve deciding where to place guards in the transformed program.

The rules governing the transformation from SILCS into the target language are determined by the semantics of that target language. The range of parallel logic programming languages available for the construction of concurrent systems is still increasing. Advances in the technology of implementing runtime systems for these languages on a variety of hardware configurations make the selection of just one target language undesirable. Thus we present below a general framework for such a transformation and modify it for some of the currently available target languages.

The 'purest' concurrent logic programming language discussed above is *Guarded Horn Clauses* since it does not have mode declarations, one-way unification, read-only variables or assignment. There are however several differences between GHC and SILCS, most notably the former's lack of the sequential operator. In the following sections we outline a methodology for deriving programs in GHC from system descriptions in SILCS.

The syntax of SILCS is a little different from that of all the target languages and a pre-processor is required to make the appropriate translations for each target language. Most of the target languages have a common syntax based on that described in Section 5.5.1 and we base our transformations on that target syntax.

5.10.1 List notation

The usual logic programming list notation is "|" for the list constructor, "[]" for the empty list, and "[", "]" for list delimiters. SILCS employs ./2 as the list constructor

and represents the empty list by `nil`. A simple addition to the term reader of a logic programming language suffices to effect this translation as SILCS programs are read in to the system. Alternatively `./2` can be declared as an operator of the appropriate precedence and type and the SILCS list structures read in then explicitly translated by the transformer program. We give such a Prolog program below:

```
:- op(200,xfy,'.').

convert_list(nil,[]).
convert_list(X.Xs, [X|Ys]):-
        convert_list(Xs,Ys).
```

5.10.2 The simultaneous operator

The "$\wedge$" connective of SILCS maps onto the "," connective in the CCL languages which all interpret "," as the parallel operator. However, the suspension mechanism of SILCS involves an interaction between the "$\wedge$" and "&" operators, and a direct mapping of the former onto "," is not sufficient (see below).

5.10.3 The sequencing operator

Parlog and its derivatives (Pandora, GDC) possess a sequential-and operator, the syntax and semantics of which is identical to that of SILCS when considered in isolation to the "$\wedge$" connective. Representing sequentiality in CCL's which do not possess the "&" connective requires the use of a variant on the short circuit technique first attributed to Takeuchi [125] and later extended by Weinbaum et al [139] and Saraswat et al [115].

The following transformation has to be applied to SILCS programs containing the sequencing operator when they are translated into languages which lack such a construct. We take the target language to be GHC in this instance.

- Add two extra arguments to all the relations in sequence c_1 & ... & c_n of calls, except for the first and last to which only one argument is added. Call these arguments I_k

and O_k for each c_k, $1 < k < n$. The single arguments added to c_1 and c_n are O_1 and I_n respectively.

- Each call c_k shares I_k with c_{k-1} and shares O_k with c_{k+1}. I.e. O_k=I_{k+1} for all calls c_k, $1 \leq k < n$.

- Add code which causes the variable O_k representing a synchronisation flag to be bound in the body of call c_k and code causing the suspension of c_{k+1} on that variable (i.e. on I_{k+1}).

- Replace all occurrences of the "&" operator in the sequence by the "," operator.

- The definition of every sub-process that can be invoked by the sequence (potentially all the processes in the system specification) is transformed to take one extra argument in the form of the tuple X-Y.

  p(...) ← true | true.
  is transformed to
  p(...,X) ← X=L-R | ← L=R.

  p(...) ←
  $p_1(\ldots), p_2(\ldots), \ldots, p_n(\ldots)$
  transformed to
  p(...,X) ←
  X = L-R |
  $p_1(\ldots,L\text{-}X_1), p_2(\ldots,X_1\text{-}X_2), \ldots, p_n(\ldots,X_{n-1}\text{-}R)$

  Note that a call ← p(...,X) will suspend until the variable X is bound to a tuple of the form X-Y. For clarity we have written the transformed clause in normalised form; we could have written the first of the above clauses as:

  p(...,L-R) ← true | ← L=R.

- If the underlying program contains a call $T1 = T2$ then it must be replaced by a call $(Left, T1) = (Right, T2)$ in the transformed program.

For example, the SILCS sequence

$$\leftarrow x(X1)\ \&\ y(Y1)\ \&\ z(Z1)$$

would be translated into the GHC.

$$\leftarrow x(X1,S1)\ ,\ y(S1,Y1,S2)\ ,\ z(S2,Z1)$$

The code of `x/1`, `y/1` and `z/1` would be augmented as follows. If the code were of the form

$x(X1) \leftarrow \ldots \mid x_1(\ldots) \ldots x_k(\ldots)$
$y(Y1) \leftarrow \ldots \mid y_1(\ldots) \ldots y_m(\ldots)$
$z(Z1) \leftarrow \ldots \mid z_1(\ldots) \ldots z_n(\ldots)$

then the outline transformed code would be

$x(X1,L\text{-}R) \leftarrow \ldots \mid x_1(\ldots,L\text{-}M_1) \ldots x_k(\ldots,M_{k-1}\text{-}R)$
$y(P\text{-}P,Y1,L\text{-}R) \leftarrow \ldots \mid y_1(\ldots,P\text{-}M_1) \ldots y_k(\ldots,M_{k-1}\text{-}R)$
$z(P\text{-}P,Z1) \leftarrow \ldots \mid z_1(\ldots) \ldots z_n(\ldots)$

Note that a call p(L-R,...) to a relation with head p(X-X,...) will suspend until L and R can be unified.

An alternative method is to use call/3 of Parlog or the shoēn metacall in GHC to introduce sequentiality in the manner described by Gregory [52]. For example, a Parlog call

$$\leftarrow a\ \&\ b$$

can be rewritten as

$$\leftarrow call(a,S,C)\ ,\ call\text{-}b(S,b)$$

since call(a,S,C) will bind S to succeeded when a has terminated. We define call-b/2 by

mode call-b(?,?).
call-b(succeeded,b) ← call(b).

5.10.4 Synchronisation

The basic SILCS communication scenario which we wish to map into a concurrent logic programming language is that of *1:n* synchronous stream communication:

$\leftarrow$ producer(Msgs) $\wedge$ consumer$_1$(Msgs) $\wedge \ldots \wedge$ consumer$_n$(Msgs)

producer(Msg.Msgs) $\leftarrow$ producer(Msgs)

consumer$_k$(Msg.Msgs) $\leftarrow$ consumer$_k$(Msgs)

None of the implementations of the target concurrent logic programming languages is totally synchronous — their semantics in this respect is that of eager producers and lazy consumers. Full synchronisation can be achieved by the use of *incomplete messages* described by both Clark [28] and Shapiro [116], the basis of which is that a producer is constrained to wait (suspend) until a consumer has instantiated a variable to a non-variable term. This technique relies on constructs in the language which invoke suspension until an argument is instantiated. Such suspension may be effected by a call to unification in the guard of a GHC program, or in a Parlog program by test-unification ==/2 or one-way unification $\Leftarrow$/2. We use GHC as the target language in the following examples.

Two variants of the incomplete message technique exist: *back-communication* and *message-tuples*.

Back-communication

Each channel variable shared between a producer and a consumer is split into two, one for the message item and the other for the synchronisation reply variable. These variables are *arguments* to the procedures; each consumer in the communication has one extra argument for the reply variable, and the producer has one extra argument for each of the consumers participating in the communication:

$\leftarrow$ producer(Msg,Reply$_1$, ..., Reply$_n$) ,
consumer$_1$(Msg,Reply$_1$), ..., consumer$_n$(Msg,Reply$_n$).

producer(Msg, Reply_1, ..., Reply_n) ←

true | Msg=m , check(Reply_1, ..., Reply_n).

check(Reply_1, ..., Reply_n) ←

Reply_1 = ok , ..., Reply_n = ok | **true.**

consumer_k(Msg, Reply_k) ←

true | consume(Msg) , Reply_k = ok.

This technique requires adding an additional argument to a producer for each consumer participating in the communication, but each consumer is 'unaware' of the other consumer partners. In the case that a sequence of messages is to be sent synchronously, the *Reply* arguments must also be sequences:

producer(Msgs, Replies_1, ..., Replies_n) ←

Msgs=[m|Msgs'] , check(Msgs', Replies_1, ..., Replies_n).

check(Msgs, [R_1|Replies_1], ..., [R_n|Replies_n]) ←

R_1 = ok , ..., R_n = ok | producer(Msgs, Replies_1, ..., Replies_n).

consumer_k([Msg|Msgs], Replies_k) ←

true | consume(Msg) , Replies_k=[ok|$\text{Replies}'_k$] , consumer_k(Msgs, $\text{Replies}'_k$).

Request-reply tuples

A message sent by a producer is wrapped up in an n-tuple whose arity is one more than the number of consumers in the communication. Each of the arguments other than the message item is a distinct variable, and the producer suspends until each of these variables has been ground by a consumer. The consumer of a message grounds the appropriate variable on receipt of the message. Thus a goal and the associated procedures representing *1:n* communication are:

← producer(Tuple) ,

consumer$_1$(Tuple) , ..., consumer$_k$(Tuple) , ..., consumer$_n$(Tuple).

producer(Tuple) ←

Tuple=tuple(msg,Reply$_1$, ..., Reply$_n$), check(Reply$_1$, ..., Reply$_n$).

check(Reply$_1$, ..., Reply$_n$) ←

Reply$_1$ = ok , ..., Reply$_n$ = ok | **true.**

consumer$_k$(tuple(Msg, Reply$_1$, ..., Reply$_k$, ..., Reply$_n$)) ←

true | consume(Msg) , Reply$_k$ = ok.

This technique requires care by the programmer to avoid the construction of code in which a consumer grounds the *Reply* variable of another consumer. The number of arguments to the producer and consumers is the same as in the SILCS program, but a new data structure has to be introduced in the form of the message tuple. Stream communication is achieved by the recursive production and consumption of message tuples.

producer(Tuples) ←

true | Tuples=[tuple(msg,Reply$_1$, ..., Reply$_n$)|Tuples],

check(Reply$_1$, ..., Reply$_n$,Tuples).

check(Reply$_1$, ..., Reply$_n$,Tuples) ←

Reply$_1$ = ok , ..., Reply$_n$ = ok | producer(Tuples).

consumer$_k$([tuple(Msg, Reply$_1$, ..., Reply$_k$, ..., Reply$_n$)|Tuples]) ←

true | consume(Msg) , Reply$_k$ = ok , consumer$_k$(Tuples).

A restricted form of synchronous communication (which in Parlog relies of the property of weak modes), can be achieved for stream-based computations using *role-reversal.* The 'consumer' originates a stream comprising variables which are instantiated to messages by the producer. In the example below, all messages consist of the constant 'message':

← producer(Items) , consumer(Items).

```
producer([Item|Items]) ←
        true | Item=message , producer(Items).

consumer(Items) ←
        true | Items = [Item | Items'] , check(Item,Items').

check(Item,Items') ←
        Item = message | consumer(Items').
```

This technique can be used only for 1 : 1 communication because the consumer instantiates the stream variable to a list pattern. In a situation involving one producer and more than one consumers, one consumer could run ahead of all the other consumers and instantiate the stream variable. The other consumers would then be *checking* the list pattern of the stream instead of producing it — the technique is too simple to ensure synchronisation among producers of an identical list pattern.

Static analysis and code transformers

Whatever method is chosen to enforce complete synchronistion on the derived programs, SILCS source will have to be analysed statically to determine the communication patterns, and the code transformed accordingly. The easiest method to implement of enforcing synchronisation is that of message tuples. The algorithm requires:

(1) Transforming the SILCS code into guarded code (see below).

(2) Determining which atoms are producers and which are consumers for each communication, and totaling the number of each. A producer of a communication is identified by having a call in its body to unify the shared variable. A consumer has a call to unify a shared variable in its guard.

(3) Transforming each message on a shared variable into a synchronisation tuple whose arity is $n + 1$ where n is the number of consumers involved.

(4) Adding calls to check the status of each reply variable in the body of the producer.

(5) Adding calls to instantiate the appropriate synchronistion variable in the body of each consumer.

5.10.5 Guards

SILCS programs contain neither guards nor cuts; all the programming languages which are targets for transformations from SILCS employ these constructs. Some heuristics will have to be employed in the translation from a guarded to an unguarded form, but the basic rule is to produce a clause very similar to Andorra Prolog's guarded form:

H :- G_t | G_b.

where G_t represents a conjunction of all the constraints in the source clause.

Problems arise due to the bi-directional nature of SILCS programs as opposed to the uni-directional nature of the RL-like languages. In some cases SILCS programs may be altered to indicate an intended direction of use. Thus there is no essential difference between a *merger* and a *splitter* definition in SILCS:

merge-split(As, Bs, Cs) ←
    (As=A.As′ ∧ Cs=A.Cs′) & merge-split(As′,Bs,Cs′)
merge-split(As, Bs, Cs) ←
    (Bs=B.Bs′ ∧ Cs=B.Cs′) & merge-split(As,Bs′,Cs′)
merge-split(As, Bs, Cs) ←
    As=nil ∧ Bs=Cs
merge-split(As, Bs, Cs) ←
    Bs=nil ∧ As=Cs

GHC code can be derived from the SILCS code by placing the input unification within a guard. In order to achieve the bidirectionality inherent in the source program, each SILCS clause must be transformed into two clauses in GHC with the appropriate guard. For example, the first clause of the SILCS merge-split/3 program given above would be transformed into the following two GHC clauses:

merge-split(As, Bs, Cs) ←

As=[A|As'] | Cs=[A|Cs'] , merge-split(As',Bs,Cs').

merge-split(As, Bs, Cs) ←

Cs=[A|Cs'] | As=[A|As'] , merge-split(As',Bs,Cs').

The suspension rules for GHC coupled with the lack of the sequencing operator and asynchronous output mean that the GHC program will behave differently to the SILCS description. A more complete translation would involve sequencing the calls in the body of each GHC clause, as described in Section 5.10.3

5.10.6 Guarded output

The concurrent logic programming languages derived from the Relational Language do not in general permit guarded output. The very limited form of atomic output unification in CP is still not powerful enough to permit the direct representation of many of the more interesting concurrent scenarios explored in previous chapters. Coupled with don't care non-determinism this causes concurrent logic programming languages to be less expressive than SILCS for describing buffer based systems.

An example of a SILCS program whose translation into a committed choice logic programming language does *not* involve guarded output is one in which the output of a producer is dependent on the state of the consumer with which it communicates. The atomic unification of SILCS can be explicitly represented in an asynchronous language by the use of request-reply tuples as described in Section 5.10.4.

However the correct implementation in a committed choice language of a system in which there is a *choice* between input and output is more problematic. For example, consider the following SILCS program and associated goal for a simplified unbounded buffer with no concurrent input and output goal:

← producer(X) ∧ buffer(X,A/B,Y) ∧ consumer(Y)

producer(X) ← X=item.Xs & producer(Xs)

```
buffer(X,Store,Y) ←
      (X=nil ∧ Store=A/A ) & Y=nil

buffer(X,Store,Y) ←
      (X=Item.Xs ∧ add(Item,Store,NewStore)) & buffer(Xs,NewStore,Y)
buffer(X,Store,Y) ←
      (Y=Item.Ys ∧ remove(Item,Store,NewStore)) & buffer(X,NewStore,Ys)

receive(Y) ← Y=Item.Ys & receive(Ys)

add(Item,X/Y, Item.X/Y)

remove(Item,X/Item.Y, X/Y)
```

A translation into GHC could be:

```
← producer(X) , buffer(X,A/B,Y) , consumer(Y)

producer(X) ← true | X=[msg(item,R)|Xs] , checkprod(R,Xs).
checkprod(R,Xs) ← R=ok | producer(Xs).
buffer(X,Store,Y) ←
      X=[] , Store=A/A | Y=[].

buffer(X,Store,Y) ←
      X=[msg(Item,R)|Xs] , add(Item,Store,NewStore) |
      R=ok , buffer(Xs,NewStore,Y).
buffer(X,Store,Y) ←
      remove(Item,Store,NewStore) | [Y=msg(Item,R)|Ys] , checkbuff(X,R,NewStore,Ys).

checkbuff(X,R,Store,Y) ← R=ok | buff(X,Store,Y).
```

receive(Y) ← Y=[msg(Item,R)|Ys] | R=ok , receive(Ys).

add(Item,X/Y, NewStore) ← **true** | NewStore = [Item|X]/Y.

remove(Item, X/[Item|Y], NewStore) ← **true** | NewStore = X/Y.

There is a danger that the system will non-deterministically commit to the third clause for buffer/3 and attempt to output an item when the consumer is not 'ready'. If the consumer is deadlocked for some reason then the system will deadlock instead of permitting unbounded addition to the buffer store. One solution is to make the buffer have input modes on both stream variables, and the consumer to output the list pattern:

```
producer(X) ← true | X=[msg(item,R)|Xs] , checkprod(R,Xs).
checkprod(R,Xs) ← R=ok | producer(Xs).
buffer(X,Store,Y) ←
      X=[] , Store=A/A | Y=[].

buffer(X,Store,Y) ←
      X=[msg(Item,R)|Xs] , add(Item,Store,NewStore) |
      R=ok , buffer(Xs,NewStore,Y).
buffer(X,Store,Y) ←
      remove(Item,Store,NewStore) , [Y=msg(Item1,R)|Ys] |
      Item = Item1 , R=ok, buffer(X,NewStore,Ys).

receive(Y) ← Y=[msg(Item,R)|Ys] , checkreceive( R , Ys).

checkreceive(R,Ys) ← R=ok | receive(Ys).
```

However, this representation does not permit buffer programs to be composed, since both modes are input on the buffer stream variables. An explicit monitor process has to be placed in between each such buffer:

← buffer(X,X_1) , monitor(X_1, X_2), buffer(X_2, X_3).

```
monitor(X,Y) ←
      true | X=[msg(I,R)|Xs] , monitor'(Xs,I,R,Y).

monitor'(X,I,R,Y) ←
      R=ok | Y=[msg(I,R1) | Ys] , monitor''(X,R1,Ys).

monitor''(X,R,Y) ←
      R=ok | monitor(X,Y).
```

producer ——request-state⟶ *consumer*
producer ⟵my-state—— *consumer*
producer ——message⟶ *consumer*

Recent proposals for concurrent logic languages which have the ability to perform guarded output have been made by Saraswat [113] and others, but these have been rejected as being too expensive to implement. At present, until the efficiency problem connected with this construct has been solved, guarded output will have to be represented by complex protocols in the target language.

There are times when we do want to specify that a process generates all possibilities, and that the environment responds to that. In the design of Pandora [5] an attempt has been made to solve the problem of responding to the environment. A (backtracking) Prolog computation can be run concurrently with Parlog processes. Andorra Prolog [53] permits backtracking in some circumstances, but behaves in a committed choice manner in others. These languages are most suited as targets for the transformation of SILCS programs which involve generate and test situations.

5.11 Conclusion

The transformations required to obtain programs in committed choice logic programming languages from SILCS programs vary according to the syntax and semantics of the target language. Most of the programs that are derived are complex and inefficient due to the

code required to explicitly implement the semantics of atomic synchronous communication. In cases where asynchronous behaviour can be tolerated this extra code can be dispensed with in the interests of efficiency. The transformation process is unlikely to be completely mechanisable in the case that the target language is guarded. Some form of heuristics is required to aid the decision as to where to place the guards.

An alternative to transformation is the interpretation of SILCS programs. The most suitable candidate for the meta language is a concurrent logic programming language whose computational model is close to that of SILCS. Andorra Prolog may be suited for this task when an acceptable implementation of the language becomes available.

The issues discussed in this chapter raise the question of the need for conformance testing. Some kind of mechanisable tool is required to facilitate the comparison of source programs with target programs and the comparison of programs with specifications. We define the conformance relation and its variants in the next chapter.

Chapter 6

Conformance testing

6.1 Introduction

In previous chapters we have proposed the use of first order logic to describe the observations of concurrent systems and shown how programs in the logic language SILCS can be derived from the first order descriptions. SILCS programs can be used as a starting point for the derivation of executable code in a variety of concurrent logic programming languages. We have defined a specification to be a description of the set of observations that can be made of a system, and an implementation to be the machine code instructions whose execution results in the behaviour described by the specification. In this chapter we describe a theory of conformance which uses first order logic to describe the relationship between specifications and implementations. This theory provides guidelines for the construction of conformance testers.

Conformance testing is an important stage in the design and construction of concurrent systems and indicates if a given system correctly implements its specification. The need for conformance testing arises from a variety of reasons. Complete automation of the process of deriving implementations from specifications is not yet possible and human guidance is required for some of the stages; this may introduce errors if the systems specified are highly complex.

6.2 Snapshot logic interpreters

We assume that each target logic implementation language has an operational semantics which is defined with reference to a Horn clause logic interpreter, possibly augmented with negation as failure. The semantics of SILCS is similarly defined and the Horn clause form of the descriptions of the sets of observations can also be executed on such an interpreter. This common mechanism provides the basis for the definition of the conformance relation and the conformance tester itself.

We recall that an observation of a concurrent system is represented by an n-tuple of the bindings made to each of the observable variables of that system. A specification of a system is the partially ordered set, or directed acyclic graph, of the observations of all the computations that the system can perform. A computation is one path in the graph from the initial state to a final state. For *stream-based* systems the ordering relation on the set is based on the predecessor relation over lists.

In order to compare an implementation with a specification we require that the implementation is executed on an interpreter augmented with the facility to produce *snapshots* of the progress of the computation. These snapshots are records of bindings made to the observable variables of the system. The interpreter must conform to the operational semantics of the language in question and the snapshot mechanism must not alter this semantics.

The simplest form of a snapshot interpreter is based on the definition of the interpreter for a pure Horn clause language outlined by Kowalski [74], augmented with mechanisms to record bindings made to variables during a computation. The definition of the augmented Horn clause interpreter is:

```
demo(Prog, Goals, nil) ←
        empty(Goals)

demo(Prog, Goals, State.States) ←
        select(Goal, Goals, Rest) ∧
        member(Clause, Prog) ,
```

renamevars(Clause, Goals, Clause′) ∧
parts(Clause, Head, Body) ∧
match(Goal, Head, Sub) ∧
copyvars(Goals, GoalVars) ∧
add(Body, Rest, InterGoals) ∧
apply(InterGoals, Sub, NewGoals) ∧
apply(GoalVars, Sub, State) ∧
demo(Prog, NewGoals, States)

Goals and Prog are bound to data structure representing respectively the object level goals to be solved and the object level program used to solve the goals. The variable States is a data structure representing the set of observations that can be made of one computation and is a list of lists of variable bindings. The state of each variable binding is represented by a tuple `V/t` where `V` is an identifier standing for the name of the original variable and `t` is the data structure to which the variable is bound. The call `copyvars(Goals,GoalVars)` ensures that variables in each state of the history are unique and hence are not instantiated by unifications occurring in subsequent states.

The call to this relation is

← demo(Prog,Goals,States)

States is a structure representing the observations that can be made of one computation with respect to the triple ⟨Prog,Goals,Control⟩ and Control is the control strategy embedded in the interpreter. We extend `demo/3` by adding an extra argument representing the control strategy to be used when evaluating ⟨*Program,Goal*⟩; the top-level call to `demo/4` is

← demo(Program,Goal,Control,States)

The control strategy dictates the computational rule (call selection) and the search rule (clause selection). The former is encoded in `select` and `add` while the latter is encoded in `member`. The call to this relation is

← demo(Prog,Goals,States)

States is a structure representing the observations that can be made of one computation with respect to the triple ⟨Prog,Goals,Control⟩ and Control is the control strategy embedded in the interpreter.

In the following sections we assume that the snapshot interpreter returns a structure representing the bindings made to the observable variables of the system interpreted for successful computations only.

6.3 Conformance

We define the conformance relation between two systems X and Y each comprising the triple ⟨P,I,C⟩, where I is the initial state of the process definitions P to be evaluated under control strategy C:

Conf 6.1

conforms(⟨Xp,Xi,Xc⟩,⟨Yp,Yi,Yc⟩) ↔

$\forall$ Obs (demo(Xp,Xi,Xc,Obs) ↔ demo(Yp,Yi,Yc,Obs))

Obs is a data structure that represents one *trace* of a system. An implementation X conforms to a specification Y iff all the observations that can be made of the implementation can be made of the specification and vice-versa. For two traces A and B to match, every edge in A must be identical to a corresponding edge in B and vice-versa.

The above sentence Conf 6.1 permits conformance of two systems which both fail, since (**false** ↔ **false**) is equivalent to **true**. The relation between the tuple (Xp,Xi,Xc) and its observations (binding history) Xo is *demo(Xp,Xi,Xc,Xo)* such that each instance of Xo is one chain in the poset of all possible observations of (Xp,Xi) under Xc.

We derive a clause in *normal form* from Conf 6.1 using the following equivalences:

$$X \rightarrow Y \equiv \neg X \vee Y$$

$$\neg X \vee Y \equiv \neg(X \wedge \neg Y)$$

$$\forall Z \neg(X \wedge \neg Y) \equiv \neg \exists Z(X \wedge \neg Y)$$

and omitting the existential quantifier when weakening the $\leftrightarrow$ form to the normal clause form. Normal program clauses were defined in Definition 3.28 on page 58.

Conf 6.2

conforms(⟨Xp,Xi,Xc⟩,⟨Yp,Yi,Yc⟩) ←

$\neg$(demo(Xp,Xi,Xc,Obs_1) $\wedge$ $\neg$demo(Yp,Yi,Yc,Obs_1)) $\wedge$

$\neg$($\neg$demo(Xp,Xi,Xc,Obs_2) $\wedge$ demo(Yp,Yi,Yc,Obs_2))

The above normal clause cannot be used to *generate* observations of the negated cases if it is interpreted on a logic programming system which implements negation as negation by failure (see [22]) since negative information cannot be deduced from negative literals. Care has to be taken in the ordering of the calls in such a clause to avoid calls to negated atoms which contain variables at the time of call.

In order to arrive at definite program clauses which are Horn clauses (see Definition 3.26 on page 57) we need to define the conformance relation using a *setof* relation. We extend the `demo/4` relation to *set-demo(Procedures,Goal,Control,SetOfObs)* in which the fourth argument is a structure representing the poset of all the possible observations that can be made of ⟨Procedures,Goal,Control⟩ under the evaluation strategy of `set-demo/4`. We define `set-demo/4` by:

set-demo(Procs,Goal,Control,SetOfObs) $\leftrightarrow$

setof-solutions(Obs, demo(Procs,Goal,Control,Obs), SetOfObs)

The setof-solutions relation has been discussed previously in Chapter 3, Section 3.10.2.1. We assume here that `set-demo/4` produces a sorted set of observations with no duplicates, using Naish's description [99].

The definition of `conforms/2` using the set of solutions of the demonstrate relation given in Conf 6.3 is similar to that of Conf 6.1.

Conf 6.3

$conforms_S$(⟨Xp,Xi,Xc⟩,⟨Yp,Yi,Xc⟩) $\leftrightarrow$

$\forall$ Obs (set-demo(Xp,Xi,Xc,Obs_X) $\leftrightarrow$ set-demo(Yp,Yi,Yc,Obs_Y) $\wedge$ sameset(X,Y))

Note that since `set-demo/4` produces an ordered set of observations, a definition of `sameset/2` is

sameset(X,Y) $\leftrightarrow$ X=Y

and we simplify 6.3 as follows:

Conf 6.4

$\text{conforms}_S(\langle Xp,Xi,Xc\rangle,\langle Yp,Yi,Xc\rangle) \leftrightarrow$

$\forall$ Obs (set-demo(Xp,Xi,Xc,Obs) $\leftrightarrow$ set-demo(Yp,Yi,Yc,Obs))

The definition Conf 6.3 may be rewritten with existential quantification over Obs since `set-demo/4` only ever produces one solution:

Conf 6.5

$\text{conforms}_S(\langle Xp,Xi,Xc\rangle,\langle Yp,Yi,Xc\rangle) \leftrightarrow$

$\exists$ Obs (set-demo(Xp,Xi,Xc,Obs) $\leftrightarrow$ set-demo(Yp,Yi,Yc,Obs))

We rewrite `conforms/2` as Conf 6.6 using conjunction and assume that `set-demo/4` never fails. This assumption is valid under the intended interpretation since we are not interested in determining conformance between terminating systems whose executions never succeed[1]

Conf 6.6

$\text{conforms}'_S(\langle Xp,Xi,Xc\rangle,\langle Yp,Yi,Yc\rangle) \leftrightarrow$

$\exists$ Obs (set-demo(Xp,Xi,Xc,Obs) $\wedge$ set-demo(Yp,Yi,Yc,Obs))

The definite (Horn) clause form of Conf 6.6 is

Conf 6.7

$\text{conforms}'_S(\langle Xp,Xi,Xc\rangle,\langle Yp,Yi,Yc\rangle) \leftarrow$

set-demo(Xp,Xi,Xc,Obs) $\wedge$ set-demo(Yp,Yi,Yc,Obs)

[1]Note that we are using the following implication

$$((X \leftrightarrow Y) \wedge true(X) \wedge true(Y)) \rightarrow X \wedge Y$$

and can be executed on a suitable logic interpreter without encountering problems connected with negation by failure.

6.4 Partial conformance

We say that an implementation X *partially conforms* to a specification Y iff all the observations that can be made of the implementation can be made of the specification, i.e.

Conf 6.8

partially-conforms(⟨Xp,Xi,Xc⟩,⟨Yp,Yi,Yc⟩) ↔

∀ Obs (demo(Xp,Xi,Xc,Obs) → demo(Yp,Yi,Yc,Obs))

We derive the following *normal clause* form of Conf 6.8 by the same transformations described for Conf 6.1 above.

Conf 6.9

partially-conforms(⟨Xp,Xi,Xc⟩,⟨Yp,Yi,Yc⟩) ←

¬(demo(Xp,Xi,Xc,Obs) ∧ ¬demo(Yp,Yi,Yc,Obs))

However, this normal clause form suffers from the same problem of negated calls with variables as does Conf 6.9. Again, we can avoid this problem by deriving a clause about partial conformance from the Conf 6.10 which uses `set-demo/4` to reason over the sets of observations. Informally, if a system X partially conforms to a specification Y then the observations of X are a subset of those predicted by Y. We describe this formally by:

Conf 6.10

partially-conforms$_S$(⟨Xp,Xi,Xc⟩,⟨Yp,Yi,Xc⟩) ↔

∀ Obs$_X$, Obs$_Y$ ((set-demo(Xp,Xi,Xc,Obs$_X$) ↔ set-demo(Yp,Yi,Yc,Obs$_Y$))

∧ subset(Obs$_X$, Obs$_Y$))

subset(X,Y) ↔ ∀ Obs (Obs ∈ X → Obs ∈ Y)

together with an appropriate definition of ∈ . Since only one instance each of Obs$_X$ and Obs$_Y$ are implied by the use of `set-demo/4` then

Conf 6.11

partially-conforms$_S$(⟨Xp,Xi,Xc⟩,⟨Yp,Yi,Xc⟩) ↔

$\exists$ Obs$_X$, Obs$_Y$ ((set-demo(Xp,Xi,Xc,Obs$_X$) ↔ set-demo(Yp,Yi,Yc,Obs$_Y$))

$\wedge$ subset(Obs$_X$, Obs$_Y$))

We define $\in$ using a list structure to represent a set

X $\in$ nil ↔ **false**

X $\in$ Y.Ys ↔ X=Y $\vee$ X $\in$ Ys

and derive a Horn clause form of the subset relation by standard transformations:

(i) by writing ← for ↔

X $\in$ Y.Ys ← X=Y $\vee$ X $\in$ Ys

subset(X,Y) ← $\forall$ Obs (Obs $\in$ X → Obs $\in$ Y)

(ii) by substituting the definition for $\in$ in subset

subset(X.Xs,Ys) ← $\forall$ Obs ((Obs=X $\vee$ Obs $\in$ Xs) → Obs $\in$ Ys)

(iii) by expansion

subset(X.Xs, Ys) ←

$\forall$ Obs1(Obs1=X → Obs1 $\in$ Ys) $\wedge$ $\forall$ Obs2(Obs $\in$ Xs → Obs $\in$ Ys)

(iv) i.e. the recursive case

subset(X.Xs, Ys) ←

X $\in$ Ys $\wedge$ subset(Xs,Ys)

(v) using subset(nil,Y) and (X $\in$ nil ↔ **false**)

subset(nil,Y) ← $\forall$ Obs (Obs $\in$ nil → Obs $\in$ Ys)

(vi) since **false** → **true**

subset(nil,Y) ←

and then by standard transformations

Conf 6.12

partially-conforms$_S$(⟨Xp,Xi,Xc⟩ ,⟨Yp,Yi,Xc⟩) ←

set-demo(Xp,Xi,Xc,Obs$_X$) ∧ set-demo(Yp,Yi,Yc,Obs$_Y$)

∧ subset(Obs$_X$, Obs$_Y$)

subset(nil,Y)

subset(X.Xs,Y) ← member(X,Y) ∧ subset(Xs,Y)

member(X,X.Ys)

member(X,Y.Ys) ← member(X,Y)

6.5 Completeness

We say that an implementation X is *complete* w.r.t. a specification Y iff all the observations that can be made of the specification can be made of the implementation. This is expressed in the definition of the relation `partially-conforms/2` below:

Conf 6.13

partially-conforms(⟨Xp,Xi,Xc⟩ ,⟨Yp,Yi,Yc⟩) ↔

∀ Obs (demo(Xp,Xi,Xc,Obs) ← demo(Yp,Yi,Yc,Obs))

or using the `set-demo/4` relation:

Conf 6.14

partially-conforms$_S$(⟨Xp,Xi,Xc⟩ ,⟨Yp,Yi,Xc⟩) ↔

∀ Obs$_X$, Obs$_Y$ ((set-demo(Xp,Xi,Xc,Obs$_X$) ↔ set-demo(Yp,Yi,Yc,Obs$_Y$))

$\wedge$ subset(Obs_Y, Obs_X))

As with partial conformance we may derive the following normal program:

Conf 6.15

partially-conforms(⟨Xp,Xi,Xc⟩,⟨Yp,Yi,Yc⟩) ←

¬(¬demo(Xp,Xi,Xc,Obs) ∧ demo(Yp,Yi,Yc,Obs))

and also:

Conf 6.16

partially-conforms$_S$(⟨Xp,Xi,Xc⟩,⟨Yp,Yi,Xc⟩) ←

set-demo(Xp,Xi,Xc,Obs_X) ∧ set-demo(Yp,Yi,Yc,Obs_Y)

∧ subset(Obs_Y, Obs_X)

6.6 Program verification and notions of conformance

The concepts of conformance, partial conformance and completeness discussed above are very similar to those of total correctness, partial completeness and correctness in terms of verification for logic programs and logic algorithms. Clark and Tärnlund [31] first formulated verification criteria and this theory was elaborated in Clark's thesis [23] where it is described as the 'theory of the program computed relation'. This relation, denoted by $\mathcal{R}$ is defined by

$$\mathcal{R} = \{T' \mid P \models R(T)\}$$

where P is the procedure set, T′ is a solution to a goal R(T). $\mathcal{R}$ is the set of all solutions computable from P, covering all possible choices of the goal n-tuple T, and is independent of the goal. Clark regards the set of procedures P as the program and a goal G as a 'use' of P.

In our treatment we follow the approach of Hogger [62] who denotes a program by the tuple ⟨P,G⟩ and an algorithm by ⟨P,G,C⟩ where P is a set of procedures, G is a goal and C is a control strategy. In our case, G is always a negated atom `init` whose arguments are the observable variables of the system in question. Hogger's definitions of partial correctness, completeness and total correctness reason about Θ, the substitutions made to

the variables in a goal R(T) to give a substitution instance TΘ. The definitions employ propositions of the form

$$A \vdash_c B$$

to express that B is provable from A using resolution controlled by the strategy C. The definitions are given w.r.t a specification S;

Partial correctness of ⟨P,G,C⟩ w.r.t. S

$$\forall\, \Theta, (S \models R(T\Theta)) \leftarrow (P \vdash_c R(T\Theta))$$

Completeness of ⟨P,G,C⟩ w.r.t. S

$$\forall\, \Theta, (S \models R(T\Theta)) \rightarrow (P \vdash_c R(T\Theta))$$

Total correctness of ⟨P,G,C⟩ w.r.t. S

$$\forall\, \Theta, (S \models R(T\Theta)) \leftrightarrow (P \vdash_c R(T\Theta))$$

6.7 Equivalence

The interpreter outlined above in Section 6.2 produces one or more binding histories each of which is associated with one successful computation. Each instance of the structure to which *States* is bound represents the bindings that are made to the variables of all the goals and subgoals during one computation. These bindings permit reasoning about strong bisimulation, discussed by Milner in [95], which we adapt to our logic-based formalism as follows:

Definition 6.1 A binary relation $\mathcal{S} \subseteq \mathcal{P} \times \mathcal{P}$ over systems is a *strong bisimulation* if $(P, Q) \in \mathcal{S}$ implies for all $\mu \in$ Act,

(i) Whenever $P \xrightarrow{\mu} P'$ then, for some Q', $Q \xrightarrow{\mu} Q'$ and $(P', Q') \in \mathcal{S}$

(ii) Whenever $Q \xrightarrow{\mu} Q'$ then, for some P', $P \xrightarrow{\mu} P'$ and $(P', Q') \in \mathcal{S}$

■

If we restrict the bindings recorded to just the *observable* variables of a system, i.e. the variables in the initial goal, then we can only reason about *weak bisimulation*. We first extend our notion of labelled transition systems introduced in Section 4.11. *Act* is a non-empty set of atomic actions and *Obs* is a non-empty set of *observable* actions. We let Act^* be a non-empty set of *sequences* of atomic actions and Obs^* be a non-empty set of sequences of *observable* atomic actions. Each member of Obs^* is called a *trace* and corresponds to a path through graph of the observations of a system. The trace of a computation is the path starting from the initial state or minimum node to a final state or maximal node in the graph.

Definition 6.2 If $t \in Act^*$, then $\hat{t} \in Obs^*$ is the sequence gained by deleting all non-observable actions from t. ■

Definition 6.3 If $t = \mu_1 \ldots \mu_n \in Act^*$, then we write $S =t\Rightarrow S'$ if
$S \xrightarrow{\mu_1} \ldots \xrightarrow{\mu_n} S'$.

■

We now define a new labelled transition system $TS =< S, Obs^*, T, s_0 >$ where:

- S is a *non-empty* set of *states*,
- Obs^* is a non-empty set of sequences of *observable* atomic actions.
- $T = \{= t \Rightarrow \subseteq S \times S \mid t \in Obs^*\}$ is a set of *transition relations*,
- $s_0 \in S$ is the *initial state*.

We can now define a weak bisimulation:

Definition 6.4 A binary relation $\mathcal{S} \subseteq \mathcal{P} \times \mathcal{P}$ over systems is a (weak) bisimulation if $(P, Q) \in \mathcal{S}$ implies for all $\mu \in$ Act,

(i) Whenever $P \xrightarrow{\mu} P'$ then, for some Q', $Q =\hat{\mu}\Rightarrow Q'$ and $(P', Q') \in \mathcal{S}$

(ii) Whenever $Q \xrightarrow{\mu} Q'$ then, for some P', $P =\hat{\mu}\Rightarrow P'$ and $(P', Q') \in \mathcal{S}$

■

We can now define observation equivalence, or bisimilarity:

Definition 6.5 Systems P and Q are *observation equivalent*, or weakly bisimilar, written $P \approx Q$, if $(P,Q) \in \mathcal{S}$ for some weak bisimulation $\mathcal{S}$. That is

$$\approx = \bigcup\{\mathcal{S} : \mathcal{S} \text{ is a bisimulation}\}$$

■

The definition of conformance (Definition 6.1) is the same as observation equivalence for complete computations.

The inclusion of bindings made on finite failure branches of the execution tree permits reasoning about *failures* equivalence, described by Hoare in [58]. We adapt the definition of failures to our formalism as follows:

Definition 6.6 A *failure* is a pair (s, L) where $s \in Obs^*$ and $L \subseteq Obs$. The failure (s, L) is said to belong to a state S if there exists S' such that

(i) $S =s\Rightarrow$ ■, or

(ii) $S =s\Rightarrow S'$ and $S' \neq \Box$ and there is no state S'' such that $S' \text{—}\mu\rightarrow S''$ (i.e. S' is deadlocked)

■

Definition 6.7 Two systems P and Q are said to be *failures-equivalent* if their initial states possess exactly the same failures. ■

6.8 The conformance relation and message types

In a specification the *types* of messages may not be defined, being left as variables in the specification text, while an implementation may produce messages which are instances

of specific types or sub-types. For example the specifier of a buffer may not wish to be concerned with the actual messages that are transmitted, other than that there should be an end-of-transmission (`eot`) in the case of a terminating system. This specification may be used to derive an implementation which can accept typed or untyped messages. The logical variables representing the specification (or implementation) messages stand for the universal type. Since first order logic is untyped any typing of messages must be represented by explicit type predicates in the specification.

The definition of *Act* as ***unification*** over terms permits bindings made to the variables of the specification during the evaluation of equivalence to be recorded. These bindings represent the types of the messages of the implementation. We require that the syntactic names of the variables in the specification are preserved, and bindings made to them during equivalence checking are returned as an argument, denoted by a subscript $_B$ to the equivalence relation. Thus $X \approx_B Y$ denotes the observation equivalence relation between specification X and implementation Y where $_B$ is the set (possibly empty) $\{X_1/t_{x_1}, \ldots, X_1/t_{x_m}, Y_1/t_{y_1}, \ldots, Y_1/t_{y_n}\}$, $n \geq 0, m \geq 0$ of bindings made to the variables of both X and Y. V/t represents the binding of variable V to term t of type τ. If the specification has not described the types of the messages then $n = 0$ and the set consists of any bindings made to the m messages. The possibilities are:

(1) $[m = n]$ the implementation messages completely type-conform to those of the specification

(2) $[m > n]$ the implementation messages are a sub-type of the specification (partial conformance)

(3) $[m < n]$ the implementation messages are a more general type compared with the specification messages (completeness)

The second case is an instance of *under-specification* and the third case an instance of *over-specification.*

A *type derivation* algorithm is required to ascertain the union of the types of messages on any one stream, using for example that proposed by Shapiro [121] or Zobel [143]. For a

stream of m messages, the type union $\mathcal{T}$ is defined by

$$\mathcal{T} = \bigcup_{0}^{m} \tau_k \quad (0 \leq k \leq m)$$

where m is the number of messages, k is a message and τ_k the type of the message k.

6.9 Conformance and deadlock

Deadlock detection is part of the conformance problem. Deadlock occurs when no more progress can be made in a system, and neither success nor failure has occurred. The interpreter for the language under consideration must be able to indicate that no more computations can be performed. The operational semantics of some languages may also permit the static analysis of programs to detect deadlock, but in general this is computationally equivalent to interpreting the program.

The indication of deadlock in a set of observations of a terminating stream-based system is that one or more chains of observations of stream variables are not terminated with an eot message and yet no more observations can be made. Given a set of observations of one or more computations of a system, an initial state and an order relation $\leq$, we can detect the presence of deadlock by:

deadlocked(Obs) $\leftrightarrow$

$\exists$ X (X $\in$ Obs $\wedge$ $\forall$ Y (Y $\in$ Obs $\rightarrow$ (Y $\leq$ X)) $\wedge$ non-terminated(X))

non-terminated(X) $\leftrightarrow$

X=tail $\vee$ $\exists$ Y,Z (X=Y.Z $\wedge$ non-terminated(Z))

The use of the existential quantifier in the above definition does not require that every computation in Obs is deadlocked, but that at least one computation is.

An implementation may terminate early w.r.t. its specification, indicated by premature termination of one or more traces. We say that the implementation is equivalent to the specification up to premature termination, and describe this by the equivalence relation $\approx_p$. We write X $\approx_p$ Y where X are the observations of a computation which terminates early w.r.t. the observations Y of a specification:

$X \approx_P Y \leftarrow X \subset Y$

We can write $X \approx_D Y$ if X are the observations of a deadlocked computation and Y is not deadlocked:

$X \approx_D Y \leftrightarrow$

$X \subset Y \wedge$ non-terminated(X) $\wedge$ terminated(Y)

terminated(Y) $\leftrightarrow$

X=nil $\vee \exists$ Y,Z (X=Y.Z $\wedge$ terminated(Z))

6.10 Conclusion

In this chapter we have described notions of conformance and equivalence based on an extrinsic, observation-based view of systems. We have shown how a logic programming system, demo(Program,Goal,Control) can be enhanced to demo(Program,Goal,Control,Obs) where Obs is bound to a data structure representing the binding states of the observable variables during a computation. Conformance and equivalence are relations on observations.

Notions of implementation and conformance are closely related. We categorize notions of implementation below, adapting the scheme described by Brinksma in [16].

(1) 'implementation' as a synonym for the real or physical system that is the subject of conformance requirements and conformance testing.

(2) 'implementation' as a

   (a) *deterministic reduction* of a given specification. For example, an implementation can be derived from a specification by resolving choices that were left open in the specification.

   (b) *non-deterministic reduction* of a given specification. In this case, all-solutions choice in a specification is implemented by committed-choice in an implementation.

In this context specification and implementation are relative notions in a hierarchy of system descriptions.

(3) 'implementation' as an *extension* of a given specification, where context specification and implementation are again relative notions in a hierarchy of system descriptions. In this case an implementation adds information that is consistent with the original specification.

(4) 'implementation' as a *refinement* of a given specification. In this case the implementation provides more detail on the subdivision of the specification itself into smaller components. The implementation and specification are extensionally equivalent in that their observable behaviour cannot be distinguished.

We do not consider an implementations as necessarily being a physical system since we describe logic programming systems which may not map directly onto a physical architecture. We have preferred to discuss reduction and extension implementation , notions which are related to partial conformance and completeness respectively. The notion of refinement is not dealt with directly since our conformance relation is on sets of observations. However, an implementation which completely conforms to a specification may be a refinement of the specification.

Chapter 7

Conclusion

This chapter summarises the work presented in this thesis, briefly describes some related areas of research and suggests future directions for research.

7.1 Summary

The central tenet of this thesis is that there can be advantages in relating the description and implementation of concurrent systems by a common and coherent underlying formalism, namely first order logic.

The essential characteristics of concurrent systems are introduced in Chapter 2 and the use of logic to describe these systems is set out in Chapter 3. Communication in a system is regarded as taking place via shared write-once (logic) variables which are incrementally bound to data structures during the execution of that system. The observations that can be made of these bindings comprise sets which are partially ordered by relations determined by the topologies of the structures to which the shared variables are bound. First order logic is a suitable formalism for the description of these sets since they are characterized by an associated order relation.

Two types of Horn clauses, $\mathcal{SET}$ and $\mathcal{PROG}$, can be derived from the first order descriptions. These forms can be regarded as logic programs in the logic languages $L_{\mathcal{S}}$ and $L_{\mathcal{P}}$ respectively. The successful execution of $\mathcal{SET}$ on a suitable logic interpreter results in a

variable in the query being bound to a data structure which represents the set of observations that can be made of the system described. The operational semantics of $L_{\mathcal{S}}$ do not have to incorporate concurrency and Prolog is a suitable model of such a language. The *behaviour* of $\mathcal{PROG}$ when executed on a suitable interpreter is that described by $\mathcal{SET}$ and hence the operational semantics of $L_{\mathcal{P}}$ must include rules governing the concurrent execution of $\mathcal{PROG}$.

Chapter 4 introduces the concurrent logic programming language SILCS, a logic language of the type $L_{\mathcal{P}}$. Programs in SILCS are Horn clauses with the sole syntactic addition of a sequencing operator whose logical meaning is AND-conjunction. Synchronisation and suspension are not indicated by textual annotations in SILCS programs, but instead are are defined as a part of the reduction strategy of calls in atomic and sequence groups. There are no guards in SILCS programs, and all-solutions non-determinancy is a part of the operational semantics of the language. SILCS therefore spans the gap between descriptions in logic of the expected behaviour of concurrent systems and programs in mainstream concurrent logic programming languages. The problem of deriving programs whose execution conforms to the descriptions predicted by the poset descriptions is examined this chapter. Equivalences between programs are investigated, in particular between stores based on data structures and process networks.

Chapter 5 describes the ways in which SILCS programs can be used to guide the construction of concurrent systems. SILCS has not been designed for the *implementation* of concurrent systems — its semantics dictate expensive computational mechanisms, for example atomic unification and synchronous communication. The 'all-solutions' model of non-determinism employed by SILCS also makes it unsuitable for the implementation of systems which exhibit *committed-choice* behaviour. Don't-care non-determinism rather than don't-know non-determinism is a characteristic of many such systems — only one possible successor state to any given state is permitted for efficiency reasons. Committed choice concurrent logic programming languages are well suited to implementing such systems. SILCS programs could be interpreted in these languages, but this results in less efficient systems than transformation techniques, the solution explored in this chapter. The range of concurrent logic programming languages available as real implementations is increasing, and general guidelines are given rather than targeting specific languages. The transfor-

mations required to obtain programs in committed choice logic programming languages from SILCS programs vary according to the syntax and semantics of the target language. The transformation process is unlikely to be completely mechanisable in the case that the target language is guarded. Some form of heuristics is required to aid the decision as to where to place the guards.

One problem that concerns the designers of concurrent systems is that of testing the conformance between an implementation and its specification. Conformance testing essentially describes the relationship between two behaviours (actual and predicted) — Chapter 6 discusses such relations using first order logic to reason about equivalent behaviours. The methodology presented is based on the comparison of the output of snapshot interpreters. Interpreters for logic programming languages can be enhanced with the facility to produce snapshots of bindings during a computation and snapshots collected into data structures representing the sets of observations that can be made of the computation. The conformance of programs to specifications is expressed in terms of the comparison of the set of bindings from the interpretation of the program with those predicted by the specification. The specifications in our methodology are Horn clauses which can themselves be interpreted (with respect to suitable goals) producing the set of predicted bindings.

During the course of the research reported in this thesis software tools were developed which served as test benches for the ideas and theories that were being investigated. These tools were prototypes, not fully fledged releasable systems.

The construction of a portable interpreter for SILCS written in sequential Prolog is reported in Appendix B. The implementation was developed in AAIS Prolog [1] on an Apple Macintosh microcomputer and was subsequently ported to a range of machines running a variety of implementations of Prolog, including SICStus Prolog [18] on a Sun 3 workstation. The compiled version of the interpreter was an effective platform for investigating the behaviour of the SILCS programs presented in Chapter 4. The interpreter was enhanced with trace, spy and snapshot facilities.

Appendix B reports the construction of a prototype conformance tester, developed to investigate the conformance between SILCS programs and their Horn clause specifications. The specifications were interpreted in Prolog and the sets of predicted observations

compared with those produced by the snapshot interpreter.

7.2 Related work

Work related to this research can be roughly divided into two areas: concurrent logic programming languages and specification techniques for concurrency.

7.2.1 Concurrent logic programming

Logic programming has developed into a rich research field since its beginning in the early 1970's (see [76] for an account of the early development of logic programming). During the last two decades significant advances have been made both in the theory and practice of logic programming, and the original logic programming language, Prolog, is available on a wide variety of machines and operating systems. The idea of using predicate logic as a language for parallel programming was investigated by van Emden and Filho [135] and Hogger [61]. IC-Prolog [30] was an early attempt to design a concurrent logic programming language, but the Relational Language described by by Clark and Gregory [26] was the stimulus for several concurrent logic programming languages including Concurrent Prolog [116], Guarded Horn Clauses [131] and Parlog [27]. These languages have been discussed earlier in Chapter 5 and we discuss here aspects of research using these languages which is pertinent to the aims of this thesis.

Some attempts have been made to give concurrent logic programming languages a semantics similar to that of algebraic techniques. Early work by Beckman [6], Ellis [38] and Hussey [64] attempted to identify the semantics of some CLPL's with that of CCS. However, the proposed translations of concurrent logic programs into a CCS-like formalism was inelegant and complex.

7.2.2 Flat Concurrent Prolog and traces

The work of Lichtenstein et al. which is summarised in Appendix C is related to our own in that the authors analyse bindings to variables made during the execution of concurrent

logic programs. The model of concurrency used by Lichtenstein et al. is that of *interleaving*, which enables them to propose a hierarchy of abstractions {traces, behaviours, labeled goals, goals} where traces capture more details of a computation than behaviours, behaviours than labeled goals and labeled goals than goals. Given a trace and a goal it is possible to uniquely reconstruct the computation from which the trace was abstracted, but interleaving semantics means that behaviours ignore some details of a computation. True concurrency coupled with atomic unification as proposed proposed in our thesis makes these distinctions unnecessary. In our view, FCP is *restricted* as a description language for concurrent systems due to its lack of an explicit sequential operator.

Further work by the same group and Gerth [46] develops a denotational semantics for FCP. The theory is complex due to the read-only variable in the version of FCP described and the attempt to cast the semantics within the trace framework of CSP.

7.2.3 A comparison of LOTOS and SILCS.

LOTOS [10] has adopted the algebraic framework of CCS [92] and CSP [58] and has a very different semantics to that of logic programming languages. In particular LOTOS has no notion of *failure*, only success and suspension, while pure logic programming admits only success and failure as permissible outcomes of computations. However SILCS in common with the concurrent logic programming languages can describe suspension as well as success and failure and there is some common ground between LOTOS and this group of languages.

Although the synchronisation mechanism of LOTOS appears to be very different from that of a concurrent logic programming language, it possesses many similarities with unification and constraint evaluation. Communication in LOTOS — participation in atomic events — is via named event gates which have an associated alphabet and the messages that are passed are typed expressions. Concurrent logic programming languages express communication by the instantiation of shared logical variables. Communication in LOTOS is completely synchronous, unlike that of most CCLPs.

The simultaneous operator of SILCS is associative, unlike the parallel constructors of many algebraic theories including LOTOS (see Appendix D).

7.2.4 CIRCAL and concurrency

Circal, an algebraic description technique for concurrent systems developed by Milne [91] permits the modeling of parallel and concurrent behaviour. The semantics of CIRCAL relevant to this thesis are discussed in Appendix D.3. The dot operator of CIRCAL has similar semantics to that of the simultaneous operator of SILCS. Parallel events can be represented as simultaneous events in both formalisms. Moreover, suspension is effectively expressed by the same mechanism in both CIRCAL and SILCS. For example in CIRCAL, if $P \Leftarrow \alpha\beta P'$ and $Q \Leftarrow \beta\alpha Q'$ then their simultaneous composition results in deadlock: $P \bullet Q \Leftarrow \Delta$. Similarly, the following SILCS process will deadlock:

$\leftarrow$ (A=x & B=y) $\wedge$ (B=y & A=x)

However the choice operators "+" and "$\oplus$" of CIRCAL permits only *exclusive* choice, and it is unlikely that a direct mapping can be made between SILCS programs and CIRCAL descriptions.

7.3 Conclusions and future research

The research reported in this thesis has demonstrated that the specification of concurrent systems is *feasible* using first order logic. Our basic assumption in this research has been that communication in a concurrent system occurs through bindings made by atomic unification to shared logic variables. Using this model we can describe the behaviour of a concurrent system by observations comprising the binding histories of the observable variables of that system.

Our research has described the use of first order logic to:

(1) describe the observations that can be made of a concurrent system as partially ordered sets,

(2) derive Horn clause programs which reason about data structures representing the sets in (1),

(3) derive programs in a concurrent logic programming language whose execution produces the behaviour described by the programs in (2),

(4) describe the operational semantics of the concurrent logic programming language referred to in (3),

(5) describe the conformance relation between specifications and programs, for example between (2) and (3).

Our research has lead to the design of the logic programming language SILCS whose operational semantics encapsulates the behaviour of the concurrent systems which are the subject of this study. The nature of the complex mechanisms required to fully implement an interpreter for the language mean that it is not feasible to implement a concurrent system using SILCS. The difficulties are associated with the implementation of atomic unification and the suspension mechanism of SILCS in a distributed system. Modification of the semantics of SILCS to facilitate the task of implementation would effectively result in the design of a new concurrent logic programming language, a subject for future research. A more plausible route to implementation is the derivation of code in committed choice logic programming languages from SILCS programs. Although SILCS differs from these languages regarding aspects of both syntax and semantics, we have outlined methods for such derivations in Chapter 5. Due to the differences between SILCS and CCLP languages these methods are only partially formalised.

The research undertaken has demonstrated the usefulness of first order logic as a formalism underlying the specification of concurrent systems in logic. Advantages of such an approach over algebraic techniques for the specification of concurrency include the more natural description simultaneous actions and the ease of building interpreters for logic languages.

The work reported in this thesis has not attempted to completely exhaust the topic of using first order logic to specify and reason about concurrent systems. There are several potential paths that future research could investigate, and we sketch a few of these below.

Automatic derivation of SILCS programs

Automating the derivation of SILCS programs from descriptions in first order logic is likely

to be a fruitful area of future research. The present research has described the transformations as being achieved by a mixture of heuristics and formal rules. Further work would probably lead to a better understanding of the general relationship between descriptions of observations as partially ordered sets and the view of executions as approximations to a final state. This would facilitate the formulation of rules for deriving SILCS programs from specifications.

Design of a new concurrent logic programming language

The mismatch between the semantics of SILCS and those of existing concurrent logic programming languages means that the transformation of SILCS programs into programs in those languages is non straightforward. As an alternative the design of a new concurrent logic programming language could be investigated which could be implemented efficiently and whose semantics are closely related to those of SILCS.

Interpreters for SILCS in concurrent languages

The SILCS interpreter described in this thesis was implemented in sequential Prolog. Other logic programming languages can be used to build concurrent interpreters for the language. One of the strongest contenders for this task is Andorra Prolog, whose basic model is still in the process of being defined. It appears that this language will be very suitable for the construction of a SILCS interpreter since the Andorra Prolog is not a committed choice language, but can express concurrency and incorporates constraints [54].

Types

SILCS is an untyped language and goals which fail because they are 'ill-typed' are not considered to be distinct from those which fail because the program did not contain clauses about them. The incorporation of a theory of types in the definition of SILCS would make the language more robust as a specification formalism. There has been an interest in type theory for logic programs [96, 121, 142] which could provide a starting point for such a line of investigation. Type derivation and static error checking can be based based on syntactic typing. However the inferred typing information is only partial and the addition of type declarations would extend the ability of a typing algorithm to detect errors.

Constraints

The development of a type theory would also enable work to be done on the incorporation of constraints into SILCS. The range of problems that can be specified using SILCS with constraints needs to be explored, and sample specifications constructed. An interpreter for this constraints form of SILCS should then be built. A formal comparison between the use of constraints in LOTOS and SILCS might lead to the development of a constraint-oriented specification style in SILCS and the development of a methodology for the comparison of constraint and unification oriented programs in SILCS.

Equivalences

The present research has not produced a detailed theory of equivalences for logic-based specifications of concurrent systems. The development of a theory of equivalences which would encompass both the descriptions of the sets of observations and SILCS programs would significantly improve the usefulness of our approach. We would expect equivalences to be defined which would enable comparisons to be made between various classes of concurrent systems including those which are data oriented and those which are process oriented. Equivalence relations are an established component of algebraic theories, forming the basis of transformational techniques and enabling the construction of tools to aid the development of specifications and programs. We envisage similar benefits to arise from the formulation of an equivalence theory for logic specifications.

Conformance Testing

The definition of conformance given in this thesis is a relationship between triples of the form ⟨Program,Goal,Control⟩. One interesting avenue of research would be to investigate the feasibility of *generating* control strategies for conforming systems, given a particular program and goal. Related to this would be the possibility of deriving specifications from programs and hence permitting comparisons to be made between logic programs.

The development of conformance test software for a variety of concurrent logic programming languages is a natural area of research that leads on from the work described in this thesis defining the conformance relation. Such software would facilitate the comparison of programs written in different logic programming languages as well as comparing programs

and specifications.

Appendix A

The SILCS interpreter

An interpreter for SILCS was implemented in Prolog with the aim of providing a platform for the development of the SILCS language and as a test bed for investigating the behaviour of programs written in SILCS. The requirements were for fast development, ease of maintenance and portability of the interpreter; these factors influenced the design decisions which were taken regarding the construction of the interpreter. The operational semantics of SILCS are defined with reference to an idealized Horn clause interpreter and form the basis for the construction of an interpreter for the language. The method of interpretation chosen in this research follows closely that described by Kowalski [74] and discussed in Section 4.6 of this thesis on page 120. Appendix A.6 contains listings for the top level Prolog code of the SILCS interpreter.

A.1 Implementation language and hardware

The use of a logic programming language to implement the interpreter facilitated the construction of the system since SILCS is a logic programming language. Some of the underlying mechanisms of the implementation language such as the representation of terms, unification and clause access were used directly in the interpreter and did not have to be reimplemented. The task of maintaining the interpreter was also made easier since logic programming languages have a semantics which can be defined in logic.

Sequential Prolog was chosen as the implementation language since at the time that the interpreter was built there was no readily available implementation of a concurrent all-solutions logic programming language. Parallel evaluation of atomic groups and the concurrent reduction of calls within an atomic group was simulated by the use of run queues. The production of a concurrent SILCS interpreter was not an aim of the research reported in this thesis.

The Prolog implementations chosen for the construction of the interpreter conformed as closely as possible to the 'Edinburgh' syntax due to the portability requirement for the interpreter. The machine on which the interpreter was initially developed was the Apple Macintosh micro-computer since this machine offered a WIMPS environment; the Prolog implementation chosen was AAIS Prolog [1]. An IBM-PC compatible was also used for some initial system development, using Arity Prolog [4]. The SILCS interpreter was subsequently ported to SICStus Prolog [18] running on a Sun 3/260 computer under the Unix operating system.

A.2 Basic design

The design of an interpreter implemented in a logic programming language can be based on the demonstrate/2 relation outlined in Kowalski's book [74], or a unary-argument variant commonly referred to as solve/1 described by Shapiro in [124]. We first describe the outline of demonstrate/2, and then show how a solve/1 interpreter can be derived from it.

(i) demonstrate/2

This relation relies on none of the built-in facilities of the meta language, for example clause access or variable renaming. It is invoked as *demonstrate(Program,Goals)* where *Goals* is a data structure representing the object language goals to be solved and *Program* is a data structure representing the object language program. The top level definition of demonstrate/2 is:

demonstrate(Program, Goals) ←

```
        empty(Goals)

demonstrate(Program, Goals) ←
        select(Goal,Goals,Rest) ∧
        member(Clause,Program) ,
        renamevars(Clause,Goals,Clause') ∧
        parts(Clause,Head,Body) ∧
        match(Goal,Head,Substitution) ∧
        add(Body,Rest,InterGoals) ∧
        apply(InterGoals,Substitution,NewGoals) ∧
        demonstrate(Program,NewGoals)
```

The *control strategy* of a logic language consists of a *computation rule* determining the order in which goals are reduced, and a *selection rule* determining the order in which clauses are chosen for matching with a goal. No control strategy is defined for a 'pure' logic programming language. The computation rule of the object language is determined by the definition of the relations *select/3* and *add/3*. Access to clauses of the object language (the selection rule) is determined by the definition of *member/2*. These relations are operations on recursively defined data structures and the order of call selection and clause search depends on the way in which the data structures are traversed. The object language *Goals* are commonly represented as a list, and *Prog* as either a list or a tree.

(ii) solve/1

An interpreter implemented using demonstrate/2 is inefficient due to

(1) storing the entire object language program as an argument to one clause of the metalevel program and

(2) explicit procedures in the meta language required for access clauses, renaming of variables apart, matching and application of substitutions.

These limitations may make such an interpreter impracticable but can be overcome by using some of the inbuilt mechanisms of the meta language. This is achieved by partially

evaluating the interpreter of the meta language itself with the demonstrate/2 program. The general outline of an interpreter built using this technique is:

```
solve(Goals) ←
        empty(Goals)

solve(Goals) ←
        select(Goal,Goals,Rest) ,
        clause(Head,Body) ,
        unify(Goal,Head) ,
        add(Body,Rest,NewGoals) ,
        solve(NewGoals)
```

We represent logical conjunction by "," in the above program to emphasise the meta language is an implemented logic programming language with its own control strategy. Each clause *Head* ← *Body* of the object language program is stored as one meta language clause in the form of *clause(Head,Body)*. The clause access mechanism of the meta language ensures that the object language variables in each object language clause are renamed apart from the object language variables of the Goal(s) during matching. Unification of the current *Goal* with the *Head* of the selected object language clause ensures that the substitutions generated are applied to the variables in the Goals. An interpreter constructed using this method is more efficient than demonstrate/2 since the clause access and variable renaming mechanisms of the meta language are made directly available to the interpreter. Larger object level programs can be interpreted using solve/2 compared with demonstrate/2 since there is generally a limit to the size of one clause in the meta language; there is one meta level clause for each clause in the object language.

The order computation rule encoded in solve/1 is determined by the definition of the relations *select/3* and *add/3*. A restriction of the technique is that clause selection is effected using the selection rule of the meta language. A different selection rule can be implemented if a setof predicate is used to gather selected object level clauses into a data-structure from which they can be explicitly chosen according to the selection rule of the object language.

Choice of interpreter design

An interpreter built according to the `solve/1` design is more efficient than one employing `demonstrate/2` if the clause access, variable renaming and substitution mechanisms of the implementation of the meta language are more efficient than those achievable by explicit coding. In an initial pilot study the performance was recorded of an interpreter of Prolog implemented. The performance of the `solve/1` design was approximately 5 to 15 times as fast as that of the `demonstrate/2` design for the naive reverse program. Computations involving deeper recursions could be explored with the former design compared with the latter before memory limitations of the meta language implementation were reached. Therefore it was decided to base the SILCS interpreter on the `solve/1` design.

A.3 Design details

A.3.1 Storing the object language code

The SILCS program to be interpreted is stored as a series of facts asserted into in the Prolog data base. SILCS code can be read in from disc or entered directly by the user. Each SILCS clause is stored twice, firstly with all variables replaced by constants so that the variable names of the object level code are preserved for listing purposes, and secondly with new meta language variables replacing the object language variables of the program text. Each fact has two arguments, the first being a tuple representing the head of the SILCS clause and the second being a list of tuples representing the calls in the body of the SILCS clause. If the SILCS clause is a fact, then the second argument to the Prolog fact is the empty list.

Each clause in the second form is *normalised*. All arguments in the head are replaced by unique variables and explicit calls to these variables with the original terms in the head are introduced into the body of the clause. Normalization of object level clauses facilitates the collection of calls into atomic groups since the unifications of arguments of a call with arguments in the head of a clause are candidates for inclusion in atomic groups together with calls in the body of the clause. For example, the SILCS clause

merge(X.Xs, Ys, X.Zs) ← merge(Xs, Ys, Zs)

is converted into the following normalised form:

merge(A, Ys, C) ← A=X.Xs ∧ C=X.Zs ∧ merge(Xs, Ys, Zs)

and stored as the Prolog facts

clause(merge('A', 'Ys', 'C') ,[merge('Xs', 'Ys', 'Zs')]).

normalised(merge(A, Ys, C) , [A=[X | Xs], X=[X | Zs], merge(Xs,Ys,Zs)]).

A.3.2 Implementing the operational semantics of SILCS

Parallelism and concurrency

The SILCS interpreter does not simulate OR-parallelism, but implements choice by backtracking. AND-parallelism is implemented by interleaving; concurrency is implemented according to the rules defining the reduction of atomic groups.

Membership of atomic groups

An idealised interpreter for SILCS collects calls using the definitions of *direct* and *indirect* constraints (Definitions 4.9 and 4.10 respectively). The operation of the idealised interpreter assumes that either *static analysis* of a SILCS specification or *dynamic analysis* of goals is employed to determine indirect constraints. Both methods are computationally expensive. The Prolog implementation of the interpreter employs the computationally less expensive method of determining membership of atomic groups according to possible constraints (Definition 4.11). Two calls are deemed to be members of the same atomic group if they share one or more variables with a third call, which is also deemed to be a member of that atomic group. Categorisation of membership by possible constraint is more conservative than by indirect constraint, but is acceptable for the task of building an efficient interpreter.

Sequence groups

The interpreter fully implements the semantics of SILCS w.r.t. sequence groups. A sequence group is defined to be those calls linked by the sequential operator (Definition 4.7, page 125) and is included in an atomic group if any of the calls that constitute the sequence group depends on any existing calls in the atomic group. Sequence groups are represented by lists of calls connected with the &/2 operator. Thus the SILCS sequence group `(a,b)&(c,d)` is represented as the term `[a,b]&[c,d]`.

Reduction of calls in a query

The reduction of calls in a query is performed according to the strategy described in Section 4.10.3. This simple algorithm does not return an indication of which calls in a query suspend. The SILCS interpreter is enhanced so that all suspended calls are collected and their states displayed to the user when no more atomic groups can be reduced. The variables in an initial query are collected in a list as `variable-variable_name` pairs so that bindings can be displayed at the end of a computation.

The reductions of atomic groups are co-routined to give the effect of and-parallel operation. The interpreter employs an overall loop to reduce atomic groups and regroup the calls resulting from reductions. An atomic group whose attempted reduction suspends is allocated to a suspension queue and a deadlocked computation is detected by an empty run queue and a non-empty suspension queue.

Reduction of calls within atomic groups

Atomic groups are treated as the unit of atomic reduction in the SILCS interpreter. No reductions are made available from an atomic group until all of its members are reduced successfully. The interpreter implements the operational semantics described in Chapter 4 (Section 4.10.4). Calls within an atomic group are processed using a local run queue. A separate suspension queue is maintained for each group instead of using a marker to detect suspension of the attempted reduction of an atomic group. The result of an attempted reduction of an atomic group is either to return success and a list of the reduced calls,

or suspended and the original atomic group.

SILCS system calls

Various SILCS system calls are implemented in the interpreter; the names of these are held in a table along with the definitions of their behaviours. There are three possible results for a call to a SILCS primitive — *success*, *suspension* or *failure*. Only the first two are returned to the calling routine; failure in a call to a system primitive results in overall failure of the computation unless tracing of the computation is enabled (see below). If a call to a primitive suspends then the variables on which it is suspended are returned along with the suspension flag.

Some SILCS system calls make use of calls to the underlying Prolog system, but with checks on their arguments before they are invoked. For example the Prolog $</2$ is used as the basis of the SILCS primitive of the same name. However if both arguments are not ground then the SILCS call suspends without invoking the Prolog primitive. The following is the entry in the system call table for $</2$ where `contains_vars/2` returns as its second argument the variables in its first argument, and `all_data/1` checks that there are no variables in its argument. Such checks reduce the speed of execution of the interpreter.

```
sys_call(A<B, suspend(Vars)):-  contains_vars(A<B, Vars).
sys_call(A<B, success):-  all_data(A<B) , A<B.
```

SILCS system calls which are not calls to Prolog system calls are similarly defined:

```
sys_call(data(X), suspend(X)):-  var(X).
sys_call(data(X), success):-  nonvar(X).
```

A.4 Enhancements

The interpreter has been enhanced with respect to the basic model outlined above by the addition of

(1) reporting of variable bindings on success,

(2) deadlock detection and reporting of variable bindings in that state,

(3) user spy and trace,

(4) snapshot of current variable bindings,

(5) set solutions to queries.

The snapshot facilities have been used to form the basis of a conformance tester (see below).

Report of variable bindings on success

The SILCS interpreter reports the state of variable bindings for successful queries. This requires the use of the predicate read/2 to read in the initial query — the term read and a list of `variable=symbolic_name` pairs are returned. For example, if the Prolog goal

```
?- read(Term, Table).
```

is executed and the user inputs the term `foo(A,B,t(A))` then `Term` will be bound to `foo(_1,_2,t(_1))` and `Table` will be bound to `[_1 = 'A', _2 = 'B']`. The technique is similar to that employed by traditional Prolog interpreters, for example SICStus Prolog [18].

The bindings to the variables in the query represent the final observable state of a successful computation about a system.

Snapshots of variable bindings

A snapshot facility is incorporated into the interpreter permitting display of binding states of the variables in the initial query at each reduction cycle for atomic groups. each snapshot corresponds to an observable state of the system being interpreted.

Deadlock detection

Deadlock detection has been added as an enhancement to that described in the basic SILCS interpreter. This implementation of the SILCS interpreter employs the possible constraints rule (Definition 4.11) and is thus over-conservative in the strategy used to form atomic groups. As a result, no queue of suspended atomic groups is required to detect deadlock. However the implementation of the interpreter *does* employ a suspension queue to permit deadlock detection and reporting of the deadlocked goals and the instantiation states of their variables. In this respect the behaviour of the interpreter is similar to that of success reporting, except that all deadlocked goals are reported, not just those in the initial query. The deadlocked queries are those queries in the suspension queue when the run queue is empty.

User spy and trace

The user spy and trace facilities incorporated into the SILCS interpreter are similar to those in conventional logic interpreters. Users can denote a set of predicates for which spy or trace is to be activated or deactivated and may turn on spying or tracing. This information is kept as asserted facts which the interpreter checks when reductions are attempted. The use of spy and trace slows down the operation of the interpreter due to the extra processing time required for performing the checks and displaying spy or trace information.

The interleaving operation of the interpreter facilitates display of spy and trace messages, which are performed as side-effects in the main reduction engine for calls. A parallel implementation would require more sophisticated display techniques, for example the output of information for each marked predicate to a separate file or window.

Although algorithmic debugging techniques have been proposed for Prolog by Shapiro [117], by Lichtenstein [83] for FCP and by Huntbach [63], these have not been incorporated into the present design of the SILCS interpreter.

A.5 Set solutions

The snapshot facility forms the basis of the implementation of the set-demo program (see Section 6.3). The setof/3 relation of the Prolog implementation can be used to provide a set-demo predicate if memory limitations permit. In practice it was found necessary to directly implement a set-demo relation using `assert` and `retract/1` due to efficiency reasons.

The demo program augmented with the snapshots of observable variables is used to provide `demo/3` whose top level call is demo(Prog,Goals,States) where States is the set of states of one complete computation. The states of the observable variables of the initial query are asserted as facts in the Prolog database, and backtracking performed without user control by use of a fail loop at the top level of the interpreter to provide information about the states of all the computations.

States are stored in the Prolog database using `assert/1`. On termination of a query a marker is asserted at the end of the states facts. The states stored as facts are gathered up into one data structure by a fail loop using `abolish/1` which terminates on encountering the marker. The data structure representing the states is sorted and duplicates removed.

A.6 Prolog code for the SILCS interpreter

```
solve(Calls, Result, Snaps) :-
        create_variable_table(Calls,VarTable),
        create_atomic_groups(Calls,Groups),
        SuspendedQ = [],
        solve(VarTable, Groups, SuspendedQ, Result, Snaps).

% no more groups in list - success
solve(VarTable,[],[],success(VarTable),[]).

% deadlock detected
```

```
solve(_,[],Susps,deadlock(Susps),[]):-
        Susps \== [].


% there are atomic groups in the runQ to be reduced - process one & loop
solve(Vars,Groups,Susp,Result,Snaps) :-
        Groups \== [],
        select1(Groups,Group,Rest),
        reduce_atomic(Group,Red,AtResult),
        solve_check(Vars,Rest,Group,Red,Susp,AtResult,Result,Snaps).


% Check on the result of an attempt to process an atomic group in the runQ

% (i)   The result was successful -
%       add the reduction to the runQ and continue
solve_check(Vs,Groups,Group,Red,Susp,succeeded,Result,[Snap|Snaps]):-
        copyvars(Vs,Snap),
        insert(Groups,Red,NewGroups),
        solve(Vs,NewGroups,Susp,Result,Snaps).


% (ii)  The attempt suspended -
%       add the list of suspended Groups list to the suspension list
solve_check(Vs,Grps,_,_,Suspended,suspended(Susps),Result, Snaps):-
        NewSusps = [Susps|Suspended],
        solve(Vs,Grps,NewSusps,Result,Snaps).


% simple left-first selection of atomic group to reduce
select1([Group|Groups],Group,Groups).
```

```
% reschedule the result of the reduction onto the end of the runQ
insert(Groups,Red,NewGroups):-
        append(Groups, Red, NewGroups).
```

Appendix B

Conformance testing

A basic conformance tester has been constructed in order to investigate the conformance techniques outlined in Chapter 6 with reference to SILCS programs.

B.1 Generating the predicted observations

The specifications against which SILCS programs were tested were derived from first order descriptions by transformation into recursive form (Chapter 3). This form can be used to generate test sets of predicted observations, or test that a given observations set is valid (predicted by the specification). Execution of the definite programs of the form $\mathcal{SET}$ derived directly from the specifications produce the complete poset of predicted observations without use of the *set-demo* relation since the specification describes that complete set. It was not necessary in practice to interpret the specifications using a specialised interpreter, and the execution mechanism of the Prolog interpreter was sufficient.

B.2 Generating the actual observations

Modifications were made to the snapshot version of the SILCS interpreter to provide an all-solutions *set-demo* relation. Given as input a system description (a set of SILCS procedures) P and a query Q about the initial state the execution of the query ?-

`set-demo(P,Q,Obs)` binds `Obs` to a data structure representing the bindings that have been made to the variables in the query by the computations. This data structure is a list of lists, each of which represents the bindings made during one computation.

B.3 Comparing the sets of observations

The data structures representing the predicted and actual observations are compared as follows (note that lists are used to represent sets):

(1) The data structure representing the actual observations is flattened into one list and duplicates are removed. An ordered tree insert algorithm is used to flatten the list.

(2) Each items in the predicted set is checked for membership in the actual set and vice-versa. Those items from each set which are not present in the other set are flagged accordingly

```
compare(Predicted, Actual, Result) :-
        flatten(Actual, ActualList),
        check(Predicted, ActualList, ExcessPredicted),
        check(ActualList, Predicted, ExcessActual),
        generate_result(ExcessPredicted, ExcessActual, Result).

flatten(X,Y) :- flatten(X,[],Y).

flatten([],Tree,List):-
        flatten_tree(Tree,List,[]).
flatten([Ob|Obs], Store, List):-
        flatten_1(Ob,Store,Temp),
        flatten(Obs,Temp,List).

flatten_tree([],List,List).
flatten_tree(t(L,Obs,R), List,Temp):-
        flatten_tree(R,R1,Temp),
```

```
        flatten_tree(L,L1,List,[Ob|R1]).

flatten_1([], List, List).
flatten_1([Ob|Obs], Store, List):-
        insert(Ob,Store,Temp), flatten_1(Obs,Temp,List).

insert(Ob,[],t([],Ob,[])).
insert(Ob,t(L,Ob,R),t(L,Ob,R)).
insert(Ob, t(L, Ob1,R), t(L1,Ob1,R)):-
        Ob @< Ob1, insert(Ob,L,L1).
insert(Ob, t(L, Ob1,R), t(L,Ob1,R1)):-
        Ob @> Ob1, insert(Ob,R,R1).

check([],Y,[]).
check([X|Xs],Y,[X|Excess]):-
        not(member(X,Y)), check(Xs,Y,Excess).
check([X|Xs],Y,Excess):-
        member(X,Y), check(Xs,Y,Excess).

member(X,[X|_]).
member(X,[_|Ys]):- member(X,Ys).

generate_result([],[],totally_conforms).
generate_result([],[_|_],complete).
generate_result([_|_],[],partially_conforms).
generate_result([P|Ps],[A|As],mismatch([P|Ps],[A|As])).
```

B.4 Performance

Limitations were experienced with the Prolog system and the hardware that were used — execution of the conformance test took a long time, and the memory limit of the system

was often exceeded. The choice of relations to be tested was limited to simple examples due to these restrictions, and the conformance testing system requires improvement if it is to be used "in anger".

Appendix C

A traces model of FCP computations

This appendix summarises work by Lichtenstein et al. reported in [84] which attempted to use some ideas from the traces model of CSP as a basis for the description of computations in FCP. A comparison between this approach for FCP and that of our own for SILCS is presented in Section 7.2.2 of this thesis.

The state of a computation is denoted by the pair $\langle R, \theta \rangle$ where R is a resolvent or one of the special symbols *fail* or *deadlock*; θ is a substitution. A communication is a substitution annotated as input or output, or *fail* or *deadlock*. An event is a triple $\langle r,c,comm \rangle$ where r and c are integers specifying respectively the goal and clause used, and *comm* is a communication. States are transformed by the application of transition rules, the application of which produces events. Thus $S_n \stackrel{e_n}{\rightarrow} S_{n+1}$ denotes the transformation of state S_n to state S_{n+1} via event e_n. A computation is either a finite sequence $C = I(G) \stackrel{e_0}{\rightarrow} S_1 \stackrel{e_1}{\rightarrow} \ldots \stackrel{e_{n-1}}{\rightarrow} S_n \stackrel{e_n}{\rightarrow} S$ or $C = I(G) \stackrel{e_0}{\rightarrow} S_1 \stackrel{e_1}{\rightarrow} S_2 \ldots$ where G is a goal, e_i events, S_i states and S a terminal state.

The authors propose two abstractions — traces and behaviours. A *trace* is the sequence of labels (events) of a computation and a behaviour is the sequence of communications of a trace. Thus the trace of a computation $C = I(G) \stackrel{e_0}{\rightarrow} S_1 \stackrel{e_1}{\rightarrow} S_2 \stackrel{e_2}{\rightarrow} \ldots$ is the sequence $Tr(C) = e_0, e_1, e_2, \ldots$. The set of all traces of a goal G and a program P is denoted by $TR_P[G]$. A *behaviour* is a series of communications, where the function *bhv* transforming

events to relations is defined as: bhv(⟨r,c,comm⟩) = comm. Thus the behaviour of a computation C with the trace $e_0, e_1, e_2, \ldots$ is the series $b_0, b_1, b_2, \ldots$ such that $b_i = bhv(e_i)$ for all i.

The authors also present an alternative approach which requires the use of time-stamping members of a Labeled Herbrand Universe to represent the states of terms during computations. Members of this universe are labeled constants and labeled function symbols. A labeled constant is a triple $\langle c,t,io \rangle$ where c is the constant, t its time and io its input/output label. A labeled function symbol is a triple $\langle f,t,io \rangle$ where f is the function symbol, t its time and io its input/output label. Variables, predicates, connectives, quantifiers and punctuation symbols have the usual form. Time labels are integers and the io elements indicate whether the instantiation was performed in an input or output transition. States are now triples $\langle R, \theta, t \rangle$ where R is a resolvent, θ a substitution over the labeled Herbrand Universe and t an integer representing time. Communications (as defined previously) are ignored, and transition rules determine state transformations denoted by $S_1 \rightarrow S_2$. A labeled computation is either a finite sequence $C = I(G) \rightarrow S_1 \rightarrow \ldots \rightarrow S_n \rightarrow S$ or $C = I(G) \rightarrow S_1 \rightarrow S_2 \ldots$ where G is a goal, S_i states and S a terminal state.

The model of concurrency used by Lichtenstein et al. is that of *interleaving*, which enables them to propose a hierarchy of abstractions {traces, behaviours, labeled goals, goals} where traces captures more details of a computation than traces, traces than labeled goals and labeled goals than goals. Given a trace and a goal it is possible to uniquely reconstruct the computation from which the trace was abstracted, but interleaving semantics means that behaviours ignore some details of a computation. The example quoted is the program

p(X) ← q(X) , r(X).
q(a).
r(a).

with respect to the goal ← p(X). Interleaving means that q(X) may be reduced before r(X) or vice-versa, a fact recorded in the traces, but not the behaviour of the computation. Labeled goals ignore variable to variable substitutions, so that the goal ← p(X,Y) w.r.t. the program

P(X,Y) ← q(X,Y) , r(X,Y).

```
q(a,a).
r(X,X).
```

gives the behaviour that either X and Y are unified before both are bound to a, or that X is bound to a and then Y is bound to a. The set of labeled goals ignores this, since an argument is only labeled when it is bound to a non-variable, and the set of labeled goals for both the query $\leftarrow p(X,Y)$ and the query $\leftarrow p(X,X)$ are identical.

The authors describe the relation between goals and labeled goals in terms of abstraction and concretisation functions of abstract interpretation (α and γ). Goals are an abstraction of labeled goals, or alternatively labeled goals are a concretisation of goals. Informally $\alpha(\{lterm\})$ removes the labels of *lterm* to give a term, and $\gamma(\{term\})$ is the set of all possible labeled terms that can be constructed from *term*.

An interpreter for traces is achieved by associating a unique identifier with each clause and interpreting the goal sequentially, recording at each stage which goal was removed from the resolvent and which clause was used to reduce it. For behaviours the goal and clause numbers are ignored and only substitutions recorded. For labelled interpretation each FCP program is transformed into a labeled program where each constant and function symbol is transformed into the triple containing the symbol and new time ans I/O variables. The interpreter executes in an interleaving manner using time-stamped read-only unification.

Appendix D

Algebraic specification techniques

A considerable amount of research has been undertaken by workers in the field of algebraic specification techniques for concurrent systems. Chapter 2 briefly reviews the theories of CCS [92], CSP [58], LOTOS [66] and CIRCAL [91]. In this appendix we look in more detail at LOTOS and describe Brinksma's proposal for LOTCAL [14], a small set of operators that suffices for the formal interpretation of LOTOS. We also review Milne's work on CIRCAL [91].

D.1 LOTOS

The most successful work has centered around the development of LOTOS which has been adopted by the International Standards Organisation (ISO) as one of the formal description techniques for the specification of the Open Systems Interconnection (OSI) computer network architecture [66]. LOTOS was developed during the ESPRIT Software Technology project ST410 "SEDOS" (Software Environment for the design of Open distributed Systems) [35, 138]. LOTOS has its roots in both CCS and CSP but also developed in response to the needs of the user community. For example, although strongly influenced by CCS regarding the specification of communication, LOTOS permits the description of multiway communication and also incorporates the notion of constraints in communication. These two additions permit the use of a constraint oriented style of specification, allowing improvements in the quality, conciseness and ease of verification of specifica-

tions. Problems associated with the implemention of systems specified using LOTOS in a constraint-oriented style are discussed by van Eijk [134], Gilbert [50] and Leon [82].

Much effort has been put into the development of tools to support the task of *specifying* systems in LOTOS — a review of these is to be found in [88]. The Hippo LOTOS simulator [133, 129] is the most widely available LOTOS tool and was based on work reported by van Eijk in his PhD thesis [132]. It is an interactive tool that symbolically executes LOTOS specifications. The simulator builds a communication tree from a given specification and permits the user to interactively step through the specification, selecting from a menu of possible events in each stage. Deadlock properties of the specification can be investigated, test sequences analysed and dynamic behaviour explored. Hippo incorporates both the abstract data type part of LOTOS (ACT-ONE) and the dynamic part.

The task of verification is closely connected with theories of conformance and test derivation and these have been discussed by Brinksma with respect to LOTOS in [16, 17, 12, 13]. Such research has encouraged the construction of tools to support the activity of verification [15].

Communication in LOTOS

Basic LOTOS is a simplified version of the language employing a finite alphabet of observable actions which can occur at named gates. The symbol **i** denote the unobservable action (the τ in CCS). A basic LOTOS specification comprises a behaviour expression which is built by applying an operator to other behaviour expressions or processes. The basic operators of LOTOS are sequencing, choice, synchronisation, hiding and disruption. The basic process of LOTOS is **stop** which indicates inaction. More complex processes can be defined by associating a symbolic name and a list of gate names (interaction points) with a behaviour expression; process definitions can be nested. E.g. the sequencing operator in LOTOS is ";" which permits the definition of the following process:

```
process buffer1 [a,b,c] :=
      a ; b ; stop
endproc
```

The operational semantics of LOTOS is given by labelled transitions: given a behaviour expression B, a labelled transition is of the form $B \text{ —x→ } B'$ indicating that B performs the action x and transforms into B'. The use of a labelled transition on its own in a definition is an *axiom*, while a *rule* is expressed by:

$$\frac{B \text{ —x→ } B'}{B_1 \text{ —x→ } B_1'} \qquad (\textit{Condition})$$

where $B \text{ —x→ } B'$ is a precondtion, B_1 composes B with other behaviour expressions, and $B_1 \text{ —x→ } B_1'$ if *Condition* is true.

In the following we denote:

G	the set of user-definable gates
$g, g_1, \ldots, g_n$	range over G
i	the unobservable action
Act	the set G ∪ {i} of user-definable actions
μ	range over Act

Additionally LOTOS describes the occurrence of successful termination (thus permitting the enabling of a subsequent process) by the special action δ which is *not* user definable. Thus we denote:

δ	the successful termination action
G+	the set G ∪ {δ} of observable actions
g+	range over G+
Act+	the set Act ∪ {δ} of actions
μ+	range over Act+

The δ action is equivalent to the use of `nil` to indicate end of transmission in SILCS.

The semantics of the *action prefix* behaviour expression is defined by the axiom:

μ; $B \text{ —}\mu\text{→ } B$

and that of *choice* by the axioms

$$\frac{B_1 \xrightarrow{\mu^+} B_1'}{B_1 \,[]\, B_2 \xrightarrow{\mu^+} B_1'}$$

and

$$\frac{B_2 \xrightarrow{\mu^+} B_2'}{B_1 \,[]\, B_2 \xrightarrow{\mu^+} B_2'}$$

Parallelism is defined in terms of interleaving for independent actions and simultaneity for dependent (concurrent) actions. If S is the set of gates $[g_1,\ldots,g_n]$, then:

$$\frac{B_1 \xrightarrow{\mu} B_1'}{B_1 \,|S|\, B_2 \xrightarrow{\mu} B_1' \,|S|\, B_2} \qquad (\mu \notin S)$$

$$\frac{B_2 \xrightarrow{\mu} B_2'}{B_1 \,|S|\, B_2 \xrightarrow{\mu} B_1 \,|S|\, B_2'} \qquad (\mu \notin S)$$

$$\frac{B_1 \xrightarrow{g^+} B_1' \text{ and } B_2 \xrightarrow{g^+} B_2'}{B_1 \,|S|\, B_2 \xrightarrow{g^+} B_1' \,|S|\, B_2'} \qquad (g^+ \in S \cup \{\delta\})$$

Note that if $(B_1 \xrightarrow{g^+_1} B_1')$ and $(B_2 \xrightarrow{g^+_2} B_2')$ and $(g^+_1 \neq g^+_2)$ and $(g^+_1 \in S)$ and $(g^+_2 \in S)$ then neither process can proceed and the behaviour expression is deadlocked, i.e. it is equivalent to **stop**.

LOTOS has the following syntactic variants of the parallel operator. *Interleaving*, "|||" is equivalent to "|[]|" (the set S is empty), while the *general* parallel operator "||" indicates that S is the set of all gates common to the two composed behaviour expressions which are thus forced to proceed synchronously.

The general parallel operator in LOTOS has to be used carefully. For example, the behaviour expression $((a;b;stop)||(a;b;stop))$ is a shorthand for $((a;b;stop)|[a]|(a;b;stop))$. The rules for parallel composition mean that the expression is equivalent to $a;((b;stop)|[a]|(b;stop))$, not $a;((b;stop)||(b;stop))$, since the latter form is a short hand for $a;((b;stop)|[b]|(b;stop))$.

Hiding causes those actions which are hidden to introduce an unobservable action into the environment in place of the hidden action:

$$\frac{B \xrightarrow{\mu^+} B'}{\textbf{hide } g_1,\ldots,g_n \textbf{ in } B \xrightarrow{\mu^+} B'} \qquad (\mu^+ \notin g_1,\ldots,g_n)$$

$$\frac{B \xrightarrow{\mu^+} B'}{\textbf{hide } g_1,\ldots,g_n \textbf{ in } B \xrightarrow{i} B'} \qquad (\mu^+ \in g_1,\ldots,g_n)$$

An example of the use of hiding is in the composition of buffer processes:

```
process buffer2 [ins,outs] :=
    hide mids in
    buffer1 |[mids]| buffer1
endproc
```

Disabling is a construct which permits the specification of one process being disrupted by another and is required for the description of disconnection or abortion:

$$\frac{B_1 \xrightarrow{\mu} B_1'}{B_1 \,[>\, B_2 \xrightarrow{\mu} B_1' \,[>\, B_2}$$

$$\frac{B_1 \xrightarrow{\delta} B_1'}{B_1 \,[>\, B_2 \xrightarrow{\delta} B_1'}$$

$$\frac{B_2 \xrightarrow{\mu^+} B_2'}{B_1 \,[>\, B_2 \xrightarrow{\mu^+} B_2'}$$

A note on the associativity of the parallel operator

The parallel operator is associative in the case that the set of gate does not vary during its application, i.e. $B_1|S|B_2|S|\ldots|S|B_n$. However, Brinksma [14, p66] remarks that the parallel operator is *not* in general associative since $B_1|S_1|(B_2|S_2|B_3)$ is not equivalent

to $(B_1|S_1|)B_2|S_2|B_3$ if $S_1 \neq S_2$ and its associativity is ill-defined if $S_1 \cap S_2 \neq \emptyset$ and $(S_1 \cup S_2) - (S_1 \cap S_2) \neq \emptyset$. The problem of associativity of parallel operators for algebraic languages has been described by Milner in [94].

The repeated application of the general parallel operator "||" to a list of processes implies that the synchronisation type varies during the application. Thus the behaviour expression $b[i, m_1]||b[m_1, m_2]||b[m_2, o]$ is equivalent to $b[i, m_1]||[m_1]|(b[m_1, m_2]||[m_2]|b[m_2, o])$. In his thesis Brinksma claims that this form of parallel composition still has the associative property 'by happy coincidence' [14, p48], but this is refuted by the statement on page 66 of the same work[1] where he states that 'the associative law for $|[a]|$ does *not* generalise to the case $B_1|[A_1]|(B_2|[A_2]|B_3) = (B_1|[A_1]|)B_2|[A_2]|B_3$ when $A_1 \neq A_2$'.

Exchange of data on synchronisation

Full LOTOS permits the exchange of data on synchronisation. Gates are postfixed by value offers and optionally constraints. Offers are expressions of values in a type defined in the ADT part of full LOTOS. An observable action prefix expression is of the form

action denotation ; *behaviour expression*

where the action denotation is of the general form g α_1 α_2 ... α_n where g is a gate name and the α's represent a finite list of attributes (value offers). Attributes can be value declarations or variable declarations.

Value offers are of the form $!E$ where E is a value expression, e.g. $!(2 + 7)$ or $!TRUE$. *Variable declarations* are of the form $?x{:}t$ where x is the name of a variable and t is its sort identifier (indicating the domain of values over which x ranges), e.g. $?x{:}integer$, $?y{:}boolean$. Selection predicates can optionally be associated with variable declarations. These have the effect of restricting the range of the sort associated with the declaration, e.g. $?x{:}integer\ [x \leq 5]$. Synchronisation occurs in full LOTOS only when gate names match and their associated attributes are compatible according to the rules below:

Value matching is the basic form of communication in LOTOS, where the value and type of the expressions involved must agree. For example given the sorts boolean, string and

[1] In his thesis Brinksma uses the notation $|_A$ for $|[A]|$.

integer with the usual associated operations then the expressions

$$g!(2+7)!TRUE!yes$$

and

$$g!(2+7)!TRUE!yes$$

match. The transition rule for value matching is

g !E; B —g<value(E)>→ B

Value passing from 'producer' to 'consumer' is best regarded as 'offers within a range'. The 'consumer(s)' offer a range of values within a type or *sort*, the range being optionally restricted by selection predicates associated with the offer. The 'producer(s)' agree on one value from this set. The sort of the offers made by the consumer(s) must agree with the sort of the offer made by the producer(s). For example a consumer may offer a range of values from the sort integer: $Min \leq Offer \leq Max$ to synchronise with one integer value offered by a producer, as in

$$(g!(2+3);\ldots)|[g]|(g?x{:}\,integer\ [5<x<10];\ldots)$$

No synchronisation occurs in this case.

Value generation occurs when two processes both offer to synchronise on a range of data values from one sort, and as with value passing, one value is selected. In the behaviour expression $(g?x{:}\,integer;\ldots)|[g]|(g?y{:}\,integer;\ldots)$ the two processes involved synchronise with $x = y = z, z \in INT$. The ranges of the sort may be constrained by selection predicates. For example in the behaviour expression

$$(g?x{:}\,integer\ [x<10];\ldots)|[g]|(g?y{:}\,integer\ [y>5];\ldots)$$

the two processes involved synchronise with $x = y = z, 5 < z < 10$. LOTOS permits multi-way communication, with the added complexity of the propagation of constraints until all offers have been made and successfully negotiated. The implementation of such a scheme is problematic and goes against the idea of a completely distributable system and requires one *active* global observer to determine final values in a negotiation.

The mechanisms involved in the latter two types of communication are thus akin to the evaluation of constraints in concurrent constraint logic programming, except that only

one final value is generated, rather than a collection of constraints being propagated. This reflects the difference in the semantics of the choice operators in each formalism: the choice operator in LOTOS is 'committed' (only one branch is ever chosen), whilst in concurrent constraint logic programming languages[2] choice operator is all-solutions (logical choice).

However, there are interesting concepts in the LOTOS framework which may be applicable in a logic specification scheme. Such a scheme which incorporates more than term unification may be of use in specifying communicating systems, but the basic formalism should be applicable to unification over terms. In LOTOS the difference between 'producer' and 'consumer' is that the latter can leave a successful communication with one of a range of values, whilst the former can only wait until expression matching, and continues with the same offer.

D.2 LOTCAL

Brinksma [14] has identified a small set of operators called LOTCAL which suffices for the formal interpretation of LOTOS and has proposed a design for extended LOTOS based on these operators. The work is an attempt to rationalise the design of LOTOS and proposals are made for a number of modifications to the original model adopted as the international standard by ISO. The potential language enhancements proposed include the use of list operators and improvements in the definitions of functionality, successful termination and parallel composition. Brinksma outlines the potential for the addition of timing in the language and argues for the addition of indexed synchronisation labels. A form of modularisation is also proposed and a more flexible interface between the data-type and behaviour-oriented part of the language in the form of explicitly defined data environments along with a more uniform syntax for both.

The problems connected with multi-way synchronisation and associativity of the parallel operator have led Brinksma to propose a new semantics for this operator in his calculus LOTCAL [14, Chapter 4], in line with those outlined by Milner in [94]. These are expressed in the following rules for "|"

[2]Except for cc($\rightarrow$) which has a committed choice operator

$$\frac{B_1 \xrightarrow{\mu} B_1'}{B_1 \mid B_2 \xrightarrow{\mu} B_1' \mid B_2}$$

$$\frac{B_2 \xrightarrow{\mu} B_2'}{B_1 \mid B_2 \xrightarrow{\mu} B_1 \mid B_2'}$$

$$\frac{B_1 \xrightarrow{\mu_1} B_1' \text{ and } B_2 \xrightarrow{\mu_2} B_2'}{B_1 \mid B_2 \xrightarrow{\mu_1 \Delta \mu_1} B_1' \mid B_2'}$$

where $\mu_1 \Delta \mu_2 =_{df} (\mu_1 \cup \mu_2) - (\mu_1 \cap \mu_2)$ (symmetric difference).

This new definition of the parallel operator is coupled with the introduction of a *restriction* operator "\" which removes all behaviour from a process that starts with an action containing an event that is restricted over:

$$\frac{B \xrightarrow{\mu} B'}{B\backslash A \xrightarrow{\mu} B'} \qquad (\mu \cap A = \emptyset)$$

Brinksma's thesis [14, Chapter 4] contains proofs of the equivalence of the LOTOS parallel operator with a combination of the parallel and restriction operators of LOTCAL.

D.3 CIRCAL

CIRCAL, proposed by Milne [91], permits the modeling of asynchronous and simultaneous behaviour using an acceptance semantics. It was designed with for the description and analysis of concurrent systems, either in hardware or software. The parallel operator "•" is similar in behaviour to that of "||" in CSP but permits several simultaneous 'particulate' actions, similar to the idea presented by Milner in [94] and permits the modeling of concurrency without recourse to interleaving. Milne has claimed with some justification that he was the first to propose such behaviour in the algebraic context. CIRCAL does not require that all components in a system are totally synchronous, but permits the modeling of a spectrum between actions occurring one at a time and the complete simultaneity of

all component interaction.

We use Milne's notation from [91] to illustrate the semantics of those operators of CIRCAL which are relevant to a comparison with SILCS. An action in CIRCAL can consists either of one event or a number of simultaneous events. Each event is represented by a label and an action is represented by a non-empty set of labels written using ( ) rather than { }. For example, a typical label set is $(\alpha\beta\gamma)$ representing the simultaneous occurrence of the actions α, β and γ. A singleton label set can be written without parenthesis. Any sort L is a subset of Λ the set of all labels. The set PROG_L is the set of all terms of sort L. The semantics of CIRCAL is based on the idea of a system responding to a stimulus, and is described in terms of labelled transition systems. For terms $T, T' \in \mathrm{PROG}_L$ and the label-set $m \subseteq L$ then $T \text{—}m\rightarrow T'$ denotes the the term T accepting m and evolving to T'. If T rejects m then we write $T \text{—}m\rightarrow *$ where $*$ is a special symbol.

Guarding, as in $(\alpha\beta)P$ where $(\alpha\beta) \subseteq L$ and $P \in \mathrm{PROG}_L$, contributes sequentiality to CIRCAL. The transitions describing guarding are:

$mP \text{—}m\rightarrow P$ where term mP accepts the stimulus given by label-set m and evolves to P.

$mP \text{—}n\rightarrow *$ where $m \neq n$.

Choice[3] is indicated by $P + Q$. The environment, other interacting terms, resolve the choice as to whether an action in P or Q occurs next. The rules are:

[3]CIRCAL possesses a nondeterministic choice operator "$\oplus$" which is not relevant to this discussion.

$$\frac{P \xrightarrow{m} P'}{P + Q \xrightarrow{m} P'}$$

$$\frac{Q \xrightarrow{m} Q'}{P + Q \xrightarrow{m} Q'}$$

$$\frac{P \xrightarrow{m} * \text{ and } Q \xrightarrow{m} *}{P + Q \xrightarrow{m} *}$$

Termination is represented by Δ and is described by

$$\Delta \xrightarrow{m} *$$

The *definition* operator "$\Leftarrow$" is used to name terms as in $P \Leftarrow Q$.

A derived operator is *summation* over the choice operator for label-sets $m_1, \ldots, m_n$, defined by

$$\sum_{i=1,n} m_i P_i \Leftarrow m_1 P_1 + m_2 P_2 + \ldots + m_n P_n$$

The guarding and choice operators are used to define the composition operator "$\bullet$"

Definition. For $A \Leftarrow \sum \lambda_i A_i$ of sort L and $B \Leftarrow \sum \mu_j B_j$ of sort M, then

$$A \bullet B \Leftarrow \sum_{\lambda_i \cap M = \emptyset} \lambda_i [A_i \bullet B] + \sum_{\mu_j \cap L = \emptyset} \mu_j [A \bullet B_j] + \sum_{(\lambda_i \cap M) = (\mu_j \cap L)} (\lambda_i \cup \mu_j) [A_i \bullet B_j]$$

If

$$\forall \lambda_i, \mu_j \lambda_i \cap M \neq \mu_j \cap L, \lambda_i \cap M \neq \emptyset, \mu_j \cap L \neq \emptyset$$

then

$$A \bullet B \Leftarrow \Delta$$

$A \bullet B$ has sort $L \cap M$

The first clause contributes to $A \bullet B$ those guards belonging to A whose labels do not intersect with M, the sort of B. Such guards appear independently of B. Similarly the second clause describes the independent appearance of guards from B.

The third clause contributes to $A \bullet B$ those guards formed by a synchronisation of guards from A and B. This clause can be replaced by the following clause:

$$\sum_{(\lambda_i \cap M)=(\mu_j \cap L)=\emptyset} (\lambda_i \cup \mu_j)[A_i \bullet B_j] + \sum_{(\lambda_i \cap M)=(\mu_j \cap L)\neq\emptyset} (\lambda_i \cup \mu_j)[A_i \bullet B_j]$$

The first clause describes the independent but simultaneous events represented by λ_i and μ_j while the second clause describes those guards which result from a synchronisation of at least some of the labels comprising a guard in A with some of the labels of a guard in B. The two label sets $(\lambda_i \cap M)$ and $(\mu_j \cap L)$ are identical and A and B intersect.

If there is no $\lambda_i \in A$ and $\mu_j \in B$ such that $\lambda_i \cap M = \emptyset$, $\mu_j \cap L = \emptyset$ or $(\lambda_i \cap M) = (\mu_j \cap L)$, then $A \bullet B \Leftarrow \Delta$. If both A and B are not themselves Δ, this corresponds to *deadlock*. This may be illustrated by the two terms P and Q with sort $\{\alpha, \beta\}$. If $P \Leftarrow \alpha\beta P'$ and $Q \Leftarrow \beta\alpha Q'$, then $\lambda_i = \{\alpha\}$, $\mu_j = \{\beta\}$, $L = M = \{\alpha, \beta\}$, $\lambda_i \cap M = \{\alpha\}$, $\mu_j \cap L = \{\beta\}$. Hence $(\lambda_i \cap M) \neq (\mu_j \cap L)$ and $P \bullet Q \Leftarrow \Delta$.

The semantics of the dot operator of CIRCAL are similar to those of the simultaneous operator of SILCS in that parallel events can be represented as simultaneous actions, and deadlock (suspension) is effectively expressed by the same mechanism.

Appendix E

Glossary of abbreviations

ADT	Abstract Data Type
cc	Concurrent Constraint Language (Saraswat)
CCS	Calculus of Communicating Systems
CLP	Concurrent Logic Programming
CCPL	Concurrent Constraint Programming Language
CLPL	Concurrent Logic Programming Language
CSP	Communicating Sequential Processes
(F)CP	(Flat) Concurrent Prolog
(F)GHC	(Flat) Guarded Horn Clauses
FOPL	First Order Predicate Logic
HCL	Horn Clause Logic
ISO	International Standards Organisation
LOTOS	Language of Temporal Ordering Specification
LP	Logic Programming
LPL	Logic Programming Language
$\mathcal{ML}$	Meta-language
$\mathcal{OL}$	Object-language
OSI	Open Systems Interconnection
PDC	Pure Definite Clauses
PHC	Pure Horn Clauses
RL	Relational language
SILCS	Specification In Logic of Concurrent Systems

Table E.1: Abbreviations

Bibliography

[1] Advanced A. I. Systems Inc., Mountain View, California. *Advanced A. I. Systems' Prolog Reference Manual Version M-1.15*, 1987.

[2] J. F. Allen. An Interval-Based Representation of Temporal Knowledge. Technical report, Department of Computer Science, University of Rochester, Rochester, NY, 1982.

[3] J. F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[4] Arity Corporation, Concord, Massachussetts. *The Arity Prolog Programming language*, 1986.

[5] R. Bahgat and S. Gregory. Pandora: Non-deterministic Parallel Logic Programming. In G. Levi and M. Martelli, editors, *Proceedings 6th International Conference on Logic programming*, pages 471–486, Lisbon, Portugal, June 1989. MIT Press.

[6] L. Beckman. Towards a formal semantics for concurrent logic programming languages. In E. Shapiro, editor, *Third International Conference on Logic Programming, 1986*, pages 335–349, London, UK, 1986. Springer-Verlag.

[7] F. Belina and D. Hogrefe. Introduction to SDL. In *FORTE88 — Formal Description Techniques 1988*, Stirling, Scotland, September 1988. (Invited paper).

[8] C. Berge. *The Theory of Graphs and its Applications.* Methuen, London, 1962.

[9] G. Birkhoff. *Lattice Theory*, volume 25 of *Colloqium Publications.* American Mathematical Society, Providence, Rhode Island, 1940.

[10] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[11] E. Brinksma. An Introduction to LOTOS. Technical report, University of Twente, Enschede, Netherlands, 1987.

[12] E. Brinksma. On the existence of Canonical Testers. Technical Report Memorandum INF-87-5, Department of Informatics, University of Twente, Enschede, Netherlands, January 1987.

[13] E. Brinksma. A Theory for the Derivation of Tests. In *IFIP Protocol Specification, Testing and Verification VIII*, Atlantic City, June 7-10, 1988. IFIP.

[14] E. Brinksma. *On the Design of Extended LOTOS; a Specification Language for Open Distributed Systems.* PhD thesis, Department of Informatics, University of Twente, Enschede, Netherlands, 1988.

[15] E. Brinksma. LOTOS Verification Aspects — Report of the SEDOS C2 Task. In P. van Eijk, C. Vissers, and M.Diaz, editors, *The Formal Description Technique LOTOS*, pages 229–234. North-Holland, 1989.

[16] E. Brinksma and G. Scollo. Formal notions of Implementation and Conformance in LOTOS. Technical Report Memorandum INF-86-13, Department of Informatics, University of Twente, Enschede, Netherlands, December 1986.

[17] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS specifications, their implementations and their tests. In *Proc. 6th Workshop on Protocol Testing and Verification*, pages 349–360, Montreal, June 1986. Noth-Holland.

[18] M. Carlsson and J. Widen. *SICStus Prolog User's Manual Version 0.6.* Swedish Institute of Computer Science, Kista, Sweden, 1988.

[19] B. Carré. *Graphs and networks.* Clarendon Press, Oxford, 1979.

[20] C. L. Chang and R. C. T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, 1973.

[21] A. Church. The calculi of lambda conversion. *Annals of Mathematics Studies*, 6, 1941.

[22] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

[23] K. L. Clark. Predicate Logic as a Computational Formalism. Technical Report Research Monograph 79/59 TOC, Department of Computing, Imperial College, December 1979.

[24] K. L. Clark. The Synthesis and Verification of Logic Programs. Technical Report DOC 81/36, Department of Computing, Imperial College, September 1981.

[25] K. L. Clark. Logic Programming Schemes. In *Proc FGCS88*, Tokyo, Japan, November 1988. ICOT.

[26] K. L. Clark and S. Gregory. A relational language for parallel programming. In *Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architectures*, pages 171–178, Portsmouth, NH, 1981. ACM.

[27] K. L. Clark and S. Gregory. PARLOG: A parallel logic programming language. Technical Report IC Research report DOC 83/5, Department of Computing, Imperial College, London, UK, 1983.

[28] K. L. Clark and S. Gregory. PARLOG: Parallel programming in Logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, 1986.

[29] K. L. Clark and S. Gregory. Parlog and Prolog United. In J.-L. Lassez, editor, *Proceedings of the 4th International Conference on Logic Programming*, pages 927–961. MIT, 1987.

[30] K. L. Clark, F. G. McCabe, and S. Gregory. IC-Prolog language features. In K. L. Clark and S. Å. Tärnlund, editors, *Logic Programming*. Academic Press, 1982.

[31] K. L. Clark and S. Å. Tärnlund. A First Order Theory of Data and Programs. In B. Gilchrist, editor, *IFIP Information Processing 77*. North-Holland, 1977.

[32] A. Colmerauer. *Prolog II: Manuel de reference et modele theoretique*. University of Aix-Marseille, 1982.

[33] A. Colmerauer. Introduction to Prolog III. Technical Report Project No. 1106, Group Intelligence Artificielle, Faculte des Sciences de Luminy, Marseilles, France, 1987.

[34] J. Crammond. *Implementation of Committed Choice Logic Languages on Shared Memory Multiprocessors.* PhD thesis, Department of Computer Science, Heriot-Watt University, Edinburgh, UK, 1988.

[35] M. Diaz, C. A. Vissers, and J.-P. Ansart. SEDOS Software Environment for the Design of Open distributed Systems. In P. van Eijk, C. Vissers, and M.Diaz, editors, *The Formal Description Technique LOTOS*, pages 3–14. North-Holland, 1989.

[36] E. W. Dijkstra. Guarded Commands, non-determinancy and formal derivation of programs. *Communications ACM*, 18(8):453–457, Aug 1975.

[37] H. Ehrig, W. Frey, and H. Hansen. ACT ONE: An algebraic specification language with two levels of semantics. Technical Report Bericht Nr 83-03, Techische Universitaet, Berlin, 1983.

[38] M. R. Ellis. A Relational Language into ECCS. Master's thesis, Department of Computing, Imperial College, London. UK, September 1986.

[39] U. Engberg and M. Nielsen. A Calculus of Communicating Systems with Label Passing. Technical report, Mathematic Institute, Aarhus University, Denmark, 1986.

[40] I. Foster. *Parlog as a Systems Programming Language.* PhD thesis, Department of Computing, Imperial College, 1988. (Also Research Report PAR 88/5).

[41] I. Foster. *Systems Programming in Parallel Logic Languages.* Prentice Hall, 1990.

[42] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[43] D. Gabbay. Temporal Logic and Computer Science. Technical report, Department of Computing, Imperial College, London, UK, May 1985.

[44] D. Gabbay. Executable Temporal Logic for Interactive Systems. Technical report, Department of Computing, Imperial College, London, UK, March 1987.

[45] H. Gericke. *Lattice Theory.* George Harrap & Co, 1966.

[46] R. Gerth, M. Codish, Y. Lichtenstein, and E. Shapiro. A Fully Abstract Denatational Semantics for Flat Concurrent Prolog. Technical report, Weizmann Institute of Science, Rehovot Israel, March 1988.

[47] G. Gierz, K. H. Hofman, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott. *A Compendium of Continuous Lattices.* Springer-Verlag, 1980.

[48] D. R. Gilbert. Executable LOTOS: Using PARLOG to implement an FDT. In *Proceedings of IFIP Protocol Specification, Testing and Verification: VII, Zurich, Switzerland, 5-8 May 1987*, Amsterdam, Netherlands, 1987. Elsevier Science, North-Holland.

[49] D. R. Gilbert. Specification and implementation of concurrent systems using PARLOG. In *Workshop on Specification and Verification of Concurrent Systems*, Stirling UK, July 1988. BCS-FACS.

[50] D. R. Gilbert. A LOTOS to PARLOG translator. In K. J. Turner, editor, *FORTE88 — Formal Description Techniques 1988*, pages 31–44. North-Holland, 1989.

[51] S. Gregory. *Design, Application and Implementation of a Parallel Pogic Programming Language.* PhD thesis, Department of Computing, Imperial College, London, UK, 1985.

[52] S. Gregory. *Parallel Logic Programming in PARLOG: The Language and its Implementation.* Addison-Wesley, London, UK, 1987.

[53] S. Haridi and P. Brand. Andorra Prolog; an integration of Prolog and committed choice languages. In *FGCS 1988*, pages 745–754, Tokyo, Japan, 1988. ICOT.

[54] S. Haridi and S. Janson. Kernel Andorra Prolog and its Computational Model. Technical Report SICS/R-90/R9002, Swedish Insitute of Computer Science, January 1990.

[55] M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *Journal of the ACM*, 32(1):137–161, January 1985.

[56] P. M. Hill and J. W. Lloyd. Analysis of Meta-programs. Technical Report CS-88-08, Department of Computer Science, University of Bristol, Bristol, UK, June 1988.

[57] M. Hirsch, W. Silverman, and E. Shapiro. Layers of protection and control in the Logix system. Technical Report Technical Report CS86-19, Department of Computer Science, Weizmann Institute of Science, Rehovot, Israel, 1986.

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

[59] C. J. Hogger. Program Synthesis in Predicate Logic. In *Proc Conference on Artificial Intelligence AISB/GI-78.* University of Hamburg, July 18-20 1978.

[60] C. J. Hogger. Derivation of Logic Programs. *Journal of the Association for Computing Machinery*, 28(2):372–392, 1981.

[61] C. J. Hogger. Concurrent Logic Programming. In K. L. Clark and S. Å. Tärnlund, editors, *Logic Programming*, pages 199–211. Academic Press, London, 1982.

[62] C. J. Hogger. *Introduction to Logic Programming.* Academic Press, 1984.

[63] M. Huntbach. Algorithmic PARLOG Debugging. In S. Haridi, editor, *Proceedings 1987 Symposium on Logic Programming*, pages 288–297, Washington, DC, USA, September 1987. IEEE.

[64] C. Hussey. Interpreting PARLOG Programs as CCS agents. Master's thesis, Department of Computing, Imperial College, London, September 1987.

[65] Inmos Ltd. *occam Programming Manual.* Prentice-Hall International, 1984.

[66] ISO. *ISO IS 8807 Information Processing Systems, Open Systems Interconnection, LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, 1989.

[67] ISO. *ISO IS 9074 Information Processing Systems, Open Systems Interconnection, ESTELLE, A Formal Description Technique Based on an Extended State Transition Model.* ISO, 1989.

[68] J. Jaffar, S. Michaylov, P. J. Stuckey, and P. H. C. Yap. The CLP(R) Language and System. Technical report, IBM, Yorktown Heights, NY, USA, April 19, 1988.

[69] G. Kahn and D. B. MacQueen. Coroutines and Networks of Parallel Processes. In B. Gilchrist, editor, *Information Processing 77*, pages 993–998. IFIP, North Holland, 1977.

[70] R. M. Keller. Formal verification of parallel programs. *Communications of the AGCM*, 19:371–384, 1976.

[71] Y. Kimura and T. Chikayama. An abstract KL1 machine and its instruction set. In *Proceedings IEEE Symposium on Logic Programming*, pages 468–477. IEEE, September 1987.

[72] R. A. Kowalski. Predicate Logic as a programming language. In *Proceedings of the IFIP Congress 1974*, pages 569–574. IEEE, 1974.

[73] R. A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.

[74] R. A. Kowalski. *Logic for problem solving.* North Holland, 1979.

[75] R. A. Kowalski. The relation between logic programming and logic specification. *Phil. Trans. R. Soc. Lond. A I*, 312:345–361, 1984.

[76] R. A. Kowalski. The early development of Logic Programming. Technical report, Department of Computing, Imperial College, London, UK, November 1986.

[77] R. A. Kowalski and M. Sergot. A Logic-based Calculus of Events. *New Generation Computing*, 4:67–95, 1986.

[78] C. Lassez. Constraint Logic Programming. *BYTE*, pages 171–176, August 1987.

[79] J.-L. Lassez. Parametric queries, linear constraints and variable elimination. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1990.

[80] J.-L. Lassez, M. J. Maher, and K. Mariott. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufman, Los Altos, California, 1987.

[81] J.-L. Lassez and K. McAloon. A Constraint Sequent Calculus. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1990.

[82] G. Leon, C. Delgado, G. Gonzaleza, and M. Ruz. ASDE: Design of a LOTOS Transformational Environment for LOTOS. In S. T. Vuong, editor, *FORTE '89, Proceedings of the Second International Conference on FORMAL DESCRIPTION TECHNIQUES for Distributed Systems and Communication Protocols*, pages 643–657, Vancouver, Canada, December 5-8, 1989.

[83] Y. Lichtenstein. Algorithmic Debugging of Flat Concurrent Prolog. Master's thesis, Feinberg Graduate School, Weizmann Institute of Science, Rehovot, Israel, August 1987.

[84] Y. Lichtenstein, M. Codish, and E. Shapiro. Representation and Enumeration of Flat Cconcurrent Prolog Computations. In E. Shapiro, editor, *Concurrent Prolog*, volume 2, pages 197–210. MIT Press, 1987.

[85] J. W. Lloyd. *Foundations of Logic Programming.* Spinger-Verlag, Berlin, second edition, 1987.

[86] M. J. Maher. Logic Semantics for a class of Committed-Choice Programs. In J.-L. Lassez, editor, *Logic Programming, Proceedings of the Fourth International Conference*, volume 2, pages 858–876, Cambridge, Mass, USA, 1987. MIT.

[87] Z. Manna and A. Pnueli. Verification of concurrent programs: the temporal framework. In R. S. Boyer and J. S. Moore, editors, *The Correctness Problem in Computer Science*, pages 215–273. Academic Press, 1981.

[88] A. K. Marshall. Introduction to LOTOS Tools. In P. van Eijk, C. Vissers, and M.Diaz, editors, *The Formal Description Technique LOTOS*, pages 339–350. North-Holland, 1989.

[89] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. *LISP 1.5 Programmers' Manual.* MIT Press, Cambridge, Mass, USA, 1965.

[90] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4, 1969.

[91] G. J. Milne. CIRCAL and the Representation of Communication, Concurrency and Time. *ACM TOPLAS*, 7(2):270–298, April 1985.

[92] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[93] R. Milner. Calculi for Synchrony and Asynchrony. *Journal of Theoretical Computer Science*, 25:267–310, 1983.

[94] R. Milner. Process Constructors and Interpretations. *Proceedings of IFIP 10th International World Computer Congress*, 10:507–514, September 1-5, 1986.

[95] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[96] P. Mishra. Towards a theory of types in Prolog. In *Proc. IEEE International Symposium Logic Programming*, Atlanta City, USA, 1984. IEEE.

[97] A. Mistry. A PARLOG to Flat PARLOG Compiler. Master's thesis, Department of Computing, Imperial College, London, September 1987.

[98] L. Naish. Heterogeneous SLD Resolution. *Journal of Logic Programming*, 1(4), 1984.

[99] L. Naish. *Negation and Control in Prolog*. Springer-Verlag, 1986.

[100] L. Naish. Parallelising Nu-Prolog. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming, Proceedings of the Fifth Conference and Symposium*, volume 2, pages 1546–1564, Cambridge, Mass, USA, 1988. MIT Press.

[101] K. Nakajima, Y. Inamaura, N. Ichioshi, K. Rokusawa, and T. Chikayama. Distributed Implementation of KL1 on the Multi-PSI/V2. In G. Levi and M. Martelli, editors, *Logic Programming, Proceeding of the Sixth International Conference*, pages 436–451. MIT Press, June 1989.

[102] C. A. Petri. Kommunication mit Automaten (English translation). Technical Report RADC-TR-65-377, Applied Data Research, Princetown, NJ, 1966. Vol 1 Suppl 1, Contract AF 30 (602)-3324.

[103] G. A. Ringwood. PARLOG86 and the Dining Logicians. *Communications of the ACM*, 31(1):10–25, January 1988.

[104] G. A. Ringwood. A Comparative Exploration of Concurrent Logic Languages. Technical report, PARLOG Group, Department of Computing, Imperial College, London, UK, January 1989.

[105] J. A. Robinson. A machine-orientated logic based on the resolution principle. *Journal of the ACM*, 12(1):23 - 49, Jan 1965.

[106] J. A. Robinson. Computational Logic: The Unification Computation. *Machine Intelligence*, 6:63–72, 1971.

[107] P. Roussel. *PROLOG: Manuel de reference et d'utilisation.* Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Universite Aix, Marseille, France, 1975.

[108] S. Safra and E. Shapiro. Meta-interpreters for real. In *Information Procesing 86*, pages 271–278. North-Holland, 1986.

[109] V. A. Saraswat. Partial correctness semantics for CP[|,|,&]. In *Fifth FST + TCS Conference, 1985.* Springer-Verlag, December 1985.

[110] V. A. Saraswat. Problems with Concurrent Prolog. Technical Report Technical report 86-100, Carnagie-Mellon University, 1986.

[111] V. A. Saraswat. Merging many streams efficiently: The importance of atomic commitment. In E. Shapiro, editor, *Concurrent Prolog*, volume 1, pages 421–445. MIT Press, 1987.

[112] V. A. Saraswat. The concurrent logic programming language CP: Definition and Operational Semantics. In *POPL 1987*, pages 49–63. ACM, 1987.

[113] V. A. Saraswat. A somewhat logical formulation of CLP synchronisation primitives. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming, Proceedings of the Fifth Conference and Symposium*, volume 2, pages 1298–1314, Cambridge, Mass, USA, 1988. MIT Press.

[114] V. A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Computer Science Department, Carnagie-Mellon University, January 1989.

[115] V. A. Saraswat, D. Weinbaum, K. Kahn, and E. Shapiro. Detecting stable properties of networks in concurrent logic programming languages. In *Proceedings ACM Conference on Distributed Computing*, 1988.

[116] E. Shapiro. A Subset of Concurrent PROLOG and Its Interpreter. Technical Report TR-003, ICOT, Tokyo, 1983.

[117] E. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.

[118] E. Shapiro. A Subset of Concurrent PROLOG and Its Interpreter. In E. Shapiro, editor, *Concurrent Prolog*, volume 1, pages 27–83. MIT Press, 1987.

[119] E. Shapiro. Concurrent Prolog: A Progress report. In E. Shapiro, editor, *Concurrent Prolog*, volume 1, pages 157–187. MIT Press, 1987.

[120] E. Shapiro. The Family of Concurrent Logic Programming Languages. Technical Report CS89-08, Weizmann Institute of Science, Rehovot, Israel, May 1989.

[121] E. Shapiro and E. Yardeni. A Type System for Logic Programs. Technical report, Dept of Computer Science, Weizmann Institute of Science, Rehovot, Israel, 1987.

[122] J. R. Shoenfield. *Mathematical Logic.* Addison-Weseley, 1967.

[123] W. Silverman, M. Hirsch, A. Houri, and E. Shapiro. The logix system user manual, version 1.21. In E. Shapiro, editor, *Concurrent Prolog, Volume 2*, chapter 21, pages 46–77. MIT Press, 1987.

[124] L. Sterling and E. Shapiro. *The Art of Prolog : Advanced Programming Techniques.* MIT Press Series in Logic Programming. MIT Press, Cambridge, Mass USA ; London UK, 1986.

[125] A. Takeuchi. How to solve it in Concurrent Prolog. (Unpublished note), 1983.

[126] A. Takeuchi and K. Furukawa. Parallel Logic Programming Languages. In E. Shapiro, editor, *Concurrent Prolog*, volume 1, pages 188–201. MIT Press, 1987.

[127] S. Taylor, S. Safra, and E. Shapiro. A parallel implementation of Flat Concurrent Prolog. *Journal of Parallel Programming*, 15(3):245–275, 1987.

[128] B. Thompson. A Calculus of Higher Order Communicating Systems. In *POPL 1989*, Austin, Texas, January 1989.

[129] J. Tretmans. HIPPO: A LOTOS Simulator. In P. van Eijk, C. Vissers, and M.Diaz, editors, *The Formal Description Technique LOTOS*, pages 391–396. North-Holland, 1989.

[130] E. D. Tribble, M. S. Miller, K. Kahn, D. G. Bobrow, and C. Abbott. Channels: A Generalization of Streams. In J.-L. Lassez, editor, *Logic Programming, Proceedings of the Fourth International Conference*, volume 2, pages 839–857, Cambridge, Mass, USA, 1987. MIT.

[131] K. Ueda. *Guarded Horn Clauses.* PhD thesis, University of Tokyo, 1986.

[132] P. van Eijk. *Software tools for the specification language LOTOS.* PhD thesis, Department of Informatics, University of Twente, Enschede, Netherlands, January 1988.

[133] P. van Eijk. The Design of a Simulator Tool. In P. van Eijk, C. Vissers, and M.Diaz, editors, *The Formal Description Technique LOTOS*, pages 351–390. North-Holland, 1989.

[134] P. van Eijk. Tools for LOTOS Specification Style Transformation. In S. T. Vuong, editor, *FORTE '89, Proceedings of the Second International Conference on FORMAL DESCRIPTION TECHNIQUES for Distributed Systems and Communication Protocols*, pages 54–62, Vancouver, Canada, December 5-8, 1989.

[135] M. H. van Emden and G. J. de Lucena Filho. Predicate Logic as a Language for Parallel Programming. In K. L. Clark and S. Å. Tärnlund, editors, *Logic Programming*, pages 189–198. Academic Press, London, UK, 1982.

[136] M. H. van Emden and R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23(4):733–742, 1976.

[137] M. H. van Emden and J. W. Lloyd. A logical reconstruction of Prolog II. In S. Å. Tärnlund, editor, *Logic Programming, Second International Logic Programming Conference*, pages 35–40, Uppsala, Sweden, July 2-6, 1984.

[138] C. A. Vissers. LOTOS backgrounds. In P. van Eijk, C. Vissers, and M.Diaz, editors, *The Formal Description Technique LOTOS*, pages 15–22. North-Holland, 1989.

[139] D. Weinbaum and E. Shapiro. Hardware description and simulation using Concurrent Prolog. In *Proceedings CHDL '87*, pages 9–27. Elsevier Science Publishing, 1987.

[140] R. Weyhrauch. Prolegomena to a theory of formal reasoning. Technical Report AIM-315, Computer Science Department, Stanford University, 1978.

[141] R. Yang. *P-Prolog: A Parallel logic Programming Language and its Implementation.* PhD thesis, Keio University, 1986.

[142] J. Zobel. Derivation of Polymorphic Types for Prolog Programs. Technical Report 86/19, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1986.

[143] J. Zobel. Derivation of Polymorphic Types for Prolog Programs. In J.-L. Lassez, editor, *Logic Programming, Proceedings of the Fourth International Conference*, volume 2, pages 817–838, Cambridge, Mass, USA, 1987. MIT.