Yale University
Department of Computer Science

Programming in Distributed Systems Logic

Kevin Tyrone Lynch

Research Report YALEU/DCS/RR-1133
May 1997

Abstract

Programming in Distributed Systems Logic

Kevin Tyrone Lynch

1997

Over the last ten years numerous logic languages have been proposed for use in programming distributed systems. However, these languages generally lack adequate features to describe the *spatial* properties of distributed programs — that is, the properties that describe how the components (i.e., pieces) of a distributed program are organized and distributed over the network of processors. As a consequence, these languages are unsuitable for describing many useful distributed applications.

In this dissertation we describe a new distributed logic programming language called NETLOG (NETwork LOGic programming) that was designed as a tool for specifying and rapidly prototyping a broad range of distributed applications. NETLOG differs from existing logic languages for distributed programming in that it is based on a subset of Distributed Systems Logic (DSL). Moreover, the language is unique in that it supports a *unified* framework in which both the *spatial* and *temporal* properties of distributed computations can be expressed in a concise and elegant fashion. Thus, the language can be used to specify both *where* (i.e., on which processors) to locate the components of a distributed computation and *when* to execute them. Accordingly, NETLOG represents a new approach to specifying and reasoning about distributed logic programs.

After giving an informal introduction to NETLOG, we present the syntax and semantics of first-order DSL. We then investigate the formal operational semantics of NETLOG and provide a Plotkin style semantics for the language using transition systems. An implementation scheme designed for implementing NETLOG on a distributed memory multiprocessor is described, and experimental results are provided based on the simulated performance of several NETLOG programs.

Programming in Distributed Systems Logic

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Kevin Tyrone Lynch

Dissertation Director: Alan Perlis, Paul Hudak
May 1997

Copyright © 1997 by Kevin Tyrone Lynch

All rights reserved.

Aknowledgements

I am deeply indebted to Alan Perlis, my advisor under whom I originally began my dissertation research. Alan's enthusiasm and deep knowledge and understanding of computer science will always be a source of inspiration and admiration for me; his untimely passing was a great loss. I am also deeply grateful to my current advisor, Paul Hudak, not just for his advice, support, encouragement, and patience over the years, but for his willingness to help me when I needed help the most.

I would also like to thank the other members of my dissertation committee — Nicholas Carriero, Zhong Shao, and Young-il Choo — for the constructive comments they provided and helpful suggestions for improving this work. In particular, Nick's insightful comments and attention to detail were most helpful.

None of my work would have been possible without the love, encouragement, and prayers of my parents and other members of my family. My mother, Amilda, supported me in every possible way, and the amusing antics of Ashley, Kimberley, and Justin provided much needed humor that helped me to keep things in perspective. I dedicate this dissertation to my family.

Contents

List of Tables v

List of Figures vii

1 Introduction 1

1.1 Motivation . . . 1

1.2 Distributed Systems Logic (DSL) . . . 4

1.3 Event-Action Model . . . 5

1.4 Formal Semantics . . . 6

1.5 Implementation . . . 7

1.6 Comparison to related work . . . 7

1.6.1 Other concurrent logic programming languages . . . 7

1.6.2 Comparison with NETLOG . . . 9

1.7 Contributions of dissertation . . . 12

1.8 Organization of thesis . . . 13

2 Introduction to programming in NETLOG 15

2.1 Programming Model . . . 15

2.1.1 Constraint based Event Model (CEM) . . . 15

2.2 Syntax of NETLOG . . . 17

2.2.1 Variables . . . 18

2.2.2 Expressions . . . 18

2.2.3 Event Descriptors 19
2.2.4 Actions 19
2.2.5 Constraints 20
2.2.6 Rules and Programs 21
2.3 Additional Constructs 22
2.4 Example: Computing Sum of Factorials 23
2.5 Nondeterminism and Constraints 25
2.6 Summary 34

3 Distributed Systems Logic **35**
3.1 Background 35
3.2 First-Order Distributed Systems Logic (FDSL) 36
3.2.1 Alphabet 37
3.2.2 Syntax 37
3.3 Networks 39
3.4 Semantics 39
3.5 Example 42
3.6 Summary 43

4 Specifying and Reasoning about Distributed Programs **45**
4.1 Specifying Distributed Programs 45
4.2 Reasoning about Distributed Programs 60
4.3 Summary 65

5 Operational Semantics **67**
5.1 Abstract Syntax 67
5.2 Semantics 69
5.2.1 Configurations 69
5.2.2 Notation and Auxiliary Definitions 71
5.2.3 State Transistion System 72

5.2.4 Program Transistion Relation 77
5.3 Example 79
5.4 Properties of the Operational Semantics 80
5.5 Summary 83

6 Further Examples **85**
6.1 Example 1: Snow White and the Seven Dwarfs 85
6.2 Example 2: Bounded Buffer message Communication 88
6.3 Example 3: Airline Reservation System 90
6.4 Example 4: Implementing LINDA in NETLOG 92
6.5 Example 5: Computing primes 94
6.6 Summary 96

7 Implementation **97**
7.1 Compiling NETLOG programs 97
7.1.1 Stage 1: Source program to intermediate form 99
7.1.2 Stage 2: Global analysis and source-to-source transformations 100
7.1.3 Stage 3: Code generation 102
7.2 Abstract Machine Design 107
7.2.1 Abstract Machine Structure and Organization 108
7.2.2 Abstract Machine Operation 109
7.2.3 Abstract Machine Support for Global Time 110
7.2.4 Machine Instructions 112
7.3 Experimental Results 121
7.3.1 Overview of simulator 121
7.3.2 Results 124
7.3.3 Summary 131

8 Conclusions and Future Research **133**
8.1 Summary of research 133

8.2 Directions for future research 134
8.3 Conclusions 135

Bibliography **137**

A **145**

B **155**

List of Tables

7.1 Test instruction set summary. 113
7.2 Bind instruction set summary. 114
7.3 Put instruction set summary. 114
7.4 Put_arg instruction set summary. 115
7.5 Control instruction set summary. 116
7.6 Execution time ratio: FGHC/C Vs NETLOG/C 127
7.7 Reverse benchmark for a variety of link communication speeds. 127
7.8 Reverse benchmark for a variety of problem sizes. 127
7.9 Delete benchmark for a variety of link communication speeds. 128
7.10 Delete benchmark for a variety of problem sizes. 128
7.11 Primes benchmark for a variety of link communication speeds. 129
7.12 Primes benchmark for a variety of problem sizes. 129
7.13 Matrix Multiplication benchmark for a variety of link communication speeds. 130
7.14 Matrix Multiplication benchmark for a variety of problem sizes. . . . 130

List of Figures

1.1 Linear array of processors. 4

2.1 Computing the sum of factorials. 23

2.2 An N-processor ring. 24

2.3 Computing the sum of factorials on an N-processor ring. 25

2.4 A Dyadic Network. 26

2.5 Sum of factorials using constraints. 32

2.6 Five dining philosophers. 33

2.7 Five dining philosophers on a ring of processors. 34

4.1 Multiple PE/single printer network. 54

4.2 GHC program for multiple PE/single printer network. 55

4.3 METATEM program for multiple PE/single printer network. 56

4.4 Annotated Concurrent Prolog program for multiple PE/single printer network. 58

4.5 NETLOG program for multiple PE/single printer network. 59

4.6 Computing the sum of factorials. 60

5.1 Abstract syntax of NETLOG 68

6.1 Snow White and the Dwarfs on a ring. 87

6.2 An 8-processor ring. 88

6.3 Bounded Buffer Communication on a 3D-Cube. 90

6.4 3-D Cube of processors. 91
6.5 Distributed program for simplified airline reservation system. 92
6.6 A star network. 93
6.7 Model of LINDA in EVENTLOG. 94
6.8 Computing primes using Linda. 95

7.1 Stages of the NETLOG compiler. 98
7.2 Intermediate program after stage 1. 100
7.3 Intermediate program after stage 2. 101
7.4 Object code generated by NETLOG compiler... 103
7.5 Object code generated by NETLOG compiler continued... 104
7.6 Object code generated by NETLOG compiler continued... 105
7.7 Object code generated by compiler continued. 106
7.8 NETLOG Abstract Machine organization. 108
7.9 Basic execution algorithm. 110

Chapter 1

Introduction

This dissertation describes the syntax, semantics, and implementation of NETLOG (NETwork LOGic programming), a new logic programming language for distributed computing. NETLOG was designed as a tool for specifying and rapidly prototyping a broad range of distributed applications and is unique in that it supports a *unified* framework in which both the *spatial* and *temporal* properties of distributed computations can be expressed in a concise and elegant fashion. Accordingly, NETLOG represents a new approach to specifying and reasoning about distributed logic programs.

In the following sections we discuss the motivation for our work and give a more detailed overview of the contents of this dissertation.

1.1 Motivation

Distributed computing systems are now readily available and cost effective for a large number of programming problems. These systems cover a wide spectrum in terms of intended application, size, and performance. There are also substantial differences in how they are programmed. Some are programmed in conventional languages, typically with the addition of several library routines for sending and receiving messages.

Others are programmed in completely new languages that have been specially designed for implementing distributed applications.

In the last 10 years there has been a great deal of research directed towards using logic as a tool for writing computer programs. The idea is attractive for a number of reasons:

- The dichotomy between specifications and programs is removed. A single notation is used for both; hence, the often error prone and time consuming step of translating specifications into a conventional programming language can be eliminated.
- Logic languages can be seen as high-level programming languages that support the rapid prototyping of applications. This allows the application design to be tested and any inconsistencies in the design to be detected early in the development cycle.
- Established mathematical techniques may be used to reason about and manipulate programs.
- Logic supports the writing of compositional and hierarchical programs.

One branch of logic programming research has lead to the development of a number of concurrent logic programming languages[Sha86,CG86,Ued86,R^{+}88,P^{+}86,Y^{+}86]. These languages were designed specifically for programming parallel and distributed systems. However, while existing concurrent logic programming languages provide support for specifying the concurrent components of a distributed computation and for expressing the synchronization and communication between these components, almost none of these languages provide comparable support for specifying the *spatial* properties of a distributed computation — that is, the properties that describe how these concurrent components are organized and distributed over the underlying network of processors.

Instead, these languages rely on the language implementation to implicitly perform this task. This may simplify the programmer's task for some applications but, in general, the implementation does not have any knowledge about the application being implemented — thus, the mapping (i.e., decomposition and distribution) strategy it employs is not specific to the problem. This is a severe restriction for many distributed applications since in practice it is often the case that an application needs to be distributed in a specific way on a particular distributed machine[B+89].

For example, many distributed applications are structured as a collection of specialized services or are inherently distributed in nature (see chapter 6). Such applications often require that certain functions (i.e., subcomputations) be performed on a particular processor in the network because that processor contains needed data or provides some specialized service. Similarly, high performance applications, such as those found in scientific computing, need to make optimal use of the available processors. In such high performance applications, decisions concerning which computations are executed on which processors are of great importance since they determine how efficiently the corresponding distributed program executes. Accordingly, in order to specify these kinds of distributed applications within the framework of logic programming, it is desirable to have a logic programming notation which allows programmers themselves to specify how their applications should best be mapped onto the network of processors — in other words, a notation which allows them to describe the spatial properties of their applications.

In this dissertation we present a new logic programming language called NETLOG which provides such facilities. NETLOG was designed as a tool for specifying and rapidly prototyping a broad range of distributed applications — both applications that require a particular distribution strategy and those that do not. The language itself is based on an executable subset of Distributed Systems Logic (DSL) and is unique in that it supports a *unified* framework in which both the *spatial* and *temporal* properties of distributed programs can be expressed in a concise and elegant fashion. Thus, in addition to specifying the concurrent components of a distributed

computation and the synchronization and communication between these components, NETLOG can also be used to specify both *where* (i.e., on which processors) to locate these concurrent components and *when* to execute them.

To illustrate the capabilities of NETLOG, we give several examples showing how the language can be used to specify a variety of well-known problems implemented on distributed architectures covering a range of network topologies. These examples include message passing using channels, distributed AI, distributed mutual exclusion, and distributed (replicated) databases. Collectively, the examples cover both transformational and reactive systems (the latter being well known as notoriously difficult to characterize formally).

1.2 Distributed Systems Logic (DSL)

Distributed Systems Logic is a modal logic that contains both spatial and temporal modal operators. NETLOG is based on a particular subset of DSL; thus, every NETLOG program represents a formula in DSL.

Formulas in DSL are interpreted with respect to a given distributed system, characterized abstractly as a network of distributed processes. One of the processes in the network is distinguished as the *origin* process and represents the reference point or origin of the network (in our programming model, computations begin at the origin process). Throughout this dissertation, networks will be represented pictorially as shown in figure 1.1. Each circle in the diagram represents a process. The circle for process p_1 has a darkened segment indicating that p_1 is the origin process. The labeled

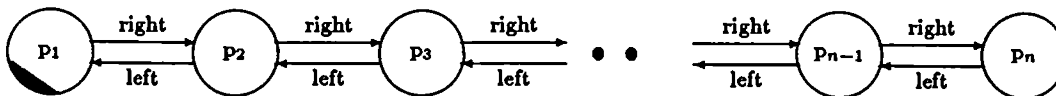

Figure 1.1: Linear array of processors.

arcs connecting the circles represent the interconnection network. As an illustration of the use of DSL, consider the following English statement: "On every processor of the linear array, X is set to the value 1 and then J is set to the value 2." This can be represented in DSL by the formula

$$\boxplus[X = 1; J = 2]$$

Here the operator $\boxplus$ corresponds to the concept "everywhere" and indicates that the subformula [X = 1; J = 2] is to be executed by every process in the network. The binary temporal operator ';', known as *chop*, indicates that the two equalities are to be executed sequentially.

Similarly, the statement — "On some processor of the linear array, X is always set to the value 10" — can be expressed in DSL as:

$$\diamond\boxdot[X = 10]$$

The spatial operator $\diamond$ can be understood as "somewhere," while the temporal operator $\boxdot$ corresponds to the concept "always."

Many of the modal operators found in DSL are also used in NETLOG.

1.3 Event-Action Model

NETLOG adopts the event-action[LS94,BC83] interpretation of logic programs. The particular event-action programming model used to describe the (informal) behavior of NETLOG programs is the Constraint based Event Model (CEM). Only the fundamental features of the model and its relationship to NETLOG are described here; a more detailed account is given in the next chapter. The event model contains two basic kinds of entities: *events* and *event handlers.* Events are abstract entities that are explicitly caused by the program as the computation proceeds. Event handlers are program constructs that are activated by the occurrence of one or more events. Once an instance of an event handler is activated it can either cause one or more

new events (which, in turn, can activate more event handlers) or impose *constraints* on the occurrence of one or more future events (e.g., such constraints can be used to restrict where and/or when a particular event occurs). Many event handlers may be activated concurrently. A computation begins by causing an initial set of events. These events activate one or more event handlers which, in turn, cause more events (and, possibly, some new constraints) and so on. The computation continues in this manner until no more events occur.

Based on this programming model, NETLOG programs can be interpreted as describing the set of events that can occur during the execution of a distributed program. Events (and the constraints imposed on them) are the basic source of control in NETLOG programs. Both deterministic and nondeterministic events can be specified in the language. NETLOG contains two kinds of event handlers, *multi-use* event handlers and *single-use* event handlers. Multi-use event handlers are permanent event handlers which exist throughout the extent of the computation. Thus, they may be activated many times during the course of a computation. In contrast, single-use event handlers are temporary event handlers. They may be activated at most once, after which they vanish. Only multi-use event handlers may impose constraints on events.

1.4 Formal Semantics

An important part of the definition of NETLOG is the inclusion of a formal operational semantics for the language. In particular, a mathematical description of the language is given using a Plotkin style operational semantics. Given such a semantics, we can formally reason about and prove important properties concerning the execution of NETLOG programs. In addition, a formal operational description can be used by programmers to answer decisively questions concerning the interaction between various constructs in the language, while implementors can use it as a reference in determining the correctness of their implementations.

1.5 Implementation

The implementation of NETLOG described in this dissertation is similar to the distributed implementation of Flat Concurrent-PROLOG (FCP) developed by Taylor for the Intel iPSC hypercube[Tay89]. In both cases the implementation strategy is based on compiling source programs into a sequence of (abstract) machine instructions for an abstract machine; nevertheless, there are a number of significant differences between the two implementations.

One difference between the two implementations is the design of the abstract machine. Our implementation is based on a new abstract machine design, the NETLOG Abstract Machine (NAM), which can be seen as an adaptation of Taylor's FCP abstract machine. The NAM includes additional features such as data structures for storing events and representing constraints, new control registers, new instructions for the creation and the matching of events, and modified control instructions to handle the scheduling of both sequential and concurrent threads.

Another difference is the approach taken in implementing the abstract machine itself. In the FCP implementation, abstract machine instructions are executed by an emulator written in the programming language C. In our implementation, abstract machine instructions are implemented using C-macros. The advantage of using the latter approach is better performance since it avoids the runtime overhead of instruction and argument decoding incurred by the C emulator[Tay89].

1.6 Comparison to related work

1.6.1 Other concurrent logic programming languages

Most of the concurrent logic programming languages that have been proposed to date are based on a particular subset of first-order logic known as Horn Clauses. Well known examples of this family of logic languages are Concurrent PROLOG[Sha86],

Guarded Horn Clauses (GHC)[Ued86], and PARLOG[CG86]. In general, these languages adopt the process interpretation of logic programs, guarded horn clauses, and committed-choice nondeterminism. In particular, clause goals are viewed as concurrent processes that synchronize and communicate through shared logical variables. A good survey of this branch of logic programming may be found in [Sha89].

Several concurrent horn clause languages have also been proposed based on alternative models wherein communication is not based on shared variables. Shared PROLOG[BC91] uses a global data structure called a *blackboard* as a mechanism for communication. Processes communicate and synchronize by adding atoms to, and/or deleting atoms from, this global blackboard. Delta PROLOG[P^+86] is based on a process model which uses *event goals* to effect communication. Two processes may communicate (and synchronize) by causing corresponding matching events. The CC family of logic languages defined by Saraswat[Sar89] is based on concurrent constraint logic programming. Processes (called agents in [Sar89]) communicate by posting constraints on a global store and synchronize by checking that constraints are entailed by the store.

More recently, other approaches to logic programming have been advocated. In particular, Temporal Logic[RU71,Kro87,MP92] has been used as the basis for several logic languages[FKTM86,AM87,Nes93] — including, TEMPURA[Mos86,HM87] and METATEM[BFG^+89,FB91,Fis93], two concurrent logic programming languages. The evaluation of programs in these languages are not based on proof procedures as is the case with horn-clause concurrent languages (see [BG88] for a more complete discussion of this topic); in addition, algorithms in these languages can be expressed using time-dependent operators such as $\boxdot$ ("always") and $\diamond$ ("sometimes").

More specifically, TEMPURA is an imperative temporal logic language that is based on a subset of Interval Temporal Logic[Mos83]. The language supports an algorithmic style of programming and contains a wide variety of constructs including several kinds of assignment statement, while loop, etc. One restriction imposed by the designers of the language is that nondeterminism is not permitted in TEMPURA

programs[Mos86]; thus, TEMPURA can only be used to express deterministic concurrent computations in which processes communicate using shared logical variables. METATEM is a concurrent temporal logic language based on classical linear-time temporal logic. METATEM programs describe collections of concurrent communicating objects, where objects consist of two parts: a body and an interface definition. The body of the object consists of a collection of rules that describe the object's behavior. The interface definition contains a list of declarations that identify which predicates appearing in the body of the object correspond to input and/or output messages. METATEM objects synchronize and communicate using broadcast message passing.

1.6.2 Comparison with NETLOG

The basic differences between NETLOG on the one hand, and horn-clause concurrent logic languages and temporal logic concurrent logic languages on the other, can be summarized as follows:

- NETLOG is based on a different logical formalism. In particular, NETLOG is based on a subset of distributed system logic, as opposed to horn clause logic or temporal logic. Thus, in addition to conventional logical operators such as $\wedge$ and $=$, and time-dependent operators such as $\boxdot$ and $\diamond$, NETLOG programs can contain spatial operators such as $\boxplus$ ("everywhere") and $\oplus$ ("somewhere").

- The programming model used to interpret programs is different. NETLOG uses an event-action model, CEM, whereas horn-clause concurrent languages and temporal logic concurrent languages are based on the process and object models. Concepts such as single-use event handlers and the like have no analog in these models. Both event handlers in NETLOG and agents in CC can impose constraints on the state of the program. In CC, however, such constraints take the form of simple equalities whereas in NETLOG a richer set of constraints is

permitted. Constraints can be inequalities and negated atoms and can include temporal and spatial operators.

- Communication and synchronization in NETLOG is achieved by causing and waiting on events. In contrast, horn-clause concurrent languages (with the exception of Delta-PROLOG) and TEMPURA use shared variables or globally shared data structures for communication and synchronization, while METATEM uses broadcast message passing.

  Compared to Delta PROLOG, the use of events in NETLOG is much more general, even with respect to communication; for example, events in NETLOG can be used to model broadcast and one-to-many communication not just point-to-point communication between two processes (as is the case for Delta PROLOG). More generally, the notion of event used in Delta PROLOG bears little relation to the event-action model underlying NETLOG.

- NETLOG programs are pure logic programs. The language does not need, nor does it include, any syntactic control constructs or other "extra-logical" features that have no foundation in logic. Thus, interesting properties (e.g., liveness, safety, etc.) of NETLOG programs can be reasoned about and proved easily. In contrast, concurrent horn-clause languages include various control constructs and features that have no logical foundation — the commit ('|') operator, mode declarations, read only annotations, etc.; likewise, TEMPURA uses the process statement and METEM includes interface declarations.

From a broader perspective, these differences reflect the different design goals which motivated the development of NETLOG. The design of NETLOG was undertaken from the point of view that the specification of a distributed computation includes specifying the spatial properties that describe how the computation is mapped onto the network of processors. Consequently, NETLOG was based on DSL, as opposed to horn-clause logic or temporal logic, since within the framework of DSL such inclusive

specifications can be expressed in a concise and elegant fashion. Indeed, the spatial operators of DSL are well suited to the task of expressing processor mappings.

In contrast, concurrent logic programming languages based on horn-clauses and temporal logic do not address the issue of how computations are mapped onto processors. To overcome this limitation, various extensions to these languages have subsequently been proposed. For example, Shapiro[Sha84] suggests extending Concurrent-PROLOG with annotations based on the language LOGO[Pap80] to express the mapping of Concurrent-PROLOG programs onto processors. Similarly, in [Fis93] various extra-logic features are suggested as extensions to METATEM. For example, the concept of "rooms" is proposed as a mechanism for partitioning and grouping objects.

One disadvantage associated with such extra-logical extensions is that the user must, in effect, deal with two different languages since the extensions constitute a language that has a (operational) semantics which is quite different (if not totally divorced) from the logical semantics associated with the rest of the language. Another disadvantage is that formal reasoning and verification of programs becomes much more difficult since such extra-logical extensions tend to compromise the declarative properties of logic programs.

NETLOG, on the hand, has the advantage of being a single language with a single semantics. Thus, the language preserves all the benefits that accompany the logic programming enterprise as cited in section 1.1.

It has been argued that languages such as NETLOG, and other language proposals that give users control over the mapping of their computations[F^+92,HS86b], are more difficult and cumbersome to use because the user is forced to deal with the details of the topology of the system and, moreover, programs in such languages are no longer portable.

While this may be true for other languages, this argument does not seem to hold in the case of NETLOG since user mappings may be expressed using a variety of operators, each of which embodies a different level of abstraction (with respect to the topology of the underlying distributed architecture). For example, the spatial

operator ◈ ("elsewhere") may be used to initiate (i.e., map) a subcomputation on another processor. In using this operator, however, no reference is made to the identity of the processor (i.e., no processor ids are used) or where it is located in the network. Thus, in NETLOG, one can use these more abstract operators, as appropriate, to avoid dealing with the topology of the system and to write programs that are abstract (i.e., architecture independent) and therefore portable.

Further discussion of the differences between NETLOG and existing concurrent logic languages may be found in chapter 4.

1.7 Contributions of dissertation

In summary, the key contributions made by this dissertation are:

- The design of a new distributed logic programming language, called NETLOG, that has the following desirable properties:
  - it is a high-level programming language that can be used to specify and rapidly prototype a broad range of distributed applications — both applications that require a particular distribution strategy and those that do not.
  - it supports a *unified* framework in which both the *spatial* and *temporal* properties of distributed computations can be expressed in a concise and elegant manner.
  - it has a well defined semantics and does not need, nor does it include, any extra-logical features or constructs. Consequently, the language facilitates reasoning formally about program behavior.
- The definition of a formal operational semantics for NETLOG — no similar semantics exists for a language of this kind.

- The introduction of the CEM event-action model as a suitable programming model for interpreting certain classes of logic languages such as NETLOG.
- The design of the NETLOG Abstract Machine, a new machine which realizes the CEM model, is described and shown to be suitable for implementing NETLOG on distributed memory multiprocessors.
- The definition and presentation of the first-order syntax and semantics of Distributed Systems Logic.

1.8 Organization of thesis

Chapter 2 contains an introduction to NETLOG. A fairly detailed, although informal, description of the syntax and operational behavior of the language is given. Chapter 3 examines the logical foundations of NETLOG. The syntax and semantics of first-order DSL is presented. In addition, using the basic operators of DSL, several additional operators that have proved useful in describing program behavior are defined. In chapter 4 we show how to reason about, and prove interesting properties of, NETLOG programs. Chapter 5 gives a formal account of the syntax and operational semantics of NETLOG. The behavior of NETLOG programs is captured using Plotkin style transition systems. In chapter 6 the utility of the language is demonstrated. The language is used to solve several well-known problems for a variety of network topologies. Chapter 7 describes a strategy for implementing NETLOG on a distributed memory multiprocessor. Experimental results based on the simulated execution of several of the example programs is presented. The dissertation concludes with chapter 8 which summarizes the research, presents our conclusions, and makes suggestions for further research.

Chapter 2

Introduction to programming in NETLOG

This chapter introduces the notation used to express NETLOG programs and gives an operational description of how programs execute. The description given here is informal but nevertheless provides enough information to understand most of the programming examples given elsewhere in this dissertation. A more comprehensive and formal description of the operational behavior of NETLOG programs may be found in chapter 5.

2.1 Programming Model

2.1.1 Constraint based Event Model (CEM)

The intuitive (i.e., informal) description of the operational behavior of NETLOG programs that we present in this chapter is given in terms of a particular programming model. The Constraint based Event Model (CEM) is an event-action[LS94,BC83] model that includes constraints. This programming model contains two basic kinds of entities: *events* and *event handlers.*

Events are abstract entities that are explicitly caused by the program as the computation proceeds. Event handlers are program constructs that are activated by the occurrence of one or more events. Once an instance of an event handler is activated it can either cause one or more new events (which, in turn, can activate more event handlers) or impose *constraints* on the occurrence of new events (i.e., such constraints place restrictions on when and/or where new events can occur). Many event handlers may be activated concurrently. A computation begins by causing an initial set of events. These events may activate one or more event handlers which, in turn, cause more events (and, possibly, some new constraints) and so on. The computation continues in this manner until no more events occur.

Based on this programming model, NETLOG programs can be interpreted as describing the set of events that can occur during the execution of a distributed program. In particular, each atom in the program, that is formula of the form, $\mathrm{p}(\mathrm{e}_1, \ldots, \mathrm{e}_n)$, denotes an event. The symbol p identifies the event type and the event arguments $\mathrm{e}_1, \ldots, \mathrm{e}_n$ represent information (i.e., data) that describes this particular event occurrence (i.e., the event arguments serve to distinguish between different occurrences of events that have the same event type). Similarly, to restrict the occurrence of an event $\mathrm{p}(\mathrm{e}_1, \ldots, \mathrm{e}_n)$, a constraint of the form $\neg\mathrm{p}(\mathrm{e}_1, \ldots, \mathrm{e}_n)$ can be used, where $\neg$ is the logical negation operator. Both deterministic and nondeterministic events can be specified in the language.

NETLOG allows two kinds of event handlers: *multi-use* event handlers and *single-use* event handlers. Multi-use event handlers are permanent event handlers which exist throughout the extent of the computation; thus, they may be activated many times during the course of a computation. Multi-use event handlers are specified using the $\Longrightarrow$ operator. In contrast to multi-use handlers, single-use event handlers are temporary event handlers. They may be activated at most once, after which they vanish. Single-use event handlers are expressed using the **atnext** and **wait** operators. Only multi-use event handlers may impose constraints on events.

Related Event Models

The concept of an event has been used in many guises in a variety of programming languages[LS94,Reu80,BC83]. The event model described above is most similar to the event model used by Reuveni[Reu80]. There are, however, some important differences.

In Reuveni's event model, for example, there are two kinds of events, single-use and multi-use (the latter corresponds to the notion of event used in our model), and only one kind of event handler (corresponds to a multi-use event handler in our model). Single-use events are deleted after being used by an event handler; thus, when used these events cause the event space (i.e., collection of existing events) to be altered or "side-effected." Such "side-effects" are not consistent with the declarative nature of logic programming, so the notion of single-use events was not included in our model. Nevertheless, the effect of an event being used at most once can easily be achieved in our model, without side-effects, by having the event activate only single-use event handlers.

Another difference between our event model and Reuveni's is that Reuveni's event model is based on a single global event space whereas our model is based on a distributed event space. In our event model, the occurrence of an event is associated with a particular processor in the network so the event space is distributed amongst the processors. Thus, we distinguish between events occurring at one location (i.e., processor) and those occurring at other locations. Moreover, the ability to use constraints to control the occurrence of events has no analog in Reuveni's event model. Due to these differences, certain computations that can be expressed in our model cannot be expressed in Reuveni's.

2.2 Syntax of NETLOG

The syntax of NETLOG includes conventional logical operators such as $=$ ("equality"), $\Longrightarrow$ ("implication"), $\wedge$ ("logical-and"), and $\vee$ ("logical-or"); temporal operators

such as $\odot$ ("next"), $\boxdot$ ("always"), $\diamond$ ("sometime"), **until**, **atnext**, and ; ("chop"); and spatial operators such as $\boxplus$ ("everywhere"), $\oplus$ ("somewhere"), $\oplus$ ("elsewhere), **nearby**, and symbols $\ell \in \bar{\mathrm{L}}$ drawn from the set of *link* symbols.

The principle syntactic categories in NETLOG programs are: *variables*, *expressions*, *event descriptors*, *actions*, *constraints*, and *rules*.

2.2.1 Variables

There are two kinds of variables in NETLOG:

- Local variables: N', K', Max', ...
- Global variables: A, B, Hen, ...

To distinguish the two kinds of variables syntactically, local variables are primed. Operationally, local variables differ from global variables in that the value of a local variable can change from one state of the computation to the next. The value assigned to a global variable, however, remains the same throughout the entire computation.

2.2.2 Expressions

Expressions are combinations of constants, variables, and operators as follows:

- Symbolic constants: a, b, done, ...
- Arithmetic constants: 0, 1, 10, ...
- Local variables: N', K', Max', ...
- Global variables: A, B, Hen, ...
- Arithmetic expressions: $e_1 + e_2$, $e_1 - e_2$, $e_1 \times e_2$, e_1/e_2, e_1 **mod** e_2, etc., where e_1 and e_2 are variables, arithmetic constants, or arithmetic expressions.

- Relational expressions: $e_1 = e_2$, $e_1 > e_2$, $e_1 \geq e_2$, $e_1 < e_2$, $e_1 \neq e_2$, etc., where e_1 and e_2 are variables, arithmetic constants, or arithmetic expressions.

- List expressions: $[e_1,\ldots,e_n]$, where $e_1,\ldots,e_n$ are expressions.

Note that constant symbols begin with a lowercase letter while variables begin with uppercase letters.

2.2.3 Event Descriptors

Event descriptors can have one of two forms depending on whether the descriptor appears in a single-use event handler or a multi-use event handler. In the following, G_i denotes a simple action and T_i denotes a relational expression:

- Single-use event descriptor, B: $G_1 \wedge \cdots \wedge G_k \wedge T_1 \wedge \cdots \wedge T_j$ $(j \geq 0,\ k \geq 1)$

- Multi-use event descriptor, H: $G \wedge T_1 \wedge \cdots \wedge T_k$ $(k \geq 0)$

2.2.4 Actions

Actions can be *simple* or *compound.* A simple action is the causing of a single event. Simple actions have the form given below, where E is an atom of the form $p(e_1,\ldots,e_n)$:

- Local event: E (cause event E)

- Directed event: $\ell_1 \cdots \ell_n$ E (cause event E on the processor reached by following directions $\ell_1 \cdots \ell_n$)

- Global event: ⊞ E (cause event E on every processor)

- Nearby event: **nearby** E (cause event E on every adjacent processor)

- Nondeterministic inclusive event: $\diamond$ E (cause event E on some processor)

- Nondeterministic exclusive event: $\diamond$ E (cause event E on some other processor)

A compound action is a sequence of actions. Let S_1 and S_2 denote compound actions, and G denote a simple action. Then compound actions are defined recursively as follows:

- Simple action: G (execute G)
- Next: $\odot S_1$ (execute S_1 in the next step of the computation)
- Sometime: $\diamond S_1$ (execute S_1 in some future step of the computation)
- Atnext: $[S_1$ **atnext** $B_1] \vee \cdots \vee [S_n$ **atnext** $B_n]$, $n \geq 1$ (execute single-use **atnext** event handler(s) until events satisfying some B_i occur, then execute the corresponding S_i and terminate)
- Chop: S_1 ; S_2 (execute S_1 and S_2 sequentially)

2.2.5 Constraints

Constraints are used to impose restrictions on the event space (i.e., the set of allowable events). Basic constraints are conjunctions of conditional constructs that have one of the following forms:

- Active, K_A: $\neg E$ **if** $\neg E_1 \wedge \cdots \wedge \neg E_k$ $(k \geq 1)$
- Passive, K_p: $\neg(G_1 \wedge \cdots \wedge G_k \wedge \neg E_{k+1} \wedge \cdots \wedge \neg E_n)$ $(n \geq k \geq 1)$

where E is a local event and G is a simple action. The global variables of a conditional clause are implicitly assumed to be universally quantified. While both kinds of basic contraints can be used to specify restrictions on the occurrence of events, only active constraints can be used to place restrictions on event arguments. That is, they can be used to bind (i.e., constrain) event variables to values (see the discussion in section 2.5 below).

General constraints have the form described below. The definition is recursive. Here, C is itself a general constraint and K is a basic constraint:

- Negation: K (constraint K holds for the current step of the computation)
- Until: K **until** B (constraint K holds until events matching B occur)
- Always: $\boxdot K$ (constraint K holds now and in all future steps of the computation)
- Weak Next: $\bigcirc C$ (constraint C holds at the next step of the computation, if there is a next step)

2.2.6 Rules and Programs

Rules represent multi-use event handlers. Let S denote a sequential action, C a constraint, and H an event descriptor. Then rules are defined as follows:

- Action rule: $H \Longrightarrow S$ (execute S whenever events satisfying H occur)
- Constraint rule: $H \Longrightarrow C$ (establish constraint C whenever events satisfying H occur)

A program consists of a set of rules $R_1, \cdots, R_n$, and an initial action A_0 (often referred to as the initial assertion) which is either a sequential statement, a constraint, or a conjunction of sequential statements and/or constraints. Programs have the following general form:

- Program: $\{R_1, \cdots, R_n\}$ **assert** A_0 (execute rules $R_1, \cdots, R_n$ on every processor and initiate action A_0 on the origin processor)

Logically, a rule (i.e., multi-use event handler) R corresponds to the DSL formula

$$\boxdot\boxplus\forall X_1, \ldots X_n \exists Y_1, \ldots Y_j (H \longrightarrow A)$$

where $X_1, \ldots, X_n$ are the global variables appearing in H, and $Y_1, \ldots, Y_j$ are the global variables appearing in A but not in H[1]. Thus, for example, the following rule

$$a(X) \Longrightarrow b(X, Y)$$

[1] Or in the scope of a basic constraint; recall, global variables within the scope of a basic constraint are considered to be universally quantified.

corresponds to the DSL formula

$$\boxdot\boxplus\forall X\exists Y(a(X) \longrightarrow b(X,Y))$$

Rules are read declaratively as "always everywhere for all $X_1,\ldots,X_n$ there exists $Y_1,\ldots,Y_m$ such that if H is true then A is true."

Finally, a program corresponds to the DSL formula:

$$\underline{R}_1 \wedge \cdots \wedge \underline{R}_n \wedge A_0$$

where $\underline{R}_1,\cdots,\underline{R}_1$ are the quantifier closed versions of $R_1,\cdots,R_n$, respectively.

In the examples that follow, we assume that the various operators in the language have the operator precedences given in section 3.2 of chapter 3. Parenthesis, (and), or square brackets, [and], may be used to explicitly control the scope of operators.

2.3 Additional Constructs

Other constructs are permitted, but these are considered extensions to the language that expand into statements already described. Here are some additional constructs along with their equivalent expansions.

Extensions to rules:

$$H \Longrightarrow C_1 \wedge \cdots \wedge C_n \equiv H \Longrightarrow C_1,\ldots,H \Longrightarrow C_n$$

Extensions to constraints:

$$\neg E \equiv \neg E \textbf{ if } \neg\text{false}$$

$$\neg(G_1 \wedge \cdots \wedge G_k \wedge \neg E_{k+1} \wedge \cdots \wedge \neg E_n) \equiv \neg\text{true} \textbf{ if } G_1 \wedge \cdots \wedge G_k \wedge \neg E_{k+1} \wedge \cdots \wedge \neg E_n$$

Other extensions:

$$S \textbf{ afternext } B \equiv \odot S \textbf{ atnext } B$$

$$\textbf{wait } B \equiv \text{true } \textbf{atnext } B$$

$$A \textbf{ atnext } (B_1 \textbf{ also } B_2) \equiv [(A \textbf{ atnext } B_1) \textbf{ atnext } B_2] \vee [(A \textbf{ atnext } B_2) \textbf{ atnext } B_1]$$

2.4 Example: Computing Sum of Factorials

Consider the simple program, shown in figure 2.1, for computing the sum of two factorials. The program consists of a body, a set of (multi-use) event handlers which are replicated across all the processors in the network, and an initial action (i.e., initial program statement) which starts the computation off.

```
  { infac(X) ⟹ fac(X,1,1),
    fac(X,N,R) ∧ N<X ⟹ ⊙fac(X,N+1,(N+1)×R),
    fac(X,N,R) ∧ N≥X ⟹ outfac(X,R),
    outfac(X,R) ⟹ ⊡¬(⊕outfac(X,Y) ∧ Y≠R)
  }
assert
  infac(3) ∧
  ⊕infac(7) ∧
  wait (outfac(3,X) also ⊕outfac(7,Y)) ; write(X+Y)
```

Figure 2.1: Computing the sum of factorials.

In this example, executing the initial action causes both a local event, **infac**(3), and a nonlocal event, **infac**(7), to occur. These events initiate the computations to compute 3! and 7!, respectively. Note the use of the spatial operator ⊕ which is used to specify that the event **infac**(7) is to occur on some processor other than the current processor. Executing the initial action also causes the (single-use) event handler **wait** (**outfac**(3,X) **also** ⊕**outfac**(7,Y)) to begin executing. This event handler is used in waiting for the results of the two factorial computations. After the results have been computed, the predefined event **write**(X+Y) is caused to display the result. Except for the constraint event handler which establishes the constraint ⊡¬(⊕**outfac**(X,Y) ∧ Y≠R) when event **outfac**(X,R) is caused, the body of our program is reminiscent of the familiar recursive implementation of the factorial relation. The importance

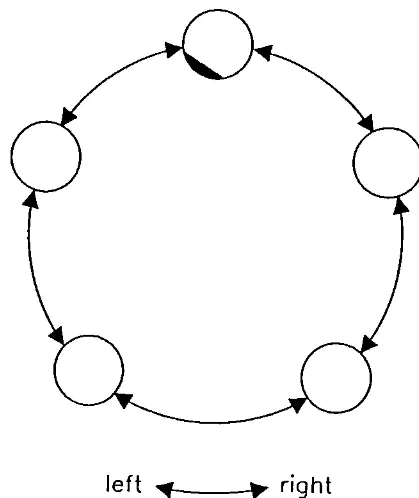

Figure 2.2: An N-processor ring.

of including such constraints is that they remove any ambiguity in the meaning of our program and, in this case, make it clear that the same value must be returned whenever we compute a particular factorial.

The program shown in figure 2.1 is quite general since it makes no assumptions about the structure of the distributed system that will perform the computation. Thus, it describes a computation that can be executed successfully on any distributed system, irrespective of its topology.

For example, the computation described by our program could be executed successfully on the N-processor ring shown in figure 2.2. However, the topology of the ring gives rise to non-uniform communication costs. Thus, for this particular architecture, we may wish to refine our program in order to be more specific about where each of the two factorials are computed. In particular, we might stipulate in our program that the two factorials are to be executed on adjacent processors, thereby minimizing the cost of interprocessor communication. A program that illustrates one way to accomplish this is shown in figure 2.3.

```
{ infac(X) ⟹ fac(X,1,1),
  fac(X,N,R) ∧ N<X ⟹ ⊙fac(X,N+1,(N+1)×R),
  fac(X,N,R) ∧ N≥X ⟹ outfac(X,R),
  outfac(X,R) ⟹ ⊡¬(⊕outfac(X,Y) ∧ Y≠R)
}
assert
  infac(3) ∧
  right infac(7) ∧
  wait (outfac(3,X) also right outfac(7,Y)) ; write(X+Y)
```

Figure 2.3: Computing the sum of factorials on an N-processor ring.

Notice that the body of the program is identical to that given in the previous example. The only difference between the two programs is the initial statement which now causes the nonlocal event `infac(7)` to occur on an adjacent processor. As these two examples illustrate, NETLOG may be used to describe distributed computations that are targeted for execution on a particular distributed architecture, as well as describe distributed computations that are more general and therefore suitable for execution on a range of distributed architectures. Moreover, tailoring (or generalizing) a NETLOG program can often be accomplished just by changing the spatial operators in the program, while leaving the remainder of the program undisturbed.

2.5 Nondeterminism and Constraints

Nondeterminism is a fundamental feature of many important distributed programming problems. NETLOG adopts the *committed-choice* (also referred to as *don't-care*) interpretation of nondeterminism[Sha89]. In this respect, NETLOG is similar to existing concurrent logic programming languages since they are also based on the

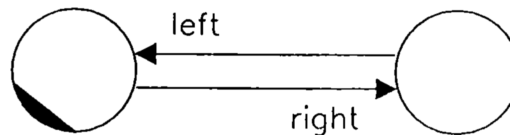

Figure 2.4: A Dyadic Network.

committed-choice approach to executing nondeterministic formula. NETLOG differs however in that nondeterminism is expressed at the language level. In particular, the language contains an extensive set of operators that allows one to explicitly specify various kinds of nondeterministic behavior. Both nondeterministic events and nondeterministic event handlers can be specified. Control over the behavior of nondeterministic events is provided for via constraints.

Nondeterministic events and constraints

As illustrated earlier, nondeterministic spatial events can be specified using the spatial operators ◇ and ⟐. Precisely *where* (i.e., on which processor) these events actually occur is determined by the constraints that have been imposed on the event space of each processor. In particular, if a nondeterministic event does not violate the constraints associated with a given processor, that processor will attempt to exclusively *commit* the event. If it succeeds, the event is *committed* to occur on that processor. Several processors may simultaneously attempt to commit the same nondeterministic event, but at most one of them will succeed. Of course, once committed, the event

may cause the activation of event handlers in the usual way.

For example, consider executing the following program on the dyadic system shown in figure 2.4. As there are no constraints present, the nondeterministic event q(2) may occur on either processor. Thus, the program would write either 4 or 6, depending on which processor actually succeeds in committing the nondeterministic event.

```
  { p(X) ⟹ [ write(X+Y) atnext q(Y) ]
  }
assert
  p(4) ∧ right p(2) ∧ ⊙⊕q(2)
```

On the other hand, due to the constraint ⊡¬q(2) being imposed, the following program will only write the number 4.

```
  { p(X) ⟹ [ write(X+Y) atnext q(Y) ],
    p(4) ⟹ ⊡¬q(2)
  }
assert
  p(4) ∧ right p(2) ∧ ⊙⊕q(2)
```

In an analogous manner, constraints may also be used to control nondeterministic temporal events. Nondeterministic temporal events can be directly expressed using the temporal operator ◇. These events remain pending until there are no constraints which prevent them from occurring; thus the following program will either print 5 followed by 6 or vice versa.

$$\{\ q(X) \Longrightarrow \text{write}(X+2)$$
$$p(X,Y) \Longrightarrow \boxdot\neg(q(X) \wedge q(Y))$$
$$\}$$

assert

$$p(3,4) \wedge \diamond q(3) \wedge \diamond q(4)$$

Temporal nondeterminism can also arise as a result of the temporal operator ';' which is used to express the concept of sequentiality. Intuitively an action of the form $A_1;A_2$ means execute A_1 and then execute A_2. However, the logical (i.e., declarative) semantics of the operator ';' (see chapter 3) allows an arbitrary delay between the execution of A_1 and the subsequent execution of A_2.

In NETLOG, the operator ';' is given a particular operational interpretation wherein action A_2 is executed immediately after action A_1 providing no constraints exist which force A_2 to be delayed. Typically, therefore, sequential actions are executed one after the other without arbitrary or fixed delays in between. Nevertheless, users may override this behavior by using constraints to explicitly control the execution of sequential actions. For example, in the following program `d(2)` cannot occur until `c(2)` occurs.

$$\{\ p(X) \Longrightarrow a(X)\,;b(X)\,;c(X),$$
$$q(X) \Longrightarrow \neg d(X)\ \textbf{until}\ c(X),$$
$$r(X) \Longrightarrow p(X)\,;d(X)$$
$$\}$$

assert

$$q(2) \wedge r(2)$$

Nondeterministic event handlers

In NETLOG, one can also express event handlers that are themselves nondeterministic; that is, event handlers that directly incorporate nondeterminism. Nondeterminism in event handlers is expressed using the ∨ operator; it may also be expressed by using the spatial operators ⟠ and/or ⟡ in event descriptors. Thus, for example, the following program will print either 2 or 6:

```
  { p(Y) ⟹ [ write(X+Y) atnext q(X)
                          ∨
             write(X-Y) atnext q(X) ]
  }
assert
  q(4) ∧ p(2)
```

Likewise, either 4 or 6 would be printed by the following:

```
  { p(Y) ⟹ [ write(X+Y) atnext ⟠q(X) ]
  }
assert
  q(4) ∧ left q(2) ∧ p(2)
```

Nondeterministic events and nondeterministic event handlers allow many practical aspects of distributed computations to be expressed in a simple intuitive manner. Load balancing and mutual exclusion, for example, can be expressed easily using nondeterministic events. In a later section, we use the familiar problem of the five dining philosophers to illustrate how nondeterministic events, in conjunction with constraint event handlers, may be used to implement mutual exclusion.

Constraints as data stores

NETLOG requires that the arguments of an event evaluate to ground terms at the time the event occurs — that is, any variables appearing in an event's argument list must be defined. One way to accomplish this is to ensure that any variables are assigned values prior to causing the event. In the program fragment below, for example, the variable X appearing in q(X) is defined by virtue of the fact that event p(1) matches the event descriptor p(X) which in turn causes X to be instantiated to 1.

```
  { p(X) ⟹ q(X),
    u(Y) ⟹ X=2×Y ; v(X)
  }
assert
  p(1) ∧ u(2)
```

In the case of event v(X), variable X is explicitly assigned a value before causing the event.

Another way to ensure that event arguments are defined at the time the event occurs is to use constraints. In addition to their use in controlling nondeterminism, constraints may also be used to ensure that the otherwise undefined variables of an event are bound (or constrained as it were) to specific values. Consider the following program fragment which describes the partial behavior of two processes executing on the dyadic system given earlier.

{ proc(Id) $\Longrightarrow$ get(Input′,Id) ; ⋯

input(L) $\Longrightarrow$ ⊡(¬get(X,N) **if** X≠L↓N)

}

assert

⊞input([3,9]) ; ⊞proc(Pid′)

The first event handler in the program describes the behavior of the processes, each process begins by getting its input data. The second event handler describes the constraints on the data to be distributed amongst the processes. The event ⊞proc(Pid′) causes one process on each node of the system to begin executing, while ⊞input([3,9]) represents the input data to be distributed amongst these two processes. Note, Pid′ is a predefined state variable that contains the unique (integer) index of the executing processor; the binary function ↓ takes a list L and an index N and returns the Nth element of L.

Operationally, due to the constraint ⊡(¬get(X,N) **if** X≠L↓N), a get(Input′,Id) event can only occur if the state variable Input′ can be unified with an element of the input list. In effect, if left undefined, the constraint instantiates the state variable Input′ associated with each process to a different element of the input list [3,9].

The data used to bind the state input variables is actually stored as part of the constraint itself. Thus, when used in this way, constraints serve as a general mechanism for storing data (e.g., information) and binding values to variables. In general, the principle advantage of this technique (i.e., using constraints to store and bind values) is programs tend to be shorter and more abstract, and therefore more pleasing from a specification standpoint. This technique is used in several of the examples given in chapter 6. Figure 2.5 shows another version of the sum of factorials program in which the values of X and Y are determined using constraints.

From an operational perspective, it is interesting to note that active constraints can be viewed as *inference* rules[Llo84]. As such, they can be seen as nothing more than prolog[CM81,SS86] rules over negative atoms. Hence, we can treat a conjunction

```
   { constraints(fac) ⟹ ⊡( ¬fac(X,Y) if ¬fac(X,1,Y) ∧
                            ¬fac(0,N,R) if N≠R ∧
                            ¬fac(X,N,R) if X≰0 ∧ ¬fac(X-1,N×X,R) )
   }
assert
   ⊞constraints(fac);
   fac(3,X);
   ⟐fac(7,Y);
   write(X+Y)
```

Figure 2.5: Sum of factorials using constraints.

of active constraints like a prolog program and checking whether or not an event satisfies the constraints can be done by simply querying the program (as in prolog). The query either fails causing the event to be rejected, or the query succeeds causing any unbound variables in the event to be instantiated as a side effect. Indeed, in checking whether an event violates a set of basic constraints, active constraints are checked first. If the event is not rejected, any unbound event arguments become bound. The resulting (ground) event is then checked against the passive constraints.

Example 2: Five Dining Philosophers problem

The problem of the five dining philosophers[Dij71] involves five philosophers who spend their time in infinite cycles of eating and thinking. The philosophers sit at a round table, and there is one fork between every two adjacent philosophers (total of five forks). In order to eat, each philosopher needs the fork to its left and the fork to its right. Figure 2.6 shows one possible solution to this problem expressed in NETLOG. Note, the program shown is not a complete program but does show the essential synchronization details

The program consists of two event handlers. The first event handler describes the behavior of the philosophers, while the second event handler describes the synchro-

```
  { phil(I,J) ⟹ ... think(some) ...;
                  getfork(I,J);
                  ... eat(some) ...;
                  putfork(I,J);
                  phil(I,J),
    getfork(I,J) ⟹ (¬getfork(X,I) ∧ ¬getfork(J,Y)) until putfork(I,J)
  }
assert
  phil(1,2);
  phil(2,3);
  phil(3,4);
  phil(4,5);
  phil(5,1)
```

Figure 2.6: Five dining philosophers.

nization constraints. Events of the form `getfork(F1,F2)` are used to denote that left fork `F1` and right fork `F2` have been obtained; similarly, event `putfork(F1,F2)` denotes that the two forks `F1` and `F2` have been returned to the table.

Mutual exclusion is assured by the language semantics since a `getfork(F1,F2)` event implies that another `getfork` event that refers to either fork `F1` or fork `F2` cannot occur until a corresponding `putfork(F1,F2)` event occurs.

The above program solves the dining philosophers problem for the case in which the philosophers are executing concurrently on the same processor (the absence of any spatial operators should make this clear). The more general problem can be solved just as easily. For example, consider solving the dining philosophers problem on the ring of processor shown in figure 2.2, where one philosopher is assigned to each processor. The program for this situation is shown in figure 2.7. It is easy to see that both of the programs presented are deadlock free. Both programs could be further enhanced to be starvation free as well.

```
    { phil(I,J) ⟹ ... think(some) ...;
                  getfork(I,J);
                  ... eat(some) ...;
                  putfork(I,J);
                  phil(I,J),
      getfork(I,J) ⟹ (¬left getfork(X,I) ∧ ¬right getfork(J,Y)) until putfork(I,J)
    }
assert
    phil(1,2);
    right phil(2,3);
    right right phil(3,4);
    left phil(5,1);
    left left phil(4,5)
```

Figure 2.7: Five dining philosophers on a ring of processors.

2.6 Summary

This chapter contained an overview of NETLOG. Through several simple examples, NETLOG was shown to be capable of expressing many important concepts in distributed programming — i.e., concurrency, synchronization, (interprocessor) communication, mutual exclusion, nondeterminism, and locality — in a concise and elegant fashion. A deeper insight into the language may be found in chapter 5.

Chapter 3

Distributed Systems Logic

In this chapter we examine the logical foundations of NETLOG. In particular, the syntax and semantics of first-order Distributed Systems Logic is presented.

3.1 Background

Distributed Systems Logic (DSL) is a modal logic that contains both temporal and spatial modalities. It is a useful formalism for specifying and reasoning about distributed computing systems with fixed topologies. The logic includes conventional operators such as $=$ ("equality"), $\wedge$ ("and"), and $\vee$ ("or"); temporal operators such as $\odot$ ("next"), $\diamond$ ("sometimes"), $\boxdot$ ("always"), ; ("chop"), and **atnext**; and spatial operators such as $\diamond\!\!\!+$ ("elsewhere"), $\boxplus$ ("everywhere"), $\Re$ ("region"), **nearby** and symbols ℓ, called *links*, drawn from the set $\overline{\mathrm{L}}$ of all link symbols.

Examples illustrating the use of operators such as ;, $\boxdot$, and $\diamond\!\!\!+$ were given in chapter 1. Here we shall give an example illustrating the use of the region operator $\Re$. The region operator is useful for describing the properties of a network in terms of its partitions. For example the DSL formula

$$\boxplus[X = 2]\Re\diamond\!\!\!+[X > 4]$$

describes a network which is partitioned into two regions where X equals 2 everywhere in one region and X is greater then 4 somewhere in the other.

DSL itself is closely related to the Multiprocessor Network Logic (MNL) introduced by Reif and Sistla[RS85]. However, our desire for a network logic that supports *compositional* reasoning[Eme90][1] has resulted in a number of important differences. One difference between the two logics is that, in DSL, the properties of a network can be described in terms of the properties of its composite regions. The concept of regions (i.e., network partitions) is not present in Reif and Sistla's logic. Another basic difference is that the concept of time is based on intervals in DSL as opposed to being point based as is the case in MNL.

Not surprisingly, these differences are also reflected in the repertoire of spatial and temporal operators each logic provides. For example, DSL contains the temporal operator ';', as well as spatial operators such as **nearby** and $\Re$[2], whereas MNL does not. As neither logic has been used extensively, it is obviously too premature to compare the various strengths of each approach. Nevertheless, our experience to date suggests that DSL does indeed facilitate the writing of more modular composeable specifications.

In the next section, we provide a formal definition of the syntax and semantics of first-order DSL. A knowledge of classical temporal logic would be helpful, but the description given below should be accessible even to those without prior knowledge of classical temporal logic.

3.2 First-Order Distributed Systems Logic (FDSL)

We now define first-order DSL.

[1]Intuitively, specification languages that support compositional reasoning contain operators which allow the properties of different parts of a program to be combined into a single property (i.e., formula). Thus, the properties of a complete program can be obtained by first specifying the properties of its constituent parts and then combining these constituent properties into a single property for the entire program.

[2]Both ';' and '$\Re$' are compositional operators.

3.2.1 Alphabet

The *alphabet* of symbols used in constructing formulas is as follows:

- A denumerable set of variables: x, y, ...
- For every $n \geq 0$, a denumerable set of n-ary function symbols: f, c, ... (also called *constant* symbols in the case $n = 0$)
- For every $n \geq 0$, a denumerable set of n-ary predicate symbols: r, q, ... (also called *propositional* symbols in the case $n = 0$)
- The binary predicate symbol: $=$
- Logical symbols: $\neg$, $\wedge$, $\exists$, (, and)
- Temporal symbols: $\odot$, $\diamond$, $\boxdot$, ;, and **atnext**
- Spatial symbols: $\diamondplus$, $\boxplus$, $\Re$, **nearby**, and link symbols $\ell \in \bar{\mathrm{L}}$

3.2.2 Syntax

The *terms* of first-order Distributed Systems Logic (FDSL) are defined inductively as follows:

- Each constant c is a term.
- Each variable x is a term.
- If f is a function symbol of arity n, and $t_1, t_2, \ldots, t_n$ are terms, then $f(t_1, t_2, \ldots, t_n)$ is a term.

The *atomic formula* of FDSL are defined as follows:

- Each atomic proposition q is an atomic formula.
- If t_1 and t_2 are terms, then $t_1 = t_2$ is an atomic formula.

- If r is a predicate symbol of arity n, and $t_1, t_2, \ldots, t_n$ are terms, then $r(t_1, t_2, \ldots, t_n)$ is an atomic formula.

We inductively define the class of *formulas* as follows:

- Each atomic formula is a formula;
- If φ and ψ are formulae then $(\varphi \wedge \psi)$ and $\neg\psi$ are formulae;
- If φ is a formula and x is a free variable in φ then $\exists x\, \varphi$ is a formula;
- If φ and ψ are formulae then $\odot\psi$, $\diamond\psi$, $\boxdot\psi$, $(\varphi\,;\psi)$, and $(\varphi\,\mathbf{atnext}\,\psi)$ are formulae;
- If φ and ψ are formulae then $\oplus\psi$, $\boxplus\psi$, $\ell\psi$, $(\varphi\Re\psi)$, and $(\varphi\,\mathbf{nearby}\,\psi)$ are formulae;

The set of formulae generated by the above rules forms the language of FDSL.

Additional operators can be introduced as abbreviations. We assume the standard abbreviations for $\vee$, $\longrightarrow$, $\equiv$, true, and false. We also assume the following abbreviations:

$$
\begin{aligned}
\oplus\varphi &\stackrel{\text{def}}{=} \varphi \vee \oplus\varphi \quad (\text{“somewhere”}) \\
\bigcirc\varphi &\stackrel{\text{def}}{=} \odot\varphi \vee \neg\odot\,\text{true} \quad (\text{“weak next”}) \\
\varphi\,\mathbf{until}\,\psi &\stackrel{\text{def}}{=} \psi\,\mathbf{atnext}\,(\varphi \longrightarrow \psi) \wedge \diamond\psi \quad (\text{“until”}) \\
\varphi\,\mathbf{if}\,\psi &\stackrel{\text{def}}{=} \varphi \longleftarrow \psi \quad (\text{“if”})
\end{aligned}
$$

In the interest of notational simplicity, we adopt the following conventions with regard to the binding power of the operators:

(highest) $=$, $\neg$, $\bigcirc$, $\odot$, $\diamond$, $\boxdot$, $\diamond$, $\diamond$, $\boxplus$, ℓ, **nearby**, **until**, **if**
atnext

$;$

$\wedge$

$\vee$

$\longrightarrow$

(lowest) $\Re$

Accordingly, we shall omit any unnecessary parentheses. Thus, for example, instead of

$$(\odot\varphi_1 \vee \varphi_2) \longrightarrow (\neg\psi_1 \wedge (\psi_2\,\mathbf{atnext}\,\psi_3) \wedge \psi_4),$$

we write

$$\odot\varphi_1 \vee \varphi_2 \longrightarrow \neg\psi_1 \wedge \psi_2\,\mathbf{atnext}\,\psi_3 \wedge \psi_4.$$

3.3 Networks

A network is a triple N=(L,P,$\hat{\mathcal{P}}$), where L$\subseteq \overline{\text{L}}$, P is a countable set of elements called processes, and $\hat{\mathcal{P}}$:P$\times$L$\rightarrow$ P is a partial mapping. Networks also satisfy the following condition, $\forall p \in \text{P}\ \exists \ell \in \text{L}(\hat{\mathcal{P}}(p,\ell) = p')$. Intuitively, for each process p $\in$P and link ℓ $\in$L, $\hat{\mathcal{P}}(p,\ell)$, if defined, is the process connected to p via link ℓ; moreover, ever process is connected via some link to at least one other process.

3.4 Semantics

A model $\mathcal{M}$, with respect to a particular network N, is a 4-tuple (N,D,Σ,$\mathcal{L}$), where

- N=(L,P,$\hat{\mathcal{P}}$) is a network.

- D is a data domain.
- $\Sigma=(\sigma_0, \sigma_1, \ldots)$ is a sequence of *distributed* states (an interval of time), where $\sigma_i : \mathrm{P} \rightarrow \mathrm{S}$ associates with each process p the process state $\sigma_i(p)(\in \mathrm{S})$ which maps the local variables of p.
- $\mathcal{L}=(\mathcal{C},\mathcal{I})$ where $\mathcal{C}$ is a global variable valuation which assigns an element of D to each global variable, and $\mathcal{I}$ is an interpretation which provides the meaning of the function and relational symbols.

The set of symbols of FDSL is assumed to be divided into two classes: the class of *global* symbols and the class of *local* symbols. The interpretation of local symbols is state dependent and can vary from state to state; whereas, the interpretation of global symbols is the same for all states. In our presentation, we assume that all function and constant symbols are global. Variable, relational, and propositional symbols may be either local or global. In the case of global symbols r, $\mathcal{I}(\mathrm{s})(\mathrm{r})=\mathcal{I}(\mathrm{s}')(\mathrm{r})$ for all $\mathrm{s}, \mathrm{s}' \in \mathrm{S}$. It is convenient to refer to the global interpretation associated with $\mathcal{I}$ by $\hat{\mathcal{I}}(\mathrm{r})=\mathcal{I}(\mathrm{s})(\mathrm{r})$, where r is a global symbol and s is any process state.

The meaning of the terms of FDSL with respect to a model $\mathcal{M}=(\mathrm{N},\mathrm{D},\Sigma,\mathcal{L})$ is given by a function $\mathcal{M}_{<i,p>}$ defined inductively as follows:

$\mathcal{M}_{<i,p>}(c) = \hat{\mathcal{I}}(c)$, where c is a constant and is therefore global;

$\mathcal{M}_{<i,p>}(x) = \mathcal{C}(x)$, where x is a global variable;

$\mathcal{M}_{<i,p>}(x) = \mathcal{I}(\sigma_i(p))(x)$, where x is a local variable;

$$\mathcal{M}_{<i,p>} = (f(t_1, \ldots, t_n)) = \hat{\mathcal{I}}(f)(\mathcal{M}_{<i,p>}(t_1), \ldots, \mathcal{M}_{<i,p>}(t_n)).$$

Given the above function, we can now define the meaning of formulas. A formula of FDSL is interpreted with respect to a model $\mathcal{M}=(\mathrm{N},\mathrm{D},\Sigma,\mathcal{L})$ as defined above. We adopt the following conventions. The relation $\mathcal{C}' \sim_x \mathcal{C}$ is defined to be true iff global interpretations $\mathcal{C}'$ and $\mathcal{C}$ are exactly the same except (possibly)

for the value they assign to variable x. We extend $\hat{\mathcal{P}}$ to the domain L^* such that $\hat{\mathcal{P}}(p,\varepsilon) = p$ and $\hat{\mathcal{P}}(p,\ell_1 \cdot \ell_2) = \hat{\mathcal{P}}(\hat{\mathcal{P}}(p,\ell_1),\ell_2)$, where ε is the empty string. For natural numbers i and j, $\mathcal{M}^{(i:j)}$ denotes the model obtained from $\mathcal{M}$ by taking the initial distributed state to be σ_i and the sequence of distributed states to be $(\sigma_i, \sigma_{i+1}, \ldots, \sigma_j)$. Finally, we write $N = N_1 // N_2$ to denote the partitions N_1 and N_2 of network N where $N_i =< L, P_i, \hat{\mathcal{P}}_i >$ $(i = 1,2)$ such that $(P_1 \cap P_2) = \emptyset$ and

$$\hat{\mathcal{P}}_i(p,\ell) = \begin{cases} \bot & \text{if } \hat{\mathcal{P}}(p,\ell) = p' \text{ and } p' \in P_{3-i} \\ \hat{\mathcal{P}}(p,\ell) & \text{otherwise} \end{cases} \qquad i = 1,2$$

We define a satisfiability relation, $\models$, which denotes the truth of a formula in an interpretation. We write $\mathcal{M}_i, p \models \varphi$ to mean that formula φ is true in model $\mathcal{M}$ at time i for process p. The satisfiability relation is defined inductively as follows.

$\mathcal{M}_i, p \models \mathrm{q}$ iff $\hat{\mathcal{I}}(\mathrm{q})$, for global atomic proposition q;

$\mathcal{M}_i, p \models \mathrm{q}$ iff $\mathcal{I}(\sigma_i(p))(\mathrm{q})$, for local atomic proposition q;

$\mathcal{M}_i, p \models r(t_1, \ldots, t_n)$ iff $(\mathcal{M}_{<i,p>}(t_1), \ldots, \mathcal{M}_{<i,p>}(t_n)) \in \hat{\mathcal{I}}(r)$,
for global relation r;

$\mathcal{M}_i, p \models r(t_1, \ldots, t_n)$ iff $(\mathcal{M}_{<i,p>}(t_1), \ldots, \mathcal{M}_{<i,p>}(t_n)) \in \mathcal{I}(\sigma_i(p))(r)$,
for local relation r;

$\mathcal{M}_i, p \models t_1 = t_2$ iff $\mathcal{M}_{<i,p>}(t_1) = \mathcal{M}_{<i,p>}(t_2)$;

$\mathcal{M}_i, p \models \varphi \wedge \psi$ iff $\mathcal{M}_i, p \models \varphi$ and $\mathcal{M}_i, p \models \psi$;

$\mathcal{M}_i, p \models \neg\varphi$ iff $\mathcal{M}_i, p \not\models \varphi$;

$\mathcal{M}_i, p \models \exists x \psi$ iff $\mathcal{M}'_i, p \models \psi$ for some $\mathcal{M}'$ with $\mathcal{C}' \sim_x \mathcal{C}$.

$\mathcal{M}_i, p \models \odot\varphi$ iff $i < |\Sigma|$ and $\mathcal{M}_{i+1}, p \models \varphi$;

$\mathcal{M}_i, p \models \Diamond\varphi$ iff $\exists j, i \leq j \leq |\Sigma|$, such that $\mathcal{M}_j, p \models \varphi$;

$\mathcal{M}_i, p \models \boxdot\varphi$ iff $\forall j, i \leq j \leq |\Sigma|$, $\mathcal{M}_j, p \models \varphi$;

$\mathcal{M}_i, p \models \varphi\,;\psi$ iff $\exists j, i \leq j \leq |\Sigma|$ such that $\mathcal{M}_0^{(i:j)}, p \models \varphi$ and $\mathcal{M}_0^{(j:|\Sigma|)}, p \models \psi$;

$\mathcal{M}_i, p \models \varphi\,\mathbf{atnext}\,\psi$ iff $\mathcal{M}_j, p \models \varphi$ for the smallest $j > i$ with $\mathcal{M}_j, p \models \psi$;

$\mathcal{M}_i, p \models \ell\psi$ iff $\hat{\mathcal{P}}(p, \ell) = p'$ and $\mathcal{M}_i, p' \models \psi$;

$\mathcal{M}_i, p \models \mathbf{nearby}\,\psi$ iff $\forall \ell \in \mathrm{L}$, $\hat{\mathcal{P}}(p, \ell) = p'$ implies $\mathcal{M}_i, p' \models \psi$;

$\mathcal{M}_i, p \models \boxplus\psi$ iff $\forall \tilde{\ell} \in \mathrm{L}^*$, $\hat{\mathcal{P}}(p, \tilde{\ell}) = p'$ implies $\mathcal{M}_i, p' \models \psi$;

$\mathcal{M}_i, p \models \Phi\psi$ iff $\exists \tilde{\ell} \in \mathrm{L}^+$ such that $\hat{\mathcal{P}}(p, \tilde{\ell}) = p'$ and $\mathcal{M}_i, p' \models \psi$;

$\mathcal{M}_i, p \models \varphi\Re\psi$ iff $\mathcal{M}'_i, p' \models \varphi$ and $\mathcal{M}''_i, p'' \models \psi$ where $\mathcal{M}' = (\mathrm{N}_1, \mathrm{D}, \Sigma, \mathcal{L})$ and $\mathcal{M}'' = (\mathrm{N}_2, \mathrm{D}, \Sigma, \mathcal{L})$ for some partition $\mathrm{N} = \mathrm{N}_1 // \mathrm{N}_2$ with $p' \in \mathrm{P}_1$ and $p'' \in \mathrm{P}_2$;

We say that a formula ψ is *satisfiable* if there exists a model $\mathcal{M}$, index i, and a process p such that $\mathcal{M}_i, p \models \psi$. A formula ψ is *valid* if ψ is true in all interpretations. Finally, A formula ψ *follows from a set* $\mathcal{F}$ of formulas, written $\mathcal{M}_i, p \models_{\mathcal{F}} \psi$ if $\mathcal{M}_i, p \models \varphi$ for every $\varphi \in \mathcal{F}$ implies $\mathcal{M}_i, p \models \psi$.

3.5 Example

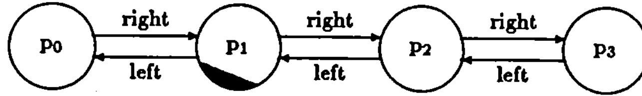

Figure 3.1: Linear array of processors.

Let $\mathrm{N} = (\{\text{left}, \text{right}\}, \{p_0, \ldots, p_3\}, \hat{\mathcal{P}})$, where $\hat{\mathcal{P}}(p_0, \text{right}) = p_1$, $\hat{\mathcal{P}}(p_1, \text{right}) = p_2$, $\hat{\mathcal{P}}(p_2, \text{right}) = p_3$, $\hat{\mathcal{P}}(p_3, \text{left}) = p_2$, $\hat{\mathcal{P}}(p_2, \text{left}) = p_1$, and $\hat{\mathcal{P}}(p_1, \text{left}) = p_0$ be a

description of a network consisting of a linear array of processors (for convenience, the corresponding network diagram is shown in figure 3.1). In addition, let I and J be local variables, **r** a binary predicate symbol, and **f** a binary function symbol. Consider the following two formulas

$$\mathrm{I} = 1 \wedge \mathbf{nearby}\neg\mathrm{r}(\mathrm{I}, \mathrm{J})$$

and

$$\Diamond \odot \mathrm{I} = \mathrm{f}(\mathrm{I}, \mathrm{J}) \Re (\mathrm{J} = 1 \wedge \mathrm{right} \odot \mathrm{J} = 1)$$

in the model $\mathcal{M}$=(N,D,Σ,$\mathcal{L}$) with $\Sigma=(\sigma_0, \sigma_1, \ldots)$, D=$\mathcal{N}$, $\hat{\mathcal{I}}(\mathrm{r})=<$. $\hat{\mathcal{I}}(\mathrm{f})=+$, and the values of variables I and J in states $\sigma_i(p)$ given by to the following tables (one for each process):

p_0	$\sigma_0(p_0)$	$\sigma_1(p_0)$	$\cdots$
I	1	0	$\cdots$
J	1	0	$\cdots$

p_1	$\sigma_0(p_1)$	$\sigma_1(p_1)$	$\cdots$
I	1	0	$\cdots$
J	0	0	$\cdots$

p_2	$\sigma_0(p_2)$	$\sigma_1(p_2)$	$\cdots$
I	1	1	$\cdots$
J	1	0	$\cdots$

p_3	$\sigma_0(p_3)$	$\sigma_1(p_3)$	$\cdots$
I	0	0	$\cdots$
J	0	1	$\cdots$

It can easily be determined that:

$$\mathcal{M}_0, p_1 \models \mathrm{I} = 1 \wedge \mathbf{nearby}\neg\mathrm{r}(\mathrm{I}, \mathrm{J})$$

$$\mathcal{M}_0, p_1 \models \Diamond \odot \mathrm{I} = \mathrm{f}(\mathrm{I}, \mathrm{J}) \Re (\mathrm{J} = 1 \wedge \mathrm{right} \odot \mathrm{I} = 1)$$

3.6 Summary

In this chapter we have presented the syntax and semantics of DSL, the logical formalism which provides the declarative (i.e., logical) meaning of NETLOG programs. In the next chapter, we shall present a different kind of semantics based on rewrite rules; this semantics, in turn, provides the operational meaning of NETLOG programs.

Chapter 4

Specifying and Reasoning about Distributed Programs

Distributed systems are being used to implement applications that are becoming increasingly more sophisticated. Yet specifying and reasoning formally about distributed programs is known to be notoriously difficult[LL87]. Three central issues must be dealt with in designing a language for distributed programming. These are concurrent execution, coordination of the concurrent parts of the program, and the spatial distribution of these concurrent parts over the network of processors. In the first half of this chapter, we compare NETLOG to other logic languages for distributed programming with respect to these three central issues. In the second half of the chapter, we show how one can formally reason about and prove interesting properties of NETLOG programs.

4.1 Specifying Distributed Programs

There are three main issues that must be addressed in a language designed for distributed programming, above and beyond other programming language issues. These are *concurrency*, *coordination* (i.e., synchronization and communication), and *par-*

titioning and assignment (i.e., the spatial assignment of concurrent components to processors).

Almost all logic languages for concurrent and distributed programming provide language constructs to describe the concurrent components of a distributed computation and to describe the synchronization and communication between these components. Few, however, provide corresponding language constructs that allow the assignment of concurrent computations to processors to be described.

In the following sections, we will compare the features of NETLOG to other logic languages for distributed computing with respect to these three central issues. Our comparison will include horn clause based languages such as Concurrent PROLOG[Sha86,Sha87], Shared PROLOG[BC91], Delta Prolog[P+86], and CC[Sar89][1], and temporal logic based languages such as METATEM[BFG+89,FB91,Fis93] and TEMPURA[Mos86,HM87]. As a group, these languages are representative of current approaches to concurrent and distributed logic programming. Our intention is to highlight the fundamental differences between NETLOG and current approaches to distributed logic programming, not to provide a complete description of each of the languages mentioned or to provide a general survey of the field.

Concurrency

Distributed systems have by definition more than one processor therefore it is possible to have different parts of a program executing at the same time. Thus, the first issue that must be addressed by languages for distributed programming is concurrent execution.

An important factor in expressing concurrent execution is the *unit* of concurrency adopted by a given language[B+89]. In NETLOG, sequential actions and constraints are the basic unit of concurrency. Hence, several sequential actions and/or constraints

[1]CC is really a family of concurrent constraint logic programming languages. For our purposes, the distinction between the various members of the family is immaterial. Thus, when referring to these languages, we shall continue to refer to them as though they were a single entity.

may be executed concurrently. Other languages have adopted other approaches. In METATEM, for example, the unit of concurrency is the so called "object" and thus many objects may be active at the same time. In TEMPURA concurrency is based on statements, where several sequential statements can be executed simultaneously. For logic languages such as Concurrent PROLOG and other members of the family of horn-clause based concurrent languages, it is the process, where a process is identified as a single goal. Some logic languages, for example, Concurrent PROLOG, Shared PROLOG, and CC, permit several forms of concurrency within programs. The term *AND/OR-parallelism* has been coined to describe the various forms of concurrency found in these languages. AND-parallelism arises from the concurrent execution of the goals appearing in the body of a guarded horn-clause and OR-parallelism from resolving a single goal with more than one guarded horn-clause. It is not surprising perhaps, given the close similarity between guarded horn-clauses and event handler rules, that NETLOG also allows similar forms of concurrency to be exploited — several events may be executed concurrently and a single event may simultaneously match more than one event handler rule. However, since the rules in NETLOG programs are based on forward-chaining rather than backward-chaining, AND/AND-parallelism would be a more accurate term in the case of NETLOG.

NETLOG also permits a third kind of concurrency that has no analog in horn-clause logic languages. In particular, a *speculative* form of concurrency can be exploited by performing event matching in parallel with event constraint verification. That is, we may begin using a nondeterministic event in forming new matching event collections before the event has been committed. An event collection containing such an event cannot be allowed to activate an event handler, however, until the constraint check is complete and the event is committed. This form of concurrency is called speculative because the event matching that is performed may have to be thrown away if the nondeterministic event fails to be committed. This form of concurrency is not yet exploited in our current implementation.

Coordination

The second issue that must be dealt with in a language for distributed programming is coordination. Coordination involves two types of interaction between the concurrent components of a distributed program: communication and synchronization. A wide variety of shared data and message passing models have been used for communication and synchronization in distributed logic languages.

One shared data model that is used widely in logic languages is the *shared logical variable*. Logical variables have the single assignment property. Once a value has been bound (via unification) to a logical variable it cannot be changed. Concurrent PROLOG and TEMPURA are examples of concurrent languages that use shared logical variables. The concurrent entities in these languages communicate by binding values to the logical variables and synchronize by suspending on unbound variables. Other shared data models that have been used are the *global store* and the *blackboard*. In the constraint based logic language CC, agents communicate by posting constraints on a global store and synchronize by checking that constraints are entailed by the store. Shared PROLOG uses a globally shared data structure called a blackboard; processes communicate and synchronize by adding atoms to, and deleting atoms from the shared blackboard.

A number of logic languages for distributed computing have also been based on message passing models. Programs in METATEM, for example, describe (concurrent) objects that communicate and synchronize using broadcast message passing. Object interface declarations control which broadcasted messages can be received by an object, as well as which messages an object can broadcast (to other objects). Delta PROLOG is another logic language based on message passing. In Delta PROLOG however, message passing is based on synchronous point-to-point communication, where *event goals* are introduced as an explicit mechanism for synchronously exchanging information. In particular, two processes may communicate and synchronize by causing corresponding matching event goals.

One significant difference between these languages and NETLOG is that communication and synchronization in NETLOG is achieved by causing and waiting on events, where constraints are used to provide additional control over the behavior (i.e., occurrence) of nondeterministic events. Events can be used to model a wide variety of communication patterns including broadcast, many-to-one, one-to-many, and point-to-point communication. Consequently, NETLOG is not based on a particular type of communication pattern as is METATEM (broadcast communication) and Delta-PROLOG (Synchronous point-to-point communication). Moreover, the notion of event used in Delta PROLOG is very limited in scope and has little to do with the event model underlying NETLOG.

Another significant difference between NETLOG and other logic languages for distributed computing is NETLOG does not need, nor does it include, any syntactic control constructs or other "extra-logical" features that have no foundation in logic, including the constructs used to express communication and synchronization. In the absence of such extra-logical features, many interesting properties of NETLOG programs can be reasoned about and proved easily. In contrast, concurrent horn-clause languages have introduced the commit operator ('|') and guards to control communication and synchronization. Additional control constructs, such as mode declarations, read only annotations and the like may also be found in these languages. Similarly, METEM includes message interface declarations to control the types of messages an object may send or receive.

It is also interesting to compare the use of constraints in NETLOG and in CC. Both languages use constraints for synchronization and for binding values to variables. In CC, however, constraints are global in scope and can only be imposed on variables, not on agents themselves. Moreover, once a constraint has been imposed it remains in effect forever. This is less powerful than the approach taken in NETLOG, which allows constraints on events as well as variables. In addition, constraints in NETLOG can include both temporal and spatial operators. Using these operators, one can express both permanent constraints and temporary constraints (i.e., constraints that last for

some finite duration); similarly, one can express constraints that are global in scope and ones that are local to some processor.

Partitioning and Assignment

The third central issue that must be addressed by languages for distributed programming is how to distribute the concurrent computations of a distributed program over the network of processors, in other words, which concurrent computation is executed on which processor at any given point in time. The assignment of concurrent computations to processors is often referred to as a *mapping*[AS86]. In general, different applications will require different mapping strategies. Thus, an important choice in the design of a distributed programming language is whether or not the mapping will be under user control.

The design of NETLOG was undertaken from the point of view that the specification of a distributed computation includes specifying the spatial properties that describe the organization of the computation and how it is mapped over the network of processors. Consequently, NETLOG was based on DSL, as opposed to horn-clause logic or temporal logic, since within the framework of DSL such inclusive specifications can be expressed in a concise and elegant fashion. Indeed, the spatial operators of NETLOG are well suited to the task of expressing processor mappings.

In contrast, neither Concurent-PROLOG, CC, METATEM, nor any of the other distributed logic languages being discussed address, at the language level, the issue of how computations are mapped onto processors. In particular, there are no language constructs in these languages to express processor mappings, so users cannot control the mapping of their programs. Instead, these languages rely on the language implementation (i.e., compiler and/or runtime system) to implicitly perform this task. In practice this can be a severe limitation[B+89].

To overcome this limitation, various extensions to some these languages have subsequently been proposed. For example, Shapiro[Sha84] suggests extending Con-

current PROLOG with annotations based on the language LOGO[Pap80] to express the mapping of Concurrent-PROLOG programs onto processors. Similarly, in [Fis93] various extra-logic features are suggested as extensions to METATEM. For example, the concept of "rooms" is proposed as a mechanism for partitioning and grouping objects.

Most of the proposals describing extensions to existing logic languages provide only an incomplete sketch of the intended facilities. One notable exception is Shapiro's annotations for Concurrent Prolog[Sha84]. This being the case, and as annotations can be seen as a general mechanism that could be used to extend any (existing) logic language[2], let us describe Shapiro's approach in more detail and compare it with the integrated approach found in NETLOG.

A specific comparison

Shapiro's approach is to extend Concurrent PROLOG(CP) with annotations derived from the programming language LOGO[Pap80]. These annotations describe how the executing CP program is to be mapped onto a grid of processors (in subsequent references we shall refer to this extended language as annotated-CP). Specifically, every Concurrent PROLOG process has a *position* and a *heading* just like a Turtle in the LOGO programming language. By default, the position and heading of a process are those of its parent (creator), but they can be altered using a sequence of turtle commands. For example, `p(X)@(left(90),forward(1))` describes an Concurrent PROLOG process `p(X)` with annotations `(left(90),forward(1))`. Turtle commands such as `forward` and `backward` take a distance as an argument and change the position of the process accordingly. Commands such as `left` and `right` take angles as arguments and change the heading of the process. Thus, if a process located on processor n and heading northward uses the rule[3]

$$A : -B, C@(\mathrm{left}, \mathrm{forward}(2)), D@(\mathrm{right}, \mathrm{forward})$$

[2] Similar annotations have also been used in the concurrent logic language Strand[FT90]

[3] If angles and distances are omitted, the default is 90 degrees and a distance of 1, respectively.

to reduce process A, then process B is created on processor n, process C is created on the second processor to the west of processor n, and process D is created on the processor to the east of processor n, where B is headed northward, C westward, and D eastward.

As this simple example suggests, both the approach taken by Shapiro and that used in NETLOG are similar in that the mapping of concurrent computations is specified in terms of the structure (i.e., topology) of the underlying distributed system. One advantage of this approach is that it is quite general; it can be used to describe the mapping strategy for distributed architectures having any network topology. Shapiro's annotations, however, reflect a "low-level" approach since they offer little in the way of convenient abstractions which the user can take advantage of. This is undesirable because it forces the user to deal with the details of the topology of the system even if this is not necessary to satisfy the mapping strategy for a particular application.

In comparison, NETLOG provides more flexibility. Indeed, the language contains a range of spatial operators that provide different levels and kinds of abstraction with respect to the topology of the underlying distributed architecture. In addition to operators that express lower-level concepts such as `left` and `right`, NETLOG includes operators that can express abstract concepts such as "everywhere," "somewhere," "nearby," "not elsewhere," etc. These latter operators can be used to express mapping strategies in more abstract terms without reference to a specific architecture or topology. A simple example of this is illustrated in our sum-of-factorials program. In that example, the two factorials can be computed independently (no communication is needed between them); hence, it is desirable to have each factorial mapped to a separate processor, without regard to the specific identity of the processors or their location. This requirement was expressed using the "elsewhere" ($\Diamond$) operator. The resulting program specification is quite general yet captures our intuition rather nicely.

More generally, annotations, and similar extra-logical extensions, have the dis-

advantage that the user must, in effect, deal with two different languages. That is, not having any logical foundations, such extensions have a semantics which is quite different, if not totally divorced, from the rest of the language. Moreover, formal reasoning and verification of programs becomes much more difficult since such extra-logical extensions tend to compromise the declarative properties of logic programs. In the case of Shapiro's annotations in particular, we believe that reasoning about the behavior of a distributed computation in terms of turtles meandering through a network is distracting and rather unnatural at best.

NETLOG, on the other hand, is a single integrated language with a single well-defined logical semantics; all of the constructs and operators in the language have a natural interpretation in DSL. Thus, we can easily reason about and prove interesting properties of NETLOG programs. In addition, as many of the operators in NETLOG correspond to concepts that are familiar to us (e.g., "elsewhere," "everywhere," etc.), they allow us to express the mapping of a distributed computation using concepts that closely reflect our natural intuition (as opposed to unrelated concepts such as wandering turtles).

Example

To further illustrate some of the ideas that have been discussed in this and preceding sections, let us consider a specific example. Consider the simple network shown in figure 4.1. It consists of eight Processing Elements (PEs) connected in a cube configuration. The network has access to a single Printer (Pr) which is controlled by a print daemon. Let us assume that each PE executes a simple cyclic process that does nothing more than prints its input data. In particular, each process has the following behavior: read the next input, send a print request to the print daemon to have the input data printed, wait for the print request to complete, then repeat the cycle by reading the next input value. The print daemon ensures that only one print request is sent to the printer at any one time; once a print request has been satisfied,

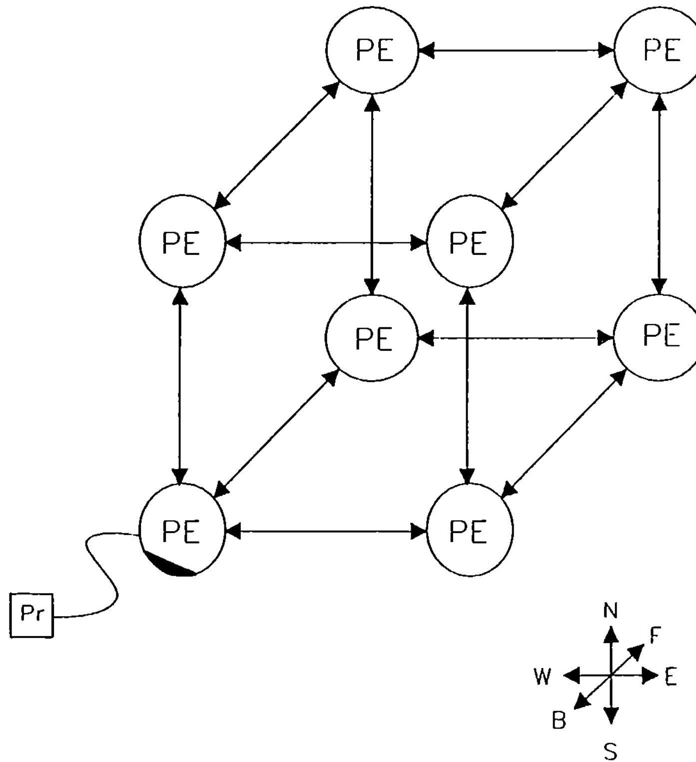

Figure 4.1: Multiple PE/single printer network.

it sends a reply to the originating process to inform the process that its data has been printed. The print daemon runs on the PE connected to the printer.

Figures 4.2–4.5 shows how one might specify (or attempt to specify) this network of distributed processes in NETLOG and a representative subset of the logic languages discussed earlier. Figure 4.2 shows a GHC program[4] for this problem. In GHC (and Concurrent Prolog) one cannot synchronize multiple appends to a single channel (i.e., stream), so each process must have its own channel (e.g., S0,...,S7) to the

[4]With little or no change, similar programs could be written in PARLOG, CC, STRAND, and other members of the concurrent horn clause logic languages.

```
proc(Id,S) :- true | read(X), S=[msg([print,X],Id,R)|Ms], wait(Id,R,Ms).
wait(Id,R,S) :- R=done | proc(Id,S).
pdaemon([msg([print,X],Id,R)|S0],S1,S2,S3,S4,S5,S6,S7) :-
        true | printing(Id,X,R), pdaemon(S0,S1,S2,S3,S4,S5,S6,S7).
pdaemon(S0,[msg([print,X],Id,R)|S1],S2,S3,S4,S5,S6,S7) :-
        true | printing(Id,X,R), pdaemon(S0,S1,S2,S3,S4,S5,S6,S7).
pdaemon(S0,S1,[msg([print,X],Id,R)|S2],S3,S4,S5,S6,S7) :-
        true | printing(Id,X,R), pdaemon(S0,S1,S2,S3,S4,S5,S6,S7).
pdaemon(S0,S1,S2,[msg([print,X],Id,R)|S3],S4,S5,S6,S7) :-
        true | printing(Id,X,R), pdaemon(S0,S1,S2,S3,S4,S5,S6,S7).
pdaemon(S0,S1,S2,S3,[msg([print,X],Id,R)|S4],S5,S6,S7) :-
        true | printing(Id,X,R), pdaemon(S0,S1,S2,S3,S4,S5,S6,S7).
pdaemon(S0,S1,S2,S3,S4,[msg([print,X],Id,R)|S5],S6,S7) :-
        true | printing(Id,X,R), pdaemon(S0,S1,S2,S3,S4,S5,S6,S7).
pdaemon(S0,S1,S2,S3,S4,S5,[msg([print,X],Id,R)|S6],S7) :-
        true | printing(Id,X,R), pdaemon(S0,S1,S2,S3,S4,S5,S6,S7).
pdaemon(S0,S1,S2,S3,S4,S5,S6,[msg([print,X],Id,R)|S7]) :-
        true | printing(Id,X,R), pdaemon(S0,S1,S2,S3,S4,S5,S6,S7).
pdaemon([],[],[],[],[],[],[],[]).
printing(Id,X,R) :- true | write(Id:X), R=done.
start :- true | proc(0,S0), proc(1,S1), proc(2,S2), proc(3,S3),
                proc(4,S4), proc(5,S5), proc(6,S6), proc(7,S7),
                pdaemon(S0,S1,S2,S3,S4,S5,S6,S7).
```

Figure 4.2: GHC program for multiple PE/single printer network.

proc0(msg)[msg]:
  start $\Longrightarrow$ $\Diamond$read(X)
  $\ominus$read(X) $\Longrightarrow$ msg([print,X],0)
  $\ominus$msg(done,0) $\Longrightarrow$ $\Diamond$read(X)

proc1(msg)[msg]:
  start $\Longrightarrow$ $\Diamond$read(X)
  $\ominus$read(X) $\Longrightarrow$ msg([print,X],1)
  $\ominus$msg(done,1) $\Longrightarrow$ $\Diamond$read(X)

proc2(msg)[msg]:
  start $\Longrightarrow$ $\Diamond$read(X)
  $\ominus$read(X) $\Longrightarrow$ msg([print,X],2)
  $\ominus$msg(done,2) $\Longrightarrow$ $\Diamond$read(X)

proc3(msg)[msg]:
  start $\Longrightarrow$ $\Diamond$read(X)
  $\ominus$read(X) $\Longrightarrow$ msg([print,X],3)
  $\ominus$msg(done,3) $\Longrightarrow$ $\Diamond$read(X)

$\ddots$

proc7(msg)[msg]:
  start $\Longrightarrow$ $\Diamond$read(X)
  $\ominus$read(X) $\Longrightarrow$ msg([print,X],7)
  $\ominus$msg(done,7) $\Longrightarrow$ $\Diamond$read(X)

pdaemon(msg)[msg]:
  msg([print,X],Id) $\Longrightarrow$ $\Diamond$printing(Id,X)
  $\ominus$printing(Id,X) $\Longrightarrow$ write(Id:X) $\wedge$ $\Diamond$msg(done,Id)
  printing(Id1,X) $\wedge$ printing(Id2,Y) $\Longrightarrow$ Id1=Id2

Figure 4.3: METATEM program for multiple PE/single printer network.

print daemon along which it sends and receives messages. While the intent of the GHC program is that each of the goals proc(0,S0),···,proc(7,S7) (and the sub goals spawned by them during a reduction) should be executed on a different PE, there is no mechanism in GHC to enforce this. Indeed, the semantics of the language permits a goal to be reduced on any processor — thus, for example, there is no way to prevent read(X) goals for different process streams being reduced on the same processor! Similar limitations are apparent in the METATEM program given in figure 4.3. The syntax of METATEM, like GHC, lacks constructs to control which objects execute on which PEs, so one cannot stipulate, for example, that the print daemon object should execute on the PE connected to the printer.

Figures 4.4 and 4.5 show the annotated Concurrent Prolog and NETLOG programs, respectively[5]. Unlike the programs in GHC and METATEM, both these programs express how the parts of the computation are to be mapped to the PEs. The fundamental difference between the two programs lies in the higher-level description provided by the NETLOG solution. One aspect of this can be seen by observing that the number of rules in the NETLOG program does not depend on the size (i.e., number of PEs) or the topology of the network. An advantage of this is that, even for networks with irregular connections and/or large numbers of PEs, one can quickly construct a high-level NETLOG description of the network that is both concise and easy to comprehend. In comparison, the annotated-CP program reflects the low-level details of the network. For example, the topology of the network is reflected in the mapping of individual goals to processors (i.e., proc(1,S1) @ **north**, proc(1,S2) @ (**north**, **forward**), etc.)[6]. Moreover, the number of clauses in the pdaemon procedure (and the number of arguments in a pdaemon goal) grows with the size of the network. Clearly, for an underlying network with more complex connections and/or large (or

[5]Several predefined state variables are used in the NETLOG program. In particular, each processor can access state variable Np′ to reference the total number of processors in the network; similarly, each processor can access state variable Pid′ to reference an integer (between 0 and Np′-1) which uniquely identifies the executing processor.

[6]The number so mapped is proportional to the size of the network.

```
proc(Id,S) :- read(X), S=[msg([print,X],Id,R)|Ms], wait(Id,R?,Ms).
wait(Id,done,S) :- proc(Id,S).
pdaemon([msg([print,X],Id,R)|S0],S1,S2,S3,S4,S5,S6,S7) :-
        printing(Id,X,R), pdaemon(S0?,S1,S2,S3,S4,S5,S6,S7).
pdaemon(S0,[msg([print,X],Id,R)|S1],S2,S3,S4,S5,S6,S7) :-
        printing(Id,X,R), pdaemon(S0,S1?,S2,S3,S4,S5,S6,S7).
pdaemon(S0,S1,[msg([print,X],Id,R)|S2],S3,S4,S5,S6,S7) :-
        printing(Id,X,R), pdaemon(S0,S1,S2?,S3,S4,S5,S6,S7).
pdaemon(S0,S1,S2,[msg([print,X],Id,R)|S3],S4,S5,S6,S7) :-
        printing(Id,X,R), pdaemon(S0,S1,S2,S3?,S4,S5,S6,S7).
pdaemon(S0,S1,S2,S3,[msg([print,X],Id,R)|S4],S5,S6,S7) :-
        printing(Id,X,R), pdaemon(S0,S1,S2,S3,S4?,S5,S6,S7).
pdaemon(S0,S1,S2,S3,S4,[msg([print,X],Id,R)|S5],S6,S7) :-
        printing(Id,X,R), pdaemon(S0,S1,S2,S3,S4,S5?,S6,S7).
pdaemon(S0,S1,S2,S3,S4,S5,[msg([print,X],Id,R)|S6],S7) :-
        printing(Id,X,R), pdaemon(S0,S1,S2,S3,S4,S5,S6?,S7).
pdaemon(S0,S1,S2,S3,S4,S5,S6,[msg([print,X],Id,R)|S7]) :-
        printing(Id,X,R), pdaemon(S0,S1,S2,S3,S4,S5,S6,S7?).
pdaemon([],[],[],[],[],[],[],[]).
printing(Id,X,R) :- write(Id:X) @ printer, R=done.
start :- proc(0,S0), proc(1,S1) @ north,
      proc(1,S2) @ (north, forward),
      proc(1,S3) @ forward, proc(1,S4) @ (forward, west),
      proc(1,S5) @ (forward, west, north), proc(6,S6) @ (west, north),
      proc(7,S7) @ west, pdaemon(S0?,S1?,S2?,S3?,S4?,S5?,S6?,S7?).
```

Figure 4.4: Annotated Concurrent Prolog program for multiple PE/single printer network.

```
    { proc(Id) ⟹ ◇read(X) ; ⊞msg([print,X],Id) ; wait msg(done,Id) ; proc(Id),
      pdaemon ⟹ startdaemons(0,Np′),
      pdaemon ⟹ ⊡¬(printing(Id1,X) ∧ printing(Id2,Y) ∧ Id1≠Id2),
      startdaemons(I,J) ∧ I<J ⟹ daemon(I) ; startdaemons(I+1,J),
      daemon(Id) ⟹ wait msg([print,X],Id) ; printing(Id,X) ; daemon(Id),
      printing(Id,X) ⟹ printer write(Id:X) ; ⊞msg(done,Id)
    }
  assert
    pdaemon ; ⊞proc(Pid′)
```

Figure 4.5: NETLOG program for multiple PE/single printer network.

even moderate) numbers of PEs, the annotated-CP solution would quickly grow in size, becoming large and unwieldy not to mention more difficult to comprehend and verify correct.

Other approaches

From a broader perspective, alternative approaches to mapping a computation onto a network of processors exist. For example, one common approach used in programming languages based on other programming paradigms (e.g., functional programming, imperative programming, etc.) is to assign each processor a unique identifier (usually an integer) and use some type of annotation to express the mapping of the computation in terms of these processor identifiers. This is the approach taken in Paralfl[HS86b], PCN[F^+92], and Vienna FORTRAN[CMZ92]. Many of the comments made earlier apply here also. In particular, concepts like "somewhere" and "nearby" cannot be directly expressed in these languages. Nor can one express the distribution from the perspective of where not to map things (e.g., "not somewhere"). Besides NETLOG, We know of no other language that offers this kind of expressiveness and flexibility. The ability to reason formally about the distributed properties of programs is also

not a central concern in most of these languages.

```
    { infac(X) ⟹ fac(X,1,1),
      fac(X,N,R) ∧ N<X ⟹ ⊙fac(X,N+1,(N+1)×R),
      fac(X,N,R) ∧ N≥X ⟹ outfac(X,R),
      outfac(X,R) ⟹ ⊡¬(⟐outfac(X,Z) ∧ R≠Z) ,
    }
assert
    infac(2) ∧
    ⟐infac(7) ∧
    wait (outfac(2,X) also ⟐outfac(7,Y)) ;write(X+Y)
```

Figure 4.6: Computing the sum of factorials.

4.2 Reasoning about Distributed Programs

NETLOG programs are pure logic programs in the sense that every NETLOG program is a formula in DSL; moreover, the operational interpretation of the constructs in the language preserves the declarative meaning of the constructs as defined by the model semantics given in chapter 3. Accordingly, we can use the declarative (i.e., model) semantics of NETLOG to reason about and prove interesting properties of NETLOG programs[7].

We will illustrate the approach by proving the (partial) correctness of the sum-of-factorials program given in chapter 2 (for convenience, the program is reproduced in figure 4.6). The correctness of the sum-or-factorials program can be expressed by the following property

$$\models \underline{P}_{fac} \longrightarrow \Diamond \text{write}(2! + 7!)$$

[7] A formal proof of the correspondence between the operational and declarative semantics may be found in chapter 5.

Here, $\underline{\mathrm{P}}_{fac}$ is the DSL formula denoted by our program. Intuitively, this property states that any computation satisfying our sum-of-factorials program (i.e., any successful computation) eventually writes the sum of 2! and 7! as output.

To prove the validity of this property, we must show that any model $\mathcal{M}$ satisfying $\underline{\mathrm{P}}_{fac}$ also satisfies $\Diamond\mathrm{write}(2! + 7!)$; in other words, for arbitrary $\mathcal{M}$, i, and p if $\mathcal{M}_i, p \models \underline{\mathrm{P}}_{fac}$ then $\mathcal{M}_i, p \models \Diamond\mathrm{write}(2! + 7!)$. We proceed by first proving the validity of several intermediate properties; our main correctness property then follows as a direct consequence. In the derivations to follow, let

$$
\begin{aligned}
\underline{\mathrm{P}}_{fac} &\equiv \underline{\mathrm{R}}_1 \wedge \underline{\mathrm{R}}_2 \wedge \underline{\mathrm{R}}_3 \wedge \underline{\mathrm{R}}_4 \wedge \mathrm{A} \\
\underline{\mathrm{R}}_1 &\equiv \boxdot\boxplus\forall X(\mathrm{infac}(X) \longrightarrow \mathrm{fac}(X, 1, 1)) \\
\underline{\mathrm{R}}_2 &\equiv \boxdot\boxplus\forall X, N, R(\mathrm{fac}(X, N, R) \wedge N < X \longrightarrow \odot\mathrm{fac}(X, N+1, (N+1) \times R)) \\
\underline{\mathrm{R}}_3 &\equiv \boxdot\boxplus\forall X, N, R(\mathrm{fac}(X, N, R) \wedge N \geq X \longrightarrow \mathrm{outfac}(X, R)) \\
\underline{\mathrm{R}}_4 &\equiv \boxdot\boxplus\forall X, R(\mathrm{outfac}(X, R) \longrightarrow \boxdot\forall Z(\neg(\Diamond\mathrm{outfac}(X, Z) \wedge R \neq Z))) \\
\mathrm{A} &\equiv \mathrm{A}_1 \wedge \mathrm{A}_2 \wedge \mathrm{A}_3 \\
\mathrm{A}_1 &\equiv \mathrm{infac}(2) \\
\mathrm{A}_2 &\equiv \Diamond\mathrm{infac}(7) \\
\mathrm{A}_3 &\equiv \mathbf{wait}\ (\mathrm{outfac}(2, X)\ \mathbf{also}\ \Diamond\mathrm{outfac}(7, Y))\,;\mathrm{write}(X + Y)
\end{aligned}
$$

In addition, when deriving the proof of some property we will, in order to facilitate derivations, feel free to condense several trivial proof steps to one. For example, when instantiating a rule, we will not hesitate to state that, say,

$$\mathcal{M}_i, p \models \underline{\mathrm{R}}_1 \;\Rightarrow\; \mathcal{M}_i, p \models \mathrm{infac}(2) \longrightarrow \mathrm{fac}(2, 1, 1)$$

is derivable. This fact is intuitively clear and could be formally proved in the following

manner:

$$
\begin{aligned}
\mathcal{M}_i, p \models \underline{\mathrm{R}}_1 \;&\Leftrightarrow\; \mathcal{M}_i, p \models \boxdot\boxplus\forall X(\mathrm{infac}(X) \longrightarrow \mathrm{fac}(X,1,1)) \\
&\Leftrightarrow\; \mathcal{M}_j, p' \models \forall X(\mathrm{infac}(X) \longrightarrow \mathrm{fac}(X,1,1)) \text{ for every } j,\, i \leq j \leq \mid \Sigma \mid \text{, and} \\
&\quad\;\; \text{every } p' \text{ such that } \hat{\mathcal{P}}(p, \ell^*) = p' \text{ for some } \ell^* \in \mathrm{L}^*. \\
&\Rightarrow\; \mathcal{M}_i, p \models \forall X(\mathrm{infac}(X) \longrightarrow \mathrm{fac}(X,1,1)) \text{ for } j = i \text{ and } p' = p. \\
&\Leftrightarrow\; \mathcal{M}'_i, p \models \mathrm{infac}(X) \longrightarrow \mathrm{fac}(X,1,1) \text{ for every } \mathcal{M}' \text{ with } \mathcal{C}' \sim_x \mathcal{C} \\
&\Rightarrow\; \mathcal{M}'_i, p \models \mathrm{infac}(X) \longrightarrow \mathrm{fac}(X,1,1) \text{ for } \mathcal{M}' \text{ with } \mathrm{C}'(X) = \hat{\mathcal{I}}(2) \\
&\Rightarrow\; \mathcal{M}_i, p \models \mathrm{infac}(2) \longrightarrow \mathrm{fac}(2,1,1)
\end{aligned}
$$

Indeed, since our proofs only make use of the semantic definitions of the logical operators[8] and simple mathematical reasoning over the natural numbers, they are quite straightforward.

The first intermediate property we wish to prove is the following:

$$\models \underline{\mathrm{P}}_{\mathrm{fac}} \longrightarrow \Diamond\mathrm{outfac}(2, 2!) \tag{4.1}$$

[8]The reader may wish to review chapter 3 at this point in order to refresh his/her memory regarding the definitions of the logical operators in DSL.

Proof of (4.1):

$$
\begin{aligned}
\mathcal{M}_i, p \models \underline{\mathrm{P}}_{fac} \;\Leftrightarrow\;& \mathcal{M}_i, p \models \underline{\mathrm{R}}_1 \text{ and } \mathcal{M}_i, p \models \underline{\mathrm{R}}_2 \text{ and } \mathcal{M}_i, p \models \underline{\mathrm{R}}_3 \text{ and}\\
& \mathcal{M}_i, p \models \underline{\mathrm{R}}_4 \text{ and } \mathcal{M}_i, p \models \mathrm{A}\\
\Rightarrow\;& \mathcal{M}_i, p \models \underline{\mathrm{R}}_1 \text{ and } \mathcal{M}_i, p \models \underline{\mathrm{R}}_2 \text{ and } \mathcal{M}_i, p \models \underline{\mathrm{R}}_3 \text{ and}\\
& [\mathcal{M}_i, p \models \mathrm{A}_0 \text{ and } \mathcal{M}_i, p \models \mathrm{A}_1 \text{ and } \mathcal{M}_i, p \models \mathrm{A}_2]\\
\Rightarrow\;& \mathcal{M}_i, p \models \mathrm{infac}(2) \longrightarrow \mathrm{fac}(2,1,1) \text{ and}\\
& \mathcal{M}_i, p \models \mathrm{fac}(2,1,1) \wedge 1 < 2 \longrightarrow \odot\mathrm{fac}(2, 1+1, (1+1)\times 1) \text{ and}\\
& [i+1 >\mid \Sigma \mid \text{ or } \mathcal{M}_{i+1}, p \models \mathrm{fac}(2,2,2) \wedge 2 \geq 2 \longrightarrow \mathrm{outfac}(2,2)] \text{ and}\\
& \mathcal{M}_i, p \models \mathrm{infac}(2)\\
\Leftrightarrow\;& [\mathcal{M}_i, p \models \mathrm{infac}(2) \text{ and } \mathcal{M}_i, p \models \mathrm{infac}(2) \longrightarrow \mathrm{fac}(2,1,1)] \text{ and}\\
& \mathcal{M}_i, p \models \mathrm{fac}(2,1,1) \wedge 1 < 2 \longrightarrow \odot\mathrm{fac}(2,2,2) \text{ and}\\
& [i+1 >\mid \Sigma \mid \text{ or } \mathcal{M}_{i+1}, p \models \mathrm{fac}(2,2,2) \wedge 2 \geq 2 \longrightarrow \mathrm{outfac}(2,2)]\\
\Rightarrow\;& \mathcal{M}_i, p \models \mathrm{fac}(2,1,1) \text{ and}\\
& \mathcal{M}_i, p \models \mathrm{fac}(2,1,1) \wedge 1 < 2 \longrightarrow \odot\mathrm{fac}(2,2,2) \text{ and}\\
& [i+1 >\mid \Sigma \mid \text{ or } \mathcal{M}_{i+1}, p \models \mathrm{fac}(2,2,2) \wedge 2 \geq 2 \longrightarrow \mathrm{outfac}(2,2)]\\
\Leftrightarrow\;& [\mathcal{M}_i, p \models \mathrm{fac}(2,1,1) \text{ and } \mathcal{M}_i, p \models \mathrm{fac}(2,1,1) \wedge 1 < 2 \longrightarrow \odot\mathrm{fac}(2,2,2)] \text{ and}\\
& [i+1 >\mid \Sigma \mid \text{ or } \mathcal{M}_{i+1}, p \models \mathrm{fac}(2,2,2) \wedge 2 \geq 2 \longrightarrow \mathrm{outfac}(2,2)]\\
\Rightarrow\;& \mathcal{M}_i, p \models \odot\mathrm{fac}(2,2,2) \text{ and}\\
& [i+1 >\mid \Sigma \mid \text{ or } \mathcal{M}_{i+1}, p \models \mathrm{fac}(2,2,2) \wedge 2 \geq 2 \longrightarrow \mathrm{outfac}(2,2)]\\
\Leftrightarrow\;& [i+1 \leq\mid \Sigma \mid \text{ and } \mathcal{M}_{i+1}, p \models \mathrm{fac}(2,2,2)] \text{ and}\\
& [i+1 >\mid \Sigma \mid \text{ or } \mathcal{M}_{i+1}, p \models \mathrm{fac}(2,2,2) \wedge 2 \geq 2 \longrightarrow \mathrm{outfac}(2,2)]\\
\Rightarrow\;& \mathcal{M}_{i+1}, p \models \mathrm{fac}(2,2,2) \text{ and } \mathcal{M}_{i+1}, p \models \mathrm{fac}(2,2,2) \wedge 2 \geq 2 \longrightarrow \mathrm{outfac}(2,2)\\
\Rightarrow\;& \mathcal{M}_{i+1}, p \models \mathrm{outfac}(2,2)\\
\Rightarrow\;& \exists j, i \leq j \leq\mid \Sigma \mid, \text{ and } \mathcal{M}_j, p \models \mathrm{outfac}(2, 2!)\\
\Leftrightarrow\;& \mathcal{M}_j, p \models \Diamond\mathrm{outfac}(2, 2!)
\end{aligned}
$$

In an analogous manner, we can prove that the following intermediate property is also valid

$$\models \mathrm{P}_{fac} \longrightarrow \Diamond\Diamond\mathrm{outfac}(7, 7!) \tag{4.2}$$

We omit the details. The next intermediate property we wish to prove is the following

$$\models \mathrm{P}_{fac} \longrightarrow \diamondcdot\mathrm{outfac}(2,X) \wedge \diamondcdot\diamondplus\mathrm{outfac}(7,Y) \wedge \diamondcdot\mathrm{write}(X+Y) \tag{4.3}$$

Proof of (4.3):

$$
\begin{aligned}
\mathcal{M}_i, p \models \underline{\mathrm{P}}_{fac} &\Leftrightarrow \mathcal{M}_i, p \models \underline{\mathrm{R}}_1 \text{ and } \cdots \text{ and } \mathcal{M}_i, p \models \underline{\mathrm{R}}_4 \text{ and } \mathcal{M}_i, p \models \mathrm{A} \\
&\Rightarrow \mathcal{M}_i, p \models \mathrm{A} \\
&\Leftrightarrow \mathcal{M}_i, p \models \mathrm{A}_0 \text{ and } \mathcal{M}_i, p \models \mathrm{A}_1 \text{ and } \mathcal{M}_i, p \models \mathrm{A}_2 \\
&\Rightarrow \mathcal{M}_i, p \models \mathrm{A}_2 \\
&\Leftrightarrow \exists k, i \leq k \leq \mid \Sigma \mid, \text{ such that} \\
&\qquad \mathcal{M}_0^{(i,k)}, p \models \mathrm{wait}(\mathrm{outfac}(2,X) \text{ also } \diamondplus\mathrm{outfac}(7,Y)) \text{ and} \\
&\qquad \mathcal{M}_0^{(k,|\Sigma|)}, p \models \mathrm{write}(X+Y) \\
&\Rightarrow [\exists j, i \leq j \leq \mid \Sigma \mid, \text{ such that } \mathcal{M}_j, p \models \mathrm{outfac}(2,X) \text{ and} \\
&\qquad \exists l, i \leq l \leq \mid \Sigma \mid, \text{ such that } \mathcal{M}_l, p \models \diamondplus\mathrm{outfac}(7,Y)] \text{ and} \\
&\qquad [\exists k, i \leq k \leq \mid \Sigma \mid, \text{ such that } \mathcal{M}_k, p \models \mathrm{write}(X+Y)] \\
&\Leftrightarrow [\mathcal{M}_i, p \models \diamondcdot\mathrm{outfac}(2,X) \text{ and } \mathcal{M}_i, p \models \diamondcdot\diamondplus\mathrm{outfac}(7,Y)] \text{ and} \\
&\qquad \mathcal{M}_i, p \models \diamondcdot\mathrm{write}(X+Y) \\
&\Leftrightarrow \mathcal{M}_i, p \models \diamondcdot\mathrm{outfac}(2,X) \wedge \diamondcdot\diamondplus\mathrm{outfac}(7,Y) \wedge \diamondcdot\mathrm{write}(X+Y)
\end{aligned}
$$

The last intermediate property we shall use is the following

$$\models \mathrm{P}_{fac} \longrightarrow \boxdot\boxplus\forall X, R, Z(\mathrm{outfac}(X,R) \longrightarrow \boxdot\neg(\diamondplus\mathrm{outfac}(X,Z) \wedge R \neq Z)) \tag{4.4}$$

The proof of 4.4 is trivial; the property follows by definition from $\underline{\mathrm{R}}_4$.

From (4.1)-(4.4), along with some trivial logical reasoning, we get the desired result:

$$\models \underline{\mathrm{P}}_{fac} \longrightarrow \diamondcdot\mathrm{write}(2!+7!) \tag{4.5}$$

To see this intuitively, observe that, due to property (4.4), no two `outfac` events with the same first argument can have different second arguments. Given this fact, and the fact that property (4.1) implies $\diamondcdot$`outfac`(2,2!) while property (4.3) implies $\diamondcdot$`outfac`(2,X), it follows that the only possible value for variable X is 2!. Using

similar reasoning, it is easy to see that Y must be 7!. This, of course, implies (by property 4.3) that we have $\Diamond$`write`(2!+7!).

The approach to proving program properties used above is not the only approach possible. Indeed, although straightforward, directly utilizing the semantic definitions of the logical operators to prove properties of programs often results in proofs that are long and cluttered with low level details involving index manipulations and the like. An alternative approach is to develop a proof theory for an appropriate subset of DSL and use the rules and axioms of the proof theory to prove program properties. Such a proof theory would facilitate shorter more abstract proofs. The development and presentation of such a theory, however, is outside the scope of this thesis and is the subject of future research.

4.3 Summary

In this chapter we have compared NETLOG to other approaches for distributed logic programming and highlighted some of the distinguishing features of the language. The spatial (and temporal) operators of NETLOG permits many distributed computations to be specified in terms of familiar concepts, such as place and time, that reflect our natural intuition. That NETLOG has a well defined semantics and does not include any "extra-logical" constructs or features allows us to formally reason about and prove interesting properties of NETLOG programs.

In the next chapter we will formalize the execution of NETLOG programs by giving a Plotkin style operational semantics for the language.

Chapter 5

Operational Semantics

In this chapter we give a formal operational semantics for NETLOG. The intuitive semantics of NETLOG was given in terms of the constraint based event model (CEM). The formal operational semantics we present in this chapter is different in spirit and more suited to practical implementations. In particular, we present a structural operational semantics for NETLOG (in the style of Plotkin[Plo81,Plo83]) which expresses the relationship between the execution of a NETLOG program and the formal models presented in chapter 3. Specifically, the successful evaluation of a NETLOG program P results in a model $\mathcal{M}$ which satisfies P (i.e., $\mathcal{M}_i, p \models$ P for some i and p).

We also prove the soundness of our operational semantics and illustrate how the semantics can be used to reason about the execution of NETLOG programs.

5.1 Abstract Syntax

The abstract syntax for NETLOG programs is shown in figure 5.1. An informal account of the meaning of the syntactic constructs of the language was given in chapter 2; we do not repeat the exposition here to avoid redundancy.

In addition to being syntactically well-formed, a legal NETLOG program also satisfies the (static) condtion that the first event in a multi-use event descriptor is

$K \in Bcon$ Basic constraints
$E_u \in Uat$ User-defined atomic events
$E_b \in Bat$ Built-in atomic events
$E \in Uat \cup Bat$

(Program)	$P \in PGM$	$::=$	$\{R\}$ assert A
(Rules)	$R \in RULE$	$::=$	$H \Longrightarrow S$
			$\mid H \Longrightarrow C$
			$\mid R_1, R_2$
(Actions)	$A \in ACT$	$::=$	S
			$\mid C$
			$\mid A_1 \wedge A_2$
(Sequential Actions)	$S \in SACT$	$::=$	G
			$\mid W$
			$\mid \odot S$
			$\mid \diamond\!\!\cdot S$
			$\mid S_1 ; S_2$
(Basic Actions)	$G \in GACT$	$::=$	E
			$\mid$ NEARBY E
			$\mid \diamond\!\!\vdash E$
			$\mid \diamond\!\!+ E$
			$\mid \boxplus E$
			$\mid \ell G$
(Constraints)	$C \in CONS$	$::=$	K
			$\mid K$ UNTIL B
			$\mid \boxdot K$
			$\mid \bigcirc C$
(Single-use Event Handler)	$W \in WSHAN$	$::=$	S ATNEXT $B \mid W_1 \vee \cdots \vee W_2$
(Single-use Event Descriptor)	$B \in BDESC$	$::=$	$G_1 \wedge \cdots \wedge G_n \quad n \geq 1$
(Multi-use Event Descriptor)	$H \in HDESC$	$::=$	$E \wedge E_1 \wedge \cdots \wedge E_n \quad n \geq 0$

Figure 5.1: Abstract syntax of NETLOG

always a user defined event while the remaing events (if any) are built-in events.

5.2 Semantics

In this section we define the operational semantics of NETLOG — that is, the effect of executing syntactically correct NETLOG programs. Our operational semantics is specified using transition systems in the Structural Operational Style of Plotkin[Plo81,Plo83] (see also Hennessy[HP79]). In the structural operational approach the semantics of a program is given in terms of the *transitions* it can make from one *configuration* to another. The execution of a program is then modelled by a finite or infinite sequence of configurations which represent successive observable states of the program's execution.

5.2.1 Configurations

In the structural operational approach, configurations typically consist of a syntactic component that represents the program text to be evaluated and one or more additional components that represent the *context* for evaluating the text. The configurations in our setting contain two additional components, a model $\mathcal{M}$ and a process p, as the context for evaluating the program.

Our informal understanding of the operational semantics of a NETLOG program is as follows. The execution of a NETLOG program P will be understood as a finite or infinite sequence $\gamma_{p_0}, \gamma_{p_1}, \ldots, \gamma_{p_k}, \ldots$ of *configurations* such that $\gamma_{p_i} \longrightarrow \gamma_{p_{i+1}}$ and $\longrightarrow$ is the *program transition* relation. The initial configuration γ_{p_0} has the form $\langle \mathrm{A}, \mathcal{M}_0, p\rangle$ where A is the initial action of P, $\mathcal{M}$ gives the initial values for the state and global variables[1] of P, and p is the origin process of the network in $\mathcal{M}$. An intermediate configuration γ_{p_k} has the form $\langle \mathrm{A}', \mathcal{M}'_k, p\rangle$ where A$'$ specifies the actions remaining to be executed and $\mathcal{M}'$ records the events and values of state and

[1]In particular, $\mathcal{M}$ is an inital model with $\Sigma = \sigma_0$, $\mathcal{I}(s)(q) = \emptyset$ for all s and q, and $\mathcal{C}(X) = \bot$ for all global variables X. $\bot$ represents an "undefined" value.

global variables after k program steps have been executed. If execution terminates in configuration γ_{p_n} then $\gamma_{p_n} = \mathcal{M}'$, where $\mathcal{M}'$ records the cumulative history of events and values bound to state and global variables for the entire computation. In particular, $\langle \mathrm{A}, \mathcal{M}_i, p\rangle \longrightarrow \langle \mathrm{A}', \mathcal{M}'_{i+1}, p\rangle$ means in one step of execution of A, $\mathcal{M}$ is transformed into new model $\mathcal{M}'$ with A′ being the remainder of A to be executed. If $\langle \mathrm{A}, \mathcal{M}, p\rangle \longrightarrow \mathcal{M}'$ execution of A transforms $\mathcal{M}$ into final model $\mathcal{M}'$ and then execution ends. (This *imperative* view of the execution of logic programs, where a logic program is viewed as "acting" on a model, is discussed in more detail in [BG88,BFG+89].)

In order to define the program transistion relation $\longrightarrow$, we first define the *state transition* relation $\hookrightarrow$. The state transition relation describes the effect of an individual action (i.e., basic action, sequential action, etc.) on $\mathcal{M}$. A (single) program transition, on the other hand, describes the effects of concurrent actions, that is, the effects of several individual actions simultaneously transforming $\mathcal{M}$. Thus program transistions model true concurrency and, as such, reflect the truly (synchronous) concurrent model of execution embodied by NETLOG programs. Indeed, a single program transition is realized by a sequence of transitions (or *microsteps* as they are called in [BC84]) in the state transition system. Configurations in the state transition relation have the general form $\langle \mathrm{A}, \nabla, \mathcal{M}_k, p\rangle$, where ∇ records the constraints imposed on $\mathcal{M}$ during the k-th program step (the other components are as described earlier).

Thus, in summary, we have the following domains and relations:

$\gamma_p :$	$\Gamma_p = (\mathrm{Act} \times (\overline{\mathrm{M}} \times \aleph) \times \overline{\mathrm{P}}) \cup \overline{\mathrm{M}}$	Program configurations
$\gamma_s :$	$\Gamma_s = (\mathrm{Act} \times \overline{\nabla} \times (\overline{\mathrm{M}} \times \aleph) \times \overline{\mathrm{P}})$	State configurations
	$\longrightarrow \subseteq (\gamma_p \times \gamma_p)$	Program transition relation
	$\hookrightarrow \subseteq (\gamma_s \times \gamma_s)$	State transition relation

Where $\overline{\nabla} \subseteq (\mathrm{Bcon} \times \mathrm{Bcon})$, $\aleph$ is the set of natural numbers, $\overline{\mathrm{M}}$ is the set of all models, and $\overline{\mathrm{P}}$ is the set of all process identifiers.

5.2.2 Notation and Auxiliary Definitions

In this section we define some notational conventions that will prove useful in providing a consise presentation of our transition rules. We also define several auxiliary functions and relations that will be used in our presentation. The following notational conventions will be used:

- θ_s denotes the bindings for the set of global variables $\overline{X} = \{X_1, \ldots, X_n\}$. That is, $\theta_s = \{T_1/X_1, \ldots, T_n/X_n\}$ where $T_1, \ldots, T_n$ are ground terms. $\bar{\theta}$ denotes the set of all such bindings.

- $A\theta_s$ denotes the formula derived from A by substituting T_i for each occurrence of $X_i$in A.

- If A is a formula with global variables $X_1, \ldots, X_n$, $\forall(A)$ and $\exists(A)$ denote $\forall X_1, \ldots, X_n(A)$ and $\exists X_1, \ldots, X_n(A)$, respectively. Similarly, $\exists\theta_s(A)$ denotes $\exists X_1, \ldots, X_n(X_1 = T_1 \wedge \cdots \wedge X_n = T_n \wedge A)$

- $\mathcal{M}_{\mathcal{C} \triangleright \theta_s}$ denotes the model $\mathcal{M}'$ which is exactly the same as $\mathcal{M}$ except that $\mathcal{C}'$ is defined to be $\mathcal{C}'(X) =$ **if** $X \in \overline{X}$ **then** $X\theta_s$ **else** $\mathcal{C}(X)$. Similarly, $\mathcal{M}_{\mathcal{I}_s \triangleright E}$, where $E \equiv Q(T_1, \ldots, T_n)$, stands for the model $\mathcal{M}'$ which is the same as $\mathcal{M}$ except that $\mathcal{I}'$ is defined by $\mathcal{I}'(u)(R) =$ **if** u = s and R = Q **then** $\mathcal{I}(u)(R) \cup \{(T_1, \ldots, T_n)\}$ **else** $\mathcal{I}(u)(R)$.

- Function $\mathtt{path}(N, p_1, p_2)$ returns the shortest path from p_1 to p_2 in network N (the function is undefined if p_1 and p_2 are not connected). Thus, if $N = < L, P, \hat{\mathcal{P}} >$, $\mathtt{path}(N, p_1, p_2) = \ell^*$ implies $\hat{\mathcal{P}}(p_1, \ell^*) = p_2$.

- Relation $A \overset{\ell}{\rightsquigarrow} A'$ is defined to be true if A′ is the same as A except that every basic action in A is now prefixed by ℓ in A′. For example, let $A \equiv p(X) \wedge \Diamond q(Y)$ and $A' \equiv \ell p(X) \wedge \Diamond \ell q(Y)$, then $A \overset{\ell}{\rightsquigarrow} A'$.

Definition 5.2.1 The effects that follow from executing event E on model $\mathcal{M}$ is described by the function "follows from", $\mathtt{Fllws}_{\{R\}}(E, \nabla, \mathcal{M}_i, p)$. The function returns

a pair consisting of a new model $\mathcal{M}'$ that satisfies E and actions A′ (i.e., the continuation) caused by E. In our semantics, this function provides the link with the execution of the rules, {R}, of the NETLOG program.

Let $\texttt{Rmatch}(E', E \wedge E_1 \wedge \cdots \wedge E_k \Longrightarrow A) =$ **if** $\text{match}(E', E) = \theta \wedge E_1\theta \wedge \cdots \wedge E_k\theta$ **then** $A\theta$ **else** $\texttt{true}$ and let $\texttt{action}(E, \{R_1, \ldots, R_n\}) = \text{Rmatch}(E, R_1') \wedge \cdots \wedge \text{Rmatch}(E, R_n')$. (Note, R_j' denotes a new copy of R_j.) We define $\texttt{Fllws}_{\{R\}}$ as follows:

$$\texttt{Fllws}_{\{R\}}(E_b, \nabla, \mathcal{M}_i, p) = \begin{cases} \langle \bigcirc\text{true}, \mathcal{M} \rangle & \text{if } E_b \equiv \text{true} \\ \langle \text{true}, \mathcal{M}' \rangle & \text{if } E_b \not\equiv \text{true and } \tilde{B}(E_b, \mathcal{M}_i, p) = \mathcal{M}' \\ \text{fail} & \text{otherwise} \end{cases}$$

$$\texttt{Fllws}_{\{R\}}(E_u, \nabla, \mathcal{M}_i, p) = \begin{cases} \langle A, \mathcal{M}_{\mathcal{I}_{\sigma_i(p)} \triangleright E'} \rangle & \text{if } \nabla \equiv \nabla_\emptyset \text{ and} \\ & A = \texttt{action}(E', \{R\}) \\ \texttt{Fllws}_{\{R\}}(E_u\theta, \nabla_\emptyset, (\mathcal{M}_{C \triangleright \theta})_i, p) & \text{if } \nabla \not\equiv \nabla_\emptyset \text{ and} \\ & \tilde{C}(E_u, \nabla_t \cup \nabla_p, \mathcal{M}_i, p) = \theta \\ \text{fail} & \text{otherwise} \end{cases}$$

$$\text{where } E' = \mathcal{M}_{<i,p>}(E_u) \qquad \nabla_\emptyset \equiv (\emptyset, \emptyset)$$

The above definition includes several primitive functions: $\tilde{B}(E, \mathcal{M}_i, p)$ is a primitive function that describes the effects of built-in (i.e., pre-defined) events E on model $\mathcal{M}$; it returns a new model $\mathcal{M}'$. $\tilde{C}(E, \overline{K}, \mathcal{M}_i, p)$ is a primitive function that performs constraint checking and returns bindings θ_x imposed by constraints $\overline{K}$ on the free (i.e., unbound) variables (if any) of E[2].

5.2.3 State Transistion System

We now define the rules that describe the state transition system. In general, the rules are structured so that the transitions for compound actions are specifed in terms of transitions involving the immediate subcomponents of the compound action. The

[2] θ_x is the empty binding if there are no unbound variables in E.

rules themselves are largely self explanitory; therefore, we present each rule with minimal commentary.

Basic Actions

In the transitions to follow, we will occasionally need to refer to the network and/or processes associated with model $\mathcal{M}$. In particular, whenever path(N,p,p') and/or $p' \in$ P appears in some transition rule, N refers to the network associated with $\mathcal{M}$ and P is the set of processes. The transition rules for basic actions are as follows.

Local Event:

$$\frac{\text{Fllws}_{\{\text{R}\}}(\text{E}, \nabla, \mathcal{M}_i, p) = \langle \text{A}', \mathcal{M}' \rangle}{\langle \text{E}, \nabla, \mathcal{M}_i, p \rangle \hookrightarrow \langle \text{A}', \nabla, \mathcal{M}'_i, p \rangle} \tag{5.1}$$

If event E satisfies current constraints ∇, causes actions A$'$, and transforms model $\mathcal{M}$ into $\mathcal{M}'$, then execution continues by executing actions A$'$ on model $\mathcal{M}'$.

Somewhere:

$$\frac{p' \in \text{P},\ \text{path}(\text{N}, p, p') = \ell^*,\ \langle \ell^*\text{E}, \nabla, \mathcal{M}_i, p \rangle \hookrightarrow \gamma_s}{\langle \oplus\text{E}, \nabla, \mathcal{M}_i, p \rangle \hookrightarrow \gamma_s} \tag{5.2}$$

Action $\oplus$E has the same effect as executing E on some p' accessible from p (this, of course, includes p since p is trivially accessible from itself).

Elsewhere:

$$\frac{p' \in \text{P},\ p \neq p',\ \text{path}(\text{N}, p, p') = \ell^*,\ \langle \ell^*\text{E}, \nabla, \mathcal{M}_i, p \rangle \hookrightarrow \gamma_s}{\langle \Phi\text{E}, \nabla, \mathcal{M}_i, p \rangle \hookrightarrow \gamma_s} \tag{5.3}$$

Action ΦE has the same effect as executing E on some p' accessible from p, other than p itself (i.e., $p \neq p'$).

Nearby:

$$\frac{\text{A} \equiv \bigwedge_{\ell_i \in \text{L}_p} \ell_i\text{E},\ \langle \text{A}, \nabla, \mathcal{M}_i, p \rangle \hookrightarrow \gamma_s}{\langle \textbf{nearby}\,\text{E}, \nabla, \mathcal{M}_i, p \rangle \hookrightarrow \gamma_s} \tag{5.4}$$

where $L_p = \{\ell \mid p' \in P \wedge \text{path}(N, p, p') = \ell\}$

Action **nearby** E has the same effect as executing E on each p' adjacent to p.

Everywhere:

$$\frac{A \equiv \bigwedge_{\ell_i^* \in L_p} \ell_i^* E, \ \ \langle A, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \gamma_s}{\langle \boxplus E, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \gamma_s} \tag{5.5}$$

where $L_p = \{\ell^* \mid p' \in P \wedge \text{path}(N, p, p') = \ell^*\}$

Action $\boxplus$E has the same effect as executing E on every p' accessible from p (which, of course, includes p itself).

Directed Event:

$$\frac{p' \in P, \ \text{path}(N, p, p') = \ell, \ A \overset{\ell}{\rightsquigarrow} A', \ \langle G, \nabla, \mathcal{M}_i, p'\rangle \hookrightarrow \langle A, \nabla', \mathcal{M}'_i, p'\rangle}{\langle \ell G, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \langle A', \nabla', \mathcal{M}'_i, p\rangle} \tag{5.6}$$

Action ℓG has the same effect as executing action G locally on p' reached by moving along link ℓ.

Note, action A has origin p' since in the antecedent transistion the reference point (i.e., origin) shifts from p to p'. Consequently, action A′ is the same action as A but with respect to the original orign p. For example, if $A \equiv d(1)$ then, with $A \overset{\ell}{\rightsquigarrow} A'$, we get $A' \equiv \ell\, d(1)$. Since from p moving along link ℓ we get back to p', action A′ has the same effect as action A.

Constraints

The transition rules for constraints are described below. It is worth noting that there are no transition rules for constraints of the form $\bigcirc$C. The reason for this should be intuitively clear, namely, such constraints refer to the following state rather than the current state; thus, no rules are needed for such constraints.

To capture the behavior of temporary constraints (i.e., constraints that only hold until some condition is satisfied), the constraints $\nabla = (\nabla_T, \nabla_P)$ imposed on model

$\mathcal{M}$ consists of two parts: ∇_P represents the (permanent) unconditional constraints on $\mathcal{M}$, and ∇_T represents the tentative (i.e., temporary) constraints on $\mathcal{M}$.

Not:

$$\frac{\mathcal{M}_i, p \models \forall(\mathrm{K})}{\langle \mathrm{K}, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \langle \mathrm{true}, \nabla', \mathcal{M}_i, p\rangle} \tag{5.7}$$

where $\nabla' = (\nabla_T, \nabla_P \cup \{\mathrm{K}\})$

After verifying that $\mathcal{M}$ satisfies K, the constraint K can be added to the set of unconditional constraints on $\mathcal{M}$.

Until:

$$\frac{\mathrm{K} \notin \nabla_T,\ \mathcal{M}_i, p \models \forall(\mathrm{K}) \wedge \neg\exists(\mathrm{B})}{\langle \mathrm{K\ UNTIL\ B}, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \langle \mathrm{K\ UNTIL\ B}, \nabla', \mathcal{M}_i, p\rangle} \tag{5.8}$$

where $\nabla' = (\nabla_T \cup \{\mathrm{K}\}, \nabla_P)$

$$\frac{\mathcal{M}_i, p \models \exists(\mathrm{B})}{\langle \mathrm{K\ UNTIL\ B}, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \langle \mathrm{true}, \nabla', \mathcal{M}_i, p\rangle} \tag{5.9}$$

where $\nabla' = (\nabla_T - \{\mathrm{K}\}, \nabla_P)$

If $\mathcal{M}$ satisfies K and condition B is false, add K to the tentetive constraints on $\mathcal{M}$ (if it has not already been added). When condition B becomes true, K is removed from the set of constraints on $\mathcal{M}$.

Always:

$$\frac{\langle \mathrm{K} \wedge \bigcirc\Box\mathrm{K}, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \gamma}{\langle \Box\mathrm{K}, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \gamma} \tag{5.10}$$

Constraint $\Box$K holds for model $\mathcal{M}$ if constraint K holds now and constraint $\Box$K holds next time (if there is a next time).

Sequential Actions

The transistion rules for basic actions have already been given. Thus, we need only give the rules for the remaining categories of sequential actions. Note, however,

for similar reasons to those discussed in the previous section there are no rules for sequential actions of the form ⊙S.

Atnext:

Intuitively, action S of an atnext construct is executed as soon as there is some collection of events matching (i.e., satisfying) condition B. Note, these matching events also generate bindings for the free (i.e., undefined) variables, if any, of B.

$$\frac{\mathcal{C}(\bar{X}) = \bar{\bot},\ \mathcal{M}_i, p \models \exists\theta_x(\mathrm{B})}{\langle \mathrm{S}\,\mathbf{atnext}\,\mathrm{B}, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \langle \mathrm{S}, \nabla, \mathcal{M}'_i, p\rangle} \tag{5.11}$$

where $\mathcal{M}' = \mathcal{M}_{\mathcal{C}\triangleright\theta_x}$

The semantics of event matching can be characterized formally as the condition that $\mathcal{M}$ satisfies B for some binding θ of the free variables (if any) of B; S is then executed in the model $\mathcal{M}'$ in which the free variables have the values determined by θ. The above rule reflects these ideas.

$$\frac{\langle \mathrm{W}_1, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \gamma_s}{\begin{array}{c}\langle \mathrm{W}_1 \vee \mathrm{W}_2, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \gamma_s \\ \langle \mathrm{W}_2 \vee \mathrm{W}_1, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \gamma_s\end{array}} \tag{5.12}$$

The disjunctive **atnext** construct has the same effect as executing any (single) one of its disjuncts.

Sometime:

$$\frac{\langle \mathrm{S}, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \gamma_s}{\langle \diamond\mathrm{S}, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \gamma_s} \tag{5.13}$$

Sequential action ◇S can be executed now, if action S can be executed.

Chop:

Without loss of generality, it is sufficient to consider only sequential actions that begin with either a basic action or an **atnext** construct [3]. We treat each case seperately.

$$\frac{\langle G, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \langle A', \nabla', \mathcal{M}_i', p\rangle}{\langle G\,; S, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \langle A' \wedge \diamond S, \nabla', \mathcal{M}_i', p\rangle} \tag{5.14}$$

$$\frac{\langle W, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \langle S', \nabla', \mathcal{M}_i', p\rangle}{\langle W\,; S, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \langle S'\,; S, \nabla', \mathcal{M}_i', p\rangle} \tag{5.15}$$

The action A′, caused by basic action G, executes in parallel with the remaining sequence S. Notice, S is executed at some future time (which, of course, could include the present). In contrast, the sequential action S′ resulting from the execution of W executes in sequence with S.

<u>Actions</u>

The transistion rules for constraints and sequential actions have already been given. All that remains is to specify the rules for concurrent actions.

And:

$$\frac{\langle A_1, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \langle A_1', \nabla', \mathcal{M}_i', p\rangle}{\begin{array}{c}\langle A_1 \wedge A_2, \nabla, \mathcal{M}_i', p\rangle \hookrightarrow \langle A_1' \wedge A_2, \nabla', \mathcal{M}_i', p\rangle \\ \langle A_2 \wedge A_1, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \langle A_2 \wedge A_1', \nabla', \mathcal{M}_i', p\rangle\end{array}} \tag{5.16}$$

The individual actions comprising a concurrent action can be executed in any order (i.e., standard interleaving semantics).

5.2.4 Program Transistion Relation

Having defined the state transition relation, we can now proceed to define the program transition relation. We will make use of the following definitions.

[3] In particular, due to the equivalences $(\diamond S_1);S_2 \Leftrightarrow \diamond(S_1\,;S_2)$ and $(\odot S_1);S_2 \Leftrightarrow \odot(S_1\,;S_2)$, we can always rewrite a sequential action so that first action is never of the form $\odot S_1$ or $\diamond S_1$

Definition 5.2.2 The predicate *halt* is defined as follows:

- `halt`$(\bigcirc A)$ holds;
- If both `halt`(A_1) and `halt`(A_2), then `halt`$(A_1 \wedge A_2)$ holds.

Definition 5.2.3 We define the *next step* relation, written $A \overset{\mathcal{M}_i,p}{\longmapsto} A'$, to be the least relation closed under the following rules:

- $\bigcirc C \overset{\mathcal{M}_i,p}{\longmapsto} \bigcirc C$
- $\odot S \overset{\mathcal{M}_i,p}{\longmapsto} \odot S$
- $\diamond S \overset{\mathcal{M}_i,p}{\longmapsto} \odot\diamond S$
- $S \,\textbf{atnext}\, B \overset{\mathcal{M}_i,p}{\longmapsto} \odot(S \,\textbf{atnext}\, B)$ if $\mathcal{M}_i, p \models \neg\exists(B)$
- $K \,\textbf{until}\, B \overset{\mathcal{M}_i,p}{\longmapsto} \odot(K \,\textbf{until}\, B)$ if $\mathcal{M}_i, p \models \forall(K) \wedge \neg\exists(B)$
- $A_1 \wedge A_2 \overset{\mathcal{M}_i,p}{\longmapsto} A_1' \wedge A_2'$ if $A_j \overset{\mathcal{M}_i,p}{\longmapsto} A_j'$, $j = 1, 2$

The predicate halt and the next step relation are similar to the *termination* function and *expansion step* relation used in [BC84]. The predicate halt is used to detect when a computation has reached termination (i.e., the actions that remain only apply *if* there is a next step). The next step relation expresses the conditions under which an action A is considered complete for the current excution step, that is, any actions that *must* be executed in the current step have been executed; It also expresses the continuation A' (i.e., strict future-time formula) of action A that must be executed in the next step of the computation.

In addition to the above definitions, for action $A \equiv S_1 \wedge \cdots \wedge S_j \wedge C_1 \wedge \cdots \wedge C_k$, we use the abbreviation $\dot{\oplus}^n A$ to denote $\odot^n S_1 \wedge \cdots \wedge \odot^n S_j \wedge \bigcirc^n C_1 \wedge \cdots \wedge \bigcirc^n C_k$[4].

Program transistion rules

We are now in a position to define the rewrite rules for the program transition relation. Let γ_p and γ_p' be program configurations. We define the program transition relation, $\gamma_p \longrightarrow \gamma_p'$, to be the least relation closed under the following rules:

[4]Recall, $\odot^n$ (resp. $\bigcirc^n$) stands for n occurrences of $\odot$ (resp. $\bigcirc$). For example, $\odot^0 A \equiv A$ and $\odot^2 A \equiv \odot\odot A$.

$$\frac{\text{halt}(A'),\ \langle A, \nabla_\emptyset, \mathcal{M}_i, p\rangle \hookrightarrow^* \langle A', \nabla', \mathcal{M}'_i, p\rangle}{\langle \oplus^i A, \mathcal{M}_i, p\rangle \longrightarrow \mathcal{M}'} \tag{5.17}$$

$$\frac{\neg\text{halt}(A'),\ A' \overset{\mathcal{M}'_i,p}{\longmapsto} \oplus A'',\ \langle A, \nabla_\emptyset, \mathcal{M}_i, p\rangle \hookrightarrow^* \langle A', \nabla', \mathcal{M}'_i, p\rangle}{\langle \oplus^i A, \mathcal{M}_i, p\rangle \longrightarrow \langle \oplus^{i+1} A'', \mathcal{M}''_{i+1}, p\rangle} \tag{5.18}$$

where $\mathcal{M}'' = \mathcal{M}'_{\Sigma \triangleright \text{nstate}(\Sigma, P)}$ and $\nabla_\emptyset \equiv (\emptyset, \emptyset)$.

The first transition rule says that if the actions in A can be executed in some sequential order resulting in halted action A′ and model $\mathcal{M}'$, then in the i-th step of the program action $\oplus^i A$ transforms $\mathcal{M}$ into $\mathcal{M}'$ and execution ends. The second rule is similar but deals with the continuation case. The primitive function nstate(Σ, P) returns a new distributed state σ to be appended to Σ for the set of processes P. For $\Sigma' = (\sigma_0, \ldots, \sigma_k, \sigma)$, the new state σ satisfies the property $\sigma(p) \neq \sigma_i(p)$ for all $i \leq k$ and $p \in P$.

5.3 Example

Our transition system can be used to reason about the operational behavior of NET-LOG programs. For example, consider the following (not particularly meaningful) program:

$$\{\ a(X) \Longrightarrow \odot\Diamond c(X),$$
$$b(X) \Longrightarrow \bigcirc\neg c(X)$$
$$\}$$

assert

$$\boxplus a(1) \wedge b(1)$$

Let $\mathcal{M}$ be the initial model with $\Sigma = \sigma_0$, $\mathcal{I}(s)(q) = \emptyset$ for all s and q, $\mathcal{C}(X) = \bot$ for all X, and N a dyadic network with origin p_0. A possible execution sequence for this program is the following (the numbers in braces identify the state transition rules used to derive each program transition):

$$\begin{array}{rcll} \langle \boxplus a(1) \wedge b(1), \mathcal{M}_0, p_0 \rangle & \longrightarrow & \langle \odot\Diamond\, \mathbf{right}\, c(1) \wedge \odot\Diamond c(1) \wedge \bigcirc\neg c(1), \mathcal{M}'_1, p_0 \rangle & \{5.1, 5.5, 5.6, 5.16\} \\ & \longrightarrow & \langle \odot \odot \Diamond c(1), \mathcal{M}''_2, p_0 \rangle & \{5.1, 5.6, 5.7, 5.13, 5.16\} \\ & \longrightarrow & \mathcal{M}''' & \{5.1, 5.13\} \end{array}$$

where $\mathcal{I}'(\sigma_0(p_0))(a) = \mathcal{I}'(\sigma_0(p_0))(b) = \{(1)\}, \mathcal{I}'(\sigma_0(p_0))(c) = \emptyset, \mathcal{I}'(\sigma_0(p_1))(a) = \{(1)\}, \mathcal{I}'(\sigma_0(p_1))(b) = \mathcal{I}'(\sigma_0(p_1))(c) = \emptyset, \mathcal{C}' = \mathcal{C}[1/X1, 1/X2, 1/X3]$ (X1, X2, and X3 are the new names for X resulting from generating fresh copies of each rule), and $\Sigma' = \Sigma.\sigma_1$. Similarly, $\mathcal{I}''(\sigma_1(p_0))(a) = \mathcal{I}''(\sigma_1(p_0))(b) = \mathcal{I}''(\sigma_1(p_0))(c) = \emptyset, \mathcal{I}''(\sigma_1(p_1))(a) = \mathcal{I}''(\sigma_1(p_1))(b) = \emptyset, \mathcal{I}''(\sigma_1(p_1))(c) = \{(1)\}, \mathcal{C}'' = \mathcal{C}'$, and $\Sigma'' = \Sigma'.\sigma_2$. Finally, $\mathcal{I}'''(\sigma_2(p_0))(a) = \mathcal{I}'''(\sigma_2(p_0))(b) = \emptyset, \mathcal{I}'''(\sigma_2(p_0))(c) = \{(1)\}, \mathcal{I}'''(\sigma_2(p_1))(a) = \mathcal{I}'''(\sigma_2(p_1))(b) = \mathcal{I}'''(\sigma_2(p_1))(c) = \emptyset, \mathcal{C}''' = \mathcal{C}''$, and $\Sigma''' = \Sigma''$.

5.4 Properties of the Operational Semantics

We now state some important properties of our transition systems. In particular, we prove the soundness of NETLOG program derivations (i.e., execution sequences based on NETLOG program transitions). Intuitively, we show that, for any NETLOG program P, a successful terminating execution sequence for P results in a model $\mathcal{M}$ which satisfies P. More specifically, $\mathcal{M}_0, p \models \underline{P}$ (for some p) and $\underline{P}$ is the DSL formula denoted by program P.

We begin by stating two basic properties of the state transistion system.

Theorem 5.4.1 *For any NETLOG program P containing rules* $R \equiv \{R_1, \ldots, R_k\}$, *and configurations* $\gamma_s \equiv \langle A, \nabla, \mathcal{M}_i, p \rangle$ *and* $\gamma'_s \equiv \langle A', \nabla', \mathcal{M}'_i, p \rangle$, *if* $\gamma_s \hookrightarrow \gamma'_s$ *then* $\mathcal{M}'_i, p \models_{\underline{R}} A' \leftrightarrow A$.

Proof: The theorem holds vacuously if $\mathcal{M}'_i, p \not\models \underline{R}_j$ for some $\underline{R}_j \in \underline{R}$. Therefore,

we need only show that the theorem holds when $\mathcal{M}'_i, p \models \underline{\mathrm{R}}_j$ for all $\underline{\mathrm{R}}_j \in \underline{\mathrm{R}}$. We prove the theorem by structural induction on γ_s.

Base case. For the base case we must show that our assertion holds when A is either a basic action, a constraint, or an **atnext** formula.

- Basic actions (G)

  $\mathrm{A} \equiv \mathrm{E}$. This case is defined by transition (5.1). From (5.1) it is easily established that $\langle \mathrm{E}, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \langle \mathrm{A}', \nabla', \mathcal{M}'_i, p\rangle$ iff $\mathcal{M}'_i, p \models \mathrm{E}$. Hence, it follows that $\mathcal{M}'_i, p \models_{\underline{\mathrm{R}}} \mathrm{A}' \rightarrow \mathrm{E}$. To show that $\mathcal{M}'_i, p \models_{\underline{\mathrm{R}}} \mathrm{E} \rightarrow \mathrm{A}'$, we need to consider individually the cases of E a built-in atom and E a user defined atom. If E is a built-in atom then $\mathrm{A}' \equiv \mathbf{true}$ and it follows trivially that $\mathcal{M}'_i, p \models_{\underline{\mathrm{R}}} \mathrm{E} \rightarrow \mathrm{A}'$. If E is a user defined atom then $\mathrm{A}' \equiv \mathrm{A}_1 \wedge \cdots \wedge \mathrm{A}_n$ where $\mathrm{A}_j \equiv \mathbf{true}$ or A_j is the body of rule R_j. For $\mathrm{A}_j \equiv \mathbf{true}$, clearly $\mathcal{M}'_i, p \models_{\underline{\mathrm{R}}} \mathrm{E} \rightarrow \mathrm{A}_j$. In the other case, A_j is the result of E transforming $\mathcal{M}$ to $\mathcal{M}'$ such that $\mathcal{M}'_i, p \models \mathrm{H}_j$ for rule $\mathrm{R}_j \equiv \mathrm{H}_j \Longrightarrow \mathrm{A}_j$. Since we have both $\mathcal{M}'_i, p \models \mathrm{H}_j$ and $\mathcal{M}'_i, p \models \underline{\mathrm{R}}_j$, $\mathcal{M}'_i, p \models \mathrm{A}_j$ follows by universal instantiation and modus ponens; hence, $\mathcal{M}'_i, p \models_{\underline{\mathrm{R}}} \mathrm{E} \rightarrow \mathrm{A}_j$. Since for each A_j, $j = 1, n$, $\mathcal{M}'_i, p \models_{\underline{\mathrm{R}}} \mathrm{E} \rightarrow \mathrm{A}_j$ it follows that $\mathcal{M}'_i, p \models_{\underline{\mathrm{R}}} \mathrm{E} \rightarrow \mathrm{A}_1 \wedge \cdots \wedge \mathrm{A}_n$; that is, $\mathcal{M}'_i, p \models_{\underline{\mathrm{R}}} \mathrm{E} \rightarrow \mathrm{A}'$.

  $\mathrm{A} \equiv \mathbf{nearby}\,\mathrm{E}$, $\mathrm{A} \equiv \diamondplus \mathrm{E}$, $\mathrm{A} \equiv \diamondplus \mathrm{E}$, $\mathrm{A} \equiv \boxplus \mathrm{E}$, $\mathrm{A} \equiv \ell\,\mathrm{G}$. The remaining cases can be proved in an analogous manner. We omit the details.

- Constraints (C)

  $\mathrm{A} \equiv \mathrm{K}$. By transition (5.7) it follows that $\langle \mathrm{K}, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \langle \mathrm{A}', \nabla', \mathcal{M}'_i, p\rangle$ iff $\mathrm{A}' \equiv \mathbf{true}$ and $\mathcal{M}'_i, p \models \mathrm{K}$. Since A' is true in any model $\mathcal{M}$, the assertion $\mathcal{M}'_i, p \models_{\underline{\mathrm{R}}} \mathrm{K} \leftrightarrow \mathrm{A}'$ follows immediately.

  $\mathrm{A} \equiv \mathrm{K}\,\mathbf{until}\,\mathrm{B}$. From transitions (5.8) and (5.9), it is straightforward to establish that $\langle \mathrm{K}\,\mathbf{until}\,\mathrm{B}, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \langle \mathrm{A}', \nabla', \mathcal{M}'_i, p\rangle$ iff ($\mathrm{A}' \equiv \mathbf{true}$ and

$\mathcal{M}'_i, p \models$ B) or (A$' \equiv$ K **until** B and $\mathcal{M}'_i, p \models$ K). In each case, $\mathcal{M}'_i, p \models_{\underline{R}}$ K **until** B $\leftrightarrow$ A$'$ follows immediately.

A $\equiv$ ⊡K. From transitions (5.10) and (5.16), we get $\langle$⊡K$, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \langle$A$', \nabla', \mathcal{M}'_i, p\rangle$ iff A$' \equiv$ true $\wedge$ ○⊡K and $\mathcal{M}'_i, p \models$ K. Clearly, $\mathcal{M}'_i, p \models_{\underline{R}}$ ⊡K $\rightarrow$ A$'$. Now, since $\mathcal{M}'_i, p \models$ K, it follows that $\mathcal{M}'_i, p \models_{\underline{R}}$ A$' \rightarrow$ ⊡K and we are done.

- Atnext (W)

A $\equiv$ S **atnext** B. By transition (5.11) we get $\langle$S **atnext** B$, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \langle$A$', \nabla', \mathcal{M}'_i, p\rangle$ iff A$' \equiv$ S and $\mathcal{M}'_i, p \models$ B. Hence, the assertion $\mathcal{M}'_i, p \models_{\underline{R}}$ S **atnext** B $\leftrightarrow$ A$'$ follows immediately.

A $\equiv$ $W_1 \vee W_2$. By transition (5.12) we get $\langle W_1 \vee W_2, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \langle$A$', \nabla', \mathcal{M}'_i, p\rangle$ iff $\langle W_j, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \langle$A$', \nabla', \mathcal{M}'_i, p\rangle$ for some $j \in \{1,2\}$. By induction on W, it follows that $\mathcal{M}'_i, p \models_{\underline{R}} W_j \leftrightarrow$ A$'$ and $\mathcal{M}'_i, p \models_{\underline{R}} W_1 \vee W_2 \leftrightarrow$ A$'$ follows by the monotonicity of disjunction.

Inductive case. For the inductive case we must show that the theorem holds when A is a compound action.

A $\equiv$ W ; S, A $\equiv$ G ; S, A $\equiv$ ◇S. From the definition of transition (5.14), $\langle$W ; S$, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \langle$A$', \nabla', \mathcal{M}'_i, p\rangle$ iff A$' \equiv$ S$'$; S and $\langle$W$, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \langle$S$', \nabla', \mathcal{M}'_i, p\rangle$. By the inductive hypothesis, $\mathcal{M}'_i, p \models_{\underline{R}}$ W $\leftrightarrow$ S$'$ from which it immediately follows that $\mathcal{M}'_i, p \models_{\underline{R}}$ W ; S $\leftrightarrow$ A$'$. Similarly, the proof of A $\equiv$ G ; S and A $\equiv$ ◇S also follows immediately from the inductive hypothesis. We omit the details.

A $\equiv$ $A_1 \wedge A_2$. By transition (5.16) we get $\langle A_1 \wedge A_2, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \langle$A$', \nabla', \mathcal{M}'_i, p\rangle$ iff A$' \equiv A_{3-j} \wedge A_j$ and $\langle A_j, \nabla, \mathcal{M}_i, p\rangle \hookrightarrow \langle A'_j, \nabla', \mathcal{M}'_i, p\rangle$ for some $j \in \{1,2\}$. Hence by the inductive hypothesis, we have $\mathcal{M}'_i, p \models_{\underline{R}} A_j \leftrightarrow A'_j$ and $\mathcal{M}'_i, p \models_{\underline{R}} A_1 \wedge A_2 \leftrightarrow$ A$'$ follows by the monotonicity of conjunction.

Corollary 5.4.2 *For any NETLOG program P containing rules* $\mathrm{R} \equiv \{\mathrm{R}_1, \ldots, \mathrm{R}_k\}$, *and configurations* $\gamma_s \equiv \langle \mathrm{A}, \nabla, \mathcal{M}_i, p\rangle$ *and* $\gamma'_s \equiv \langle \mathrm{A}', \nabla', \mathcal{M}'_i, p\rangle$, *if* $\gamma_s \hookrightarrow^* \gamma'_s$ *then* $\mathcal{M}'_i, p \models_{\underline{\mathrm{R}}} \mathrm{A}' \leftrightarrow \mathrm{A}$.

Proof: Follows from the corresponding theorem by induction on the length of the state transition sequence.

The following theorem, along with its corollary, establishes the soundness and partial correctness of NETLOG program derivations.

Theorem 5.4.3 *For any NETLOG program P containing rules* $\mathrm{R} \equiv \{\mathrm{R}_1, \ldots, \mathrm{R}_k\}$, *and configurations* $\gamma_p \equiv \langle \mathrm{A}, \mathcal{M}_i, p\rangle$ *and* $\gamma'_p \equiv \langle \mathrm{A}', \mathcal{M}'_i, p\rangle$, *if* $\gamma_p \longrightarrow \gamma'_p$ *then* $\mathcal{M}'_i, p \models_{\underline{\mathrm{R}}} \mathrm{A}' \leftrightarrow \mathrm{A}$. *Furthermore, if* $\gamma_p \longrightarrow \gamma'_p$ *and* $\gamma'_p \equiv \mathcal{M}'$ *then* $\mathcal{M}'_i, p \models_{\underline{\mathrm{R}}} \underline{\mathrm{R}}_1 \wedge \cdots \wedge \underline{\mathrm{R}}_k \wedge \mathrm{A}$

Proof: The proof follows from corollary 5.4.2 using structural induction on γ_p.

Corollary 5.4.4 *Let P be a NETLOG program with rules* $\mathrm{R} \equiv \{\mathrm{R}_1, \ldots, \mathrm{R}_k\}$ *and initial action* A. *Let* $\gamma_p \equiv \langle \mathrm{A}, \mathcal{M}_0, p\rangle$ *be the initial configuration for P, and let* $\gamma'_p \equiv \langle \mathrm{A}', \mathcal{M}'_n, p\rangle$ *be a subsequent configuration, if* $\gamma_p \longrightarrow^* \gamma'_p$ *then* $\mathcal{M}'_0, p \models_{\underline{\mathrm{R}}} \mathrm{A}' \leftrightarrow \mathrm{A}$. *Moreover, if* $\gamma_p \longrightarrow^* \gamma'_p$ *and* $\gamma'_p \equiv \mathcal{M}'_n$ *then* $\mathcal{M}'_0, p \models \underline{\mathrm{R}}_1 \wedge \cdots \wedge \underline{\mathrm{R}}_k \wedge \mathrm{A}$.

Proof: Follows from theorem 5.4.3 by simple induction on the length of the execution sequence.

Although not surprising, corollary 5.4.4 establishes that interesting (non-trivial) properties of NETLOG programs can be shown to hold merely by reasoning with the model theory underlying the program. Indeed, examples of such reasoning were given in chapter 4.

5.5 Summary

A formal operational semantics for NETLOG has been presented and its soundness proved. The operational semantics could be used as the basis for implementing an

interpreter for the language. The rules themselves are somewhat more complex than those for conventional logic languages, but this is hardly surprising given that NETLOG contains many new constructs and features that have no counterpart in conventional languages.

The remainder of this dissertation is devoted to giving programming examples and describing the implementation of NETLOG. Although we will be informal in our description, the precise meaning of all of the examples given in this dissertation can be understood using the formal semantics presented in this chapter.

Chapter 6

Further Examples

In this chapter we illustrate how NETLOG may be used to solve a number of familiar problems. Our aim is not necessarily to present the most concise solutions to the problems described or the most efficient — rather, our goal is to convey some idea of the utility and capability of the language. In so doing, we hope to give an indication of the kinds of problems for which the language seems suitable.

6.1 Example 1: Snow White and the Seven Dwarfs

A distinctive characteristic of many problems in distributed AI is the presence of one or more objects that exhibit intelligent behavior. To give an example of how a simple problem of this kind may be expressed in NETLOG, we consider the problem of Snow White and the Seven dwarfs[FB91]. The problem description is as follows.

Snow White has a bag of sweets. All of the dwarfs want sweets, though some of them more than others. If a dwarf asks Snow White for a sweet, she will give him one, but maybe not straight away. Snow White is only able to give away one sweet at a time. Each dwarf has a particular strategy that it uses in asking for sweets:

Eager initially asks for a sweet and, from then on, whenever he receives a sweet, asks for another.

Mimic asks for a sweet whenever he sees eager asking for one.

Jealous asks for a sweet whenever he sees eager receiving one.

Insistent asks for a sweet as often as he can.

Courteous asks for a sweet only when eager, mimic, jealous, and insistent have all asked for one.

Generous asks for a sweet only when eager, mimic, jealous, insistent, and courteous have all received one.

Shy only asks for a sweet when he sees no one else asking.

Figure 6.1 shows one way to solve this problem in NETLOG (recall, the notation ℓ^n stands for n occurrences of ℓ. For example, **left**^{2} stands for **left left**.). For concreteness, we assume our solution is intended for a network having the topology shown in figure 6.2. As can be seen from figure 6.1, Snow White and the Dwarfs communicate with each other by broadcasting messages across the system. Requests to Snow White are represented by events of the form **ask**(d), where d is the name of the requesting dwarf. Similarly, Snow White gives a sweet to a particular dwarf through events of the form **give**(d), where d is the name of the receiving dwarf.

In contrast to the solution given in [FB91], the NETLOG solution has the property that the pattern of communication between the entities in the program is explicitly reflected in the NETLOG specification itself. This explicitness facilitates reasoning about the correctness of the solution. Indeed, reasoning about the communication within a distributed program can often be one of the most difficult aspects of understanding its behavior. Furthermore, the NETLOG solution makes it clear that the computation is to be distributed across the entire network, one object (i.e., fairy tale character) per processor. An important advantage of being explicit about such matters is that one can specifically rule out any trivial or unanticipated implementations (e.g., mapping all the objects to just one processor!). This issue is ignored in [FB91].

```
{ dwarf(eager) ⟹ ◇⊞ask(eager) ; wait give(eager) ; dwarf(eager),
  dwarf(mimic) ⟹ wait ask(eager) ; ⊞ask(mimic) ; dwarf(mimic),
  dwarf(jealous) ⟹ wait give(eager) ; ⊞ask(jealous) ; dwarf(jealous),
  dwarf(insistent) ⟹ ◇⊞ask(insistent) ; dwarf(insistent),
  dwarf(courteous) ⟹ ◇⊞ask(courteous) ; dwarf(courteous),
  dwarf(courteous) ⟹¬ask(courteous) until ask(eager) ∧
                     ¬ask(courteous) until ask(mimic) ∧
                     ¬ask(courteous) until ask(jealous) ∧
                     ¬ask(courteous) until ask(insistent),
  dwarf(generous) ⟹ ◇⊞ask(generous) ; dwarf(generous),
  dwarf(generous) ⟹¬ask(generous) until give(eager) ∧
                    ¬ask(generous) until give(mimic) ∧
                    ¬ask(generous) until give(jealous) ∧
                    ¬ask(generous) until give(insistent) ∧
                    ¬ask(generous) until give(courteous),
  dwarf(shy) ⟹ ◇⊞ask(shy) ; dwarf(shy),
  dwarf(shy) ⟹ ⊡¬(ask(X) ∧ ask(shy) ∧ X≠shy),
  snowwhite(X) ⟹ wait ask(X) ; ⊞give(X) ; snowwhite(X),
  constraint(snowwhite) ⟹ ⊡¬(give(X) ∧ give(Y) ∧ X≠ Y),
}
assert
  left dwarf(eager) ∧ left² dwarf(mimic) ∧ left³ dwarf(jealous) ∧
  left⁴ dwarf(insistent) ∧ left⁵ dwarf(courteous) ∧ left⁶ dwarf(generous) ∧
  left⁷ dwarf(shy) ∧ snowwhite(eager) ∧ ··· ∧ snowwhite(shy) ∧
  constraint(snowwhite)
```

Figure 6.1: Snow White and the Dwarfs on a ring.

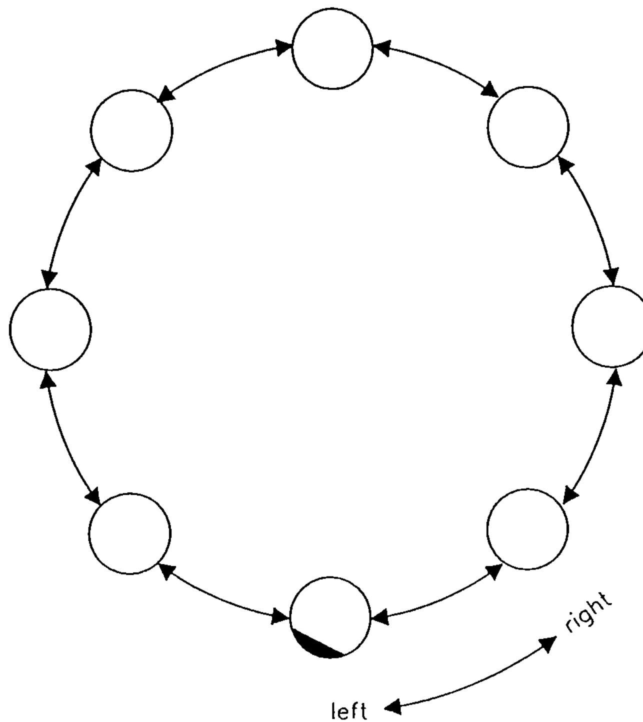

Figure 6.2: An 8-processor ring.

6.2 Example 2: Bounded Buffer message Communication

Interprocessor communication based on message passing primitives such as **send** and **recv** have been used widely in imperative languages for distributed computing (see for example [Hoa78]). Abstractly, we can view the sender and receiver as being connected by a *channel* along which messages are sent between the two. Messages sent by the sender are stored in a finite or *bounded* buffer at the receiver's end of the channel. Thus, in order to avoid message overlays, messages can only be sent if the message

buffer is not full.

Many concurrent logic languages for distributed programming rely on data structures called *streams*[1][Cia92] to (indirectly) model a sequence of messages sent via some channel. NETLOG, however, does not permit the use of streams. It is interesting therefore to see how message passing can be expressed in our language. The approach we take is to simulate the behavior of the channel by specifying how its state changes as a result of `send` and `recv` (the interested reader is encouraged to explore other ways of representing message passing in NETLOG).

Specifically, in this example we show how the behavior of two message passing threads, which communicate via a 1-bounded communications channel (i.e., the message buffer can hold at most 1 message), can be specified in NETLOG; specifying an N-bounded channel is equally straightforward. The two threads, `odd` and `even`, cooperate to compute the squares of the natural numbers according to the formula

$$K^2 = (K-1)^2 + (2K-1) \qquad K \geq 0$$

In our example, we assume the threads are executing on processor p_1 and processor p_6 of the cube architecture shown in figure 6.4. The corresponding program is shown in figure 6.3.

The thread `odd` computes successive odd numbers starting with 1 and sends them to `even`. Thread `even` writes the current square and then computes the next square in the sequence by adding the odd number received (from thread `odd`) to the square just written. Events `send`(X,a) and `recv`(X,a) represent the sending and receiving, respectively, of a message `X` on channel `a`. The state of the channel is represented by events of the form `chan`(Buff,Id), where `Id` is the name of the channel and `Buff` is the buffer.

Constraints are used to stipulate that a `recv` cannot be performed if the buffer is empty and that a successful `recv` always returns the current contents of the buffer. Similarly, a `send` cannot be performed if the buffer is full.

[1] A stream is an incomplete list whose "tail" is an unbound variable

```
  { odd(X) ⟹ send(X,a) ; odd(X+2),
    even(Y) ⟹ write(Y) ; recv(X,a) ; even(X+Y),
    chan(Buff,Id) ⟹ [ chan([],Id) afternext recv(X,Id) ∨
                        chan([X],Id) afternext east north send(X,Id) ],
    chan(Buff,Id) ⟹ (¬recv(X,Id) if Buff≠[X] ∧
                        ¬(east north send(X,Id) ∧ Buff≠[])) until chan(Buf1,Id) ∧ Buf1≠Buff
  }
assert
  chan([],a) ; even(0) ; east north odd(1)
```

Figure 6.3: Bounded Buffer Communication on a 3D-Cube.

6.3 Example 3: Airline Reservation System

Many distributed applications are by nature *reactive*. That is, instead of simply reading a set of inputs and producing, on termination, a set of outputs, reactive systems maintain an ongoing interaction with their environment and usually do not terminate[FB91]. Reactive systems are notoriously difficult to characterize and model formally[LL87].

A well known example of a reactive system is an airline reservation system[BD82]. An airline reservation system can also be viewed as an instance of the readers/writers problem since its operation involves synchronizing the reading and updating of flight information. The reservation system we describe is a simplified system composed of an airline database and a collection of agencies. The database is accessed concurrently by the agencies.

The airline database maintains several flights, each of 300 seats, and can recognize two kinds of requests: `info` and `resv`. An `info` request for flight number `f` returns the current number of available seats for the flight specified. Similarly, a `resv` request for flight number `f` reserves `n` seats on the flight specified, if sufficient seats are available. In particular, a positive acknowledgment, `ack`, is returned if sufficient seats are available and a negative acknowledgment, `nack`, is returned if there are not enough

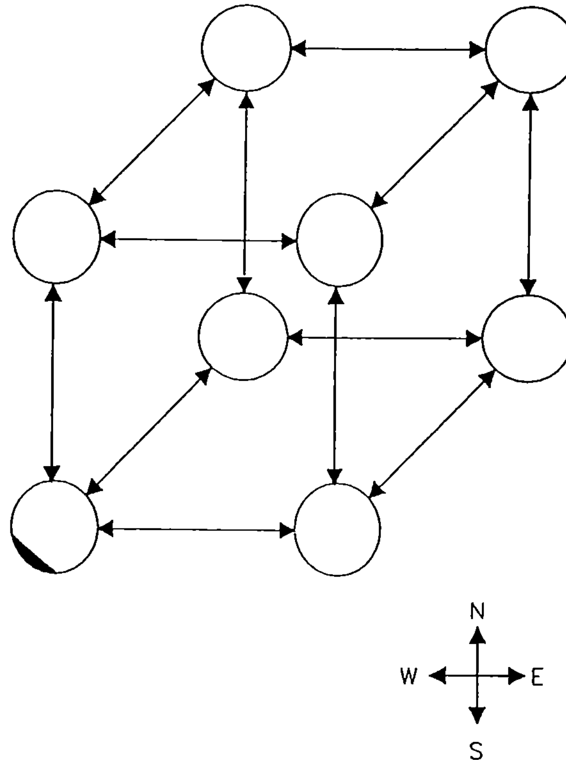

Figure 6.4: 3-D Cube of processors.

seats. Each agency reads a request from the user and queries the airline database. Upon receiving a reply from the database, the agency prints the reply and repeats the cycle by reading the next request.

Let us consider the distributed system shown in figure 6.6, a fairly natural topology for an airline reservation system. Typically, the airline database would reside on the "hub", that is, the central node, while the agencies execute on the satellite nodes. This distribution of the components of the airline reservation system is reflected in the NETLOG specification shown in figure 6.5. Although oversimplified and therefore not a realistic reservation system, the program does show the kernel of a possible

more realistic system. Several queries, directed at different flights, can be processed

```
{ agency(Id) ⟹ read(C);
                dblink query(Id,C);
                wait dblink reply(Id,R);
                write(R);
                agency(Id),
  db(F,S) ⟹ [ db(F,S); reply(Id,S) afternext query(Id,[info,F]) ∨
              db(F,S); reply(Id,nack) afternext query(Id,[resv,F,N]) ∧ S<N ∨
              db(F,S-N); reply(Id,ack) afternext query(Id,[resv,F,N]) ∧ S≥N ],
  airsys ⟹ db(101,300); ··· ; db(109,300); nearby agency(Pid'),
  airsys ⟹ ⊡¬(query(I1,[X1,F|Z1]) ∧ query(I2,[X2,F|Z2]) ∧ I1≠I2)
}
assert
  airsys
```

Figure 6.5: Distributed program for simplified airline reservation system.

concurrently; however, only one query for an individual flight is allowed at any one time. This ensures that any updates to the flight information are properly serialized and that the most up to date information is always returned when requested. This restriction is reflected in the constraints associated with the system. (Note, `Pid'` is a predefined state variable that contains a unique constant which identifies the executing processor.)

6.4 Example 4: Implementing LINDA in NETLOG

NETLOG is sufficiently versatile that it can be used to simulate the features of other distributed programming models. We have already shown how message passing programs can be expressed in NETLOG. In this example we show how NETLOG can

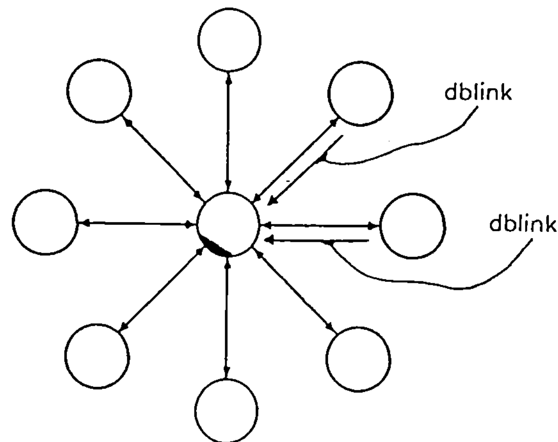

Figure 6.6: A star network.

be used to model LINDA[Gel85,Car87b][2], a programming model centered around the concept of a Tuple Space (TS).

Tuple Space (TS) is conceptually a global shared memory accessible to every process. The elements of TS, called *tuples*, are ordered sequences of values. Three atomic actions are defined on TS: **out**(t) which adds a tuple t to TS, **in**(t) which removes from TS a tuple matching t (any variables in t are updated with the corresponding values of the deleted tuple), and **read**(t) which is like **in**(t) except that the matching tuple is not removed from TS. For operations **in**(t) and **read**(t), if there are no matching tuples the operation blocks until another process adds a tuple that does match.

Figure 6.7 shows a NETLOG program that models LINDA and its associated operations. In our model, TS is replicated on all the nodes in the network. An **out**(t) operation causes an add tuple request `req(add,t)` to be broadcast to all the processors so that they can add tuple `t` to their copy of TS; similarly, an **in**(t) operation causes a delete tuple request `req(del,t)` to be broadcast which results in

[2] We do not include the LINDA operation *eval* in our model since it is used primarily for process creation rather than communication.

tuple **t** being deleted from each copy of TS. (Note, the function $L_1 + +L_2$ adds the elements of L_2 to L_1 and returns the result; similarly, the function $L_1//L_2$ deletes the elements in L_2 from L_1 and returns the result.)

The conditional constraints associated with TS stipulate that an **in**(t) or **rd**(t) operation cannot occur unless there is a matching tuple currently residing in tuple space. The permanent constraints placed on the LINDA system stipulate that at most one **in**(t) or one **out**(t) can performed at any given instance of time. Note, however, there are no constraints on the number of **rd**(t) operations that can occur provided, of course, there are matching tuples in tuple space. This example illustrates

```
{ in(T) ⟹ ⊞req(del,T),
  out(T) ⟹ ⊞req(add,T),
  ts(Ts) ⟹ (¬in(T) if T∉Ts ∧ ¬rd(T) if T∉Ts) until ts(Tx) ∧ Tx≠Ts,
  ts(Ts) ⟹ [ ts(Ts//[T]) afternext req(del,T) ∨
             ts(Ts++[T]) afternext req(add,T) ],
  linda(Ts) ⟹ ts(Ts),
  linda(Ts) ⟹ ⊡( ¬( in(X) ∧ ⟠out(Y) ) ∧
                  ¬( in(U) ∧ ⟠in(V) ) ∧
                  ¬( in(U) ∧ in(V) ∧ U≠V ) ∧
                  ¬( out(S) ∧ ⟠out(T) ) ∧
                  ¬( out(S) ∧ out(T) ∧ S≠T ) )
}
assert
  ⊞linda([])
```

Figure 6.7: Model of LINDA in EVENTLOG.

further how a replicated database can be specified in NETLOG.

6.5 Example 5: Computing primes

As our final example, we show how the model of LINDA given in the previous

```
{ master(I,N) ∧ I<N ⟹ out([int,I]);master(I+1,N);rd([int,I,R]);print(I,R),
  master(I,N) ∧ I=N ⟹ out([int,I]);rd([int,I,R]);print(I,R),

  worker ⟹ in([int,I]);prime(I,[2,...,sqrt(I)]),

  prime(I,[]) ⟹ out([int,I,prime]),
  prime(I,[N|Ns]) ∧ I%N=0 ⟹ out([int,I,notprime]),
  prime(I,[N|Ns]) ∧ I%N≠0 ⟹ prime(I,Ns),

  print(I,prime) ⟹ write(I),
  print(I,notprime) ⟹ true,

  out(T) ⟹ ···
  in(T) ⟹ ···
        ⋱
  linda(Ts) ⟹ ···
}
assert
  ⊞linda([]);Max=2000;master(2,Max);⊞worker
```

Figure 6.8: Computing primes using Linda.

example may be used to write a distributed program to compute all the prime numbers between 1 and `Max`. The program to accomplish this is shown in figure 6.8.

The program is based on the master/worker(s) paradigm[CG89] in which the master creates a number of tasks for the worker(s) to perform. When the work (i.e., tasks) has been completed, the master then gathers the results. In our example, the master initially creates a tuple of the form `[int,I]` for each number in the range 2 to `Max`. The worker(s) withdraw these tuples from TS and replace them with tuples of the form `[int,I,R]`, where `R` is either `prime` or `notprime`. These latter tuples are then input by the master and only the integers marked prime are printed.

The worker(s) determine if integer `I` is prime by verifying that no integer between 2 and the square root of `I` is a divisor of `I`. The notation [I,...,J] denotes the list

expression that generates the list of consecutive integers from I to J, that is, the list [I,I+1,I+2,...,J]. The function `sqrt(I)` is an integer function that returns the smallest integer greater than the square root of I. The symbol `%` stands for the `mod` function.

As this example illustrates, program specifications may be composed easily in NETLOG. Typically, all that is required is to combine, into a single body, the bodies of the programs to be composed. The initial assertions are then merged appropriately to form the initial assertion of the final program. This is essentially what was done in our example.

6.6 Summary

This chapter illustrated how a variety of well-known problems may be expressed in NETLOG. The NETLOG solutions to these problems covered distributed architectures with a range of network topologies. The examples included problems based on distributed AI, (simulated) message passing communication, distributed (replicated) databases, and others. Collectively, these examples covered both transformational and reactive systems.

Chapter 7

Implementation

In this chapter we address the issue of implementing NETLOG. We describe an implementation that can be used to implement the language on any network of processors but is most suitable for tightly coupled distributed memory machines. The implementation strategy is based on the NETLOG Abstract Machine (NAM), a particular virtual machine which realizes the constraint based event model of computation. At runtime, each processor in the network emulates an instance of the NAM. A NETLOG compiler is used to translate NETLOG programs into sequences of abstract machine instructions that execute on this network of (virtual) NAMs.

In the following sections, we describe the compilation process, the design of the NAM, and the experimental results obtained from the simulation of several NETLOG programs.

7.1 Compiling NETLOG programs

In this section we describe the compilation process for NETLOG programs. The NETLOG Abstract Machine (described in detail in the next section) is the target machine for our compiler. The NETLOG compiler translates source programs into object programs consisting of sequences of NAM instructions. The instruction set

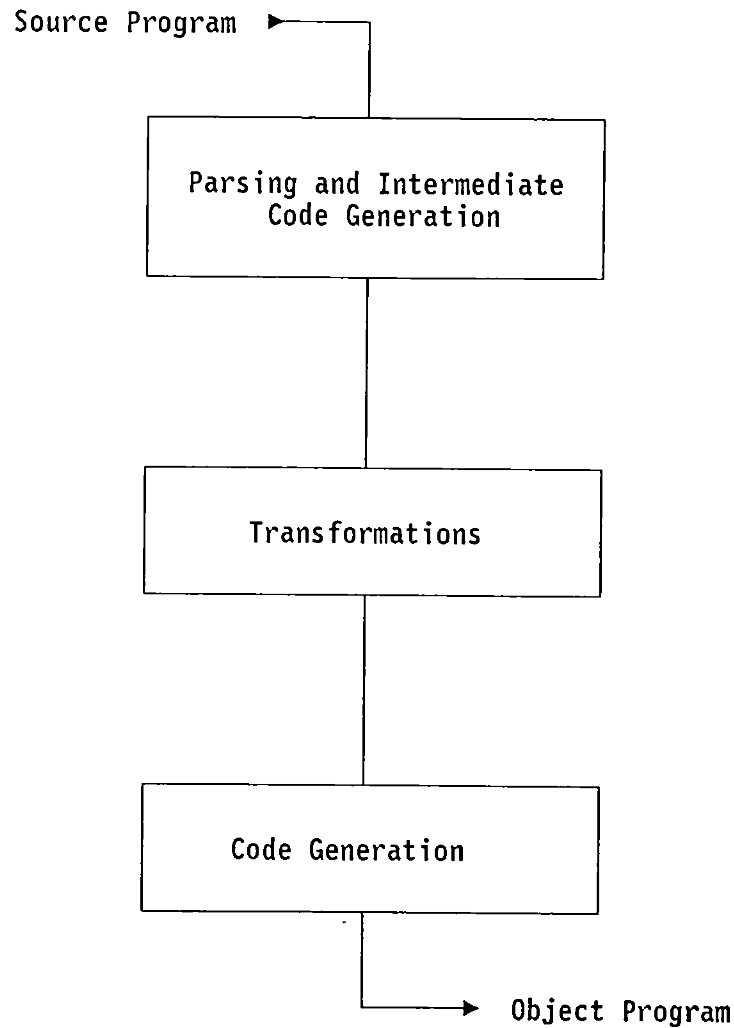

Figure 7.1: Stages of the NETLOG compiler.

of the NAM includes instructions to create (i.e., cause) events, define event descriptors, and search for matching event collections (for a particular event descriptor). The instruction set also includes more general instructions which schedule the event handlers and cause messages to be sent to, and received from, other processors.

The compilation of NETLOG programs can conceptually be divided into three stages as shown in figure 7.1. The first stage of the compiler performs lexical analysis,

parsing, and the generation of the intermediate form of the program. The second stage performs a simple global analysis of the program to obtain information needed in the code generation phase. During this stage, we also perform several source-to-source transformations on the intermediate form. These transformations are applied to transform the intermediate program into a form which is suitable for generating NAM code. Several optional transformations may also be applied to improve the final object program. The third and final stage of the compiler is the code generation stage. During this stage, the intermediate program is translated into a sequence of (abstract) machine instructions for the NAM. These abstract machine instructions are actually encoded using C-macros[Tay89,GH79]; thus, the output of our compiler is really a C program which is subsequently compiled into an executable load module using the local C compiler.

Throughout the following description, we shall use the factorial program given earlier (see figure 2.1) as the source program to illustrate the effects of various stages of the compilation process.

7.1.1 Stage 1: Source program to intermediate form

The first stage of the compiler performs lexical analysis and parsing of the source program. This results in the source program being converted into an intermediate form. The intermediate program has essentially the same structure as the source program except that constructs in extended syntax are expanded into their equivalent base language statements and a unique label is associated with each (multi-use) event handler; thus, a simple syntax directed translator is used to translate the source program into the intermediate program. An example of the output produced by this stage of the compiler is shown below (see figure 7.2).

Notice that the initial statement is converted into several *pseudo* event handlers, thereby making the intermediate program uniform throughout. These event handlers are treated differently from the others in that they are activated only once at the

```
{ H1: infac(X) ⟹ fac(X,1,1),
  H2: fac(X,N,R) ∧ N<X ⟹ ⊙fac(X,N+1,(N+1)×R),
  H3: fac(X,N,R) ∧ N≥X ⟹ outfac(X,R),
  H4: outfac(X,R) ⟹ ⊡(¬true if ◈outfac(X,Y) ∧ Y≠R),
  S1: start() ⟹ infac(2),
  S2: start() ⟹ ◈infac(7),
  S3: start() ⟹ [ (true atnext outfac(2,X)) atnext ◈outfac(7,Y) ∨
                    (true atnext ◈outfac(7,Y)) atnext outfac(2,X) ];
                  write(X+Y)
}
```

Figure 7.2: Intermediate program after stage 1.

start of the program.

7.1.2 Stage 2: Global analysis and source-to-source transformations

During the second stage of the compiler a simple global analysis is performed on the intermediate program. The purpose of the analysis is to determine, for each event type appearing in the program, the set of labels associated with the (multi-use) event handlers that could be activated by events of that type. The result is a partial mapping $\phi : \overline{\mathrm{N}} \mapsto \overline{\mathrm{H}}$, a function from the set of event names $\overline{\mathrm{N}}$ to the set of event handler labels $\overline{\mathrm{H}}$, that is used during the code generation stage of the compiler. The analysis itself starts with an initial map ϕ_{init} (which maps each event name appearing in the intermediate program to the empty set) and proceeds to examine each (multi-use) event handler in the program. For each event type appearing in the event descriptor of the handler, the label of the event handler is added to the set of

labels mapped by that event type. Thus, for the factorial program, we would get ϕ(`infac`) = {`H1`}, ϕ(`fac`) = {`H2,H3`}, and ϕ(`outfac`) = {`H4`}, for example.

During this stage, we also transform the intermediate program using source-to-source transformations. The principle source-to-source transformation performed during this stage is the renaming of the variables in the intermediate program to reference registers in the NAM. This "register allocation" is particularly simple in our case since we are dealing with an abstract machine and therefore can allocate as many registers as we need. In practice, however, the registers used by an event handler rarely exceeds 6-12 abstract machine registers, a number that is far less than the number of real registers on most of today's machines. Thus, the registers used could easily be mapped to real registers in the underlying machine, although currently no attempt is made to do this. After register allocation, our intermediate program would be transformed as follows (note, Xn is used to denote the nth X-register. See the NAM description below for more details.):

```
{ H1: infac(X0) ⟹ fac(X0,1,1),
  H2: fac(X0,X1,X2) ∧ X1<X0 ⟹ ⊙fac(X0,X1+1,(X1+1)×X2),
  H3: fac(X0,X1,X2) ∧ X1≥X0 ⟹ outfac(X0,X2),
  H4: outfac(X0,X1) ⟹ ⊡(¬true if ⟐outfac(X0,X2) ∧ X2≠X1),
  S1: start() ⟹ infac(2),
  S2: start() ⟹ ⟐infac(7),
  S3: start() ⟹ [ (true atnext outfac(2,X0)) atnext ⟐outfac(7,X1) ∨
                   (true atnext ⟐outfac(7,X1)) atnext outfac(2,X0) ];
                 write(X0+X1)
}
```

Figure 7.3: Intermediate program after stage 2.

To improve the efficiency of the code generated, a number of optional source-to-

source transformations may also be performed during this stage of the compiler. For completeness, we briefly describe those that have been implemented.

Eliminating redundant events

Aside from predefined events such as `write(...)`, `read(...)`, etc., events that are not referenced by at least one event handler (i.e., appear in some event descriptor or constraint) can be removed from the program without changing its behavior. Clearly, such events (when they occur) have no influence on the subsequent actions performed by the program.

Eliminating intermediate events and event handlers

This transformation essentially eliminates some classes of intermediate events and reduces the number of event handlers which must be managed at runtime. We treat only the simplest case in which a user defined event is the sole action in the body of an event handler. Furthermore, the only reference to the event is in the event descriptor of a single multi-use event handler and the event descriptor does not contain any relational conditions. In this case, the occurrence of the user defined event can be replaced by the actions caused by the corresponding event handler. The event handler itself can then be removed from the program.

Note, before each substitution is performed, the global variables appearing in the event handler must be replaced by new variables (i.e.,"renamed apart") and the arguments of the event must be substituted for the corresponding variables appearing in the event descriptor.

7.1.3 Stage 3: Code generation

The third, and final, phase of the compiler performs code generation. This is accomplished by making a single pass over the intermediate program using a simple syntax directed translation scheme to generate the abstract machine code. An outline

```
H1:  LOAD(1)              @ Load parameter into reg. X0
     ALLOCATE(Erecord)    @ Allocate an event record
     PUT_ARG_ADDR(H2)     @ Set address of event handler code
     PUT_ARG_ADDR(H3)     @ Set address of event handler code
     SKIP_SECTION         @ Skip to event section of entry
     EVENT(fac,local)     @ Specify event type: fac
     PUT_ARG_VAL(X0)      @ Argument of event: fac
     PUT_ARG_INT(1)       @ Argument of event: fac
     PUT_ARG_INT(1)       @ Argument of event: fac
     POST(local,L1)       @ Post the event
     TERMINATE            @ Abort the execution of the program
L1:  END                  @ End of execution
H2:  LOAD(3)              @ Load arguments into reg. X0, X1, X2
     LT(X1,X0)            @ Test if X1 < X0
     BNZ(L2)              @ Branch if CC is non-zero
     SUSPEND_NEXT         @ Suspend execution until next time step
     ALLOCATE(Erecord)    @ Allocate an event record
     PUT_ARG_ADDR(H2)     @ Set address of event handler code
     PUT_ARG_ADDR(H3)     @ Set address of event handler code
     SKIP_SECTION         @ Skip to event section of entry
     EVENT(fac,local)     @ Specify event type: fac
     PUT_ARG_VAL(X0)      @ Argument of event: fac
     LOADI(X3,1)          @ Load 1 into reg. X3
     ADD(X4,X1,X3)        @ X4 ← X1 + X3
     PUT_ARG_VAL(X4)      @ Argument of event: fac
     MUL(X5,X4,X2)        @ X5 ← X4 × X2
     PUT_ARG_VAL(X5)      @ Argument of event: fac
     POST(local,L2)       @ Post the event
     TERMINATE            @ Abort the execution of the program
L2:  END                  @ End of execution
```

Figure 7.4: Object code generated by NETLOG compiler...

```
H3:  LOAD(3)                     @ Load arguments into reg. X0, X1, X2
     GTE(X1,X0)                  @ Test if X1 ≥ X0
     BNZ(L3)                     @ Branch if CC is non-zero
     ALLOCATE(Erecord)           @ Allocate an event record
     PUT_ARG_ADDR(H4)            @ Set address of event handler code
     SKIP_SECTION                @ Skip to argument section of entry
     EVENT(outfac,local)         @ Specify event type: outfac
     PUT_ARG_VAL(X0)             @ Argument of event: outfac
     PUT_ARG_VAL(X2)             @ Argument of event: outfac
     POST(local,L3)              @ Post the event
     TERMINATE                   @ Abort the execution of the program
L3:  END                         @ End of execution
H4:  LOAD(2)                     @ Load arguments into reg. X0, X1
     ALLOCATE(Crecord)           @ Allocate a constraint record
     SKIP_SECTION                @ Skip to event section of entry
     EVENT(¬true,local)          @ Specify event type: ¬true
     SKIP_ENTRY                  @ Skip to next entry
     EVENT(outfac,somewhere)     @ Specify event type: outfac
     PUT_ARG_VAL(X0)             @ Argument of event: outfac
     PUT_ARG_VAR(X2)             @ Argument of event: outfac
     SKIP_ENTRY                  @ Skip to next entry
     EVENT(¬equal,local)         @ Specify event type: ¬equal
     PUT_ARG_VAL(X2)             @ Argument of event: ¬equal
     PUT_ARG_VAL(X1)             @ Argument of event: ¬equal
     POST(local,L4)              @ Post the constraint
     TERMINATE                   @ Abort the execution of the program
L4:  END                         @ End of execution
```

Figure 7.5: Object code generated by NETLOG compiler continued...

```
S1:  LOAD(0)                      @ Load zero arguments
     ALLOCATE(Erecord)            @ Allocate an event record
     PUT_ARG_ADDR(H1)             @ Set address of event handler code
     SKIP_SECTION                 @ Skip to event section of entry
     EVENT(infac,local)           @ Specify event type: infac
     PUT_ARG_INT(2)               @ Argument of event: infac
     POST(local,L5)               @ Post the event
     TERMINATE                    @ Abort the execution of the program
L5:  END                          @ End of execution
S2:  LOAD(0)                      @ Load zero arguments
     ALLOCATE(Erecord)            @ Allocate an event record
     PUT_ARG_ADDR(H1)             @ Set address of event handler code
     SKIP_SECTION                 @ Skip to event section of entry
     EVENT(infac,local)           @ Specify event type: infac
     PUT_ARG_INT(7)               @ Argument of event: infac
     POST(elsewhere,L6)           @ Post the event
     TERMINATE                    @ Abort the execution of the program
L6:  END                          @ End of execution
S3:  LOAD(0)                      @ Load zero arguments
     ALLOCATE(Drecord)            @ Allocate a descriptor record
     PUT_ARG_ADDR(L8)             @ Set branch address
     SKIP_SECTION                 @ Skip to event section of entry
     EVENT(outfac,local)          @ Specify event type: outfac
     PUT_ARG_INT(2)               @ Argument of event: outfac
     PUT_ARG_VAR(X0)              @ Argument of event: outfac
     ALLOCATE_RELATED             @ Allocate related descriptor record
     PUT_ARG_ADDR(L9)             @ Set branch address
     SKIP_SECTION                 @ Skip to event section of entry
     EVENT(outfac,elsewhere)      @ Specify event type: outfac
     PUT_ARG_INT(7)               @ Argument of event: outfac
     PUT_ARG_VAR(X1)              @ Argument of event: outfac
L7:  MATCH                        @ Find events satisfying event descriptor(s)
     SUSPEND                      @ Wait for new events to occur
     GOTO(L7)                     @ Branch
```

Figure 7.6: Object code generated by NETLOG compiler continued...

```
L8:  ALLOCATE(Drecord)            @ Allocate a descriptor record
     PUT_ARG_ADDR(L12)            @ Set branch address
     SKIP_SECTION                 @ Skip to event section of entry
     EVENT(outfac,elsewhere)      @ Specify event type: outfac
     PUT_ARG_INT(7)               @ Argument of event: outfac
     PUT_ARG_VAR(X1)              @ Argument of event: outfac
L11: MATCH                        @ Find events satisfying event descriptor(s)
     SUSPEND                      @ Wait for new events to occur
     GOTO(L11)                    @ Branch
L12: GOTO(L10)                    @ Branch
L9:  ALLOCATE(Drecord)            @ Allocate a descriptor record
     PUT_ARG_ADDR(L14)            @ Set branch address
     SKIP_SECTION                 @ Skip to event section of entry
     EVENT(outfac,local)          @ Specify event type: outfac
     PUT_ARG_INT(2)               @ Argument of event: outfac
     PUT_ARG_VAR(X0)              @ Argument of event: outfac
L13: MATCH                        @ Find events satisfying event descriptor(s)
     SUSPEND                      @ Wait for new events to occur
     GOTO(L13)                    @ Branch
L14: NOP                          @ No-op
L10: SUSPEND                      @ Wait for new events to occur
     ALLOCATE(Erecord)            @ Allocate a event record
     SKIP_SECTION                 @ Skip to event section of entry
     EVENT(write,local)           @ Create a new event: write
     ADD(X3,X0,X1)                @ X3 ⟶ X0 + X1
     PUT_ARG_VAL(X3)              @ Argument passed to: write
     POST(local,L15)              @ Post the event
     TERMINATE                    @ Abort the execution of the program
L15: END                          @ End of execution
```

Figure 7.7: Object code generated by compiler continued.

of the translation scheme is shown in appendix A. During code generation the compiler keeps track of which registers have previously been loaded with values. This information is used to determine when to generate code to allocate new variables. In particular, the first reference to a register that has not been previously loaded with a value causes the compiler to generate code to load the register with a pointer to a newly created variable.

Figures 7.4–7.7 shows the final object program generated for our factorial program[1]. To actually run the program it must be compiled, along with the C macro definitions that emulate each of the abstract machine instructions, using a C compiler.

7.2 Abstract Machine Design

A wide variety of abstract architectures[HS86a,G+86,KC87,Lev86,Tay89] have been used in implementing logic languages, most being adaptations of the Warren Abstract Machine (WAM)[War83]. In this section we describe the structure and operation of a new abstract machine design, the NETLOG Abstract Machine (NAM), the virtual machine architecture emulated by each processor in the distributed implementation. The structure of the NAM is most similar to the abstract machine developed by Taylor[Tay89] in his implementation of Flat Concurrent-PROLOG. The NAM has a different execution algorithm, however, and includes additional features such as data structures for storing events and representing constraints, new control registers, new instructions for the creation and the matching of events, and modified control instructions to handle the scheduling and execution of event handler threads. In the following description, our intention is to describe the distinctive features of the NAM and to give an exposition of how it executes NETLOG programs. To facilitate the exposition, a number of low-level aspects of the implementation (e.g., how logical variables are represented, etc.) that are based on standard implementation techniques are not described. Such details are readily available in the literature; see, for example,

[1]The remarks following the @ symbol are hand generated comments

[CM81,SS86,CG85,Fos88,M+85,TSS87,Car87a,Gre87,I+87,Sha87,Cra88,Con88,NT88,Tic91].

7.2.1 Abstract Machine Structure and Organization

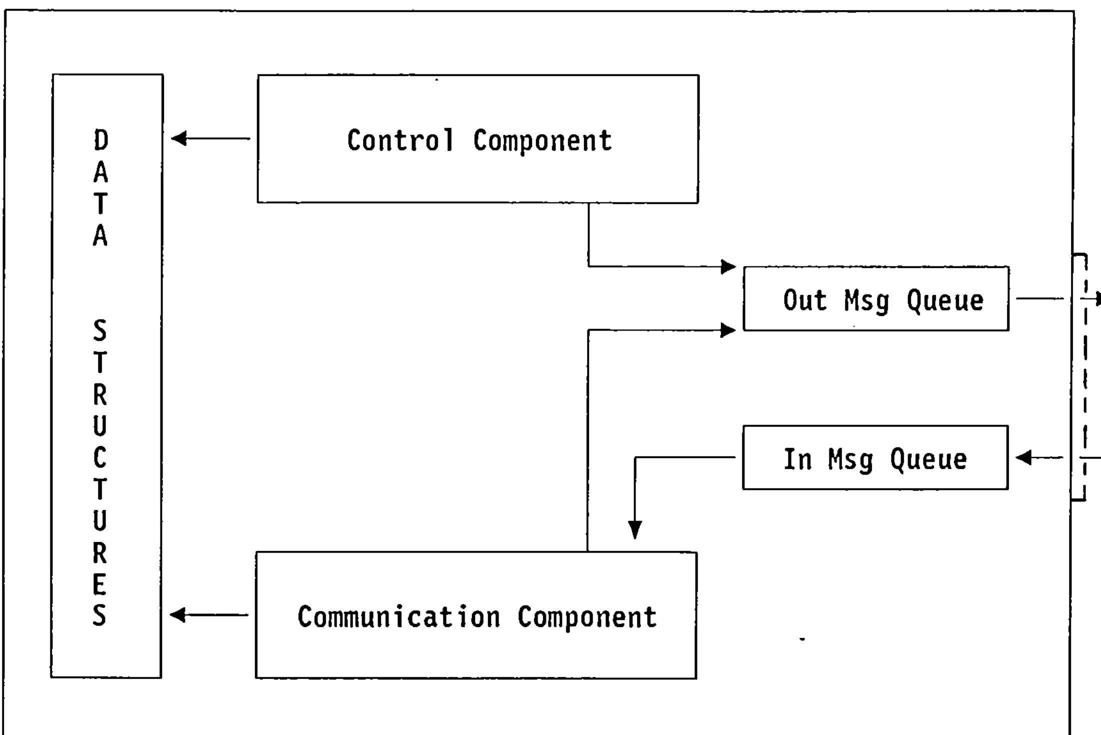

Figure 7.8: NETLOG Abstract Machine organization.

The conceptual structure and organization of the NETLOG Abstract Machine (NAM) is shown in figure 7.8. The NAM consists of a *control* component, which executes event handler code, a *communication* component, which handles messages destined

for, and received from, other processors, and a *storage* component, which is used for allocating and storing various data structures. We also implicitly assume the existence of a network *interface* component (shown in dotted lines) that runs continuously and handles the low-level details of transmitting messages placed on the outgoing message queue and similarly receiving new messages from other processors and depositing them on the incoming message queue.

7.2.2 Abstract Machine Operation

The control component maintains the *active*, *pending*, and *next-time* queues, and performs the basic algorithm outlined in figure 7.9. On each iteration of the algorithm, an event handler thread is removed from the front of the active queue and executed. The event handler code executes to completion unless it suspends. The code may suspend (temporarily), causing it to be placed on the pending queue, if the event handler must wait on one or more events that have yet to occur. The event handler code may also suspend if there are constraints which prevent it from proceeding (e.g., constraints may exisit which prevent the creation of an event at the current time). If the active queue becomes empty, any threads on the pending queue are activated and returned to the active queue to resume execution. Threads that must be delayed until the following time step are placed on the next-time queue. When the global clock is advanced, these threads are woken up and placed on the active queue.

The communication component is invoked at the end of each iteration of the algorithm and handles messages that arrive on the incoming message queue. Depending on the messages received, the global time register may be incremented, various local data structures may be updated, and new messages may be deposited on the outgoing message queue. There are four kinds of messages that may be received:

ACTION: An action message describes an action, i.e., the creation of a new event, to be performed on the receiving processor.

UPDCLK: An update clock message indicates that the global time register should

```
WHILE computation not terminated DO
 IF there are event handlers on the active or pending queue THEN
   IF the active queue is empty THEN
     move event handlers on the pending queue to the active queue
   ENDIF
   dequeue next event handler on active queue
   execute event handler actions
   IF the event handler suspends THEN
     enqueue event handler on appropriate queue
   ENDIF
 ENDIF
 invoke communication component
 IF the global clock has advanced THEN
   move event handlers on the next time queue to the active queue
 ENDIF
ENDWHILE
```

Figure 7.9: Basic execution algorithm.

be updated. It contains the new value to be placed in the register.

STATUS: A status message represents the status token which is circulated around the network. The status of the processor is placed in the token which is then sent to the next processor.

INFO: An info message is used to request information from, or send information to, a remote processor. In particular, these messages are used to convey information about the events that have occurred on a particular processor.

7.2.3 Abstract Machine Support for Global Time

In contrast to most languages for distributed programming, NETLOG has a semantics based on time. One implication of this is that *logically* the same view of time must

be observed by all the processors in the network. Thus, the language implementation needs to provide support for a *global clock.*

One common approach to modeling a global clock on a distributed system is to use global synchronization[RH90]. In this approach, one of the processors in the network is designated to act as the *global synchronizer.* The global synchronizer ensures that all of the processors in the network engage in a global synchronization step at the end of each computation step (i.e., logical time step) before proceeding to the next step of the computation. This method of simulating a global clock has the advantage of being easy to understand and can be implemented quite efficiently on tightly coupled distributed memory machines[RH90]. Indeed, this is the method used in our implementation where the origin processor acts as the global synchronizer. For loosely coupled distributed memory machines, other approaches to implementing a global clock may be more suitable[Ray88].

The NAM maintains a global time register. The value of the global time register is the same for all processors and represents the index of the current time step. Events created by a processor and messages sent by a processor are timestamped with the value of the global time register. The timestamps in events are used to ensure that, when executing event handler code that searches for events satisfying some event descriptor, only events created during the timestep in which the code is executing are considered (i.e., the timestamp in the event and the value in the global time register must be the same). The timestamp in messages are used in implementing the synchronization algorithm which detects when all of the processors have completed the current timestep, and thus the global clock can be advanced.

The particular algorithm used to detect when all of the processors have completed the current timestep is taken from [Tay89] (as is the description to follow) and is an extension of Dijkstra's algorithm[D+83] for termination detection in synchronous systems. The algorithm is based on the following simple observations:

- The difference between the number of messages sent and received during the

current timestep is zero when all messages sent have been received.

- If this difference is zero and no processor is actively working on the current timestep, the current timestep is complete.

The algorithm itself consists of circulating a *token* through all the processors beginning at the origin processor. Each processor maintains a *count* of the difference between the number of messages it has sent and received for the current timestep. It also maintains a *color* which is set to *black* whenever the processor creates an event or performs any communication during the current timestep; the color is set to *white* when the token is present (i.e., when the token enters the processor).

The token carries a *sum* and *color*. At the beginning of each cycle the color of the token is set to *white* and the sum is set to zero by the origin processor. The token propagates from one processor to the next when the current processor becomes white. As it propagates it accumulates the sum of the counts of each white processor encountered. If the token enters a *black* processor its color becomes black. The current timstep is complete if on *two successive cycles* the token arrives at the origin processor with color *white* and sum *zero*. The origin processor can then broadcast a message which informs each processor to advance its global time register and proceed with the next step of the computation.

7.2.4 Machine Instructions

The instruction set of the NAM consists of five categories of instructions: *Test* instructions, *Bind* instructions, *Put* instructions, *Put_arg* instructions, and *Control* instructions. In addition to manipulating the data registers (i.e., X-registers), instructions in several of these categories also manipulate one or more *control registers*, the registers that maintain the dynamic state of the NAM. These control registers will be introduced informally in the sections which describe the instruction that use them. A summary of the NAM instruction set is given in tables 7.1–7.5 (note, in the instruction descriptions, 'Int' denotes an integer, 'R' denotes a data register, 'M' denotes

a state memory address, 'Const' denotes a symbolic constant (e.g., an event name), and 'L' denotes a code address label).

Test Instructions

Test Instructions
test_nil(R)
test_int(R,Int)
test_atom(R,Const)
test_list(R1,R2)

Table 7.1: Test instruction set summary.

Test instructions are used to test the data type of values. These instructions operate on data registers (i.e., X-registers). The **test_nil(R)**, **test_int(R,Int)**, and **test_atom(R,Const)** instructions test the contents of a register for the empty list, an integer, and a constant, respectively. The **test_list(R1,R2)** instruction take two registers. It tests the first register to verify that it refers to a list. If it does, pointers to the head (first element of list) and tail (remainder of list after first element is removed) of the list is loaded into consecutive registers starting with the second register. These instructions also (implicitly) set the condition code register (CC-register) to a non-zero value if the value tested does not have the correct data type.

Bind Instructions

Bind instructions are used to bind values to variables. Bind instructions operate on data registers. The **bind_nil(R)**, **bind_int(R,Int)**, and **bind_atom(R,const)** instructions bind the empty list, an integer, and a constant, respectively, to the variable referenced by a register. Note, if the register refers to a variable that is already defined, the bind instruction verifies that the variable has the same value as the intended bind value.

Bind Instructions
bind_nil(R)
bind_int(R,Int)
bind_atom(R,Const)
bind_list(R,L1,L2)
unify(R1,R2)

Table 7.2: Bind instruction set summary.

The `unify(R1,R2)` instruction performs general unification. The instruction takes two registers which refer to the two arguments to be unified. Both the bind and unify instructions (implicitly) set the condition code register (CC-register) to a non-zero value if they fail to execute successfully.

Put Instructions

Put Instructions
put_nil()
put_int(Int)
put_atom(Const)
put_list(R)
put_val(R)
put_var(R)

Table 7.3: Put instruction set summary.

The memory area of the NAM is partitioned into two parts: the *main* memory and the *state memory*. The main memory of the NAM is organized as a *heap* and is used for storing data structures. Put instructions are used to allocate data structures such as integers, global variables, lists, etc. in the main memory. Other structures such as the records used to describe events, event descriptors, and constraints are also allocated on the heap. Local variables are allocated in the state memory. This memory is organized as a linear array of addressable locations (i.e., standard memeory organization).

In general, put instructions place a data item on the heap at the position indicated by the structure pointer register (SP-register) and then increment the SP-register in preparation for placing the next data item. The end of the heap is pointed to by the heap pointer register, HP-register.

The `put_nil()`, `put_int(Int)`, and `put_atom(Const)` instructions place the empty list, an integer, and a constant, respectively, at the heap location pointed to by the SP-register. The `put_val(R)` instruction copies the contents of a register into the location pointed to by the SP-register. The `put_var(R)` instruction allocates a variable at the end of the heap and places a reference to the variable at the SP-register. It also returns a reference to the variable in a register.

Put Argument Instructions

Put_arg Instructions
put_arg_nil()
put_arg_int(Int)
put_arg_atom(Const)
put_arg_list(R)
put_arg_var(R)
put_arg_val(R)
put_arg_addr(L)

Table 7.4: Put_arg instruction set summary.

For each put instruction there is a corresponding `put_arg` instruction. Put argument instructions are used to build the arguments appearing in events, event descriptors, and constraints. With the exception of the `put_arg_addr(L)` instruction, the put argument instructions operate in the same manner as the put instructions except that they place values at the location referenced by the argument register, A-register, rather than the structure pointer register, SP-register. The `put_arg_addr(L)` instruction places the code address associated with a label L at the location referenced by the A-register.

Control Instructions

Control Instructions	
allocate(Const)	allocate_related
event(Const,Const)	match
post(Const,L)	post_temp(Const,R,L)
delete(Const,R)	skip_arg
skip_section	skip_entry
suspend	suspend_next
terminate	end
load(Int)	loadI(R,Int)
cop(R1,R2)	aop(R1,R2,R3)
bnz(L)	bz(L)
goto(L)	set_svar(R,M)
get_svar(R,M)	

Table 7.5: Control instruction set summary.

The basic execution algorithm given in figure 7.6 is implemented using control instructions. These instructions perform the posting of events, the scheduling of event handlers from the active, pending, and next-time queues, etc. They are also responsible for invoking the communication component whenever an event handler terminates (or suspends) and are responsible for sending messages to other processors as necessary.

Events, event descriptors, and constraints are represented by *event records*, *event descriptor records*, and *constraint records* respectively. Event records describe the event type, the arguments of the event, and the addresses of the event handlers (if any) to be activated by the event. Similar, event descriptor records describe the collection of events that must be matched, and constraint records describe individual constraint clauses.

Each active event handler is represented by a *control record* which contains the arguments of the event that activated the event handler and the address of the code to execute when the event handler is scheduled. Space is also provided for saving the

state of various control registers whose values are needed to resume the event handler should it suspend.

The scheduling of event handlers is as follows. The event handler on the front of the active queue is always the next event handler to be scheduled. If there are event handlers on the next-time queue when the global time register is advanced, indicating the start of the next global time step, they are removed and added to the active queue before the next event handler is scheduled. Finally, If the active queue becomes empty, any event handlers on the pending queue are removed from that queue and placed on the active queue. The next event handler is then scheduled.

Scheduling the next event handler involves dequeuing its control record from the active queue, restoring the control registers, and making the event handler *active* by setting the active thread register (AT-register) to point to its control record. The code at the program counter is then executed. The control instructions are as follows:

`load(N)`: is always the first instruction of an event handler. It loads the N arguments begining at the A-register into consecutive data registers (i.e., X-registers) starting at register zero (X0).

`loadI(R,Int)`: loads integer Int into register R.

`allocate(Const)`: allocates a new event, descriptor, or constraint record[2]. The value of Const specifies the record type. When a new record. The value of Const specifies the record type. When a new record is allocated it is implicitly assigned a timestamp (i.e., the current value of the global time register) and a token which uniquely identifies the record. The record register (R-register) and the related record register (RR-register) are set to the address of the new record and the argument register (A-register) is set to point to the begining of the first entry of the new record in preparation for instructions which build the record's entries.

[2]Note, the format of all of these records are actually the same. A record consists of several identical entries, each of which can be used to describe the attributes of a single event. The entries in a record will be filled in differently, however, depending on whether the record is to be used to describe an event, event descriptor, or constraint.

`allocate_related`: allocates a new event, descriptor, or constraint record and chains it onto the list of records pointed to by the record register (R-register). The related record register (RR-register) is set to the address of the new record and the argument register (A-register) is set to point to the begining of the first entry of this new record in preparation for instructions which build the record's entries.

`skip_entry`: sets the argument register (A-register) to point to the begining of the next entry in the current record.

`skip_section`: sets the argument register (A-register) to point to the begining of the next section of the current record entry.

`skip_argument`: increments the argument register (A-register).

`event(Const1,Const2)`: places a description of the event type in the heading of the current record entry. Const1 specifies the event type, and Const2 specifies the scope of the event.

`post(Const,L)`: is used to add a new event or new constraint to existing events or constraints, respectively, on the system. Const specifies where (i.e., on which processors) the new event or constraint should be added.

If a new event is being posted, the instruction checks that there are no constraints which prevents the event from occurring and then posts the event at the location(s) specified by Const. For non-local events, posting an event involves sending messages to other processors informing them of the new event to be added to their event lists. Similarly, if a new constraint is being posted, the post instruction verifies that the constraint holds (i.e., is true) at the location(s) specified by Const and then posts the constraint. For new events, the event record contains the addresses of the event handlers (if any) activated by the event. For each address found, a new control record is allocated to represent

the activated event handler. The address of the corresponding event handler is placed in the control record at the location reserved for saving/restoring the program counter register. In addition, the arguments of the event are copied into the control record and a pointer to the first argument is stored in the area of the control record reserved for saving/restoring the argument register[3]. The control record is then added to the active queue.

If the post instruction succeeds (i.e., the new event or constraint has been added to existing events or constraints), it causes a transfer of control to the address associated with label L by modifying the program counter (PC-register). Otherwise, execution continues with the next instruction.

`post_temp(Const,R,L)`: behaves exactly the same as `post(Const,L)` except that register R is assigned a copy of the (internal) token which uniquely identifies the constraint that was posted. The instruction is used to post temporary constraints.

`delete(Const,R)`: deletes a temporary constraint that was posted using the `post_temp(Const,R,L)` instruction. Register R contains the token that identifies the constraint to be deleted and Const specifies the location(s) of the constraint. Execution continues with the next instruction.

`match`: tries to find an event collection that matches the current event descriptor record. If a matching event collection is found, any unbound variables in the event descriptor are instantiated by the corresponding values from the matching events. The program counter (PC-register) is set to the branch address stored in the descriptor record which causes the instruction at the branch address to receive control.

If a matching event collection is not found and the current descriptor record

[3]Hence, when the event handler is scheduled, the PC-register and A-register will be restored with the initial values needed to load the arguments and begin executing the event handler code.

does not point to additional (i.e., related) descriptor records, the instruction fails and execution continues with the next instruction. If there are additional descriptor records, the instruction tries each one in turn until a successful match is found or the instruction fails.

`suspend`: causes the executing event handler to be suspended. The execution state of the event handler (e.g., the control registers, etc.) is saved in the event handler's control record and the control record is then added to the pending queue. Since the execution of the event hadler has ended, the communication component is invoked to service the message queues and then another event handler is scheduled.

`suspend_next`: behaves exactly the same as `suspend` except that the control record is added to the next-time queue.

`terminate`: is used when an error condition has been detected. It abnormally terminates the entire computation.

`end`: is the last instruction of an event handler. The instruction terminates the execution of the event handler and de-allocates the event handler's control record, returning it to the heap for reuse. The communication component is invoked to service the message queues and then the next event handler is scheduled.

`aop(R1,R2,R3)`: denotes a generic arithmetic operation on registers. In particular, R1 is the result obtained by performing the specified arithmetic operation on R2 and R3 (i.e., R1 $\leftarrow$ R2 aop R3). Typical arithmetic operations included are ADD (addition), SUB (subtraction), MUL (multiplication), etc.

`cop(R1,R2)`: denotes a generic comparative operation on registers. Comparative operations include LT (less than), GTE (greater than or equal), etc. If the comparative test is false, the condition code register (CC-register) is set to a non-zero value. Execution then continues with the next instruction.

`bz(L)`: is used to specify a conditional branch. If the condition code register (CC-register) is zero, It causes a transfer of control to the address associated with label L by modifying the program counter (PC-register).

`bnz(L)`: is used to specify a conditional branch. If the condition code register (CC-register) is non-zero, It causes a transfer of control to the address associated with label L by modifying the program counter (PC-register).

`goto(L)`: is used to specify an unconditional branch. It causes a transfer of control to the address associated with label L by modifying the program counter (PC-register).

`get_svar(R,M)`: loads register R with the value stored in the state location referenced by address M.

`set_svar(R,M)`: stores the value in register R in the state location referenced by address M.

7.3 Experimental Results

As described in previous sections, a compiler for our language and the runtime code to support the abstract machine implementation have been written. However, we have not yet implemented the language on a real distributed machine. Thus the results we present in this section were obtained by executing our implementation under the control of a simulator.

7.3.1 Overview of simulator

The simulator itself is quite simple and decidedly unsophisticated. It was not designed to simulate the characteristics of a specific distributed machine, rather it was designed to simulate the two central features that characterize all distributed machines —

namely, a collection of Processing Elements (PE), each of which executes part of the distributed program's instruction stream, and the interconnection network which connects them (and via which the PEs communicate by sending mesaages)[FD90]. Thus, the simulator provides a simple tool via which a few basic measures can be obtained for subsequent use in gauging the relative merits of different implementation strategies.

The simulator is written in C and runs on a SPARCstation 1. To perform a simulation, the abstract machine program produced by the NETLOG compiler and the simulator itself are compiled separately (using a C compiler) and then linked together to form a single executable load module. By appropriately setting a number of control variables, the simulator can be configured to simulate a number of different network topologies (e.g., linear array, ring, etc.) with varying numbers of processors. Our simulator is hybrid in the sense that the computation times reported are a sum of measured and calculated time. Specifically, the abstract machine program (or more accurately, the C code representing the abstract machine program) is branched to by the simulator so it runs directly on the SPARCstation 1. Thus, its execution time can be measured using UNIX's start and stop timing functions. The times for message communication (i.e., sending a message from one processor to another) must however be computed by calculation (see below) since the transfer of messages between processors is performed by simulation. The hybrid nature of our simulator allows us to directly measure the uniprocessor performance of the abstract machine implementation.

At the beginning of each simulation, the simulator creates a list of Simulation Control Blocks (SCB), one SCB for each processor in the network, that captures the current state of each processor. For each processor the SCB contains various measurement data, such as the total execution time, the total number of bytes transferred via each (outgoing) link connected to the processor, etc., and a pointer to the location in memory where the processor state (e.g., control registers, data registers, etc.) is stored.

During each cycle of the simulation, a SCB is selected and dequeued from the list, the time slice register is reset, the state of the corresponding processor (i.e., abstract machine) is restored, and execution is resumed at the point were it was last suspended. When the time slice has expired or the processor becomes idle (i.e., there are no event handlers on the active or pending queues and all incoming messages have been read), execution of the processor is suspended. The state of the processor is saved, any messages on its outgoing message queue are sent to their destination (as described below), and the execution statistics for the processor are updated in the SCB which is then enqueued back onto the SCB list. The simulation continues in this fashion until the computation being simulated terminates, at which point the execution statistics gathered during the simulation are displayed.

The simulator allows direct communication only between adjacent processors, thus messages destined for non-adjacent processors must be routed through one or more intermediate processors before arriving at their final destination. The simulator assumes messages may be received on one link at the same time other messages are being sent on another link. Note, however, only one message may be transmitted across any single link at a given point in time. To simulate the sending of an outgoing message from one processor to an adjacent processor, the message is removed from the outgoing message queue of the sending processor and appended to the incoming queue of the adjacent processor. The following linear formula is used to calculate the transmission time (T_r) for a single message to traverse a link connecting two processors:

$$T_r = T_s + T_c * \text{Bytes}$$

where T_s is the message startup time, T_c is the time to transmit a single byte across the link, and Bytes is the total number of bytes in the message.

7.3.2 Results

This section contains the results obtained by simulating the execution of our NAM implementation on an assortment of short programs. The assortment includes programs to compute the sum of two vectors, reverse a list, delete multiples from a list, perform matrix multiplication, and find prime numbers[4]. The NETLOG specifications for these programs may be found in appendix B.

Table 7.6 focuses on the uniprocessor performance of our NAM. Looking at the uniprocessor performance of the NAM allows us to calibrate the raw performance of the implementation. In particular, since we are dealing with a single processor, interprocessor communication is not a factor; thus the execution times shown are the actual (measured) times to execute the sum of vectors benchmark on a NAM running on a SPARCstation 1. Table 7.6 also gives the ratio of the execution times for our NETLOG program and the same problem coded in C[5]. For comparison, similar results for FGHC (taken from [Tic91]) are also included in the table. As can be seen, our NETLOG program gives comparable performance to the FGHC program (35 times slower than the corresponding C program Vs 33 times slower). Since many aspects of the implementation were adopted for reasons of convenience and simplicity, we believe its performance could be substantially improved using state-of-the-art compiler technology. For example, native code logic programming implementations and advanced compiler techniques such as rule indexing would greatly reduce the emulation overheads and the number of threads that need to be scheduled and maintained at runtime. Such techniques have resulted in systems that execute logic programs 20%–40% faster[Tic91].

[4]Although our simulator was designed to be general (and, as such, not designed to simulate the particular characteristics of a specific distributed machine), the accuracy of our simulation results were randomly checked by taking the compiled code for several of the benchmark programs and manually tracing their execution. The results from the manual execution were then compared with the results obtained by the simulator. In all cases the simulation results corresponded with the manual results to within 5%.

[5]The time for the C code was obtained by performing the addition of two 10,000 element vectors one hundred times in a loop; similarly, the NETLOG time corresponds to adding two 10,000 element vectors 100 times.

A mesh topology was used in simulating the distributed execution of each of the benchmark programs. A variety of problem sizes and communication speeds were used in the simulations. The communication speeds (i.e., $T_r[T_s = 1\,ms, T_c = 1\,MB/s]$, $T_r[T_s = 400\,\mu s, T_c = 10\,MB/s]$, and $T_r[T_s = 1\,\mu s, T_c = 100\,MB/s]$[6]) were chosen to be representative of the broad range of distributed systems ranging from those based on Wide Area Networks (WANs) and Metropolitan Area Networks (MANs) at one end of the spectrum, intermediate systems based on Local Area Networks (LANs), and tightly coupled systems at the other end of the spectrum[AS88,Sta88,Sta90,For92,H+90,Cor90]. Tables 7.7-7.14 show the execution times and speedups[7] for each of the simulated benchmarks. In each case, the time reported is the average of two or three repeat runs. Since the simulations were performed on a sequential machine, the execution times were consistently repeatable with low variance. To keep operating system effects (e.g., page fault interupts and paging delays, etc.) to a minimum, relatively small problem sizes were used. It is worth mentioning that the cost of executing the synchronization algorithm during the simulation of a program is accounted for as part of the overall communication cost (i.e., time) incurred by the program. Specifically, very little in the way of actual computation is performed during the synchronization phases of a computation — most of the activity consists of sending messages between processors (i.e., circulating "token" messages). These "token" messages are treated in the same way as any other message, and thus their transmission times are included in the cost of message passing.

Tables 7.7, 7.9, 7.11, and 7.13 show how execution time varies with communication speed, while tables 7.8, 7.10, 7.12, and 7.14 show how execution time varies with problem size[8]. We can see that larger problems execute more efficiently. We can

---

[6] We use the notation $T_r[T_s{=}X, T_c{=}Y]$ to denote the transmission time T_r determined by message start up time (i.e., latency) X and byte transfer rate (i.e., bandwidth) Y. For example, $T_r[T_s = 1\,ms, T_c = 1\,MB/s]$ denotes the formula $T_r = 1.0 \times 10^{-3} + (1.0 \times 10^{-6}) \times \text{Bytes}$.

[7] Various definitions of "speedup" have appeared in the literature. The definition of speedup used here is defined as the ratio between the execution time of a program on one PE and the execution time of the same program on multiple PEs. Speedup defined in this way has sometimes been called "parallelizability"[HQ91].

[8] By setting the appropriate simulator control variable(s), the execution rate of the Processing

also see that, as the communication speed decreases, the efficiency of the computation decreases. These results are consistent with empirical studies (see, for example, [F+88]) which show that the performance of a distributed program is determined by the ratio of the time the program spends performing calculations (i.e., computation time) to the time it spends sending messages (i.e., communication time). For our benchmark programs, increasing the problem size tends to increase the computation time thereby giving a higher ratio and better performance; in contrast, decreasing the communication speed tends to increase the communication time resulting in a lower ratio and worse performance.

The data also suggests that our implementation of the benchmark programs would give reasonable speedups at all but the slowest of the three communication speeds. However, it should be pointed out that, given a more sophisticated compiler, the speedups obtained would be somewhat lower than those reported in tables 7.7–7.14. In particular, a more sophisticated compiler would generate more efficient code resulting in a decrease in the execution times of the benchmark programs (e.g., the time to execute a benchmark on a single processor would decrease). The communication costs incurred during the execution of each benchmark remains the same however, so the speedups obtained would decrease.

While the data presented provides us with some useful information, it does not represent an attempt to provide a comprehensive account of the performance of NET-LOG. Such an undertaking would be rather premature since our focus has been on the expressiveness of the language as opposed to focusing on the performance of the implementation. Rather, an implementation based on state-of-the-art logic compilation techniques should be constructed first. With such an implementation in hand, the performance of the system on a real distributed machine can be then be analyzed in-depth.

Elements (PEs) can be scaled to model a variety of processor speeds. In the data reported in tables 7.7 – 7.14, the simulator was configured to model PEs running at 200 MHz.

C (secs)	FGHC (secs)	FGHC/C	C (secs)	NETLOG (secs)	NETLOG/C
5.15	173.2	33.6	3.41	122.28	35.8

Table 7.6: Execution time ratio: FGHC/C Vs NETLOG/C

List Reverse (10,000)						
		1	2	4	8	16
$T_r[T_s = 1\,ms, T_c = 1\,MB/s]$	Time	0.113	0.102	0.076	0.064	0.050
	Speedup	1.0	1.108	1.487	1.766	2.260
	Efficiency	1.0	0.55	0.37	0.22	0.14
$T_r[T_s = 400\,\mu s, T_c = 10\,MB/s]$	Time	0.113	0.065	0.036	0.023	0.017
	Speedup	1.0	1.738	3.138	4.91	6.647
	Efficiency	1.0	0.87	0.78	0.61	0.42
$T_r[T_s = 1\,\mu s, T_c = 100\,MB/s]$	Time	0.113	0.060	0.032	0.017	0.009
	Speedup	1.0	1.85	3.53	6.65	12.55
	Efficiency	1.0	0.93	0.88	0.83	0.78

Table 7.7: Reverse benchmark for a variety of link communication speeds.

List Reverse ($T_r[T_s = 400\,\mu s, T_c = 10\,MB/s]$)						
		1	2	4	8	16
Size=5000	Time	0.057	0.035	0.020	0.013	0.010
	Speedup	1.0	1.628	2.85	4.38	5.7
	Efficiency	1.0	0.81	0.71	0.55	0.36
Size=10000	Time	0.113	0.065	0.036	0.023	0.017
	Speedup	1.0	1.738	3.138	4.91	6.647
	Efficiency	1.0	0.87	0.78	0.61	0.42
Size=20000	Time	0.227	0.127	0.069	0.041	0.028
	Speedup	1.0	1.787	3.289	5.536	8.107
	Efficiency	1.0	0.89	0.82	0.69	0.51

Table 7.8: Reverse benchmark for a variety of problem sizes.

Delete (10,000)						
		1	2	4	8	16
$T_r[T_s = 1\,ms, T_c = 1\,MB/s]$	Time	0.298	0.195	0.120	0.087	0.063
	Speedup	1.0	1.528	2.483	3.425	4.730
	Efficiency	1.0	0.76	0.62	0.43	0.29
$T_r[T_s = 400\,\mu s, T_c = 10\,MB/s]$	Time	0.298	0.158	0.082	0.046	0.029
	Speedup	1.0	1.886	3.634	6.478	10.276
	Efficiency	1.0	0.94	0.91	0.81	0.64
$T_r[T_s = 1\,\mu s, T_c = 100\,MB/s]$	Time	0.298	0.153	0.077	0.040	0.022
	Speedup	1.0	1.948	3.870	7.450	13.545
	Efficiency	1.0	0.97	0.96	0.93	0.85

Table 7.9: Delete benchmark for a variety of link communication speeds.

Delete ($T_r[T_s = 400\,\mu s, T_c = 10\,MB/s]$)						
		1	2	4	8	16
Size=5000	Time	0.149	0.080	0.043	0.027	0.018
	Speedup	1.0	1.862	3.465	5.518	8.278
	Efficiency	1.0	0.93	0.87	0.69	0.52
Size=10000	Time	0.298	0.158	0.082	0.046	0.029
	Speedup	1.0	1.886	3.634	6.478	10.276
	Efficiency	1.0	0.94	0.91	0.81	0.64
Size=20000	Time	0.597	0.310	0.161	0.089	0.052
	Speedup	1.0	1.926	3.708	6.708	11.481
	Efficiency	1.0	0.96	0.93	0.84	0.72

Table 7.10: Delete benchmark for a variety of problem sizes.

Primes (between 1 and 5120)						
		1	2	4	8	16
$T_r[T_s = 1\,\text{ms}, T_c = 1\,\text{MB/s}]$	Time	1.218	0.716	0.380	0.210	0.119
	Speedup	1.0	1.701	3.205	5.800	10.235
	Efficiency	1.0	0.85	0.80	0.72	0.64
$T_r[T_s = 400\,\mu\text{s}, T_c = 10\,\text{MB/s}]$	Time	1.218	0.705	0.369	0.196	0.105
	Speedup	1.0	1.727	3.301	6.214	11.600
	Efficiency	1.0	0.86	0.82	0.77	0.72
$T_r[T_s = 1\,\mu\text{s}, T_c = 100\,\text{MB/s}]$	Time	1.218	0.702	0.364	0.190	0.099
	Speedup	1.0	1.735	3.346	6.410	12.303
	Efficiency	1.0	0.87	0.84	0.80	0.76

Table 7.11: Primes benchmark for a variety of link communication speeds.

Primes ($T_r[T_s = 1\,\text{ms}, T_c = 1\,\text{MB/s}]$)						
		1	2	4	8	16
Size=2560	Time	0.483	0.287	0.155	0.089	0.055
	Speedup	1.0	1.683	3.116	5.427	8.782
	Efficiency	1.0	0.84	0.77	0.67	0.55
Size=5120	Time	1.218	0.716	0.380	0.210	0.119
	Speedup	1.0	1.701	3.205	5.800	10.235
	Efficiency	1.0	0.85	0.80	0.72	0.64
Size=15360	Time	5.123	2.992	1.575	0.845	0.452
	Speedup	1.0	1.712	3.253	6.063	11.334
	Efficiency	1.0	0.86	0.81	0.76	0.71

Table 7.12: Primes benchmark for a variety of problem sizes.

Matrix Mulitiplication (64x64)						
		1	2	4	8	16
$T_r[T_s = 1\,ms, T_c = 1\,MB/s]$	Time	4.581	2.357	1.232	0.709	0.466
	Speedup	1.0	1.943	3.718	6.461	9.830
	Efficiency	1.0	0.97	0.93	0.81	0.61
$T_r[T_s = 400\,\mu s, T_c = 10\,MB/s]$	Time	4.581	2.326	1.185	0.628	0.356
	Speedup	1.0	1.969	3.866	7.294	12.868
	Efficiency	1.0	0.98	0.96	0.91	0.80
$T_r[T_s = 1\,\mu s, T_c = 100\,MB/s]$	Time	4.581	2.322	1.178	0.616	0.339
	Speedup	1.0	1.973	3.888	7.437	13.513
	Efficiency	1.0	0.99	0.97	0.93	0.84

Table 7.13: Matrix Multiplication benchmark for a variety of link communication speeds.

Matrix Multiplication ($T_r[T_s = 1\,ms, T_c = 1\,MB/s]$)						
		1	2	4	8	16
Size=(64x64)	Time	4.581	2.357	1.232	0.709	0.466
	Speedup	1.0	1.943	3.718	6.461	9.830
	Efficiency	1.0	0.97	0.93	0.81	0.61
Size=(96x96)	Time	16.556	8.430	4.317	2.340	1.367
	Speedup	1.0	1.964	3.835	7.075	12.111
	Efficiency	1.0	0.98	0.96	0.88	0.76

Table 7.14: Matrix Multiplication benchmark for a variety of problem sizes.

7.3.3 Summary

In this chapter a scheme for implementing NETLOG on a tightly coupled distributed memory machine has been described. The implementation scheme is based on the NETLOG Abstract Machine (NAM), a particular virtual machine which realizes the constraint based event model of computation. The instruction set for the NETLOG abstract machine was described, and experimental results showed that the uniprocessor performance of NETLOG programs is comparable to that of conventional concurrent logic languages.

Chapter 8

Conclusions and Future Research

8.1 Summary of research

In this dissertation we have described the syntax, semantics, and implementation of NETLOG, a new logic language for distributed computing that is based on an executable subset of distributed systems logic (DSL).

NETLOG differs from other logic languages for concurrent and distributed computing in that both the notion of time and the notion of location are intrinsic to the semantics of the language. Consequently, the language includes a variety of temporal and spatial constructs which provide new tools for expressing distributed algorithms. Several programming examples were given to illustrate the use of these novel constructs and to illustrate the general capabilities of the language.

Indeed, the programming examples showed that important concepts in distributed programming — such as concurrency, synchronization, (interprocessor) communication, mutual exclusion, nondeterminism, and locality — can be expressed easily within the language. They also demonstrated that the language could be used to specify distributed computations for a diverse spectrum of applications and architectures.

Another important aspect of our presentation was the development of a formal operational semantics for the language. The semantics relates the logical interpretation

of NETLOG programs to their corresponding operational behavior. In particular, the semantics showed that the execution of a NETLOG program is related to the construction of a model for the program; thus, if a NETLOG program executes successfully, that is, without errors, then the logical formula it denotes is satisfiable. It was also demonstrated how the operational semantics could be used to reason about the execution of NETLOG programs.

Finally, a detailed description of a compiler for NETLOG was presented. The compilation process involved several stages which, collectively, transformed the source program into a sequence of abstract machine instructions for the NETLOG abstract machine. The instruction set for the NETLOG abstract machine was described, and experimental results showed that the uniprocessor performance of NETLOG programs was comparable to that of conventional concurrent logic languages.

8.2 Directions for future research

The research described in this dissertation can be pursued in a number of directions. We mention a few of them here.

Language design

NETLOG contains a relatively small set of future time temporal operators. For the most part, these operators allow one to model quite easily the temporal behavior of a wide range of distributed computations. There are, however, distributed computations which could be modeled more easily given a more comprehensive set of temporal operators. Further investigation is needed to determine the trade-offs and/or benefits of expanding NETLOG (and the underlying logic) to include additional future time temporal operators and, perhaps, some past time temporal operators as well.

Verification tools

Another worthwhile area of investigation is the development of tools for verifying NETLOG programs. We believe that semantics based tool generators could take the formal operational description given in chapter 5 as input and generate as output a program which could, at least partially, automate the process of verifying NETLOG programs. Similar research into the development of semantically based tool generators has already been conducted for programming languages based on other paradigms (e.g., functional programming[Pau82]).

Language implementation

More research is needed to develop suitable algorithms for analyzing and extracting program information that could be used to optimize the code generated by the NETLOG compiler. Based on our experience, it seems clear that a number of the standard compiler analysis and optimization techniques[ASU86] could be adapted for use in compiling NETLOG programs. Furthermore, we anticipate that the more novel aspects of our language will encourage the discovery of additional new techniques for implementing the language.

8.3 Conclusions

NETLOG is a simple yet versatile high-level logic programming language that can be used to specify and rapidly prototype a broad range of distributed applications. It can be used to specify both applications that require a particular distribution strategy and those that do not. We do not claim that the language is the ultimate solution to all the world's distributed programming problems, but we have been pleasantly surprised by its utility to date.

Many aspects of the current implementation of NETLOG were adopted for reasons of convenience and simplicity. Consequently, there is considerable room for improve-

ment and more research needs to be done before implementations of NETLOG rival those of traditional imperative programming languages.

From a broader perspective, NETLOG seems to provide a natural bridge between our conception of a problem solution and its subsequent expression in the context of a programming language[J+85]. In particular, space and time are basic concepts that seem to prescribe and permeate every facet of human activity, including the way we think and reason about the world. NETLOG provides intrinsic support for spatial/temporal reasoning within the framework of distributed programming. Thus, many familiar patterns of thought such as relating the activities and events that occur in one place to those that occur in another, reasoning about current and future events, etc. can be expressed directly within the language — that is, without requiring users to first reformulate their ideas in terms of less abstract lower-level machine concepts.

Lastly, and perhaps most importantly, distributed programs can be very deceptive[LL87]. A program that looks simple may be quite complex permitting unexpected behaviors. Therefore, rigorous reasoning is required to determine whether or not a program is doing what it was designed to do. Rigorous reasoning requires a formal foundation. NETLOG, having both a formal logical and operational semantics, facilitates such rigorous reasoning.

Bibliography

[AM87] M. Abadi and Zohar Manna. Temporal logic programming. In *International Symposium on Logic Programming*, pages 4–16, sept 1987.

[AS86] G. Andrews and F. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, 1986.

[AS88] W. Athas and C. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, 21(8):9–24, 1988.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compliers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[B+89] H. Bal et al. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, 1989.

[BC83] G. Berry and L. Cosserat. ESTEREL: Towards a synchronous and semantically sound high level language for real time applications. In *IEEE Real-Time Systems Symposium*, pages 255–269, 1983.

[BC84] G. Berry and L. Cosserat. The ESTEREL programming language and its mathematical semantics. *Sci. Comput. Programming*, 1984.

[BC91] A. Brogi and P. Ciancarini. The concurrent language shared Prolog. *ACM Trans.*, 13(1):99–123, 1991.

[BD82] R. Bryant and J. Dennis. Concurrent programming. In *Operating Systems Engineering*, pages 426–452, 1982. (Published in *Lecture Notes in Computer Science*, volume 143, Springer Verlag).

[BFG+89] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. METATEM: A Framework for Programming in Temporal Logic. In *Proceedings of REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, Mook, Netherlands, June 1989.

(Published in *Lecture Notes in Computer Science*, volume 430, Springer Verlag).

[BG88] H. Barringer and D. Gabbay. Executing temporal logic: Review and prospects. In *International Conference on Concurrency*, pages 104–105, 1988. (Published in *Lecture Notes in Computer Science*, volume 435, Springer Verlag).

[Car87a] M. Carlsson. Freeze, indexing, and other implementation issues in the WAM. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 40–58, 1987.

[Car87b] N. Carriero. *The Implementation of Tuple Space Machines.* PhD thesis, Yale University, 1987. (Also published as Research Report 567).

[CG85] K. Clark and S. Ggregory. Notes on the implementation of PARLOG. *Journal of Logic Programming*, 2(1):17–42, 1985.

[CG86] K. Clark and S. Gregory. PARLOG: Parallel programming in logic. *ACM TOPLAS*, 8(1):1–49, 1986.

[CG89] N. Carriero and D. Gelernter. How to write parallel programs. *ACM Computing Surveys*, 21(3):323–357, 1989.

[Cia92] P. Ciancarini. Parallel programming with logic languges: A survey. *Comput. Lang.*, 17(4):213–239, 1992.

[CM81] W. Clocksin and C. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, 1981.

[CMZ92] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1:31–50, 1992.

[Con88] J. Conery. Binding environments for parallel logic programs in non-shared memory multiprocessors. *International Journal of Parallel Programming*, 17(2):125–152, 1988.

[Cor90] Intel Scientific Corporation. *IPCS/2 and IPCS/860 User Guide.* INTEL Corporation, Beaverton, OR, 1990.

[Cra88] J. Crammond. *Implememtation of Commited Choice Languages on Shared Memeory Multiprocessors.* PhD thesis, Heriot-Watt University, 1988.

[D+83] E. Dijkstra et al. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16:217–219, 1983.

[Dij71] E. Dijkstra. Hierarchical ordering of sequential programming. *Acta Informatica*, 1:115–138, 1971.

[Eme90] E. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*. Springer-Verlag, 1990.

[F+88] G. Fox et al. *Solving Problems on Concurrent Processors.* Prentice-Hall, Englewood Cliffs, N.J., 1988.

[F+92] I. Foster et al. Productive parallel programming: The PCN approach. *Scientific Programming*, 1:51–66, 1992.

[FB91] M. Fisher and H. Barringer. Concurrent METATEM Processes — A Language for Distributed AI. In *Proceedings of the European Simulation Multiconference*, Copenhagen, Denmark, June 1991.

[FD90] B. Fagin and A. Despain. The performance of parallel Prolog programs. *IEEE Transactions on Computers*, 39(12):1434–1445, 1990.

[Fis93] Michael Fisher. Concurrent METATEM — A Language for Modeling Reactive Systems. In *Parallel Architectures and Languages, Europe (PARLE)*, Munich, Germany, June 1993. Springer-Verlag.

[FKTM86] M. Fujita, S. Kono, H. Tanaka, and T. Motooka. Tokio: Logic programming language based on temporal logic and its compilation to PROLOG. In *Third International Conference on Logic Programming*, pages 695–709, July 1986.

[For92] P. Fortier. *Handbook of LAN Technology.* McGraw-Hill, New York, N.Y., 1992.

[Fos88] I. Foster. Parallel implementation of PARLOG. In *Proceedings of the International Conference of Parallel Processing*, pages 9–16, 1988.

[FT90] I. Foster and S. Taylor. *STRAND: New Concepts in Parallel Programming.* Prentice Hall, New York, 1990.

[G+86] S. Gregory et al. An abstract machine for the implementation of PARLOG on uniprocessors. *Journal of New Generation Computing*, 6(4):389–420, 1986.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[GH79] M. Griss and A. Hearn. A portable standard Lisp compiler. Technical report, University of Utah, 1979.

[Gre87] S. Gregory. *Parallel Logic Programming in PARLOG: The Language and its Implementation.* Addison-Wesley, England, 1987.

[H+90] M. Heath et al. Early experience with the Intel IPSC/860 at oak ridge national laboratory. Technical report, Oak Ridge National Laboratory, 1990.

[HM87] R. Hale and B. Moszkowski. Parallel programming in temporal logic. In *International Symposium on Logic Programming*, pages 277–295, sept 1987.

[Hoa78] C. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[HP79] M. Hennessy and G. Plotkin. Full abstraction for a simple parallel programming language. *LNCS*, 74:108–120, 1979.

[HQ91] P. Hatcher and J. Quinn. *Data-Parallel Programming.* MIT Press, 1991.

[HS86a] A. Houri and E. Shapiro. A sequential abstract machine for Flat Concurrent Prolog. Technical Report CS86-20, Weizmann Institute of Science, Rehovot, Israel, 1986.

[HS86b] P. Hudak and L. Smith. Para-functional programming: A paradigm for programming multiprocessor systems. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, pages 243–254, New York, January 1986. ACM.

[I+87] N. Ichiyoshi et al. A distributed implementation of Flat GHC on the Multi-PSI. In *Proceedings of the 4th International Conference on Logic Programming*, pages 257–275, 1987.

[J+85] R. Jernigan et al., editors. *The Role of Language in Problem Solving.* North-Holland, 1985.

[KC87] Y. Kimura and T. Chikayama. An abstract KL1 machine and its instruction set. In *Proceedings of the IEEE Symposium on Logic Programming*, pages 468–477, 1987.

[Kro87] F. Kroger. *Temporal Logic of Programs.* Springer-Verlag, 1987.

[Lev86] J. Levy. A GHC abstract machine and instruction set. In *Proceedings of the 3rd International Conference on Logic Programming*, pages 157–171, 1986.

[LL87] L. Lamport and N. Lynch. Distributed systems. In *Handbook of Theoretical Computer Science.* Springer-Verlag, 1987.

[Llo84] J. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1984.

[LS94] L. Y. Liu and R. K. Shyamasundar. RT-CDL: A distributed real-time language and its operational semantics. *Computer Languages*, 20(1):1–23, 1994.

[M+85] C. Mierowsky et al. The design and implementation of Flat Concurrent Prolog. Technical Report CS85-09, Weizmann Institute of Science, Rehovot, Israel, 1985.

[Mos83] B. Moszkowski. *Reasoning about Digital Circuits.* PhD thesis, Stanford University, 1983.

[Mos86] B. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, 1986.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.

[Nes93] L. Ness. L.0: A truly concurrent executable temporal logic language for protocals. *IEEE Transactions on Software Engineering*, 19(4):410–423, 1993.

[NT88] M. Nilsson and H. Tanaka. Massively parallel implementation of Flat GHC on the connection machine. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 1031–1040, 1988.

[P+86] L. Pereira et al. Delta Prolog: A distributed backtracking extension with events. In *Third International Conference on Logic Programming*, pages 69–83, July 1986.

[Pap80] S. Pappert. *Mindstorms: Children, Computers, and Powerful Ideas.* Basic Books, 1980.

[Pau82] L. Paulson. A semantics-directed compiler generator. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages.* ACM, 1982.

[Plo81] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[Plo83] G. Plotkin. An operational semantics for CSP. In *Formal Description of Programming Concepts II*, pages 199–223, 1983.

[R+88] T. Reynolds et al. Brave — a parallel logic language for artificial intelligence. *Future Generations Computing*, 4(1):69–75, 1988.

[Ray88] M. Raynal. *Distributed Algorithms and Protocols.* John Wiley Sons, Chichester, England, 1988.

[Reu80] A. Reuveni. *The Event Based Language and its Multiple Processor Implementations.* PhD thesis, MIT, 1980.

[RH90] M. Raynal and J. Helary. *Synchronization and Control of Distributed Systems and Programs.* John Wiley Sons, Chichester, England, 1990.

[RS85] J. Reif and A. P. Sistla. A multiprocess network logic with temporal and spatial modalities. *Journal of Computer and System Sciences*, 30:41–53, 1985.

[RU71] N. Rescher and A. Urquhart. *Temporal Logic.* Springer-Verlag, 1971.

[Sar89] V. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, 1989.

[Sha84] E. Shapiro. Systolic programming: A paradigm of parallel processing. In *Proceedings of the Internation Conference on Fifth Generation Computer Systems*, pages 458–471, Tokyo, January 1984. ICOT.

[Sha86] E. Shapiro. Concurrent Prolog: A progress report. *IEEE Computer*, pages 1–49, 1986.

[Sha87] E. Shapiro. *Concurrent Prolog: Collected Papers.* MIT Press, Cambridge, Mass., 1987.

[Sha89] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, Cambridge, MA, 1986.

[Sta88] W. Stallings. *Local Network Technology.* Computer Society Press, Washington, D.C., 1988.

[Sta90] W. Stallings. *Local and Metgroplolitan Area Networks.* Macmillan, New York, N.Y., 1990.

[Tay89] S. Taylor. *Parallel Logic Programming Techniques.* Prentice-Hall, 1989.

[Tic91] E. Tick. *Parallel Logic Programming.* MIT Press, Cambridge, Massachusetts, 1991.

[TSS87] S. Taylor, S. Safra, and E. Shapiro. A parallel implementation of Flat Concurrent Prolog. *International Journal of Parallel Programming,* 15(3):245–275, 1987.

[Ued86] K. Ueda. Guarded Horn Clauses. In *Logic Programming*, pages 168–179, 1986. (Published in *Lecture Notes in Computer Science*, volume 221, Springer Verlag).

[War83] D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park CA 94025, 1983.

[Y+86] R. Yang et al. P-Prolog: A parallel logic language based on exclusive relation. In *Third International Conference on Logic Programming*, pages 255–269, July 1986.

Appendix A

Code generation

A syntax directed translation scheme is used to generate the NAM object code from the intermediate program. The translation scheme is realized by a collection of functions — each function translates one (or more) of the syntactic categories found in the intermediate program. In particular, $\mathcal{P}$ is the top-level function that translates the input program and its constituent rules; $\mathcal{S}$ translates actions; $\mathcal{G}$ translates basic actions; $\mathcal{C}$ translates constraints; $\mathcal{K}$ translates basic constraints; $\mathcal{H}$ translates multi-use descriptors; $\mathcal{B}$ translates single-use descriptors; $\mathcal{Z}$ translates predefined events; and, finally, $\mathcal{T}$ translates terms. We make use of the function $\overline{\phi}$ which is an extension of ϕ (see chapter 7) to events E. It is defined by $\overline{\phi}(\mathrm{P}(\mathrm{T}^*)) = \phi(\mathrm{P})$.

Note, some of the translation rules use the "no-op" instruction (i.e., NOP) as the target of a branch. This allows us to generate the NAM object code in a single pass over the intermediate program. Once the object code has been generated, most (if not all) of these "no-op" instructions are removed using control-flow optimization, a standard peephole optimization technique[ASU86]. The translation scheme is as follows.

Programs and Rules:

$$\mathcal{P}[\![L_1 : R_1, \ldots, L_n : R_n]\!] \equiv \begin{array}{l} \mathcal{P}[\![L_1 : R_1]\!] \\ \ddots \\ \mathcal{P}[\![L_n : R_n]\!] \\ \mathcal{S}[\![\mathrm{start}()]\!] \end{array}$$

$$\mathcal{P}[\![L : H \Longrightarrow S]\!] = \begin{array}{ll} L: & \mathcal{H}[\![H]\!] \\ & \mathrm{BNZ}(L_1) \\ & \mathcal{S}[\![S]\!] \\ L_1: & \mathrm{END} \end{array}$$

where L_1 is a new label.

$$\mathcal{P}[\![L : H \Longrightarrow C]\!] = \begin{array}{ll} L: & \mathcal{H}[\![H]\!] \\ & \mathrm{BNZ}(L_1) \\ & \mathcal{C}[\![C]\!] \\ L_1: & \mathrm{END} \end{array}$$

where L_1 is a new label.

Multi-use Descriptor:

$$\mathcal{H}[\![E \wedge E_1 \wedge \cdots \wedge E_k]\!] = \begin{array}{l} \mathrm{LOAD}(n) \\ \mathcal{Z}[\![E_1]\!] \\ \ddots \\ \mathcal{Z}[\![E_k]\!] \end{array}$$

where $E \equiv P(X_0, \ldots, X_n)$

Single-use Descriptor:

$$\mathcal{B}[\![G_1 \wedge \cdots \wedge G_n]\!] = \begin{array}{l} \mathcal{G}[\![G_1]\!] \\ \mathrm{SKIP_ENTRY} \\ \mathcal{G}[\![G_2]\!] \\ \ddots \\ \mathrm{SKIP_ENTRY} \\ \mathcal{G}[\![G_n]\!] \end{array}$$

Basic Constraints:

$$\mathcal{K}[\![\neg \mathrm{E}\ \mathrm{if}\ \mathrm{G}_1 \wedge \cdots \wedge \mathrm{G}_k \wedge \neg \mathrm{E}_{k+1} \wedge \cdots \wedge \neg \mathrm{E}_n]\!] = \begin{array}{l} \mathcal{G}[\![\neg \mathrm{E}]\!] \\ \mathrm{SKIP_ENTRY} \\ \mathcal{G}[\![\mathrm{G}_1]\!] \\ \mathrm{SKIP_ENTRY} \\ \quad \ddots \\ \mathrm{SKIP_ENTRY} \\ \mathcal{G}[\![\mathrm{G}_k]\!] \\ \mathrm{SKIP_ENTRY} \\ \mathcal{G}[\![\neg \mathrm{E}_{k+1}]\!] \\ \mathrm{SKIP_ENTRY} \\ \quad \ddots \\ \mathrm{SKIP_ENTRY} \\ \mathcal{G}[\![\neg \mathrm{E}_n]\!] \end{array}$$

$$\mathcal{K}[\![\mathrm{K}_1 \wedge \cdots \wedge \mathrm{K}_n]\!] = \begin{array}{l} \mathrm{ALLOCATE(Drecord)} \\ \mathrm{SKIP_SECTION} \\ \mathcal{K}[\![\mathrm{K}_1]\!] \\ \mathrm{ALLOCATE_RELATED} \\ \mathrm{SKIP_SECTION} \\ \mathcal{K}[\![\mathrm{K}_2]\!] \\ \quad \ddots \\ \mathrm{ALLOCATE_RELATED} \\ \mathrm{SKIP_SECTION} \\ \mathcal{K}[\![\mathrm{K}_n]\!] \end{array}$$

Constraints:

$\mathcal{C}[\![\mathrm{K}]\!]$ =

```
        𝒦[[K]]
        POSTR(local, Xn, L1)
        TERMINATE
  L1 :  SUSPEND_NEXT
        DELETE(local, Xn)
```

where L_1 is a new label and Xn is an unallocated register.

$\mathcal{C}[\![\mathrm{K\ until\ B}]\!]$ =

```
        𝒦[[K]]
        POSTR(local, Xn, L1)
        TERMINATE
  L1 :  ALLOCATE(Drecord)
        PUT_ARG_ADDR(L3)
        SKIP_SECTION
        ℬ[[B]]
  L2 :  MATCH
        SUSPEND
        GOTO(L2)
  L3 :  DELETE(local, Xn)
```

where L_1, L_2, and L_3 are new labels and Xn is an unallocated register.

$\mathcal{C}[\![\Box\mathrm{K}]\!]$ =

```
        𝒦[[K]]
        POST(local, L1)
        TERMINATE
  L1 :  NOP
```

where L_1 is a new label.

$\mathcal{C}[\![\bigcirc\mathrm{C}]\!]$ =

```
  SUSPEND_NEXT
  𝒞[[C]]
```

Events:

$$
\begin{array}{rcl}
\mathcal{G}[\![\mathrm{P}(\mathrm{T}_1,\ldots,\mathrm{T}_n)]\!] & = & \mathrm{EVENT}(\mathrm{P},\mathrm{local}) \\
 & & \mathcal{T}[\![\mathrm{T}_1]\!] \\
 & & \quad\ddots \\
 & & \mathcal{T}[\![\mathrm{T}_n]\!] \\
\mathcal{G}[\![\neg\mathrm{P}(\mathrm{T}_1,\ldots,\mathrm{T}_n)]\!] & = & \mathrm{EVENT}(\neg\mathrm{P},\mathrm{local}) \\
 & & \mathcal{T}[\![\mathrm{T}_1]\!] \\
 & & \quad\ddots \\
 & & \mathcal{T}[\![\mathrm{T}_n]\!] \\
\mathcal{G}[\![\ell_1\cdots\ell_n\,\mathrm{P}(\mathrm{T}_1,\ldots,\mathrm{T}_n)]\!] & = & \mathrm{EVENT}(\mathrm{P},/\ell_1/\cdots/\ell_n) \\
 & & \mathcal{T}[\![\mathrm{T}_1]\!] \\
 & & \quad\ddots \\
 & & \mathcal{T}[\![\mathrm{T}_n]\!] \\
\mathcal{G}[\![\mathrm{nearby}\,\mathrm{P}(\mathrm{T}_1,\ldots,\mathrm{T}_n)]\!] & = & \mathrm{EVENT}(\mathrm{P},\mathrm{nearby}) \\
 & & \mathcal{T}[\![\mathrm{T}_1]\!] \\
 & & \quad\ddots \\
 & & \mathcal{T}[\![\mathrm{T}_n]\!] \\
\mathcal{G}[\![\Diamond\!\!\!\!+\,\mathrm{P}(\mathrm{T}_1,\ldots,\mathrm{T}_n)]\!] & = & \mathrm{EVENT}(\mathrm{P},\mathrm{somewhere}) \\
 & & \mathcal{T}[\![\mathrm{T}_1]\!] \\
 & & \quad\ddots \\
 & & \mathcal{T}[\![\mathrm{T}_n]\!] \\
\mathcal{G}[\![\Diamond\!\!\!\!\dashv\,\mathrm{P}(\mathrm{T}_1,\ldots,\mathrm{T}_n)]\!] & = & \mathrm{EVENT}(\mathrm{P},\mathrm{elsewhere}) \\
 & & \mathcal{T}[\![\mathrm{T}_1]\!] \\
 & & \quad\ddots \\
 & & \mathcal{T}[\![\mathrm{T}_n]\!] \\
\mathcal{G}[\![\boxplus\,\mathrm{P}(\mathrm{T}_1,\ldots,\mathrm{T}_n)]\!] & = & \mathrm{EVENT}(\mathrm{P},\mathrm{everywhere}) \\
 & & \mathcal{T}[\![\mathrm{T}_1]\!] \\
 & & \quad\ddots \\
 & & \mathcal{T}[\![\mathrm{T}_n]\!]
\end{array}
$$

Actions:

$$
\begin{array}{rcll}
\mathcal{S}[\![\mathrm{E}]\!] & = & & \mathrm{ALLOCATE(Erecord)} \\
 & & & \mathrm{PUT_ARG_ADDR}(\mathrm{L}_1) \\
 & & & \quad \ddots \\
 & & & \mathrm{PUT_ARG_ADDR}(\mathrm{L}_n) \\
 & & & \mathrm{SKIP_SECTION} \\
 & & & \mathcal{G}[\![\mathrm{E}]\!] \\
 & & & \mathrm{POST}(\mathrm{local}, \mathrm{L}_0) \\
 & & & \mathrm{TERMINATE} \\
 & & \mathrm{L}_0: & \mathrm{NOP} \\
 & & \multicolumn{2}{l}{\text{where } \mathrm{L}_0 \text{ is a new label and } \overline{\phi}(\mathrm{E}) = \{\mathrm{L}_1, \ldots, \mathrm{L}_n\}.} \\
\mathcal{S}[\![\ell_1 \cdots \ell_n\ \mathrm{E}]\!] & = & & \mathrm{ALLOCATE(Erecord)} \\
 & & & \mathrm{PUT_ARG_ADDR}(\mathrm{L}_1) \\
 & & & \quad \ddots \\
 & & & \mathrm{PUT_ARG_ADDR}(\mathrm{L}_n) \\
 & & & \mathrm{SKIP_SECTION} \\
 & & & \mathcal{G}[\![\mathrm{E}]\!] \\
 & & & \mathrm{POST}(/\ell_1/ \cdots /\ell_n, \mathrm{L}_0) \\
 & & & \mathrm{TERMINATE} \\
 & & \mathrm{L}_0: & \mathrm{NOP} \\
 & & \multicolumn{2}{l}{\text{where } \mathrm{L}_0 \text{ is a new label and } \overline{\phi}(\mathrm{E}) = \{\mathrm{L}_1, \ldots, \mathrm{L}_n\}.} \\
\mathcal{S}[\![\mathbf{nearby}\ \mathrm{E}]\!] & = & & \mathrm{ALLOCATE(Erecord)} \\
 & & & \mathrm{PUT_ARG_ADDR}(\mathrm{L}_1) \\
 & & & \quad \ddots \\
 & & & \mathrm{PUT_ARG_ADDR}(\mathrm{L}_n) \\
 & & & \mathrm{SKIP_SECTION} \\
 & & & \mathcal{G}[\![\mathrm{E}]\!] \\
 & & & \mathrm{POST}(\mathrm{nearby}, \mathrm{L}_0) \\
 & & & \mathrm{TERMINATE} \\
 & & \mathrm{L}_0: & \mathrm{NOP} \\
 & & \multicolumn{2}{l}{\text{where } \mathrm{L}_0 \text{ is a new label and } \overline{\phi}(\mathrm{E}) = \{\mathrm{L}_1, \ldots, \mathrm{L}_n\}.}
\end{array}
$$

$$
\begin{array}{rrl}
\mathcal{S}[\![\oplus E]\!] & = & \text{ALLOCATE(Erecord)} \\
& & \text{PUT_ARG_ADDR}(L_1) \\
& & \quad \ddots \\
& & \text{PUT_ARG_ADDR}(L_n) \\
& & \text{SKIP_SECTION} \\
& & \mathcal{G}[\![E]\!] \\
& & \text{POST(somewhere}, L_0) \\
& & \text{TERMINATE} \\
& L_0: & \text{NOP} \\
& & \text{where } L_0 \text{ is a new label and } \overline{\phi}(E) = \{L_1, \ldots, L_n\}. \\
\mathcal{S}[\![\Diamond E]\!] & = & \text{ALLOCATE(Erecord)} \\
& & \text{PUT_ARG_ADDR}(L_1) \\
& & \quad \ddots \\
& & \text{PUT_ARG_ADDR}(L_n) \\
& & \text{SKIP_SECTION} \\
& & \mathcal{G}[\![E]\!] \\
& & \text{POST(elsewhere}, L_0) \\
& & \text{TERMINATE} \\
& L_0: & \text{NOP} \\
& & \text{where } L_0 \text{ is a new label and } \overline{\phi}(E) = \{L_1, \ldots, L_n\}. \\
\mathcal{S}[\![\boxplus E]\!] & = & \text{ALLOCATE(Erecord)} \\
& & \text{PUT_ARG_ADDR}(L_1) \\
& & \quad \ddots \\
& & \text{PUT_ARG_ADDR}(L_n) \\
& & \text{SKIP_SECTION} \\
& & \mathcal{G}[\![E]\!] \\
& & \text{POST(everywhere}, L_0) \\
& & \text{TERMINATE} \\
& L_0: & \text{NOP} \\
& & \text{where } L_0 \text{ is a new label and } \overline{\phi}(E) = \{L_1, \ldots, L_n\}.
\end{array}
$$

$$
\begin{array}{rrl}
\mathcal{S}[\![\mathrm{S}\ \textbf{atnext}\ \mathrm{B}]\!] = & & \mathrm{ALLOCATE(Drecord)} \\
& & \mathrm{PUT_ARG_ADDR}(L_1) \\
& & \mathrm{SKIP_SECTION} \\
& & \mathcal{B}[\![\mathrm{B}_1]\!] \\
& L_0: & \mathrm{MATCH} \\
& & \mathrm{SUSPEND} \\
& & \mathrm{GOTO}(L_0) \\
& L_1: & \mathcal{S}[\![\mathrm{S}_1]\!] \\
\mathcal{S}[\![\mathrm{S}_1\ \textbf{atnext}\ \mathrm{B}_1 \vee \cdots \vee \mathrm{S}_n\ \textbf{atnext}\ \mathrm{B}_n]\!] = & & \mathrm{ALLOCATE(Drecord)} \\
& & \mathrm{PUT_ARG_ADDR}(L_1) \\
& & \mathrm{SKIP_SECTION} \\
& & \mathcal{B}[\![\mathrm{B}_1]\!] \\
& & \mathrm{ALLOCATE_RELATED} \\
& & \mathrm{PUT_ARG_ADDR}(L_2) \\
& & \mathrm{SKIP_SECTION} \\
& & \mathcal{B}[\![\mathrm{B}_2]\!] \\
& & \ddots \\
& & \mathrm{ALLOCATE_RELATED} \\
& & \mathrm{PUT_ARG_ADDR}(L_n) \\
& & \mathrm{SKIP_SECTION} \\
& & \mathcal{B}[\![\mathrm{B}_n]\!] \\
& L_0: & \mathrm{MATCH} \\
& & \mathrm{SUSPEND} \\
& & \mathrm{GOTO}(L_0) \\
& L_1: & \mathcal{S}[\![\mathrm{S}_1]\!] \\
& & \mathrm{GOTO}(L_{n+1}) \\
& L_2: & \mathcal{S}[\![\mathrm{S}_2]\!] \\
& & \mathrm{GOTO}(L_{n+1}) \\
& & \ddots \\
& L_n: & \mathcal{S}[\![\mathrm{S}_n]\!] \\
& L_{n+1}: & \mathrm{NOP}
\end{array}
$$

$$\begin{aligned}
\mathcal{S}[\![\odot \mathrm{S}]\!] &= \mathrm{SUSPEND_NEXT} \\
&\quad \mathcal{S}[\![\mathrm{S}]\!] \\
\mathcal{S}[\![\diamond \mathrm{S}]\!] &= \mathrm{SUSPEND} \\
&\quad \mathcal{S}[\![\mathrm{S}]\!] \\
\mathcal{S}[\![\mathrm{S}_1 ; \mathrm{S}_2]\!] &= \mathcal{S}[\![\mathrm{S}_1]\!] \\
&\quad \mathrm{SUSPEND} \\
&\quad \mathcal{S}[\![\mathrm{S}_2]\!]
\end{aligned}$$

Appendix B

This appendix describes the suite of benchmark programs that was used to collect the results presented in tables 7.7–7.14. All of the benchmark programs are written in the *data-parallel* style[HQ91]. In the data-parallel style, the input data is distributed evenly across all the processors and each processor executes the same algorithm on its piece of the input. The individual results are then combined to form the final result. This style of programming is easily expressed in NETLOG and can be used to rapidly prototype transformational programs.

Several predefined state variables are used in the programs below. In particular, each processor can access state variable Np′ to reference the total number of processors in the network; similarly, each processor can access state variable Pid′ to reference an integer (between 0 and Np′-1) which uniquely identifies the executing processor. We also make use of several predefined events. `split`(N,L_1,L_2) partitions the list L_1 into seperate N element lists and binds the result to L_2 (e.g., `split`(2,[1,2,3,4],L) would result in L=[[1,2],[3,4]]), and `join`(L_1,L_2) performs the reverse operation (e.g., `join`([[1,2],[3,4]],L) gives L=[1,2,3,4]). The expression L[n] returns the nth element of list L. A brief description of each program is given below.

Program 1: Computing the sum of two vectors

The following program computes the sum of two vectors, A and B, of length N. The input vectors are represented as lists. The input vectors to be added by a particular processor is represented by sum(A,B). The result is represented by sum(A,B,C), where C is the vector that results from adding A and B.

```
{ sum([],[]) ⟹ sum([],[],[]),
  sum([A | As],[B | Bs]) ⟹ sum(As,Bs) ;
                            sum([A | As],[B | Bs],[A+B | Cs]) atnext sum(As,Bs,Cs),
  main(N,A,B) ⟹ nlist(Np',Cs) ;
                 split(N/Np',A,As) ;
                 split(N/Np',B,Bs) ;
                 ⊞sum(As[Pid⌉],Bs[Pid⌉]) ;
                 wait ⊞sum(As[Pid⌉],Bs[Pid⌉],Cs[Pid⌉]) ;
                 join(Cs,C) ;
                 write(C)
}
assert
  read(N,A,B) ; main(N,A,B)
```

Program 2: Reversing a list

The following program reverses the elements of a list L. The computation performed on each processor is initiated by an event of the form `rev(L,?)`. The result is given by `rev(L,R)`, where R is the reverse of L. These individual results are then combined (in reverse order) to form the final reversed list.

```
{ rev(X,?) ⟹ rev(X,X,[]),
  rev(X,[],L) ⟹ rev(X,L),
  rev(X,[Y|Ys],L) ⟹ rev(X,Ys,[Y|L]),
  main(N,L) ⟹ nlist(Np',Rs);
                split(N/Np',L,Ls);
                ⊞rev(Ls[Pid'],?);
                wait ⊞rev(Ls[Pid'],Rs[Np' − (Pid' + 1)]);
                join(Rs,R);
                write(R)
}
assert
  read(N,L); main(N,L)
```

Program 3: Delete all occurrences of X from list L

The following program prints the elements remaining after deleting all multiples of X from list L. The computation on each processor is initiated by an event of the form `del`(X,L). The result is given by `del`(X,L,M), where M is the list of elements remaining after deleting all multiples of X from L.

```
{ del(X,[]) ⟹ del(X,[],[]),
  del(X,[L|Ls]) ∧ L%X=0 ⟹ del(X,Ls); del(X,[L|Ls],Ms) atnext del(X,Ls,Ms),
  del(X,[L|Ls]) ∧ L%X≠0 ⟹ del(X,Ls); del(X,[L|Ls],[L|Ms]) atnext del(X,Ls,Ms),
  main(N,X,L) ⟹ nlist(Np',Ms);
                 split(N/Np',L,Ls);
                 ⊞del(X,Ls[Pid']);
                 wait ⊞del(X,Ls[Pid'],Ms[Pid']);
                 join(Ms,M);
                 write(M)
}
assert
  read(N,X,L); main(N,X,L)
```

Program 4: Generate primes between 1 and N

The following program computes the prime numbers between 1 and N. The computation on each processor is initiated by an event of the form `primes`(Is,?), where Is is a list of integers. The result is `primes`(Is,Ps), where Ps is the list of primes found in Is. The predefined event `cond`(B,A_1,A_2,K) binds K to A_1 if B is true and binds K to A_2 if B is false.

```
{ primes([],?) ⟹ primes([],[]),
  primes([I|Is],?) ⟹ relprime(I,[2,...,sqrt(I)],?);
                      primes(Is,?);
                      wait (relprime(I,[],B) also primes(Is,Ps) ∧ Ps≠?);
                      cond(B,[I|Ps],Ps,K);
                      primes([I|Is],K),
  relprime(P,[],?) ⟹ relprime(P,[],true),
  relprime(P,[I|Is],?) ∧ P%I=0 ⟹ relprime(P,[],false),
  relprime(P,[I|Is],?) ∧ P%I≠0 ⟹ relprime(P,Is,?),
  main(N,L) ⟹ nlist(Np',Ps)';
               split(N/Np',L,Ls);
               ⊞primes(Ls[Pid'],?);
               wait ⊞primes(Ls[Pid'],Ps[Pid']);
               join(Ps,P);
               write(P)
}
assert
  read(N); main(N,[1,...,N])
```

Program 5: Matrix multiplication

The following program multiplies two $N \times N$ matrices, A and B, and displays the result. The input and output matricies are represented using a list of lists. Input matrix B is assumed to be in transposed form. The computation on each processor is initiated by an event of the form mm(A,B), where A and B are matrices. The result is mm(A,B,C), where matrix C is the product of A and B.

```
{ mm([],B) ==> mm([],B,[]),
  mm([A|As],B) ==> vm(A,B);
                   mm(As,B);
                   mm([A|As],B,[C|Cs]) atnext vm(A,B,C) also mm(As,B,Cs),
  vm(A,[]) ==> vm(A,[],[]),
  vm(A,[B|Bs]) ==> ip(A,B,C);
                   vm(A,Bs);
                   vm(A,[B|Bs],[C|Cs]) atnext vm(A,Bs,Cs),
  print(Rs) ==> join(Rs,R); write(R),
  main(N,A,B) ==> nlist(Np',Cs);
                  split(N/Np',A,As);
                  ⊞mm(As[Pid⌉],B);
                  print(Cs) afternext ⊞mm(As[Pid⌉],B,Cs[Pid⌉])
}
assert
  read(N,A,B); main(N,A,B)
```