

THE UNIVERSITY OF CALGARY

**CSP* – A DISTRIBUTED LOGIC
PROGRAMMING LANGUAGE
FOR DISCRETE EVENT SIMULATION**

BY

Xining Li

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

JUNE, 1989

© Xining Li 1989



National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

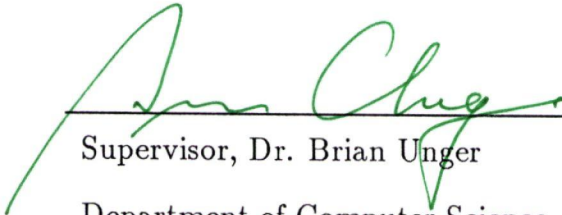
L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

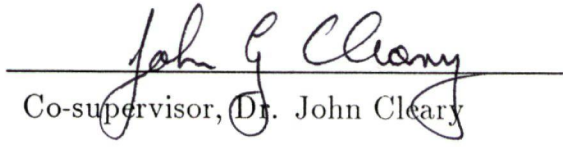
ISBN 0-315-54272-1

THE UNIVERSITY OF CALGARY
FACULTY OF GRADUATE STUDIES

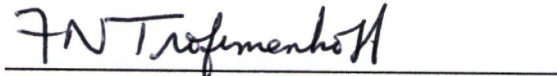
The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "CSP* - A Distributed Logic Programming Language for Discrete Event Simulation" submitted by Xining Li in partial fulfillment of the requirements for the degree of Doctor of Philosophy.



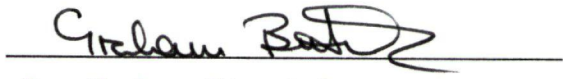
Supervisor, Dr. Brian Unger
Department of Computer Science



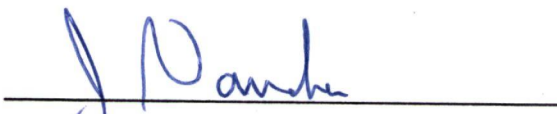
Co-supervisor, Dr. John Cleary
Department of Computer Science



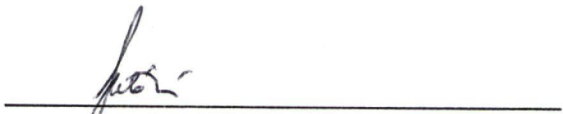
Dr. Fred Trofimenkoff
Department of Electrical Engineering



Dr. Graham Birtwistle
Department of Computer Science



Dr. Jean Vaucher
University of Montreal



Dr. Ivan Futo
Multilogic Computing Ltd. Budapest

Date: 17 July 1989

ABSTRACT

Computer simulation is a technique for predicting the behavior of real or hypothetical systems as these systems operate in real or hypothetical environments. The development of a large simulation is a complex and difficult task. In many research fields, expensive computers and human resources are devoted exclusively to simulations.

The objective of this thesis is to reduce the costs of simulation in two ways: simplifying simulation development by providing a new programming language which can be used for both model specification and program implementation, and enabling the use of more cost effective parallel computers to execute simulations.

First, an abstract distributed logic programming model is described. The model is based on first order logic theory with extensions for temporal, cooperative execution. Within this language framework, the programmer is able to describe simulation models in a declarative way.

Secondly, an implementation of the language framework is presented. Based on an optimistic synchronization mechanism – the Time Warp system [Jef85], the implementation not only provides a temporal coordinate system for measuring computational progress and defining synchronizations but also provides a spatial coordinate system for supporting nondeterministic computations.

Finally, a practical language proposal – Communicating Sequential Prolog (CSP*) is introduced. Several programming examples of discrete event simulation reveal the simplicity, flexibility, understandability and expressive power of CSP*.

ACKNOWLEDGEMENTS

I would like to express great thanks to my supervisor, Dr. Brian Unger, for his support and guidance over the course of this research. His editorial suggestions made a substantial improvement to the literary quality of this dissertation. His interest in and excitement over the topic of this dissertation was often inspirational.

I would also like to extend special thanks to Dr. John Cleary for providing valuable advice and suggestions during this long effort. I do not believe this work have been possible without his insights and assistance.

I am deeply indebted to my wife, Ling Liu, for her love and continuous encouragement.

I would like to thank my parents, my parents-in-law and my son. Although I have not seen them for four years, I could feel their love and understanding over time and space.

Finally, I would like to thank Greg Lomow, Darrin West, Zhongge Xiao and Mike Bonham for their helpful discussions and comprehensive comments.

Contents

Approval Page	ii
ABSTRACT	iii
ACKNOWLEDGEMENTS	iv
List of Figures	vii
1 INTRODUCTION	1
1.1 The Simulation Development Process	2
1.2 Thesis Motivation and Objectives	3
1.3 Thesis Outline	5
2 SURVEY OF RELATED WORK	8
2.1 Distributed Programming Languages	9
2.1.1 The Procedural Programming Languages	10
2.1.2 The Declarative Programming Languages	13
2.2 Synchronization Mechanisms	17
2.2.1 Logical Clocks	18
2.2.2 The Network Paradigm	20
2.2.3 The Time Warp Mechanism	22
2.3 Base System and General Comparisons	25
2.3.1 A Base Language and Synchronization Scheme	26
2.3.2 Comparisons with Closely Related Approaches	28
3 A DISTRIBUTED LOGIC PROGRAMMING MODEL	33
3.1 The Model	33
3.1.1 Declarative Semantics	37
3.1.2 Operational Semantics	42
3.1.3 Temporal Constructs	45
3.2 Implementation Issues	48
3.2.1 Simulation Time and The Computation Rule	48
3.2.2 Communication and Synchronization	49
3.2.3 The Sensor	51
3.3 The Basic System Structure	53

4	A MODIFIED TIME WARP KERNEL	55
4.1	Jefferson's Time Warp Kernel	55
4.1.1	Local Control	58
4.1.2	Global Control	60
4.2	Modifications and Extensions	61
4.2.1	The Extended Kernel	62
4.2.2	Global Backtracking and Termination Functions	65
5	A LOGIC PROCESS INTERPRETER	68
5.1	New Concepts and Data Structures	69
5.2	The Algorithm	78
5.2.1	Forward Execution	79
5.2.2	Backward Execution	81
5.2.3	An Example	88
5.3	Correctness of the Algorithm	91
6	COMMUNICATING SEQUENTIAL PROLOG (CSP*)	105
6.1	Basic Constructs and Programming Style	105
6.1.1	Syntax and Semantics	106
6.1.2	Process Naming, Creation and Destruction	107
6.1.3	Built-in Predicates	110
6.1.4	Pragmatics	114
6.2	Discrete Event Simulation in CSP*	118
6.2.1	A Resource Allocation Process	118
6.2.2	A Queue Process	121
6.2.3	Random Number Generator	124
6.2.4	Single Server Queueing Model	125
6.2.5	Bank Robbery	128
6.2.6	Hierarchical Health Care System	130
6.3	Summary	134
7	CONCLUSIONS AND DIRECTIONS FOR FUTURE WORK	136
7.1	Conclusions	136
7.2	Future Work	140
	Bibliography	142
	APPENDIX	147

List of Figures

3.1	The Proof-trees of the Example	39
3.2	The Temporal Property of Procedure Calls	47
3.3	The Basic System Structure	54
4.1	The Interfaces of Kernel Process	62
5.1	A Parent-Child Layer	70
5.2	A Possible Partial ξ -tree of P_i	71
5.3	The Relationship between t_b , $status$ and ξ_j^n	72
5.4	An Example for Constructing a State	75
5.5	The ξ -trees of the Example	88
5.6	Example of Process Transitions	95
5.7	Transition-tree of the Example	95
6.1	A B-tree Network	109
6.2	A Homogenous Nondeterministic Computation Model	115
6.3	A Single Server Queueing Model	126
6.4	A Health Care Model	130

Chapter 1

INTRODUCTION

Simulation is a problem solving technique that involves modeling a dynamic system and observing its behavior over time. A system may be defined as a collection of inputs which pass through certain processing phases to produce outputs. For example, a manufacturing system may use crude oil as an input to a cracking plant for crude oil processing that produces various types of oil and gasoline as outputs.

A model of a system is an abstraction of the system which can be a theoretical representation, an empirical representation, or a combination of both. Simulation enables experimentation with systems which do not yet exist, or for which it is difficult to get first hand experience. It also enables repeated experimentations with a system, under controlled conditions, to optimize performance.

Three classes of simulation can be defined: discrete, continuous and combined. Discrete event simulation refers to a modeling technique that enables instantaneous changes in the state of a model to be made at arbitrary points in time. Continuous simulation implies that changes in the state of a model occur smoothly and continuously in time. Combined simulation is a technique which simulates systems with both discrete and continuous characteristics.

It is possible to write simulation programs in computer languages such as SIMULA, GPSS, *etc.*. However, simulationists tend to prefer a programming environment which reduces the time devoted to simulation programming and debugging, as well as, the time of simulation execution. This thesis concerns the design of such a

programming system in the domain of discrete event simulation.

1.1 The Simulation Development Process

In general, practical simulation work involves the following steps:

1. **Definition of the problem and simulation objectives.** This step concerns what questions need to be answered, what aspects of system behavior need to be modeled, and what results or performance measures need to be observed as outputs.
2. **Model specification and data collection.** A system can be modeled through a decomposition process in which model components and the interactions among these components are defined. Model specification involves the correct description of component representations and a set of transformation rules which define the behavior and relationships among the set of components comprising the system. Data should be collected from the system of interest and used to estimate input parameters and to obtain probability distributions for the random variables used in the model.
3. **Construction of a computer program.** Operation of the system model is represented by the execution of a program written in some programming language. A good simulation language may reduce the required programming effort significantly and may also lead to a shorter simulation execution time.
4. **Verification and validation of the simulation program and model.** Verification refers to the consistency of the model with the model specification.

Validation involves determining whether or not the behavior of the simulation program mimics the behavior of the system being investigated with acceptable accuracy.

5. **Simulation experiments and analysis.** Production runs are made to provide performance data specified by steps 1 and 2. Statistical techniques are used to analyze the output data from the production runs. Typical goals are to construct a confidence interval for a measure of performance for one particular system design or to decide which simulated system is best relative to some specified measure of performance.
6. **Documentation and implementation of the results.** Because simulation models are often used for more than one application, it is important to document the assumptions, inputs, outputs, results and conclusion drawn from a set of experiments for further analysis.

1.2 Thesis Motivation and Objectives

Traditionally, people follow the above systematic procedure to carry out a simulation task. They use a natural language or a kind of formal language such as automata or Petri nets to specify the simulation model. Then they convert the specification into a program through either a simulation-oriented or a general purpose programming language.

Since there are gaps between the model and the model specification, and between the specification and the program implementation, substantial effort is required to verify model correctness and to validate the simulation model. Current estimates

indicate that as much as 75 percent of a typical programming budget goes to program modification and debugging [Gol85]. As a result, with the increased sophistication demanded from simulation models, it is increasingly complex, difficult and costly to carry out the above simulation steps.

Thus, the first motivation of this research is drawn from the question: is it possible to find a tool both for model specification and program implementation. If we can provide such a tool, steps 2, 3 and partly 4 in the simulation development process are integrated into one step. Thereby the simulation development shifts away from the traditional concern with the consistency of the model, the specification and the implementation, and towards an understanding of the simulation problem itself. That is, a tool which enables the simulationist to specify a model in a problem oriented language which can also be directly executed can greatly simplify the simulation development process.

Secondly, most practical simulations take a long time to execute because useful models tend to be large and complex, and because their simulation programs are executed sequentially. However, interesting classes of simulation models exhibit a high degree of natural parallelism, *i.e.*, they can be decomposed into a set of concurrently operating objects. Typical examples are health care systems, traffic control systems, communication systems, computer systems, banking systems, and most daily human activities. This fact, and the emergence of highly parallel, distributed computer systems, have led many scientists to attempt distributed, parallel, solutions to simulation problems.

The goal of distributed simulation is to speed up simulation by exploiting the availability of more cost-effective parallel computer systems. This is made possible

by the parallelism inherent in many real systems and their models.

Keeping these motivations in mind, the common thread that runs through this thesis is an attempt to provide simulation programmers with a new programming language that makes it *easier* to design a simulation whose execution can be made arbitrarily *fast*.

1.3 Thesis Outline

Reaching our objectives involves two major steps: constructing a language framework to specify a set of parallel activities and providing an efficient synchronization mechanism to coordinate these activities. Chapter 2 surveys some recent proposals which deal with these subjects. From this investigation and discussion, we conclude that a distributed logic programming language in conjunction with a run-time kernel based on the Time Warp mechanism offers the greatest potential to achieve our goals.

The programming model presented in Chapter 3 provides a framework for discussing distributed logic programming. The model is not yet a practical logic programming language, although it does capture the important aspects of a distributed logic programming system. In the model, a distributed logic program is represented by a *virtual space* – a set of processes which are logic representations of system objects and are coordinated through communication and synchronization. The semantics of the model are based on first order logic theory, which is extended to handle problems in the dynamic and parallel domains. Since the communication facilities in the model cannot be defined with first order logic, a meta-logic rule is introduced to

check the synchronization of underlying processes. Issues of implementing a practical distributed logic programming language are also addressed in this chapter, which include the temporal construct, communication predicates, completeness and the global backtracking capability.

Chapter 4 introduces a modified version of the Time Warp mechanism which implements *virtual time* for organizing and synchronizing distributed systems, typically distributed discrete event simulations. This new version maintains some of the original functions, such as manipulating input/output queues, recognizing rollback requirements and treating anti-messages, but leaves the state-saving and rollback mechanism to the language level. It also provides new functions to handle the cooperation of logic processes in a *virtual space*.

By combining the new version of the Time Warp mechanism with the proposed distributed logic programming model, a logic process interpreter algorithm is described in Chapter 5. This algorithm is a standard Prolog interpreter that is extended to control the rollback and global backtracking activities of a logic process. The rollback facility is used to deal with failures on *virtual time* while the global backtracking facility is used to handle failures on *virtual space*.

The soundness and partial completeness of the algorithm are also proved in Chapter 5. It is shown that the proposed distributed logic programming system not only provides a temporal coordinate system which can be used to measure computational progress and to define synchronizations among logic processes, but also provides a spatial coordinate system to support distributed, nondeterministic computations.

Chapter 6 presents a practical language proposal – Communicating Sequential Prolog, abbreviated to CSP*. CSP* is a distributed logical programming language

for discrete event simulation. It inherits most of its features from standard Prolog and provides a process-oriented programming environment to users.

By partially exploiting the AND-parallelism of logic programming, a CSP* program consists of a set of dynamically spawned sequential processes which act as autonomous objects and cooperate through message passing. Execution of a CSP* program relies on a set of extended interpreters proposed in Chapter 5 which evaluate their assigned logic processes in parallel and allow backtracking within processes to be combined with concurrent activities among processes. Some programming examples are described which reveal the simplicity and expressive power of CSP*.

I have tried to avoid demanding a wide background of the reader. However, as this thesis deals with a great range of subjects – programming languages, communication and synchronization mechanisms, logic programming theory, and discrete event simulation, some knowledge of these areas, especially logic programming, is necessary to appreciate the problem investigated in the thesis.

Chapter 2

SURVEY OF RELATED WORK

Much of human knowledge about the real world is concerned with the way things are done. This knowledge is often described as a set of cooperative action sequences for achieving a particular goal and is usually relevant to a kind of *time* metrology. Since these action sequences can be simulated by computers and coordinated through communications, we call each action sequence a *process* and a set of these cooperative action sequences a *distributed computing system*.

Of course, discrete event simulation is an important application area of distributed computing systems. When we seek an ideal distributed programming system for discrete event simulation, we are drawn to the following questions:

1. What theoretical language model offers the opportunity for programmers to create running programs by providing specifications of simulation models, without having to proceed with the transformation sequence “model \rightarrow specification \rightarrow implementation”?
2. What language constructs are suitable to describe simulation models such that they not only make programs easier to understand and debug, but also make it easier to characterize programs at a high level of abstraction in a natural way?
3. What synchronization mechanism can best support the cooperation of processes and be implemented efficiently?

4. What inter-process communication facility can best describe the dynamic behavior of distributed systems?

In this chapter, we survey some of the models and techniques which have been proposed for answering these questions. We first examine different distributed programming languages, and then explore several process synchronization mechanisms. The purpose of the investigation is to find possible alternatives for achieving our goal.

2.1 Distributed Programming Languages

The usual way to give directions to a computer system is with a program written in some programming language. Traditional programming languages for discrete-event simulation are sequential or pseudo-concurrent languages. Some of the more popular discrete simulation languages are GPSS [IBM77], SIMULA [Bir79], and SIMSCRIPT [Fis78]. These languages are procedural, *i.e.*, a program explicitly specifies the steps which must be performed to reach a solution. Another kind of language has recently been used in simulation, *e.g.*, T-Prolog [FS82]. It is a declarative language, *i.e.*, it is only necessary to describe the problem in terms of facts and rules that define relationships among the objects in question.

Recently, many proposals have been put forward for distributed programming, including CSP [Hoa78], DP [Han78], PLITS [Fel79], E-CLU [Lis79], *MOD [Coo80], Cell [Sil81], Soma [Kes81], NIL [SY85], CSM [SL87], Ada¹ [DOD80], PARLOG [CG86], Concurrent Prolog [Sha83], GHC [Ued85] and CS-TProlog [Fut88].

¹Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

These languages all support concurrent computations. An interesting phenomenon is that almost all of these languages use the concept of communicating sequential processes (explicitly or implicitly) to show concurrency, although they may use different names for the same concept.

In this section, we shall briefly survey these languages by dividing them into two groups, *i.e.*, the procedural and the declarative programming languages.

2.1.1 The Procedural Programming Languages

The distributed procedural programming languages inherit most features from conventional programming languages. These include support for abstraction, particularly abstract objects; support for modularization, including separate compilation of modules; support for sequential execution flow control; support for strong typing and data encapsulation; support for block scoping and information hiding; and support for error and exception handling.

However, as a distributed program resides and executes at communicating, but geographically distinct, nodes of a network, a distributed programming language must provide the functions of distribution and communication/synchronization. These features constitute the major differences from sequential programming languages.

Distribution means process dynamics which describes the change in number and variety of processes through the execution of a distributed program. Two methods are commonly used to create new processes.

Some languages allow programs to create new processes during execution (dynamic processes). The syntactic mechanisms supporting dynamic process creation are explicit allocation and lexical program elaboration. Languages with explicit al-

location have a statement to create a new process, such as Ada, NIL and PLITS. Lexical elaboration creates processes by combining declarations with recursive program structures. That is, if procedure p declares process q and then calls itself recursively, the recursive invocation of p creates another instance of q . Cell, Ada, and PLITS create new processes by lexical elaboration. Dynamic process creation is flexible and can be efficient. However, since processes are created and destroyed during execution, it is more difficult to debug such a program.

Another method requires that all processes are spawned at system creation (static processes). Languages such as CSP, DP, *MOD, and Soma adopt this method. Static process creation makes it convenient to analyze the inter-process communication structure at compile time, and the overall program is easier to understand. However, it is inconvenient to describe large programs involving thousands of processes, and difficult to characterize systems whose components vary with time.

Logically, synchronization can be defined as the establishment of some form of agreement among a set of processes. For example, in discrete event simulation, process events at a given simulation time may depend on events that occur at earlier simulation time. Even though applications may have different criteria for synchronization, inter-process message passing is a common mechanism for establishing synchronization in distributed systems.

Communication facilities at the language level can be classified as synchronous and asynchronous. In a synchronous scheme, every communication request is matched by a reception; a process cannot send the second message until the first one has been handled. In an asynchronous scheme, processes send messages without regard to their reception; a process is free to send a message and continue computing. PLITS

and Soma use an asynchronous communication scheme. NIL provides both synchronous and asynchronous communication. The most others adopt synchronous communication.

Asynchronous communication is primitive and natural in simulating a changing world. It also offers the potential to explore the maximum parallelism in any simulation model. However, a special mechanism is required to handle the message overflow problem, because it is possible that an asynchronous communication system will create an unlimited number of messages.

Other issues in distributed programming languages are communication connection and message control. These two issues have a close relationship for accomplishing communication and establishing synchronization among processes.

Communication connection is a naming problem. Three different mechanisms – ports, names and entries, are used to channel communication. Communication through a special typed symbol is communication through a port. A port can be referenced by communicating processes through global declaration or ownership transfer. *MOD, PLITS, NIL, Soma and E-CLU use ports (possibly using other names such as “mailbox”). Several languages – Ada, Cell, and DP, focus communication on an entry in the called process (another name for this mechanism is remote procedure call). A called process can have several entries and accept requests from them in an order determined by program control. CSP uses process names to communicate directly. In order to exchange messages, two processes must identify each other by their names in input and output statements. In this case, even though the communication connection looks explicit, the lack of anonymous communication makes it difficult to build program libraries.

Message control concerns the actions that processes take to communicate, including the facilities they have for choosing a communication partner, segregating and decoding incoming messages. For example, CSP treats processes as equals. It introduces asymmetric unidirectional message flow. Input guards provides concurrency control. Alternative commands combined with input guards can segregate incoming messages indeterministically. Other languages specify roles for the “calling” and “called” processes. Ada, Cell and PLITS allow the called process some freedom in choosing which request to serve because all incoming messages are segregated into groups by entry queues.

Discrete event simulation systems have been built using some of these languages, such as CSP [KH85] and Ada [ULB84]. This practical work shows that the concepts of dynamic process and inter-process communication embedded within a procedural programming language provides the user with a wide range of powerful and completely general facilities to accomplish discrete event simulations.

However, as distributed procedural programming languages have complex syntax and informal semantics, they are not adequate for model specification, and it remains difficult to verify and validate procedural simulation applications.

2.1.2 The Declarative Programming Languages

Declarative programming languages separate the logic and control aspects of an algorithm, hide control details from programming, and allow very high-level descriptions of desired relationships among values. A logic programming language is a typical declarative programming language in the sense that its clauses are interpreted on first order logic. In this section, we focus our attention on varieties of logic programming

languages.

Prolog, a manifestation of logic programming, was developed about 10 years ago. It has been very popular in Europe and is now targeted as the core language of the Japanese Fifth Generation Computer Project. Several Prolog-like systems have been used for discrete event simulation [CGU85, BG84, VL87]. The major advantages of Prolog-like languages are summarized as follows:

1. They provide declarative semantics based on logic in addition to the usual procedural semantics. It is possible to use them as executable specification languages.
2. Program and data are identical in form and therefore can be easily manipulated.
3. Arguments of clauses are not fixed as input or output parameters as in procedural programming languages and clauses may have multiple inputs and outputs.
4. Backtracking is used to find a complete set of solutions for a given problem. Therefore, nondeterministic computation becomes a natural application area.
5. The basic elements (atoms, variables and compound terms) provide a general and flexible data structure superior to the arrays and records used in procedural programming languages.
6. The language designs are well suited to parallel search and are, therefore, excellent candidates for future powerful computers incorporating parallel processing.
7. Programs are usually significantly shorter than programs written in most procedural programming languages (typically 1/5 to 1/10 the size).

However, logic programming techniques are not yet fully mature. The execution of a logic program usually requires more memory and faster execution capabilities than an equivalent procedural program. Furthermore, Prolog has been shown to be marvelous for describing static knowledge but difficult and awkward for specifying large dynamic systems such as simulation models. For example, standard Prolog does not provide facilities for describing model dynamics, hiding information, decomposing simulation components, and manipulating time-dependent activities, though these facilities are very important in simulation specification and implementation.

In order to speed up the execution of logic programs and apply logic programming techniques to large and dynamic systems, both concurrent logic programming and object(process)-oriented logic programming have become attractive research areas.

Logic programming offers two kinds of parallelism: AND-parallelism is the parallel solution of more than one goal in a given goal sequence; OR-parallelism is the parallel creation of many solutions for a given goal. These two kinds of parallelism are a consequence of nondeterminism in logic programming: we are free to choose any order in which to satisfy several subgoals in the body of a clause; and, when evaluating a selected subgoal, we are free to choose any clause which can match the subgoal.

There have been attempts to design systems using either one of these types of parallelism (or a combination of both) [CG86, Sha83, Ued85, Con87]. These proposals inherit most syntactic and semantic features of the standard Prolog, use implicit processes to exploit concurrency and shared variables as communication channels among processes.

Problems in the implementation of OR-parallelism are the combinatorial explo-

sion in the number of processes and the representation of variable binding environments. AND-parallelism, although offering advantages such as being able to exploit parallelism in deterministic programs, has been difficult to implement due to the overhead involved in the handling of shared variable bindings and the problem in preserving the “don’t know” nondeterministic semantics (for a definition see Chapter 3) of logic programs. Proposed AND-parallel logic programming languages usually sacrifice the completeness of logic programs (not all possible solutions may be found) in order to minimize these overheads.

Researchers are attempting to extend the above concurrent logic programming frameworks toward an object-oriented programming style [KTMB86, YC87]. Perpetual processes are viewed as passive objects. An input parameter of a process is a stream of events to the object and other parameters represent the state of the object. An object waits for a particular event to hold, takes behaviors corresponding to the event, such as generating another event and changing its own state, and then makes a recursive call to itself to start the next working circle. These proposals retain the incompleteness problem from their original frameworks. Moreover, finding a satisfactory semantics for perpetual processes is still an open problem.

Other approaches to the use of using logic programming for large, dynamic systems combines both logic and process/object-oriented programming in a natural and efficient way, such as CS-TProlog [Fut88], and POOPS [VL87]. By partially exploiting AND-parallelism, these proposals provide explicit object (process) declarations, temporal constructs and communication facilities. Processes are executed in parallel, they sequentially evaluate their own goals and coordinate through explicit communications. Programs written in these languages are quite easy to understand. The

specification of simulation models can be natural in these languages.

However, an important problem of in these latter approaches is that the standard model theoretic semantics of first order logic are not powerful enough to describe the behavior of these programs. A new model theory is needed to define the formal semantics of a distributed logic program. Furthermore, an efficient global backtracking algorithm is required in order to preserve “don’t know” nondeterminism.

2.2 Synchronization Mechanisms

Traditionally, discrete event simulation is performed by maintaining an *event list* which is used to sequentially schedule events for a set of cooperating activities, *i.e.*, to synchronize these cooperating activities in the order in which events occur. Each event is stamped by a value of an imaginary clock which ticks the simulation time. Thereby all events in a simulation can be ordered with respect to their timestamps. In fact, simulation time is an abstract data structure to represent the progress of a simulation. It is totally independent of the real execution time of the simulation.

When we apply distributed computing techniques to a discrete event simulation in which a model has been decomposed into processes, we have to solve the problem of exchanging information among processes and synchronizing them so that events occur in a correct order. Methods of using a central clock to tick simulation time have been proposed [KH85, BG84]. However more attractive suggestions use distributed clocks [Lam78, CM79, CM81, Jef85].

The central clock scheme is a trivial solution. It simulates the sequential scheduling mechanism of traditional discrete event simulation by using a central controller.

The central control process gathers together all the synchronization requests and schedules the execution of processes according to the timestamps of these requests.

The method of distributed clocks allocates the control to all processes. Each process is associated with a local clock and a local controller. The major responsibility of a local controller is to handle communications of its process with others and to establish the agreement of its local clock with others.

Chandy [CM79] uses two terms to distinguish these approaches. The former is called the *time driven* simulation system because synchronization of processes is driven by a controller with respect to a central clock, and the later the *time exchange* simulation system because synchronization of processes is established upon the values of distributed clocks collected through inter-process communications.

Since a *time driven* scheme has a central controller which is a bottle-neck for concurrent computing, we focus discussion on the *time exchange* simulation systems in the rest of this section. First, we review the origin of distributed logical clocks. Then we discuss two typical synchronization mechanisms proposed for distributed discrete event simulation.

2.2.1 Logical Clocks

The proverb: “A person with one watch knows what time it is; a person with two or more watches is never sure.” is commonly accepted, *e.g.*, Lamport [Lam78]. He notes that it is sometimes impossible to say that one of two events occurred first in a distributed system:

...Most people would probably say that an event *a* happened before an event *b* if *a* happened at an earlier time than *b*. They might justify this

definition in terms of physical theories of time. However, if a system is to meet a specification correctly, then that specification must be given in terms of events observable within the system. If the specification is in terms of physical time, then the system must contain real clocks. Even if it does contain real clocks, there is still the problem that such clocks are not perfectly accurate and do not keep precise physical time.

Lamport carefully reexamined the events in a distributed system and found that real-time temporal order, simultaneity, and causality between events bear a strong resemblance to the same concepts in special relativity. From the “space-time diagram” in his paper, he showed that the real time temporal relationships *happens before* and *happens after* only form a partial order. So he introduced logical clocks to extend this partial order to a total order. Being able to totally order the events can be very useful in implementing a distributed system.

Each process has an associated logical clock which is used to assign a number to an event, where the number is thought of as a timestamp which defines when the event occurred. If an event represents a mutual exclusive command, a process can execute such a command timestamped t when it has learned of all commands issued by all other processes with timestamps *before* t .

This method allows one to implement any desired form of multiprocess synchronization in a distributed system. However, there are two shortcomings. The first problem is that it involves a large volume of communication traffic among processes because each process has to broadcast its command to all other processes, and receive the same number of acknowledgements to make a synchronization. Second, a

process must know all the commands issued by other processes, so that the failure of a single process will be fatal.

Although Lamport's work did not concentrate on distributed discrete-event simulation, it does help in understanding the basic problems of distributed systems and the importance of logical clocks. The total ordering property of system events not only provides a criterion for establishing synchronization, but also provides a foundation for abandoning the concept of mutual exclusion and preventing deadlock.

2.2.2 The Network Paradigm

Most past work on distributed simulation has been based on the network paradigm [CM79, CHM79, PWM79]. In contrast with Lamport's work, Chandy notices that the communication relations among a set of cooperating processes in most simulations are not completely connected. Thus, he proposes that each node in a network represents one cooperating process and each directed line in the network represents a one way communication channel between two processes (nodes). As in Lamport's approach, each process has a logical clock which moves forward in time in an asynchronous manner, and tells how far the associated process has progressed in the simulation.

The key points of the network paradigm are summarized as follows:

1. There is a separate time variable associated with each incoming line of a process to indicate the timestamp of the earliest unreceived message on the line;
2. Processes communicate only by passing messages that include timestamps;
3. Each process attempts to move its clock as far ahead in time as possible, based

upon currently available information;

4. The output message on a line may state that no messages will arrive on that line between the current clock-time and some future time;
5. Since the sequence of clock times on a line are monotonically increasing, merging of time variables at a process can be achieved using the well known merging algorithm.

Ideally, if each process (node) in the simulation were assigned to a different processor and all could execute concurrently, then it would be possible to achieve an optimal n -fold speedup over the single processor case.

Unfortunately, this is not usually true in actual simulations. For example, a process may have one or more input lines on which no message is currently available (empty queues) when it tries to receive the next message. If this happens the process must wait until all of its input lines are non-empty, and then select the next message with the lowest timestamp. In this situation, two severe problems might arise, that is, deadlock and memory overflow.

Chandy proposed a mechanism that directly deals with both problems. First, he requires that each message queue have a bounded length. In addition to blocking whenever one of its input line is empty, a process must also block whenever it sends a message along an output line where the message queue at the other end is full. Second, the distributed simulations always run with a deadlock detection algorithm, as soon as a deadlock situation has been detected, a deadlock breaking algorithm is activated.

The network paradigm proposed a distributed solution for simulation problems

which are typically solved in a sequential manner. It showed that the time required to run a distributed program that implements a queuing network simulation is generally less than the time required to run equivalent sequential programs. This efficiency is achieved since there is no global process which could be a bottleneck.

However, the concurrency is limited by blocking when an input line is empty or the output line is full. The technique for solving the memory overflow problem exacerbates the deadlock problem. Since a process can block while either sending or receiving, a deadlock situation can occur around any undirected cycle in the simulation network, rather than just in the directed cycles. Furthermore, the communication connection among processes needs to be stable. Dynamic changes in communication connections makes the system much more complicated.

2.2.3 The Time Warp Mechanism

It has been shown that performing a set of look-ahead computations often results in faster execution even when some of these optimistic computations are wasted, *i.e.*, not used. This idea is called the *Never wait rule* which states that it is better to give a processor a task that may or may not be used than it is to let the processor sit idle.

Before Jefferson's innovative work [JS82, Jef85] nearly all the proposals for distributed discrete event simulation were based on conservative synchronization mechanisms. A synchronization mechanism is conservative if it involves "waiting". For example, in Lamport's proposal, a process has to wait for all other processes being in agreement on its synchronization request; in Chandy's method, a process has to wait for messages from all input channels before continuing its computation. These

proposals do not follow the *Never wait rule*.

On the other hand, the Time Warp mechanism never involves waiting. Jefferson uses the term *virtual time* in connection with the Time Warp mechanism to be synonymous with simulation time and describes the main idea as follows:

... A *virtual time* system is a distributed system that executes in coordination with an imaginary global virtual clock that ticks virtual time. Virtual time is a temporal coordinate system used to measure computational progress and define synchronization. ... Each process executes continuously, processing messages that have already arrived in increasing virtual receive time order as long as it has any messages left. All of its execution is provisional, however, because it is constantly gambling that no message will ever arrive with a virtual receive time less than the one stamped on the message it is now processing. As long as it wins this bet execution proceeds smoothly. The novelty is that whenever the bet is lost the process pays by rolling back to the virtual time when it should have received the late message. The situation is quite similar to the gamble that paging mechanisms take in the implementation of virtual memory: They are constantly betting with every memory reference that no page fault will occur. Execution is smooth as long as the bet is won, but a comparatively expensive drum read is necessary when it is lost.

In the Time Warp paradigm, each process always charges ahead, blocking only when its input queue is exhausted, and then only until another messages arrives. Whenever a message with a timestamp "in the past" arrives at a process's input

queue, the Time Warp mechanism automatically “rolls back” that process – restores it to a state from a virtual time earlier than the timestamp of the late message, cancels any side effects that it may have caused in other processes, and then starts the process forward again.

Although some computational effort is “wasted” when a projected future is thrown away, a conservative mechanism would keep the process blocked for the same amount of *cpu* time, so the *cpu* time may be “wasted” anyway. According to the *Never wait rule*, the Time Warp mechanism offers the potential of speeding up almost any large simulation by exploiting the concurrency within it.

A question here is: to what extent can the Time Warp mechanism win via this type of gambling? If it always loses, even if it does not waste time on synchronizations, it has to pay a lot of costs for rollback actions.

Like the paging systems, the Time Warp mechanism is also based on a *locality assumption*. Locality manifests itself in both time and space. Temporal locality is locality over time. Spatial locality means that nearby items tend to be used together. Locality is observed in operating system environments, particularly in the area of storage management. It is an empirical property rather than a theoretical one. It is never guaranteed but is often likely.

Actually, locality is quite reasonable in distributed simulation systems when one considers the way programs are written and communication is organized. In particular, temporal locality means that if process p sends a message to process q , it is most likely that process p sends another message to, or receives a reply from process q in the near future, and that such subsequent messages will have increasing time-stamps. An example of this is the communication between an event-generating process and

an event-consuming process. Spatial locality means that the communication connections between processes are often stable, *i.e.*, they may change but at a relatively slow rate in comparison to message interaction rate. The success of the Time Warp mechanism is based on this locality assumption, though it needs to be verified and characterized by further experiments.

In this section, we outlined the fundamentals of the Time Warp mechanism. Further details can be found in Chapter 4. It appears that the Time Warp mechanism is an attractive paradigm for distributed simulation. It is deadlock-free, can be made completely transparent to the programmer, and seems to have significant advantages over conservative mechanisms.

However, there are problems which need further exploration. One problem is process state-saving. In the case of rollback, the rollback mechanism of a process must hold a state queue which saves copies of the process's past states. State-saving is a low level system activity. In general, what should be saved and what should not, is not known, so the entire data space of a process is saved each time, which is not only space-consuming, but also time-consuming.

2.3 Base System and General Comparisons

The programming languages and synchronization mechanisms we have discussed span the important ideas for distributed computations. These ideas attempt to answer the questions enumerated at the beginning of this chapter. A base language and synchronization scheme is selected in this section which provides the best answers to these questions.

After the selection of a base system, two closely related systems are then compared with the work in this thesis. We use several general criteria to compare these approaches while ignoring their implementation details.

2.3.1 A Base Language and Synchronization Scheme

It is generally accepted that logic programming techniques have great potential for defining executable specifications. Hoare [Hoa82] points out:

A specification is a predicate describing all observations of a complex system. ... Specification of complex systems can be constructed from specifications of their components by connectives in the *predicate calculus*.
... A program is just a predicate expressed using a restricted subset of connectives, codified as a programming language.

Thus, an attractive answer to the first question is that a logic programming framework, such as Prolog, can provide an automatic way to turn a specification into an efficient program.

However, Prolog is not sufficient for describing distributed discrete event simulation. Among varieties of procedural programming languages and concurrent logic programming proposals, we choose the process/object-oriented logic programming approach to answer the second question. With a minimal impact on standard Prolog, this new framework provides the advantages of procedural programming – model decomposition, information hiding, object abstraction and system synchronization – and at the same time retains the benefits of logic programming – declarative and procedural understanding, nondeterministic computation, general pattern matching

and logic variable manipulation.

Comparing different synchronization mechanisms, we focus our attention on the Time Warp mechanism. Based on the *Never wait rule* and the *Locality assumption*, the Time Warp mechanism provides the possibility to speed up large simulations. It also provides asynchronous communication primitives for describing dynamic behaviors of distributed systems. Clearly, discrete event simulation is one of the most appropriate applications of the Time Warp paradigm, because the order of events is completely specified by the user. Furthermore, the Time Warp mechanism is totally transparent to the user, because the same conceptual methodology for building a sequential, object-based simulation can be used for building a concurrent, process-oriented simulation. We believe that the Time Warp mechanism is the best choice for answering questions 3 and 4 listed at the beginning of this chapter.

But, these advantages are not without cost. As Jefferson [Jef85] expected, the Time Warp mechanism uses several times as much memory as other methods in order to achieve speed-up. The major memory cost is due to state-saving. The Time Warp mechanism has no knowledge about what should be saved and what should not, and thus the entire data space of a process is saved periodically. If a process manipulates a large amount of data, say, a matrix with 100×100 integers, saving successive states may be prohibitive.

One way to reduce the cost of state-saving is to save only a few specified variables that represent those parts of the process state that have changed, instead of saving the entire data space. In this case, the ball is put in the user's court. A programmer has to isolate a set of representative variables from the data domain of his application, then present them through some linguistic declarations to the Time Warp

system. The defect of this scheme is the destruction of transparency. Another way to minimize state saving and state management overhead is to use special purpose hardware, such as the *rollback chip* [Fujimoto88]. Since this scheme needs hardware support, the discrete event simulation based on the Time Warp paradigm becomes machine-dependent.

Fortunately, we find that the backtracking facility in standard Prolog offers another possible way to overcome the shortcoming of non-knowledgeable state-saving. When Prolog backtracks and re-satisfies a goal, it returns to the most recently instantiated variables and attempts to instantiate them with alternative values. If this is not possible it backs up further to the next most recently instantiated variables, *etc.*. Thus, the set of variables changed on a given computation path is completely known. This knowledge can be used to determine the exact amount of information that must be saved for any given execution path. In other words, state-saving in Prolog is based on knowledge and is transparent to applications.

To conclude, the selected model for this thesis is a distributed logic programming framework in conjunction with a run-time kernel based on the Time Warp mechanism. The language implemented from this model is called *Communicating Sequential Prolog (CSP*)*. We believe that this distributed logic programming system offers potential advantages over other approaches.

2.3.2 Comparisons with Closely Related Approaches

Two current approaches closely related to the work in this thesis are CS-Prolog proposed by Futo [Fut88] and A-Prolog proposed by Cleary [CUL88]. CS-Prolog is a Prolog based simulation language and is implemented on a multi-transputer system.

A-Prolog is a pure Prolog system which uses the Time Warp mechanism to execute Prolog on multiprocessor and distributed systems. The similarities and differences in several general criteria among these systems are described in Table 2.1.

Criteria/System	CS-Prolog	A-Prolog	CSP*
purpose	simulation	general	simulation
logic property	impure	pure	impure
parallelism	partial AND	full AND	partial AND
process	explicit	implicit	explicit
process creation	built-in predicate(new)	AND-goal evaluation	process literal evaluation
communication protocol	asynchronous	asynchronous (Time-Warp)	asynchronous (Time-Warp)
communication medium	message	shared variable	message
communication connection	symbolic process name	channel of shared var.	symbolic process name
temporal property	explicit logical clock	none	explicit logical clock
global backtracking	dead-lock detection	distributed backtracking	GCT-controlled backtracking

Table 2.1: General Comparisons of Three Systems

Detailed explanations of Table 2.1 are as follows:

purpose: CS-Prolog is a distributed version of T-Prolog [FS82]. Its purpose is to perform special goal-oriented simulation, where the simulation is directed to find the appropriate activities and synchronization of processes to reach a predefined final state, the *goal state*. A-Prolog is a general purpose logic programming language. It preserves the standard semantics of Prolog and uses the Time-Warp mechanism to coordinate parallel execution of Prolog on multiprocessor and distributed systems. CSP* is a distributed logic programming language for discrete event simulation. Its primary goal is to speed up simulations through the use of parallelism, while as far as possible, preserving standard semantics of Prolog.

logic property: CS-Prolog and CSP* are both impure logic programming languages. The impurity is caused by the use of explicit communication predicates which cannot be defined in first order logic. On the other hand, A-Prolog implements pure Prolog, that is, clauses of A-Prolog are in the scope of first order logic.

parallelism: These three languages all use AND-parallelism within logic programming. CS-Prolog and CSP* provide specific linguistic tools to specify potential parallelism while A-Prolog potentially evaluates all AND-goals in parallel.

process: In explicit process systems, the programmer causes process instances to be created. CS-Prolog and CSP* are explicit process systems. On the other hand, A-Prolog implements standard Prolog in a distributed way. Processes in A-Prolog system are transparent to the programmer.

process creation: Processes in CS-Prolog are dynamically created by a built-in predicate *new*. Processes in CSP* are dynamically created by lexical elaborations, that is, goals which match process clauses are treated as independent processes. Processes in A-Prolog are created for goals in conjunction (AND-goals).

communication protocol: In an asynchronous protocol, processes send messages without regard to their reception; a process is free to send a message and continue computing. These three systems all adopt asynchronous communication protocols. The communication mechanism used by CSP* and A-Prolog is the Time Warp mechanism.

communication medium: CS-Prolog and CSP* define explicit messages as the information transfer medium while A-Prolog uses shared variables to carry information.

communication connection: The syntactic form to channel communication in CS-Prolog and CSP* are symbolic process names. Each message in these systems indicates a one-to-one communication connection. On the other hand, communications in A-Prolog are channeled by shared variables. Bindings of a variable are transferred (in a broadcastlike fashion) to processes which share this variable.

temporal property: Clauses in CS-Prolog and CSP* are temporal clauses. They use a temporal parameter (logical clock) and the computation rule of standard Prolog to simulate a temporal resolution. A difference between them is that

the former provides an explicit time advance predicate while the later advances a logical clock by the side-effect of communication.

global backtracking: Different global backtracking algorithms are adopted by these systems. CS-Prolog accomplishes global backtracking by a dead-lock detection algorithm, *i.e.*, the system chooses a process to backtrack only if all processes are blocked. A-Prolog implements a distributed global backtracking algorithm, *i.e.*, every process is able to backtrack if a variable of the process is bound to a conflict value. CSP* uses a *GCT*-controlled global backtracking algorithm which is a distributed algorithm but is invoked by a system-defined global virtual time.

Table 2.1 provides a very general comparison of three approaches which are based on parallel logic programming and (or) Time Warp mechanism. In the following chapters, a distributed logic programming model, a global backtracking algorithm, and the CSP* language are presented. As these topics are discussed in greater detail, further comparisons are made with the CS-Prolog and A-Prolog approaches.

Chapter 3

A DISTRIBUTED LOGIC PROGRAMMING MODEL

A distributed logic programming model is presented in this chapter. The model emphasizes the salient properties of *distributed* logic programming, that is, organizing and synchronizing numerous logic processing agents for partitioning and resolving a common goal. We first present the model from a theoretic point of view and then discuss problems relevant to an implementation of the model.

3.1 The Model

The model is based on first order logic. To specify its syntax, we must specify the alphabet of symbols to be used in the model and how these symbols are to be put together into legitimate expressions. A lot of work has been done on the study of logic programming [Llo84]. In order to simplify the syntactical description of the proposed model, we extend the original specification of the alphabet [Llo84] as follows:

1. **variables:** syntactically, variables are denoted by the letters x , y , and z . Informally, a variable is an object whose structure is unknown. As a computation progresses, the variable may be *instantiated*, or *bound* to another term which, therefore, becomes the value of the variable. A variable is a *ground variable* if it has a value.

2. **constants:** constants will normally be denoted by the letters a , b , and c . They are purely symbolic and have no inherent interpretation.
3. **functions:** functions are denoted by letters f , g , and h . A function defines a mapping of elements in a given domain.
4. **predicates:** in this model, predicates are distinguished into two disjoint sets: one for *process names* and one for *procedure names*. Process predicates are normally denoted by letters p , q and r and procedure predicates are denoted by letters d and e . Predicates define relations of elements in a given domain.
5. **connectives:** connectives in the model are " \leftarrow ", " \Leftarrow " and " $,$ ".
6. **punctuation symbols:** the model adopts the following punctuation symbols: " $($ ", " $)$ ", " $,$ " and " $.$ ".

From these symbols, we construct basic classes of expressions as follows:

1. **terms:** terms are the basic data structures in logic programs. A term is a constant, a variable or an n -ary function applied to n terms, such as $f(t_1, \dots, t_n)$.
2. **atoms:** atoms are the basic components of logic programs. An atom is a predicate or an n -ary predicate applied to n terms as arguments, such as $p(t_1, \dots, t_n)$.
3. **literals:** literals are symbolic representations of atoms. Literals with procedure names are *procedure literals*, denoted by A , B and C , and literals with process names are *process literals*, denoted by P , Q and R .

Now, we are ready to define the primitive units of a distributed logic program. To avoid using too many symbols in our discussion, the syntactic symbols may be subscripted.

Definition 3.1 *A procedure clause is a clause of the form*

$$A \leftarrow B_1, \dots, B_n.$$

where $n \geq 0$. A is called the head and B_1, \dots, B_n is called the body of the procedure. A procedure clause with an empty body ($n = 0$) is called a unit procedure clause.

Procedure clauses have the same syntactic form and semantics as the program clauses described in [Llo84]. The informal semantics of $A \leftarrow B_1, \dots, B_n$ is “for each assignment of each variable, if B_1, \dots, B_n are all true, then A is true”. Thus, if $n > 0$, a procedure clause is conditional. On the other hand, a unit procedure clause $A \leftarrow$ is unconditional. Its informal semantics is “for each assignment of each variable, A is true”. If a set of procedure clauses have the same predicate in the head, then they constitute the *definition* of a procedure.

Definition 3.2 *A process clause is a clause of the form*

$$P \Leftarrow A_1, \dots, A_n.$$

where $n \geq 0$. P is called the head and A_1, \dots, A_n is called the body of the process. A process clause with an empty body ($n = 0$) is called a unit process clause.

The largest syntactic unit in a distributed logic program is a *logic process*. A logic process is defined by *process clauses* with the same predicate in the head.

The proposed model adopts explicit logic processes as the primitive, concurrent-processing objects. The same informal semantics of a procedure clause can be applied to a process clause, except we use “ P is successful” to replace “ P is true”.

Definition 3.3 *A distributed logic program \mathcal{P} is a finite set of process clauses and procedure clauses.*

Definition 3.4 *A goal clause is a clause of the form*

$$\leftarrow Q_1, \dots, Q_k$$

where each Q_i is a process literal, and Q_i 's do not share any non-ground variables.

The declarative reading of a goal clause, i.e., $\leftarrow Q_1, \dots, Q_k$, is to “show that Q_1, \dots, Q_k are successful simultaneously”. In the following discussion, we call the literals in a clause body *procedure calls*, and the literals in a goal clause *process instances*.

Since the interesting classes of problems we are dealing with are coordinated computing systems which are concerned with inter-process communication and synchronization, we define two communication predicates for the model. For the sake of simplicity, we assume for the moment that communication predicates are time-independent.

Definition 3.5 *Communication predicates are defined by built-in procedures of the form*

$$\uparrow(x) \quad \text{and} \quad \downarrow(x)$$

where x is a term.

Clearly, communication predicates cannot be defined in clausal form because their truth values depend on the cooperation of the logic processes they reside in.

3.1.1 Declarative Semantics

The declarative semantics of a distributed logic program is intended to describe what is true about the underlying system of logic processes. If there is no cooperation among logic processes, the proposed model is just another variant of a traditional logic programming model with multiple resolution streams and inherits all properties discussed in [Llo84]. However, if a distributed logic program involves inter-process communication and synchronization, traditional model theoretic semantics of first order logic is not powerful enough to explain the semantics of such a program.

In this section, a new method is presented for the purpose of semantic analysis. Generally, the distributed logic programming model can be seen as a set of resolution theorem provers with each corresponding to a logic process. When establishing the separate proofs, a logic process prover “guesses” the truth value for each encountered communication predicate. When the proofs are combined, these guesses have to be checked for consistency using a synchronization test.

The first step of the semantic analysis starts by constructing all *proof-trees* for each logic process. A proof-tree consists of finite nodes and edges which represent the goals reduced during the construction [SS86]. In the course of building a proof-tree, a goal is called *reduced* if it is replaced by the body of a clause whose head is identical to the goal and the new formed goals are *derived*.

For a given logic process, the root of a proof-tree is the process literal, the nodes of the tree are goals reduced from their immediate parent node(s) in one reduction

step. There is a directed edge from a node to each node corresponding to a derived goal of the reduced goal. A proof-tree is the representation of a proof of its root (a logic process), using clauses as implication. It also reflects the invocation relations when clauses are treated as procedures.

An important property of building a proof-tree is that all ground communication predicates are assumed to be directly reducible. In other words, a logic process prover always assigns “true” to an encountered ground communication predicate by guess. If a communication predicate involves an unbound variable during the construction of a proof-tree, then we suppose that the variable is instantiated to each ground term in the *Herbrand universe* [Llo84] of the program, thereby we get a different proof-tree for each different instantiation. A communication predicate is always a leaf node in a proof-tree.

If a proof-tree contains communication predicates, it is a *proof-tree by guess*. In the following discussion, we use *proof-tree* as a synonym of *proof-tree by guess*.

For example, consider a logic process definition:

$$p \Leftarrow \uparrow(a), \uparrow(b).$$

$$p \Leftarrow \uparrow(c), d.$$

$$p \Leftarrow .$$

$$d \Leftarrow e.$$

$$e \Leftarrow .$$

The proof-trees of process p are shown in Figure 3.1. Examining these proof-trees we find that the tree (a) contributes a proof of p , because it does not make any guess. In other words, p is proved by proof-tree (a) without cooperation with other logic

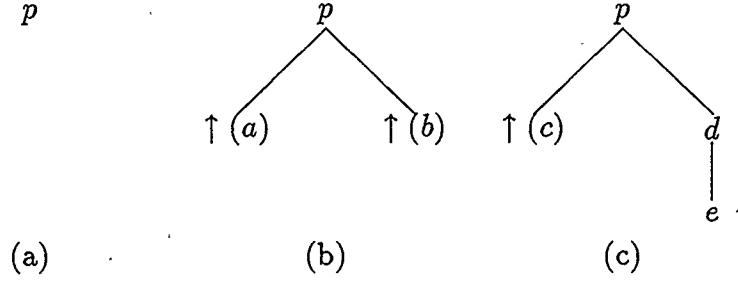


Figure 3.1: The Proof-trees of the Example

processes. However, (b) and (c) are not yet proofs of p , because they involve guesses at the truth values of communication predicates. Now, our job is to define under what circumstances the communication predicates can be proved to be “true”.

Definition 3.6 *A guess of a proof-tree is a multi-set (a set which allows multiple occurrences of an element) which includes all communication predicates in the tree.*

Definition 3.7 *The guess-set of a logic process is the set of guesses drawn from all proof-trees of the process.*

We use gs_p^i to denote a *guess* of p where i is used to index different guesses of p , and $GS(p)$ to denote the *guess-set* of p . For the above example, we have:

$$(a) \quad gs_p^1 = \{\}$$

$$(b) \quad gs_p^2 = \{\uparrow(a), \uparrow(b)\}$$

$$(c) \quad gs_p^3 = \{\uparrow(c)\}$$

$$GS(p) = \{gs_p^1, gs_p^2, gs_p^3\}$$

For a given distributed logic program, we can construct the *guess-sets* for all the underlying logic processes in the same way. Based on these sets, we describe the semantics of the program.

Definition 3.8 *A synchronous couple is a pair of ground communication predicates with the same argument and opposite operators.*

For example, $(\uparrow(a), \downarrow(a))$ is a *synchronous couple*, but $(\uparrow(a), \downarrow(b))$ is not. Informally, a *synchronous couple* defines a time-independent, one-to-one matching semantics of a pair of communication predicates. We say that the communication predicates in a synchronous couple *complement* each other.

Definition 3.9 (Synchronization-Test) *Given a multi-set of the form*

$$\{t_1, t_2, \dots, t_k\}, k \geq 1$$

where the t_i 's are ground communication predicates. Then we say that the set satisfies the Synchronization-Test if it becomes empty after deleting all synchronous couples of $(t_i, t_j), i \neq j, i, j = 1..k$.

For example, set $\{\downarrow(a), \uparrow(b), \downarrow(b), \uparrow(a)\}$ satisfies the Synchronization-Test, but set $\{\downarrow(a), \uparrow(c)\}$ does not.

Definition 3.10 *Let $S_1 = \{u_i : i \in I\}$ and $S_2 = \{v_j : j \in I\}$, where u_i and v_j are sets of literals and I is the integer set. The cross union of S_1 and S_2 is defined as*

$$S_1 \uplus S_2 = \{u_i \cup v_j : i, j \in I\}.$$

For example, suppose $S_1 = \{\{U_1, U_2\}, \{U_3, U_4\}\}$ and $S_2 = \{\{V_1, V_2\}, \{V_3\}\}$, then

$$S_1 \uplus S_2 = \{\{U_1, U_2, V_1, V_2\}, \{U_1, U_2, V_3\}, \{U_3, U_4, V_1, V_2\}, \{U_3, U_4, V_3\}\}.$$

Definition 3.11 Let \mathcal{P} be a distributed logic program, $G \leftarrow Q_1, \dots, Q_k$ be a goal. Then G is a logical consequence of \mathcal{P} iff there exists at least one element in the cross union of $GS(Q_i)$'s which satisfies the Synchronization-Test.

Definition 3.12 Let \mathcal{P} be a distributed logic program. The meaning of \mathcal{P} , denoted $M(\mathcal{P})$, is the set of all goals which are a logical consequence of \mathcal{P} .

For example, a distributed logic program \mathcal{P} consists of three logic processes p_1 , p_2 and p_3 . Their definitions, guesses and guess-sets are described as follows:

$$\begin{array}{ll} p_1 \Leftarrow \uparrow(a_1), \downarrow(b_1). & gs_{p_1}^1 = \{\uparrow(a_1), \downarrow(b_1)\} \\ p_1 \Leftarrow \downarrow(a_2), \uparrow(b_2). & gs_{p_1}^2 = \{\downarrow(a_2), \uparrow(b_2)\} \\ p_2 \Leftarrow \downarrow(a_1), \uparrow(a_3). & gs_{p_2}^1 = \{\downarrow(a_1), \uparrow(a_3)\} \\ p_2 \Leftarrow \uparrow(a_2), \downarrow(b_2). & gs_{p_2}^2 = \{\uparrow(a_2), \downarrow(b_2)\} \\ p_3 \Leftarrow \uparrow(b_1), \downarrow(a_3). & gs_{p_3}^1 = \{\uparrow(b_1), \downarrow(a_3)\} \\ p_3 \Leftarrow. & gs_{p_3}^2 = \{\} \\ GS(p_1) = \{gs_{p_1}^1, gs_{p_1}^2\} \\ GS(p_2) = \{gs_{p_2}^1, gs_{p_2}^2\} \\ GS(p_3) = \{gs_{p_3}^1, gs_{p_3}^2\} \end{array}$$

In order to find the meaning of \mathcal{P} , we have to examine the cross unions of $GS(p_i)$'s with respect to different goals. Here we use the symbol " \wedge " to represent the conjunction of process literals in case of confusion.

First, we investigate goals with a single process literal. It is easy to see that p_1 and p_2 are not logical consequences of the program because no elements in $GS(p_1)$ or $GS(p_2)$ succeeds the *Synchronization-Test*. However, the element $gs_{p_3}^2$ in $GS(p_3)$ satisfies the *Synchronization-Test*, so p_3 is a logical consequence of \mathcal{P} .

Secondly, we check the goals with two process literals. If the goal is $p_1 \wedge p_2$, we find that the element $gs_{p_1}^2 \cup gs_{p_2}^2$ in $GS(p_1) \uplus GS(p_2)$ fulfilling the test, therefore $p_1 \wedge p_2$ is in $M(\mathcal{P})$. However, if the goal is $p_1 \wedge p_3$ or $p_2 \wedge p_3$, they all fail in the *Synchronization-Test*.

Finally, the element $gs_{p_1}^1 \cup gs_{p_2}^1 \cup gs_{p_3}^1$ in $GS(p_1) \uplus GS(p_2) \uplus GS(p_3)$ succeeds in passing the test.

Putting pieces together, we have

$$M(\mathcal{P}) = \{p_3, p_1 \wedge p_2, p_1 \wedge p_2 \wedge p_3\}.$$

It means that p_3 succeeds individually, $p_1 \wedge p_2$ succeed cooperatively and $p_1 \wedge p_2 \wedge p_3$ succeed cooperatively.

3.1.2 Operational Semantics

The operational semantic analysis is a way of describing procedurally the meaning of a distributed logic program. Lloyd [Llo84] defines the procedural semantics of logic programs by using an interpreter based on SLD-resolution. The interpreter adopts a unification algorithm and benefits from the properties of the Most General Unifier (MGU). In his book, he also proves that for logic programs, the SLD-resolution is sound, complete, and computation rule independent. We accept all these results and assume that the same interpreter is applied to each of the logic processes in a

coordinated computation. Therefore, for a given logic program \mathcal{P} and a goal G , the number of interpreters in solving $\mathcal{P} \cup \{G\}$ is equal to the number of process literals in G .

However, a serious problem is that if a selected goal is a communication predicate, what actions does a local interpreter take? It is certain that a local interpreter is not able to reduce a communication predicate without knowledges about the current coordinated computation of \mathcal{P} . A local interpreter must be able to sense the global world to the extent of determining the success or failure of communication predicates. Thus, we extend the standard SLD-resolution to a distributed domain, assume that the system has sensor capabilities [GLB85] for detecting satisfaction of all communication predicates. In other words, there is a *sensor* in each interpreter which must be able to sense the world to the extent of determining the truth values of communication predicates nondeterministically. If a communication predicate is true with respect to a given goal of \mathcal{P} , the *sensor* returns its *complement*, NULL otherwise. With this assumption, we extend the abstract interpreter proposed in [SS86] as follows:

```
interpreter( $P$ : program,  $Q$ : process literal);
```

```
begin
```

```
 $T$ : resolvent;
```

```
 $\theta$ :  $MGU$ ;
```

```
  choose a clause  $Q' \leftarrow A_1, \dots, A_n$ , such that  $Q$  and  $Q'$  are unifiable; if no such
    clause exists, return  $NO$ .
```

```
   $\theta := \text{unify}(Q, Q')$ ;
```

```
   $T := A_1, \dots, A_n$ ;
```

```
  apply  $\theta$  to  $T$  and  $Q$ ;
```

```

while  $T \neq \text{NULL}$  do
begin
  choose a subgoal  $A$  from  $T$ ;
  if  $A$  is a communication predicate then
  begin /* extension part */
     $A' := \text{sensor}(A)$ ;
    if  $A' \neq \text{NULL}$  then
      begin
         $\theta := \text{arg-unify}(A, A')$ ; /* unify argument part */
        remove  $A$  from  $T$ ;
      end;
    else exit the while loop; /* no solution */
  end;
  else /* standard SLD-resolution */
  begin
    choose a clause  $A' \leftarrow B_1, \dots, B_n$ , such that  $A$  and  $A'$  are unifiable;
    if no such clause exists, exit the while loop;
     $\theta := \text{unify}(A, A')$ ;
    remove  $A$  from  $T$  and add  $B_1, \dots, B_n$  to  $T$ ;
  end;
  apply  $\theta$  to  $T$  and  $Q$ ;
end /* while */
if  $T \equiv \text{NULL}$  return  $Q$  else NO
end /* interpreter */

```

This abstract interpreter solves a query Q with respect to a program P . The output of the interpreter is an instance of Q , if a proof of such an instance is found, or *NO*, if a failure has occurred during the computation and cooperation with other interpreters (if any). An instance of Q for which a proof is found is called a *local solution* of Q . Note that the interpreter may also fail to terminate.

As explained by Sterling[SS86], there are two choices in the above interpreter: choosing the goal to reduce (computation rule) and choosing the clause to effect

the reduction (search rule). The choice of goal to reduce is arbitrary; it does not matter which is chosen for the computation to succeed. If there is a successful computation by choosing a given goal, then there is a successful computation by choosing any other goal. On the other hand, the choice of the clause to effect the reduction is nondeterministic. Not every choice will lead to a successful computation. In addition, to understand the *sensor* capability we also need our nondeterministic imagination. Thus, computations of distributed logic programs via a set of abstract interpreters resolve the issue of nondeterminism by always making the correct choice.

Definition 3.13 *Let \mathcal{P} be a distributed logic program, G a goal. A global solution of $\mathcal{P} \cup \{G\}$ is the conjunction of local solutions computed by applying the interpreter to each process literal in G concurrently.*

Definition 3.14 *The operational meaning of \mathcal{P} , denoted $O(\mathcal{P})$, is the set of all global solutions of \mathcal{P} .*

The declarative semantics defines a set of behaviors for a distributed logic program. The operational semantics also defines a set of behaviors for the program, but this set depends on the search rule and the sensor capabilities used in the above interpreter. If we assume that the sensors and the search mechanism make correct decisions with respect to each coordinated computation, then the declarative and operational semantics are consistent. As a consequence, the distributed SLD-resolution is *sound* and *complete* (for a proof see [Llo84]).

3.1.3 Temporal Constructs

As we discussed in the previous chapter, the notion of time is central in most distributed systems, especially in distributed simulation systems. Introducing a temporal construct into the model results in a specification language for describing a changing world.

Several temporal logic programming models are discussed in [AM87]. In these approaches, time is expressed directly by logic, that is, procedure clauses are associated with certain temporal operators, such as *next*, *eventually*, *until*, *precedes*, etc.. These temporal logic programming languages usually have complicated syntaxes and semantics (compared with Prolog). Moreover, the lack of quantitative representation of time makes it difficult to describe discrete event simulations in these languages.

Another way to treat time is to associate time with the resolution procedure of the traditional logic programming model. That is, traditional resolution with explicit time parameters is used to simulate temporal resolution [AM87]. In this scheme, each procedure literal is associated with two extra time parameters: t_{in} and t_{out} . Parameter t_{in} is the first parameter of a procedure and indicates the time at which the procedure is called; parameter t_{out} is the second parameter of a procedure and indicates the time at which the procedure call returns. For a procedure literal $d(t_{in}, t_{out})$, we define that $t_{in} \leq t_{out}$. In addition, each clause must involve additional steps to reason about time. These steps define the *temporal property* of a clause. For example, the temporal properties of discrete event simulation are discrete, linear, and extending infinitely towards the future. Thus, a temporal clause in such an

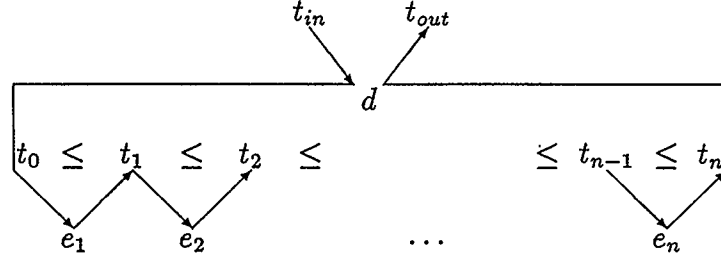


Figure 3.2: The Temporal Property of Procedure Calls

application can be defined as follows:

$$d(t_0, t_n) \leftarrow e_1(t_0, t_1), e_2(t_1, t_2), \dots, e_n(t_{n-1}, t_n), t_0 \leq t_1, \dots, t_{n-1} \leq t_n.$$

If a goal $d(t_{in}, t_{out})$ is reduced by the above clause, the *temporal property* of procedure calls is illustrated intuitively by Figure 3.2.

Proposals such as [Fut88] are based on this scheme. We also adopt this scheme in our distributed logic programming language, because it makes distributed logic programs easier to understand both declaratively and procedurally, and the quantitative representation of time parameters facilitates the implementation of simulation time in discrete event simulation.

Let E be any literal, $E(t_{in}, t_{out})$ be a literal with t_{in} and t_{out} as the first two parameters. We define the *temporal property* of clauses in the proposed model as follows:

process clause:

$$P(t_0, t_n) \Leftarrow A_1(t_0, t_1), A_2(t_1, t_2), \dots, A_n(t_{n-1}, t_n), t_0 \leq t_1, \dots, t_{n-1} \leq t_n.$$

procedure clause:

$$A(t_0, t_n) \leftarrow B_1(t_0, t_1), B_2(t_1, t_2), \dots, B_n(t_{n-1}, t_n), t_0 \leq t_1, \dots, t_{n-1} \leq t_n.$$

goal clause:

$$\leftarrow Q_1(0, t_{out}^1), \dots Q_k(0, t_{out}^k).$$

3.2 Implementation Issues

The model described above addresses the formal, logical understanding of distributed logic programs. It is not yet a practical distributed logic programming language because the “control” aspects of the model, *i.e.*, the computation rule and the *sensor*, have not been defined.

Moreover, as designed for distributed simulations, the language should provide a mechanism to synchronize the activities of logic processes with respect to a *simulation time*. Therefore, we have to extend the model to include programs with a simulation time and redefine communication predicates to reflect communication partnerships, synchronization times, *etc.*

In this section, we are going to consider these implementation issues which convert the theoretic model into a practical language.

3.2.1 Simulation Time and The Computation Rule

Interestingly, we find that the temporal property of a temporal clause discussed above mimics the procedural reading of a clause in standard Prolog. In brief, Prolog’s computation rule is characterized by selecting the leftmost goal instead of an arbitrary one, and substituting the search rule for the nondeterministic choice of a

clause by the depth-first search and backtracking. The procedural reading of a clause $A \leftarrow B_1, B_2, \dots, B_n$ is:

to solve A , first solve B_1 and then B_2 and ... and then B_n .

Such a reading defines not only the logical relations between the head of the clause and the goals in the body, but also the temporal order in which the goals are processed.

Thus, if Prolog's computation rule is used to evaluate a goal and a system-manipulated time parameter is used to indicate the progress of the goal evaluation, then it is possible to remove the explicit time parameters and the time reasoning steps from a temporal clause. In implementation, we use the concept of *virtual time* [Jef85] and inter-process communication to simulate temporal SLD-resolution. In other words, time parameters are represented by a system variable (logical clock), and the temporal property of a clause is implicitly compelled by the underlying control facility.

From the user's point of view, simulation time is a system-manipulated variable which tells a process what its time is and is used for one process to schedule an event for execution by another in the future. On the other hand, viewed by the system, simulation time is implemented by *virtual time* which defines a temporal coordinate system used to measure computational progress and specify synchronization.

3.2.2 Communication and Synchronization

So far in this chapter we have described the abstract communication and synchronization aspect of coordinated computations. Here we become highly practical – describing the “real” communication predicates, which involve how to specify com-

munication partners, how to define synchronizations and how to transfer messages.

We have chosen the Time Warp mechanism as the underlying communication system. We simply use its communication primitives, *send* and *receive*, to replace the abstract predicates \uparrow and \downarrow . The parameter list of communication predicates is defined as (Sender, Send-Time, Receiver, Receive-Time, Message), where Sender and Receiver are process predicates (process names), Send-Time and Receive-Time are simulation times (timestamps) and Message can be any data structure. In order to avoid naming conflicts in communication, we assume, for the moment, that each logic process has only one instance. A new naming method will be discussed in Chapter 6.

Predicate *send*(S, Ts, R, Tr, M) means that process S at current simulation time Ts sends a message M to process R with receive time Tr. Predicate *receive*(S, Ts, R, Tr, M) means that process R at Tr receives a message M from process S sent at Ts. In both cases, $Tr \geq Ts$. A predicate *send* succeeds if the message it sent will eventually be consumed by the specified receiver. Its evaluation has no influence on the simulation time of its process. On the other hand, at each process messages are processed strictly in receive time order. A predicate *receive* succeeds if the earliest unreceived message matches the parameters of the predicate. Whenever a process evaluates a *receive* successfully, its simulation time is automatically advanced to the specified receive time. In other words, time advancing in a logic process comes from the side effect of evaluating a *receive* predicate.

In Chapter 2, we have defined *synchronization* as the establishment of some form of agreement between a set of processes. Most traditional simulation systems only demand that processes agree on “time”, because these processes are *deterministic*.

However, the distributed logic programming system presented here deals with *non-deterministic* computation, *i.e.*, it defines a two-dimensional *virtual space*. Thus, synchronization of processes demands that these processes agree not only on *virtual time* (simulation time), but also on *virtual space*. Such agreements are established through inter-process communication and are controlled by the *sensors* and the underlying Time Warp mechanism with respect to the timestamps in the communications.

Going a step further, we simplify communication predicates as

$$\text{send}(\text{R}, \text{M}, \text{DT}) \quad \text{and} \quad \text{receive}(\text{S}, \text{M})$$

where DT is a non-negative delay interval of simulation time, and therefore the receive time T_r is the sum of DT and the simulation time T_s of the sender at which the message is sent. These two simplified predicates have the same semantics as the original ones (see how messages are constructed in the following chapters) and are used in the rest of this thesis.

3.2.3 The Sensor

Now, the question is how to realize the *sensor* used in the abstract interpreter. In general, a distributed logic program divides its problem domain into subdomains and declares them in the underlying logic processes. A logic process declaration defines a search tree [SS86] of a logic process instance and such a search tree is called a *virtual space* in this thesis. The interpreter of a logic process searches its *virtual space* for a local solution. These searches span a range from deterministic *virtual spaces* to nondeterministic *virtual spaces*.

In a deterministic *virtual space*, at any point the interpreter knows clearly which

alternative is to be applied to a selected goal, that is, the choice of what to consider next is independent of what choices have already been made.

In a nondeterministic *virtual space*, there are many potential useful alternatives at any point, if the interpreter chooses an alternative arbitrarily which always leads to a solution, then the *virtual space* provides “don’t care” nondeterminism, otherwise it has “don’t know” nondeterminism. “Don’t know” nondeterminism is common in logic programming.

Although we cannot build nondeterministic machines by our current knowledge, we can simulate them on existing computers. The standard Prolog is a typical example which approximates “don’t know” nondeterminism by sequential search and backtracking. However, retaining “don’t know” semantics of distributed logic programs is a much harder task, because it requires the ability to coordinate multiple, simultaneous, nondeterministic activities on a set of *virtual spaces*. The implementation of the *sensor* requires the function of *global backtracking*.

One way to implement global backtracking is to use a central scheduler to control the executions of all processes. T-PROLOG [FS82] adopts this scheme and simulates the “parallel” executions of processes. The drawback of this method is that the central scheduler is a bottle-neck in distributed environments.

Another way is to implement a distributed global backtracking algorithm by deadlock detection [Fut88]. In this scheme, when a process evaluates a *send* predicate, it sends the message out and assumes the predicate succeeds; when a process evaluates a *receive* predicate and there are no matched messages, the process is suspended. If all processes are suspended (some of them terminated), a deadlock detector is called to recognize processes which are deadlocked and select one of them to backtrack.

One problem in this approach is inefficiency, a deadlock situation can be found only when all processes are suspended. The second problem is that if there are several “perpetual” processes in a program which never block and terminate, how can one detect partial deadlocks of nonperpetual processes.

In Chapter 5, we present a new distributed global backtracking algorithm. The algorithm collects global knowledge through inter-process communications, uses the Time Warp rollback concept to deal with global backtracking and captures heuristics in that earlier synchronizations may make subsequent synchronizations more likely to succeed.

3.3 The Basic System Structure

To conclude, we have constructed a theoretic distributed logic programming model and discussed the transition to a practical one. The implementation discussions about the search rule, the simulation time and the communication predicates flavors the model with some “procedural” stuffings. This is because the only declarative aspect is not always sufficient for describing time-sensitive models, such as simulation models. Nevertheless, the declarative understanding of distributed logic programs is still partially retained. If the programmer’s program specification is declaratively correct, then it is relatively easy to get a correct, working simulation program.

Now, the practical model is notably similar to a distributed logic programming language system, although it still glosses over some practical details. Figure 3.3 summarizes the basic structure of the whole system. We have the following *process* definition:

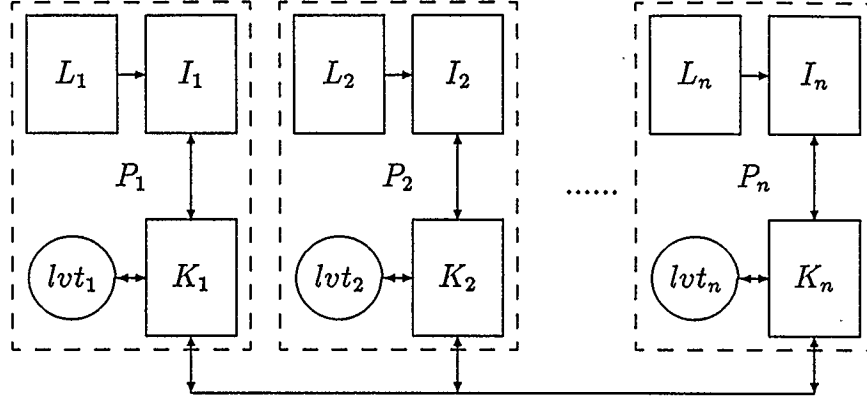


Figure 3.3: The Basic System Structure

Definition 3.15 A process P_i in the distributed logic programming system is a quadruplet (L_i, I_i, K_i, lvt_i) , where L_i is a logic description of P_i , I_i is the interpreter of L_i , K_i is the kernel server of I_i , and lvt_i is the local virtual time of P_i .

A closer observation of the system reveals that each process P_i in the system consists of two concurrently-executing subprocesses: an *interpreter process* I_i which manipulates the *virtual space* defined by L_i , that is, evaluates the assigned logic process L_i ; and a *kernel process* K_i which manipulates the *virtual time* ticked by lvt_i and handles the inter-process communications.

In the next two chapters, we describe the system's implementation. We first discuss the *kernel* part – a modified version of the Time Warp mechanism, and then present the *interpreter* part – a logic process interpreter with a global backtracking capability.

Chapter 4

A MODIFIED TIME WARP KERNEL

The Time Warp mechanism is an optimistic asynchronous inter-process communication protocol that relies on generalized process look-ahead and rollback to implement *virtual time*. *Virtual time* provides a temporal coordinate framework to define notions of synchronization and timing in distributed (simulation) systems.

This chapter describes a modified version of the Time Warp kernel [Jef85]. We begin by discussing the principles of the original mechanism, and then we present the important modifications and extensions. This new version of Time Warp constitutes the *kernel* part of the proposed distributed logic programming system.

4.1 Jefferson's Time Warp Kernel

Let $\mathcal{P} = \{P_i : P_i \text{ is a process, } i = 1..n\}$ be a distributed program running in the proposed system. Associated with each P_i is a *local virtual clock* variable, lvt_i , that ticks virtual time. At any moment local virtual clocks in \mathcal{P} may have the same or different values, but this fact is invisible to the processes themselves because they can only access their own virtual clocks. It is important to realize that lvt_i is not necessarily the same as the simulation time of P_i , for example, lvt_i may involve the simulation time of P_i as well as some extra information for system control purpose, but usually there exists a direct mapping between them.

Message passing is the only way for processes in \mathcal{P} to exchange information or

establish synchronization. The data structure of a *message* is defined as:

$$message = (P_s, t_s, P_r, t_r, msg)$$

which is read as “the sender P_s at virtual time t_s schedules the receipt of msg by P_r at virtual time t_r .” Whenever a message is sent, the virtual send time t_s is copied from the sender’s local virtual clock, the virtual receive time t_r is set by the sender according to its schedule strategy; whenever a message is received, the receiver’s local virtual clock is advanced to the virtual receive time t_r . Here we define that sending or receiving a message is a primitive *event* in a process, regardless of whether it identifies an “event” in simulation applications, and we use the “dot” notation to reference a component of a data structure. Therefore, a message m defines two primitive events: a sending event $(m.P_s, m.t_s)$ and a receiving event $(m.P_r, m.t_r)$.

Consequently, the primitive events executed by the system can be defined by:

$$E = \{(P_i, t_i) : i \in I\}$$

where each event in E represents a virtual space-time coordinate (later we will see that every P_i defines a two-dimension search space).

For the sake of simplicity, we adopt Lamport’s proposal [Lam78] to define a total ordering relation *happened before*, denoted by \rightarrow , on E . The relation requires that all processes obey the following implementation rules:

1. Each process P_i increments lvt_i between any two successive events;
2. For each message $(P_s, t_s, P_r, t_r, msg)$, $t_s < t_r$.

Thus, for any two events $e_1 = (P_i, t_1)$, $e_2 = (P_j, t_2)$, and $e_1, e_2 \in E$, we say that $e_1 \rightarrow e_2$ if and only if either $t_1 < t_2$, or $i < j$ if $t_1 = t_2$. It is easy to see that

\rightarrow defines an irreflexive total ordering relation on E , that is, for any pair of events $e_m, e_n \in E$, either $e_m \rightarrow e_n$ or $e_n \rightarrow e_m$.

For any $P_i \in \mathcal{P}$, the state of P_i is said to be *consistent* if all events that P_i has processed have *happened before* the events that P_i has yet to process. However, since P_i 's are executed in an asynchronous manner, we cannot guarantee that all processes are in a consistent state during their execution. If an inconsistent state of P_i is detected by the system, P_i is forced to roll back to an "earlier" state, cancel any side effects that may have been caused by messages sent to other processes, and then execute forward again. Thus, each process P_i has to remember enough of its history so that rollback can be accomplished when necessary.

The execution history of P_i is defined by the following information streams:

Input Queue(IQ_i): IQ_i contains all recent incoming messages and is ordered by \rightarrow wrt the receiving event part. A message m in IQ_i is *received* if $m.t_i \leq lvt_i$, otherwise m is *unreceived*.

Output Queue(OQ_i): OQ_i contains copies of the messages P_i has recently sent and is ordered by \rightarrow wrt the sending event part.

State Queue(SQ_i): SQ_i contains saved copies of P_i 's recent states where each state (s, t) is defined as a snapshot of the entire data space of P_i , including its execution stack, its own variables, and its program counter, at virtual time t at which an event occurs. SQ_i is ordered by $<$ wrt t .

4.1.1 Local Control

As mentioned before, a process P_i consists of two subprocesses: a *kernel* process K_i and an *interpreter* process I_i . The kernel process K_i sits in between I_i and the outside world, and acts as a local controller. The major responsibilities of K_i are:

1. providing a set of primitives, such as *send* and *receive*, to I_i ;
2. manipulating communications from (to) the outside world and recognizing roll-back requirements;
3. monitoring and controlling the execution of I_i , such as state-saving and rolling-back.

When I_i calls a *send* primitive, K_i saves a copy in OQ_i and then sends the message. When a message from the outside world arrives, K_i stores the message into IQ_i . When I_i executes a *receive* primitive, K_i returns the earliest unreceived message from IQ_i and assumes that no messages will ever arrive with a virtual receive time in the “past”. As long as the assumption holds, the execution of I_i proceeds smoothly. However, the local virtual clocks of processes do not necessarily agree during executions, some of them may charge ahead while others lag behind. So it is possible for K_i to meet a message from the outside world whose t_r is less than that of one already in IQ_i . For example, let t be the current local virtual time of P_i , $m_1 \dots m_{k-1} m_{k+1} \dots m_l$ be the current IQ_i and m_k be the new incoming message such that

$$m_{k-1} \rightarrow m_k \rightarrow m_{k+1}.$$

If this happens, K_i takes the following actions:

1. it inserts m_k between m_{k-1} and m_{k+1} ;
2. if $m_k.t_r > t$ then m_k represents a receiving event in the future and there is no influence on the execution of I_i ; otherwise
3. K_i rolls I_i back from t to the virtual time just before $m_k.t_r$, say t' , which includes restoring the state (s', t') from SQ_i , un-receiving all message events originally consumed during t' to t , canceling all side effects. Then K_i resumes I_i at t' to execute forward again.

When the third action is taken, we say that an inconsistent state of I_i has been detected by K_i . Because I_i was executed in an inconsistent state, it may have sent any number of messages to other processes, causing side effects and possibly inconsistent states in them. In order to remove such side effects, K_i “unsends” those messages originally sent by I_i during the interval t' to t . This is completed simply by sending an anti-message for each of them. We use \tilde{m}_k to indicate an anti-message of m_k and we assume that \tilde{m}_k arrives after its complement m_k in physical time (the assumption only simplifies our discussion, the opposite situation is easy to deal with in an implementation).

When an anti-message \tilde{m}_k with destination P_i arrives, K_i searches the IQ_i to find its complement m_k and

1. if $m_k.t_r > t$, then m_k is annihilated and IQ_i is left with no record that m_k ever existed; otherwise
2. deletes m_k from IQ_i and then takes the third step in the above algorithm.

4.1.2 Global Control

Theoretically, the Time Warp system does not need a global control mechanism because the local control and rollback facilities implement *virtual time* correctly. However, as the processes go forward, they save their execution histories periodically and some of them may attempt to do physical input and output activities. Without a time measurement to prevent memory overflow, to detect global termination, and to handle I/O and errors, the Time Warp mechanism cannot be practical. For this reason, *Global Virtual Time (GVT)* is introduced.

GVT is a virtual time value with respect to the whole system at a given real time. It is defined to be the greatest lower bound of the set of all virtual times shown by all local virtual clocks in an instantaneous snapshot of a virtual time system. We use $GVT(\tau)$ to denote the global virtual time at real time τ . Let $L(\tau)$ be a snapshot of all local virtual clocks at τ and $M(\tau)$ be the set of send times of all unreceived messages including messages in transmission at τ , we define

$$GVT(\tau) = \min(t \in L(\tau) \cup M(\tau)).$$

If there is a saved state for each event, the consequence of the definition is that *GVT* never decreases, *i.e.*, no process can ever roll back to virtual time before *GVT*. If this were to happen, the corresponding local control process would have met a message event with the virtual send time less than *GVT*, thus the scenario leads to a contradiction with the definition of *GVT*.

The nondecreasing property makes *GVT* a floor for the virtual times to which any process can ever again roll back. It is appropriate to use *GVT* as the time measurement to release execution histories (such as input, output and state queues), to

detect program termination, to handle program errors and to execute I/O activities.

Even though we use the term “global control” here, this does not necessarily create a bottle-neck. Instead, a distributed *GVT* estimation algorithm can be used in an implementation.

4.2 Modifications and Extensions

Jefferson’s Time Warp kernel is modified and extended in two basic ways:

1. Only part of the rollback mechanism is implemented in the the *kernel*. State saving and part of the rollback mechanism are moved to the *interpreter* process;
2. Special functions are provided to support nondeterministic computation.

Of course, the first modification simplifies the *kernel* part. A *kernel* process K_i now looks more like a communication server to its *interpreter* process I_i . It provides a set of service primitives and a request-service-reply protocol to I_i . It manipulates communications from (to) the outside world by an asynchronous communication protocol. However, it does not “monitor” and “control” I_i any more, instead, whenever a rollback/backtrack requirement has been detected, it notifies I_i by a special reply and starts new services only if I_i has been rolled back (or backtracked) to an earlier, consistent state.

To realize the second extension, the new *kernel* provides special functions to assist global backtracking. As discussed in Chapter 3, global backtracking is used to approximate “don’t know” nondeterminism of distributed logic programs. The key notation here is the concept of *objection* messages. Briefly, an objection message is

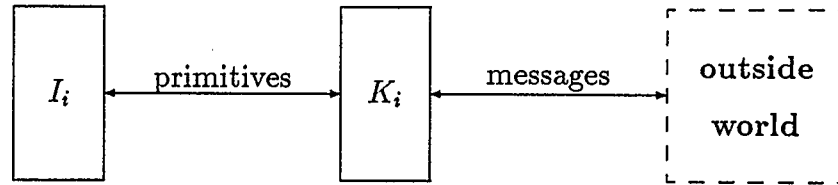


Figure 4.1: The Interfaces of Kernel Process

used by one process to change the search path of another process. In this section, we only discuss how a *kernel* deals with incoming or outgoing objection messages. The semantics of objection messages and the strategies for making objections will be investigated in the next chapter.

4.2.1 The Extended Kernel

Since a kernel process K_i resides between its interpreter process I_i and the outside world, K_i has two interfaces, as shown in Figure 4.1. The interface between K_i and the outside world is sending and receiving of various types of messages. The interface between K_i and I_i is a set of primitives. The major function of K_i can be described as follows:

loop

receive(message) or *accept*(primitive);
 if *message-received*, process the message;
 if *primitive-accepted*, process the primitive;

endloop.

K_i may receive four types of messages from the outside world. They are *normal*, *anti*, *objection* and *GCT* messages. Normal messages are used by applications. Anti

messages are used by the Time Warp kernel to correct mistakes on *virtual time*. Objection messages are used by the global backtracking algorithm to correct mistakes on *virtual space*. *GCT* (*Global Closed Time*) messages are used by the new kernel to control the effectiveness of objections (*GCT* plays an important role in global backtracking, it will be discussed in latter sections). Therefore, there are four cases for processing an incoming message:

NORMAL: insert the message into IQ_i ; set a pending *rollback* requirement if the virtual receive time of the message is in the “past”;

ANTI: annihilate the complement of the *anti* message; set a pending *rollback* requirement if the virtual receive time of the complement message is in the “past”;

OBJECTION: set a pending *backtrack* requirement;

GCT: possibly *sustains* a suspended objection (for further details see the next section);

On the other hand, each primitive is carried out by a request-service-reply transaction. The typical primitives called by I_i , and the corresponding actions taken by K_i are summarized as follows (suppose no pending rollback/backtrack requirement):

SEND(m): send m to its destination in the outside world, keep a copy of m in OQ_i and **REPLY(ok);**

RECEIVE: **REPLY(ok , ($SYSTEM$, $+\infty'$, \neg , $+\infty$, \neg))** if IQ_i is empty (here we define that $+\infty' < +\infty$ in order to maintain the total ordering relation); otherwise, **REPLY(ok , m),** where m is the first unreceived message in the current IQ_i , and mark m *received*;

OBJECT($t_c, obj, type$): invoke objection algorithm (for further details see the next section), **REPLY**(ok) when the objection message obj is sustained;

BACKTO(t): reset pointers of IQ_i and OQ_i with respect to t , delete side effects and send *anti* messages if necessary, and **REPLY**(ok).

TERMINATE: invoke termination algorithm (see the next section).

SEND and **RECEIVE** are communication primitives for *normal* messages. **OBJECT** is used for global backtracking and will be discussed later. In order to maintain a consistent lvt_i between K_i and I_i , **BACKTO** is called by I_i upon each rollback or global backtracking activity, so that K_i can adjust different queues with respect to the correct virtual time. Such a request-service-reply transaction is based on the assumption that I_i is in a consistent state. However, if there is a pending rollback/backtrack requirement in K_i , a special acknowledgement

REPLY($backtype, backtime, objector$)

would be returned to deny any request of I_i .

It is important to note that allowing global backtracking will destroy the nondecreasing property of GVT discussed in Section 4.1.2. This is because the proposed logic programming system deals with nondeterministic computations while the estimation of GVT is based on deterministic computations. For example, a process in our system may backtrack to any earlier choice point such that the entire execution history of the process must be preserved. At the moment, we do not consider GVT and assume that there is infinite memory available and there are no I/O activities. A solution for estimating GVT will be discussed in Chapter 6.

4.2.2 Global Backtracking and Termination Functions

Global backtracking functions provided by the *kernel* consists of two algorithms: one for incoming *objection* messages and another for outgoing *objection* messages.

The algorithm for treating an incoming objection message is quite simple: it just sets up a pending backtrack requirement. In this case, process K_i will deny any service request from I_i by replying with the pending backtrack information. Subsequently I_i will backtrack to find another alternative at a specified virtual time.

On the other hand, the algorithm for treating an outgoing objection message is more complex. To illustrate, assume that I_i issues a call $\text{OBJECT}(t_c, \text{obj}, \text{flag})$, where t_c is the objection *competition* time, *obj* is the objection message, and *flag* indicates whether the process to be objected has a *closed* (*open*) virtual space (which will be discussed in the next chapter) with respect to the objection. The algorithm is as follows:

1. If $\text{flag} = \text{closed}$, K_i sends *obj* to the destination immediately. We say that the objection is *sustained*.
2. Otherwise, the objection and thereby, I_i , is *suspended* until
 - (a) a rollback/backtrack requirement is detected. Then K_i denies the objection request. We say that the objection is *overruled*. Or
 - (b) a *GCT* message is received and $GCT = t_c$. Then K_i sustains the objection.

Global Closed Time (GCT) indicates that the current global virtual space is closed at *GCT*. It is quite similar to *GVT* and is defined as:

$$GCT(\tau) = \min(\text{competition times of all suspended processes at } \tau, \\ \text{timestamps of all messages in transmission at } \tau, \\ \text{ } lvt\text{'s of unsuspended processes at } \tau)$$

GCT can be estimated in the way that GVT is estimated, but with a slight modification. A detailed GVT estimation algorithm can be found in [XUC⁺86]. From which we can build a GCT estimation algorithm by inhibiting GCT in the course of global backtracking and recalculating a new GCT whenever global backtracking is done.

Like GVT , GCT guarantees (from its definition) that no process in the current global virtual space can ever *roll back* to a virtual time before GCT . Unlike GVT , GCT possibly decreases, because it is possible that a process will *backtrack* to virtual time before GCT , that is, the new GCT after a global backtracking is less than the old GCT before the global backtracking. Therefore, GCT can only be used as a time measurement to control global backtracking.

When I_i encounters an empty goal, it issues a TERMINATE request to K_i . There are two possible situations: IQ_i is empty or some unreceived messages are still pending in IQ_i . K_i executes the following algorithm to control termination:

1. $REPLY(ok)$ if and only if GCT becomes $+\infty$ and there is no pending message in IQ_i ;
2. If some unreceived message are still pending in IQ_i or a new message arrives during waiting for GCT , then suppose $(P_j, t_s, P_i, t_r, -)$ be the first unreceived message in IQ_i , $REPLY(backtrack, +\infty, (P_j, t_s, P_i, +\infty, closed))$. Later in the

next chapter we will see that such a reply will force P_i backtracking to try alternatives before $+\infty$.

In summary, we have described the major modifications and extensions of the Time Warp mechanism. The new version provides simple interfaces and efficient algorithms which support both rollback and global backtracking activities. Such a version can be implemented either as an operating system kernel or by individual *kernel* processes. More detailed discussion of the global backtracking functions is one of the subjects of the next chapter.

Chapter 5

A LOGIC PROCESS INTERPRETER

The logic process interpreter defined in this chapter is an extended standard Prolog interpreter. The major extensions are parts of the rollback facility and a global backtracking facility.

As Jefferson pointed out [Jef85], the Time Warp mechanism gambles on *virtual time*. Every time a process handles a message m with timestamp t it makes a bet that no message will arrive that has a timestamp earlier than t . If it wins that bet no time is lost. If it loses the bet, the time lost is the delay involved in restoring the process to an earlier state, removing any side effects and running it forward to the point when message m can be processed. The rollback facility is used to correct mistakes on such gambles and *anti* messages are used to remove side effects (if any) from these mistakes.

A useful comparison can be made between this gamble on *virtual time* with the search rule of a logic process interpreter that gambles on *virtual space*. Every time a process reduces a goal it makes a bet that using the depth-first, unifiable, clause to reduce the goal will lead to a successful solution. If it wins that bet no time is lost. If it loses the bet, the time lost is the delay involved in backing to that goal and finding an alternative clause (if any) in depth-first. The (global) backtracking facility is used to handle failures on such gambles, and *objection* messages are used to recompose the search paths of processes (if necessary).

The similarity between the global backtracking facility and the rollback facility is

that they both depend on a saved execution history to undo erroneous computation. The primary difference is that the backtracking facility tries alternatives at a previous choice point of the process, while the rollback facility directly restores a previous state for a specified virtual time and then continues execution of the process forward along the original evaluation path. Since the standard Prolog interpreter has a built-in state-saving and backtracking mechanism, we can easily extend this mechanism to accomplish rollback and global backtracking.

In the following sections, we start by describing important concepts and data structures of the logic process interpreter. Then based on the standard Prolog interpreter, we introduce the extensions for achieving rollback and global backtracking. Finally, we prove that the proposed logic programming system is *sound* and *partially complete*.

5.1 New Concepts and Data Structures

An important new concept which is used to control global backtracking is the *Global Backtracking Coordinator*. A *Global Backtracking Coordinator*, denoted by ξ , is an imaginary predicate used by the system and invisible to the user. A ξ is put at the beginning of each process and immediately following each communication predicate.

During a process evaluation, ξ 's form a layered *parent-child* structure. A typical parent-child layer is shown in Figure 5.1, where the ξ on the top level is the parent of ξ 's on the lower level. The edge between a parent-child pair of ξ 's represents a partial computation which consists of zero or more normal Prolog procedure calls and exactly one communication predicate. If an edge involves a *receive* predicate,

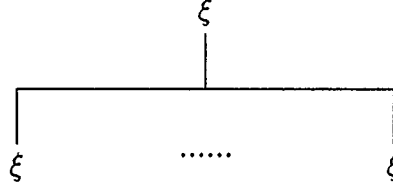
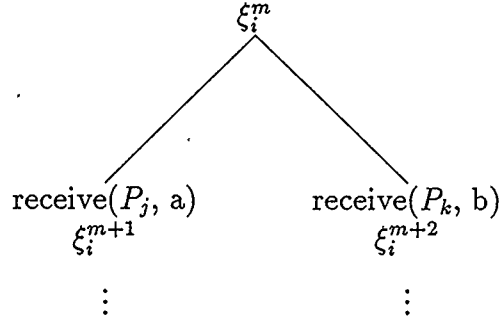


Figure 5.1: A Parent-Child Layer

it is a *receive branch* of the parent ξ ; if an edge involves a *send* predicate, it is a *send branch* of the parent ξ , otherwise, it is a *termination branch* of the parent ξ . By using the leftmost-goal-first computation rule and the depth-first search rule, the *virtual space* of a logic process can be unambiguously represented by a ξ -tree. As a matter of fact, a ξ -tree is a simplified *search-tree* [SS86], it denotes all possible communication/synchronization points of the process with other processes. Suppose each node in a ξ -tree is uniquely labeled, we use ξ_i^m to indicate a *search position* of process P_i where the superscript is a unique label of the node and the subscript is the index of the process. Each branch in a ξ -tree from the root consists of a sequence of ξ 's and is called a *search path* of the process.

The *behavior* of ξ_i^m is defined as the possible executions of P_i at or under ξ^m before reaching any child ξ . The *forward transition* of ξ_i^m implies that the execution of P_i from ξ^m successfully reaches a child ξ . *Local backtracking* is a backtrack in between ξ 's which has no influence on other processes. *Global backtracking* is a backtrack passing a ξ which may cause effects on other processes.

The notion of global knowledge is the basis for global backtracking. Before discussing how a process establishes such knowledge, we introduce another new concept:

Figure 5.2: A Possible Partial ξ -tree of P_i

depends on relation.

The *depends on* relation is a dynamic relation which describes the dependency of two processes with respect to their virtual spaces during execution. For a couple of *search position*:

$$(\xi_i^m, \xi_j^n),$$

we say that ξ_i^m *depends on* ξ_j^n if the *forward transition* of ξ_i^m relies upon the behavior of ξ_j^n . ξ_j^n is a *sponsor* of ξ_i^m if ξ_i^m depends on ξ_j^n . ξ_j^n is a *closed* sponsor of ξ_i^m if P_j has exhausted all the alternatives under ξ_j^n ; otherwise, ξ_j^n is an *open* sponsor of ξ_i^m .

For example, a possible partial ξ -tree of P_i is shown in Figure 5.2. It is clear that ξ_i^m depends on some ξ in P_j or P_k , because P_i can go forward only if the message delivered by its kernel was sent by either P_j or P_k and the message unifies one of the receive predicates under ξ_i^m . If such a message is received by P_i , it transits to a new search position ξ_i^{m+1} or ξ_i^{m+2} ; otherwise, P_i backtracks to ξ_i^m , rejects the wrong message and demands an alternative message by making an *objection* to one of its sponsors.

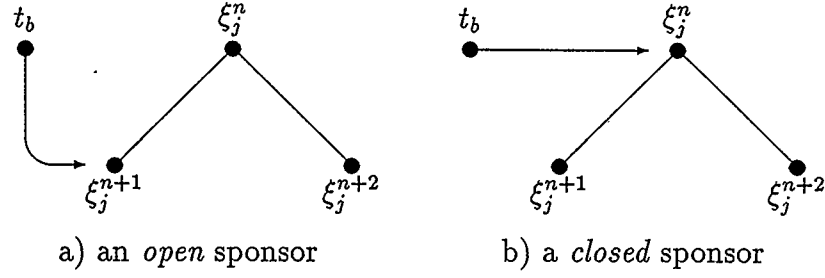


Figure 5.3: The Relationship between t_b , *status* and ξ_j^n

In implementation, the following data structure is used to represent a *sponsor* ξ_j^n :

$$\xi_j^n = (P_j, t_b, \text{status}),$$

where P_j is the sponsor process, t_b is the suggested global backtracking time of P_j , and *status* indicates if ξ_j^n is *open* or *closed*. As mentioned before, ξ_j^n indicates a unique search position in P_j 's virtual space. This search position is uniquely specified by t_b and *status* if the execution of P_j has passed ξ_j^n , i.e., ξ_j^n is in P_j 's current *search path*. This is because each search position of P_j in a given *search path* is associated with a unique *virtual time* (recall the total ordering relation defined in Chapter 4). Later we will see that the proposed algorithm guarantees the above “if” condition. If ξ_j^n is *open*, then t_b is the virtual time of a child ξ of ξ_j^n . If it is *closed*, t_b is the virtual time of ξ_j^n . Figure 5.3 shows the relationship between t_b , *status* and ξ_j^n . The reason for doing so is to make the global backtracking algorithm simple and efficient.

The basic means through which it becomes possible for a process to gather global knowledge is communication. “Communication in a distributed system can be viewed (and often should be viewed) as the act of transforming the system’s state of knowledge” [HM84]. Since a process knows only those things that it has explicitly met

during its execution and communication, we use a *state queue*, denoted by SQ_i , to memorize important historical events in the current search path of P_i .

SQ_i is a sequence of timestamped states and is maintained by the logic process interpreter I_i . A *state* is saved at each ξ in the current search path and is defined as:

$$\xi.state = (t, t_c, status, S_w, S_r)$$

where t is the virtual time at which ξ is executed, t_c is the *objection competition time* at ξ , *status* indicates if ξ is *open* or *closed*, S_w is the *working* sponsor set of ξ and S_r is the *reserved* sponsor set of ξ .

The *objection competition time* is used to control the effectiveness of an *open* objection. If the current ξ has no *receive branch*, then $t_c = t$; otherwise, t_c is set to be the virtual receiving time of the message delivered by the *kernel*. More detailed discussion about t_c and its relation with *GCT* is delayed to Section 5.2.3.

The *status* is used to control the evaluation of ξ 's branches. If it is *open*, then any branch can be searched with respect to the *search rule*. If it is *closed*, then *send branches* are forbidden to be evaluated. The reason of doing so will be described later.

The working sponsor set S_w in $\xi.state$ maintains sponsors established in evaluating branches of ξ . Entries in S_w may be *open* or *closed*. The reserved sponsor set S_r maintains sponsors transferred from ξ 's children or from S_w . Entries in S_r are all *closed*. Detailed discussion about the usage of these sets is in the next section. Here we introduce two operations for a sponsor set.

Operation $put(S, sp)$ inserts a sponsor sp into S . The algorithm of put is as follows (recall that the data structure of sponsor is defined as $(p, t_b, status)$):

1. find an sp' from S such that $sp.p == sp'.p$;
2. if $sp' == \text{NULL}$, insert sp into S , return;
3. if $sp'.status == \text{open}$, sp overwrites sp' , return;
4. if $sp'.status == \text{closed}$, return;

This algorithm maintains exactly one entry for each sponsor process. An important property of the algorithm is that if a sponsor will eventually becomes *closed*, then S will eventually contain only *closed* sponsors.

Function $select(S)$ is used to choose a sponsor from S for making an objection. The function takes the following rules:

1. divide S into two disjunctive subsets S_c and S_o , where entries in S_c are *closed* sponsors and entries in S_o are *open* sponsors (if any); if there exist entries $sp_1 \in S_o$ and $sp_2 \in S_c$ such that $sp_1.p == sp_2.p$ and $sp_1.t_b == sp_2.t_b$, then remove sp_1 from S_o and S , (a search position cannot be both *open* and *closed*);
2. if $S_o \neq \{\}$, select an entry sp from S_o , such that for all $sp' \in S_o$, $sp \rightarrow sp'$;
3. else select an entry sp from S_c , such that for all $sp' \in S_c$, $sp' \rightarrow sp$;
4. remove sp from S and return sp .

The algorithm tells us an important property: a process objects a *closed* sponsor at a ξ only if all its sponsors are *closed*.

Now, we briefly discuss how sponsors are established from a simple example illustrated in Figure 5.4; and a more detailed algorithm for general cases will be

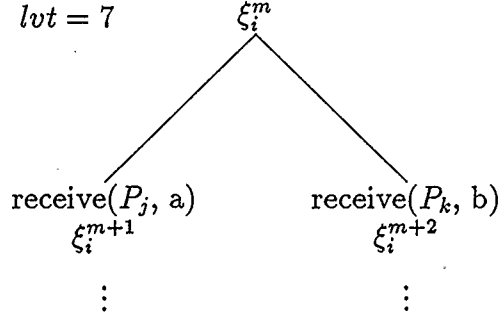


Figure 5.4: An Example for Constructing a State

described in the next section. In this example, when process P_i at virtual time 7 executes ξ_i^m , it creates an initial state:

$$\xi_i^m.state = (7, 7, open, \{\}, \{\}).$$

Then P_i chooses the left branch (by the search rule) to execute. When P_i encounters the call $receive(P_j, a)$, it issues a RECEIVE primitive. Now, there are several possible situations:

1. Suppose the message replied by the *kernel* is $(P_h, 5, P_i, 10, c)$, which means that the sender P_h at virtual time 5 schedules the receipt of c by P_i at virtual time 10. P_i tries to match this message with $receive(P_j, a)$ branch, but fails. Thus P_i constructs a sponsor entry $(P_j, 10, open)$ and inserts the entry into the current S_w . This entry says “a message from P_j is expected with virtual receive time in the interval from 7 to 10”. Then P_i backtracks locally and tries to match the message with the next branch $receive(P_k, b)$, but fails again. A new sponsor entry $(P_k, 10, open)$ is inserted into the current S_w . Then P_i backtracks to ξ_i^m (no more alternatives), it inserts two sponsor entries, $(P_h, 5, open)$ and

$(P_i, 7, closed)$, into the current S_w . In addition, P_i changes the value of t_c in the current state to 10 and the *status* to *closed*. At this time, the state of ξ_i^m becomes:

$$\xi_i^m.state = (7, 10, closed, S_w, S_r)$$

where

$$S_w = \{(P_h, 5, open), (P_j, 10, open), (P_k, 10, open), (P_i, 7, closed)\},$$

$$S_r = \{ \}.$$

Function $select(S_w \cup S_r)$ will return $(P_h, 5, open)$ to ξ_i^m for making an objection.

2. Suppose the IQ_i is empty at the moment the RECEIVE primitive is called, the message replied by the *kernel* is $(SYSTEM, +\infty', -, +\infty, -)$. Since the message matches neither branch of ξ_i^m , when process P_i backtracks to ξ_i^m , we have

$$\xi_i^m.state = (7, +\infty, closed, S_w, S_r),$$

where

$$S_w = \{(SYSTEM, +\infty', open), (P_j, +\infty, open), (P_k, +\infty, open), (P_i, 7, closed)\},$$

$$S_r = \{ \}.$$

Thus $select(S_w \cup S_r)$ operation will return $(SYSTEM, +\infty', open)$ to ξ_i^m for making an objection.

3. Suppose the message replied by the *kernel* is $(P_j, 5, P_i, 10, a)$, which means that the sender P_j at virtual time 5 schedules the receipt of a by P_i at virtual time 10. P_i tries to match this message with the left branch $receive(P_j, a)$ and succeeds.

Thus P_i inserts an entry $(P_j, 5, open)$ into the current S_w and transits to ξ_i^{m+1} at virtual time 10. However, if some time later ξ_i^{m+1} fails, P_i backtracks to try the next branch and finally backtracks to ξ_i^m . In this situation, we have

$$\xi_i^m.state = (7, 10, closed, S_w, S_r),$$

where

$$S_w = \{(P_j, 5, open), (P_k, 10, open), (P_i, 7, closed)\},$$

$$S_r = S_r \cup \xi_i^{m+1}.S_r.$$

Function $select(S_w \cup S_r)$ will return $(P_j, 5, open)$ to ξ_i^m for making an objection.

From the above analysis, we can see that whenever a ξ makes an objection, all alternatives under the ξ must have been exhausted and there must be at least a sponsor entry in either S_w or S_r for each branch under the ξ . A more detailed algorithm and example are in the next section.

Finally, we declare a set of global variables and constants for each logic process interpreter:

rs: a reference to the current state;

rp: a reference to the parent state;

backtime: a virtual time variable to indicate rollback or global backtracking time;

backtype: a variable of enumerated type

(Rollback, Global-Backtrack, Local-Backtrack)

to indicate current backward execution type, its initial value is *Local-Backtrack*;

candidate: a *message* variable with initial value NULL to denote the current incoming message;

objection: a *message* variable with initial value NULL to save the current objection message;

lvt: local virtual time of the host process;

me: the name of the host process.

5.2 The Algorithm

Before delving into the details of a logic process interpreter I_i , let us glance at the basic idea. Since the logic process L_i is modeled by its ξ -tree, I_i starts from the root and evaluates L_i along the left-most branch. The forward execution is almost the same as standard Prolog. During the evaluation, L_i may receive messages from other processes. Thus I_i establishes sponsor relations with these communication partners.

However, when an evaluation fails, I_i follows the standard Prolog procedure to locally backtrack or invokes global backtracking algorithm when the backtracked goal is a ξ .

The intuition behind the global backtracking algorithm is that whenever I_i backtracks to a ξ , it assumes that the synchronizations made before the ξ are correct and gambles on that earlier synchronizations may make subsequent synchronizations more likely to succeed. Therefore, the interpreter does not simply fail the ξ . Instead, it chooses a process (possibly itself) to object, because according to I_i 's knowledge, the process to be objected is most probably in a wrong search position.

The algorithms provided in this section are an extension of a standard Prolog interpreter to control rollback and global backtrack activities. There are two main procedures to support forward execution and backward execution.

5.2.1 Forward Execution

The *Forward Execution Algorithm (FEA)* simply mimics a forward evaluation step of the original interpreter, except that it gives special treatment to a selected goal if the goal is a ξ or a communication predicate.

Normally, if the goal is a ξ , it saves the old state, creates a new state and then succeeds; if the goal is a *send*, it constructs a message, calls SEND request to transfer the message and then succeeds; if the goal is a *receive*, it establishes a sponsor relation and tries to match the predicate with the first unreceived message coming from the current *IQ*, if they match, the predicate succeeds with a *mgu* (see the definition in Chapter 3), otherwise, the goal fails. Sometimes a request may be denied by a special reply if there is a pending rollback/backtrack requirement. In this case, the algorithm fails the selected goal, and in consequence, the *Backward Execution Algorithm (BEA)* is invoked.

In the algorithm, two frequently used functions are *Fail* and *Succeed*. Function *Fail* means backtracking to the most recently evaluated goal and executing *BEA*. Function *Succeed(θ)* means applying θ (a *mgu*) to the goal sequence, selecting the next goal (by the computation rule) and executing *FEA*. For the sake of simplicity, we assume that process names in communication predicates are ground terms, so that correct sponsor relations can be established upon the evaluation of these predicates. In practice, different bindings of a variable process name can be easily traced.

Boolean function *Ok-Reply* is called to check the acknowledgement of a primitive call. It returns *true* if the call has been served, and the variable *candidate* is assigned to the first unreceived message when the call is a RECEIVE; otherwise, it returns *false* and assigns global variables, such as *backtype*, *backtime*, and *objection*, to the corresponding values.

In addition, boolean function *Root(rs)* checks if the current state is the root state; boolean function *Unifiable(T1, T2)* returns *true* if *T1* and *T2* are unifiable; function *Save(rp)* saves state *rp* to the current *SQ*; and function *Parent(rp)* returns a reference to *rp*'s parent state.

Forward Execution Algorithm

Input parameters:

g: selected goal;

begin

(1) If *g* == ξ , then

if not *Root(rs)* then *Save(rp)*;
 rp = *rs*;
 rs = *new(lvt, lvt, {}, open)*;
 Succeed;

(2) If *g* == *send(D, M, T)*, then

if *rs.status* == *open*, then
 call *SEND((me, lvt, D, lvt + T, M))*;
 if *Ok-Reply* then *Succeed* else *Fail*;
 else *Fail*;

(3) If *g* == *receive(S, M)*, then

if *candidate* == NULL, then


```

        issue a RECEIVE request;
        if not Ok-Reply then Fail;
    if Unifiable( $S$ ,  $candidate.P_s$ ), then
        put( $rs.S_w$ , ( $S$ ,  $candidate.t_s$ , open));
    else
        put( $rs.S_w$ , ( $S$ ,  $candidate.t_r$ , open));
    if Unifiable(( $candidate.P_s$ ,  $msg$ ), ( $S$ ,  $M$ )), then
         $\theta = unify((candidate.P_s, msg), (S, M))$ ;
         $lvt = candidate.t_r$ ;
         $candidate = NULL$ ;
        Succeed( $\theta$ );
    else Fail;
(4) If  $g == NULL$ , then
     $lvt = +\infty$ ;
    issue a TERMINATE request;
    if Ok-Reply then exit else Fail;
(5) If  $g == others$ , then
    standard Prolog procedure;
end

```

5.2.2 Backward Execution

The *Backward Execution Algorithm (BEA)* deals with three possible cases whenever it is invoked. They are rollback, local backtracking and global backtracking. If the backtracked goal is not a ξ , according to the current *backtype*, *BEA* either continues rolling back or invokes the standard Prolog backtracking procedure. If the backtracked goal is a ξ , *BEA* invokes an algorithm with respect to the current case.

When a rollback requirement is detected, the underlying *kernel* process tells the logic process interpreter the *backtime* and *backtype*. The *Rollback-To- ξ Algorithm*

(*RTA*) keeps rolling back until it finds a previous ξ whose timestamp is less than or equal to the *backtime*. At this point, the algorithm resets the process's global variables and returns control to *FEA* again.

Backward Execution Algorithm

Input parameters:

g: backtracked goal;

begin

(1) If $g == \xi$, then

 if *backtype* == *Local-Backtrack*, then *Local-To- ξ* ;

 if *backtype* == *Rollback*, then *Rollback-To- ξ* ;

 if *backtype* == *Global-Backtrack*, then *Global-To- ξ* ;

(2) else if *backtrack* == *Local-Backtrack*

 standard Prolog procedure.

(3) else

Fail; /* continue rolling back */

end

Rollback-To- ξ Algorithm

begin

if *backtime* \geq *rs.t*, then

backtype = *Local-Backtrack*;

lvt = *rs.t*;

candidate = NULL;

 call BACKTO(*lvt*);

 if *Ok-Reply* then *Succeed* else *Fail*;

```

else
     $rs = rp$ ;
     $rp = Parent(rp)$ ;
    Fail;
end

```

Possible situations leading to global backtracking are that (1) the local backtracked goal is a ξ , or (2) the logic process interpreter is informed to backtrack by an objection.

If the local backtracked goal is a ξ , it means that all branches under the ξ have been tried and all failed. In other words, the search space of the process is *closed* at the ξ . Thus, the *Local-To- ξ Algorithm (LTA)* completes the construction of the current sponsor set and invokes the *Objection Algorithm*.

Local-To- ξ Algorithm

```

begin
    if not Root( $rs$ ), put( $rs.S_w, (me, rs.t, closed)$ );
    if candidate  $\neq$  NULL, then
        put( $rs.S_w, (candidate.P_s, candidate.t_s, open)$ );
         $rs.t_c = candidate.t_r$ ;
    candidate = NULL;
     $rs.status = closed$ ;
    Objection Algorithm;
end

```

Two steps are taken by the *Global-To- ξ Algorithm (GTA)* to deal with an objection. First, it rolls the process back to a search position specified by *backtime*. Second, if the objection is an *open* objection, the algorithm simply fails the current path, and the next alternative branch of ξ 's parent (if any) will be selected (by the search rule); otherwise, the process chooses a new candidate to make a subsequent objection.

Global-To- ξ Algorithm

begin

(a) if $backtime \geq rs.t$ and $objection.type == open$, then

backtype = *Local-Backtrack*;
 $put(rp.S_w, (objection.P_s, objection.t_s, closed));$
 $rs = rp;$
 $rp = Parent(rp);$
 $lvt = rs.t;$
 $candidate = NULL;$
 call BACKTO(lvt);
Fail;

(b) if $backtime \geq rs.t$ and $objection.type == closed$, then

backtype = *Local-Backtrack*;
 $put(rs.S_w, (objection.P_s, objection.t_s, closed));$
Objection Algorithm;

else

$rs = rp;$
 $rp = Parent(rp);$
Fail;

end

Objection Algorithm (OA) chooses a sponsor from the sponsor sets of the current state for making an objection. If the backtracking time of the selected sponsor is $-\infty$, then no solution can be found (for a proof see Section 5.3), the program terminates. If the selected sponsor represents the current ξ itself (self-objection), then the *Objection Algorithm* transfers ξ 's S_w and S_r to ξ 's parent, and backtracks to try the next alternative branch (if any) of ξ 's parent. Otherwise, the algorithm constructs an objection message and makes the objection. If an *open* objection is sustained, the algorithm only evaluates receive branches under the ξ , because the *open* sponsor is established from one of ξ 's receive branches. If a *closed* objection is sustained, the algorithm will reset the working sponsor set and reopen all search branches under the ξ . Whenever an objection is *sustained*, the control is transferred to *FEA*, otherwise, *BEA* is invoked again.

Objection Algorithm

begin

- (a) $sp = select(rs.S_w \cup rs.S_r);$
- (b) if $sp.tb == -\infty$, then
 - terminate with no solution;
- (c) if $sp.p == me$, then /* self-objection */
 - $rp.S_r = rs.S_w \cup rs.S_r;$
 - $rs = rp;$
 - $rp = Parent(rp);$
 - $lvt = rs.t;$
 - call BACKTO(lvt);
 - Fail;
- (d) /* construct objection message */

```

if  $Root(rs)$  and  $sp.status == closed$ , then
     $obj = (me, -\infty, sp.p, sp.tb, closed)$ ;
else
     $obj = (me, rs.t, sp.p, sp.tb, sp.status)$ ;
(e) call  $OBJECT(rs.tc, obj, sp.status)$ ;
(f) if Ok-Reply, then
    if  $sp.status = closed$  then
         $rs.status = open$ ;
         $rs.S_r = rs.S_r \cup rs.S_w$ ;
         $rs.S_w = \{\}$ ;
        Succeed;
    else goto BEA;
end

```

As it was presented in the algorithms, objection messages play a very important role in global backtracking. However, a question we haven't carefully answered concerns which objection message comes into effect if several processes have made objections. To answer this question, we examine two different objections: the objection to an *open* sponsor (*open* objection) and the objection to a *closed* sponsor (*closed* objection), respectively.

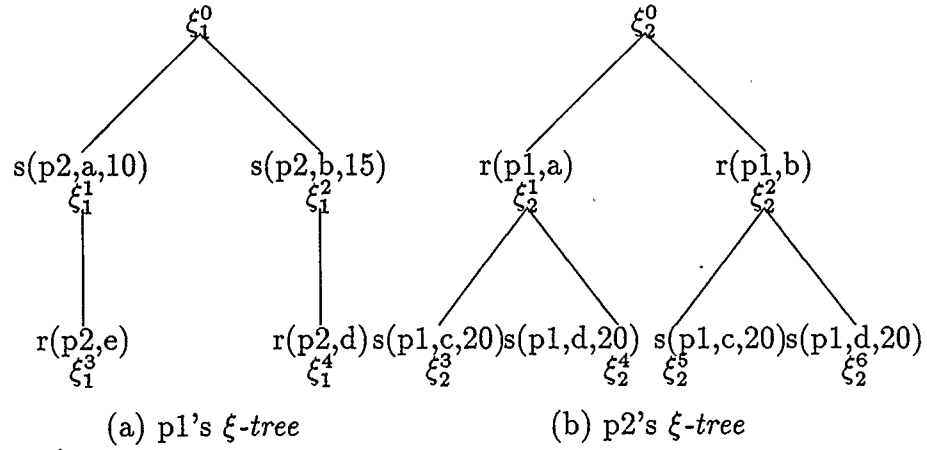
Let us consider an example. In virtual time order, process P_i should receive a message from P_1 first and then a message from P_2 . If P_2 's message arrives earlier than the message from P_1 in real time, P_i objects P_2 because it is now waiting for a message from P_1 and ignores the fact that P_2 's message may be correct in the future. Of course, such an objection is unreasonable. The strategy adopted by the proposed algorithm is to suspend any *open* objection until its competition time equals GCT

(see the algorithm in Chapter 4), because according to the definition of *GCT*, it is guaranteed that no other processes will ever send messages with timestamps earlier than the competition time of the objection. Thus, the objection to P_2 is suspended and will be overruled when the message from P_1 arrives.

Precisely speaking, a suspended objection is overruled by an incoming message, such as an anti-message, a normal message or an objection message, which *happens before* the objection. Whenever an objection is overruled, it subsequently appears that the objection never existed.

On the other hand, a *closed* objection is caused by a failure on *virtual space*, because P_i can make such an objection only if all its sponsors are *closed* (see the property of *select* operation). That is, P_i has failed in trying to cooperate with each possible alternative of its sponsors under their *closed* ξ 's. Therefore, a *closed* objection is always sustained immediately.

When process P_i makes an objection at a ξ , its search space under the ξ is closed. Does P_i reopen its search space under the ξ as soon as the objection is sustained? The answer once again depends on the type of the objection. P_i will reopen its search space under the ξ only if the objection it made is a *closed* objection. In general, P_i may have both *send* branches and *receive* branches under a ξ . When P_i backtracks to such a ξ and makes an *open* objection, it means that all *send* branches have failed but *receive* branches are still trying by the *open* objection (recall that *open* sponsors are created only at receive branches). For this moment, P_i does not re-try *send* branches (see *FEA*), until all *open* sponsors of ξ 's become *closed*. Then P_i chooses a *closed* sponsor to object and reopens the search space under ξ .

Figure 5.5: The ξ -trees of the Example

5.2.3 An Example

Now, we use a simple example to illustrate how the algorithms work. The example consists of two logic processes:

$p1 \leftarrow \text{send}(p2, a, 10), \text{receive}(p2, e).$

$p1 \leftarrow \text{send}(p2, b, 15), \text{receive}(p2, d).$

$p2 \leftarrow \text{receive}(p1, a), g.$

$p2 \leftarrow \text{receive}(p1, b), g.$

$g \leftarrow \text{send}(p1, c, 20).$

$g \leftarrow \text{send}(p1, d, 20).$

Figure 5.5 shows the ξ -trees of the program. For a given query " $\leftarrow p1, p2$ ", two processes $p1$ and $p2$ will be executed in parallel. The following is a possible execution trace of $p1$ and $p2$. In the trace, when we say a process is "waiting for a message", we means that the process's IQ is empty and the process has made an *open* objection

against the *SYSTEM*. In addition, *virtual time* in the trace is defined as $t_1 : t_2$, where t_1 represents *simulation time* while t_2 is an extension to maintain the total ordering relation defined in Section 4.1. The increment of t_2 is 1 in a ξ 's transition if the simulation time does not advance in that transition.

- (1) **p1**: goes through ξ_1^0 at 0:0, send(p2, a, 10) succeeds, goes through ξ_1^1 at 0:1 and waits for a message;
- (2) **p2**: goes through ξ_2^0 at 0:0, receive(p1, a) succeeds, goes through ξ_2^1 at 10:0, send(p1, c, 20) succeeds, goes through ξ_2^3 at 10:1 and terminates at $+\infty$;
- (3) **p1**: receive(p2, e) fails (because the first unreceived message is (p2, c)), backtracks to ξ_1^1 (*Local-To- ξ*), selects a sponsor from:

$$\xi_1^1.S_w = \{(p2, 10 : 1, open), (p1, 0 : 1, closed)\}$$

objects (p2, 10:1, open) with $t_c = 30 : 0$, sustains the objection ($GCT = \min(30 : 0, +\infty)$), waits for a message at ξ_1^1 ;

- (4) **p2**: backtracks over ξ_2^3 (by the *open* objection made in step (3)), send(p1, d, 20) succeeds, goes through ξ_2^4 at 10:1 and terminates at $+\infty$;
- (5) **p1**: receive(p2, e) fails (because the first unreceived message is (p2, d)), takes the similar actions in step (3);
- (6) **p2**: backtracks over ξ_2^4 (by the *open* objection made in step (5)), backtracks to ξ_2^1 , selects a sponsor from:

$$\xi_2^1.S_w = \{(p2, 10 : 0, closed), (p1, 0 : 1, closed)\},$$

backtracks over ξ_2^1 (self-objection), receive(p1, b) fails (because the first unreceived message is (p1, a)), backtracks to ξ_2^0 , selects a sponsor from:

$$\xi_2^0.S_w = \{(p1, 0 : 1, open), (p2, 0 : 0, closed)\}$$

$$\xi_2^0.S_r = \{(p1, 0 : 1, closed)\}$$

objects (p1, 0:1, closed) (recall the *select* operation, a sponsor cannot be both *open* and *closed*), sustains the objection immediately and waits for a message at ξ_2^0 ;

- (7) **p1:** backtracks to ξ_1^1 (by the *closed* objection made at step (6)), selects a sponsor from:

$$\xi_1^1.S_w = \{(p1, 0 : 1, closed), (p2, 0 : 0, closed)\}$$

backtracks over ξ_1^1 (self-objection), send(p2, b, 15) succeeds, goes through ξ_1^2 at 0:1 and waits for a message;

- (8) **p2:** receive(p1, a) fails (because the first unreceived message is (p1, b)), local backtracks, receive(p1, b) succeeds, goes through ξ_2^2 at 10:0, send(p1, c, 20) succeeds, goes through ξ_2^5 at 10:1 and terminates at $+\infty$;

- (9) **p1:** receive(p2, d) fails (because the first unreceived message is (p2, c)), backtracks to ξ_1^2 , selects a sponsor from:

$$\xi_1^2.S_w = \{(p1, 0 : 1, closed), (p2, 10 : 1, open)\}$$

objects (p2, 10:1, open) with $t_c = 35 : 0$, sustains the objection ($GCT = \min(35 : 0, +\infty)$), and waits for a message at ξ_1^2 ;

- (10) **p2**: backtracks over ξ_2^5 (by the *open* objection made at step (9)), `send(p1, d, 20)` succeeds, goes through ξ_2^6 at 10:1 and terminates at $+\infty$;
- (11) **p1**: `receive(p2, d)` succeeds, goes ξ_1^4 at 35:0 and terminates at $+\infty$.
- (12) **p1 and p2**: algorithm terminates with a solution.

Without counting *GCT* and *anti* messages, this example spent 10 messages – six *normal* messages and four *objection* messages – to find a solution. The cost is due to the nondeterminism of the example.

The above algorithms and discussions illustrate the basis of our work. In brief, we have extended the standard Prolog interpreter to include the abilities of rollback and global backtracking. Thereby combining logic programming technique and the Time Warp mechanism not only provides a temporal coordinate system which measures computational progress and defines synchronizations but also provides a spatial coordinate system which supports nondeterministic computations.

5.3 Correctness of the Algorithm

The correctness of a logic programming system consists of two aspects: soundness and completeness. Informally, for a logic program \mathcal{P} , the system is sound if every computed solution of \mathcal{P} is a logical answer of \mathcal{P} ; the system is complete if whenever there exists a solution of \mathcal{P} , the system will eventually find the solution.

For the subsequent discussion we use the following assumptions:

ASM(1): Each process of a distributed logic program defines a finite ξ -tree, *i.e.*, the execution of a distributed logic program terminates in finite evaluation steps.

ASM(2): The standard Prolog interpreter is sound (assume the “occur check” problem has been solved [Llo84]) and complete (for programs with finite *SLD-trees*, see **ASM(1)**), *i.e.*, in order to find a solution, the standard Prolog interpreter searches its (partial) virtual space exhaustively if no ξ 's are involved in the (partial) virtual space.

ASM(3): The *kernel* is reliable and the rollback facility is transparent, which means that all messages are delivered in correct virtual time order. In other words, the temporal property of a logic process is guaranteed by its *kernel* and the rollback algorithm.

ASM(4): The estimation of *GCT* is correct, that is, at any real time, the estimated value of *GCT* is always less than or equal to the real *GCT* (see the definition) and will eventually reach the real *GCT*.

As described in our algorithms, a trace of a process can be described as a finite sequence of evaluated ξ 's. Thus, a process is characterized by a set of all its possible traces and is “displayed” on a finite ξ -tree (by *ASM(1)*). In a ξ -tree, ξ 's form a layered *parent-child* structure. The edge connecting a parent-child pair of ξ 's consists of derivations of zero or more normal Prolog goals and exactly one communication predicate. In the following discussion, we assume that a normal Prolog goal is always derivable. Therefore, in a transition from a parent to a child, we only need to pay attention to the derivation of the communication predicate which occurred in the transition. As we have assumed that each ξ is uniquely labeled, by saying that the control of process P_i is after l , we mean that the ξ_i^l was the last one to be evaluated in P_i . Additionally, we use Φ to denote the label of the ξ placed at the beginning of

a process and Ω to indicate the termination of a process.

Definition 5.1 *Let \mathcal{P} be a distributed logic program and G a goal. If the evaluation of $\mathcal{P} \cup \{G\}$ invokes processes P_1, \dots, P_n , then (P_1, \dots, P_n) is called a coordinated computation of \mathcal{P} .*

Definition 5.2 *Let (P_1, \dots, P_n) be a coordinated computation. A state of (P_1, \dots, P_n) is (l_1, \dots, l_n) iff the control of processes P_1, \dots, P_n is after l_1, \dots, l_n respectively. As well, we say that the initial state of (P_1, \dots, P_n) is (Φ_1, \dots, Φ_n) and a final state is $(\Omega_1, \dots, \Omega_n)$.*

Definition 5.3 *Let $(l_1, \dots, l_n), (l'_1, \dots, l'_n)$ be two states of (P_1, \dots, P_n) . A state transition $(l_1, \dots, l_n) \Rightarrow (l'_1, \dots, l'_n)$ must satisfy (for $i = 1..n$):*

1. *the edge connecting l_i and l'_i is a termination branch and IQ_i is empty forever;*
or
2. *the edge connecting l_i and l'_i is a send branch; or*
3. *the edge connecting l_i and l'_i is a receive branch and the message received was sent in an earlier state transition and the message unifies the receive predicate in the receive branch; or*
4. *$l_i = l'_i$.*

(Note that ASM(3) guarantees that a message is sent to its destination in \rightarrow order and a message is received in \rightarrow order)

A trace of (P_1, \dots, P_n) is described as a sequence of state transitions from its initial state into a unique state, possibly a final state of the coordinated computation,

which in addition reflects the execution history of how it reached that state. Thus, the evaluation of (P_1, \dots, P_n) is characterized by a set of all its possible traces and can be “displayed” on a *transition-tree* defined as follows: the root of the tree is its initial state, nodes of the tree are intermediate states, there is an edge from a node for each state transition, and leaves of the tree are states which have no more transitions or are final states of the coordinated computation. In a *transition-tree*, nodes with the same parent are siblings.

Consider the example in Figure 5.6, we draw the *transition-tree* of (p1, p2) in Figure 5.7. Each node in the tree is in the form of (l_1, l_2) where l indicates the label of a ξ and the subscript of l is the index of a process.

Lemma 5.1 *A state transition of (P_1, \dots, P_n) is implemented by FEA.*

Proof: Let (l_1, \dots, l_n) be the current state of (P_1, \dots, P_n) , and (l'_1, \dots, l'_n) be the state after a transition. *FEA* transits process P_i from l_i to $l'_i, i = 1..n$, as follows:

1. If P_i terminates with an empty IQ_i , then $l'_i = \Omega_i$ (by *FEA(4)* and *ASM(1,3)*).
2. If P_i evaluates a *send* predicate and l_i is *open*, then l'_i is the label of the ξ immediately following the *send* predicate (by *FEA(2)*). Note that l_i becomes *closed* only if all branches under l_i have failed.
3. If P_i evaluates a *receive* predicate which matches the message delivered by the *kernel*, then l'_i is the label of the ξ immediately following the *receive* predicate (by *FEA(3)* and *ASM(3)*).

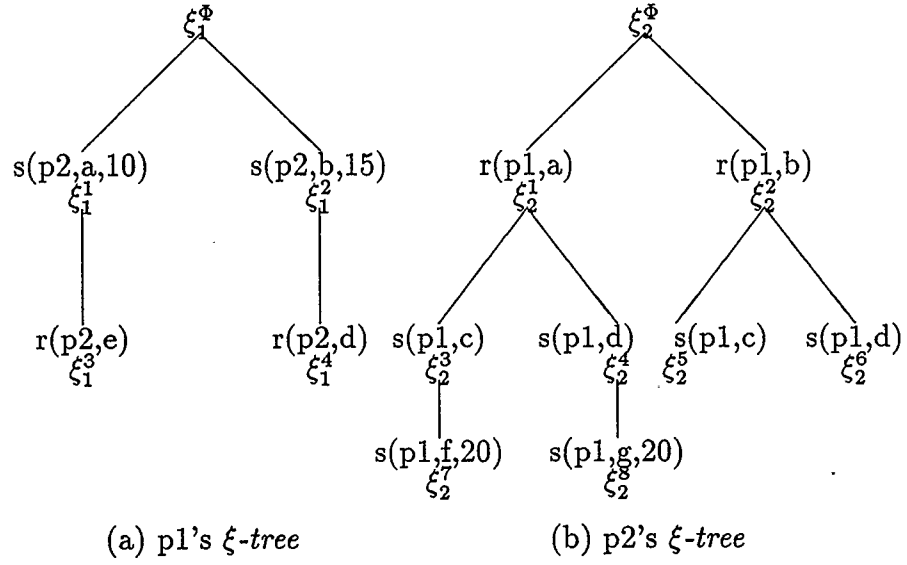


Figure 5.6: Example of Process Transitions

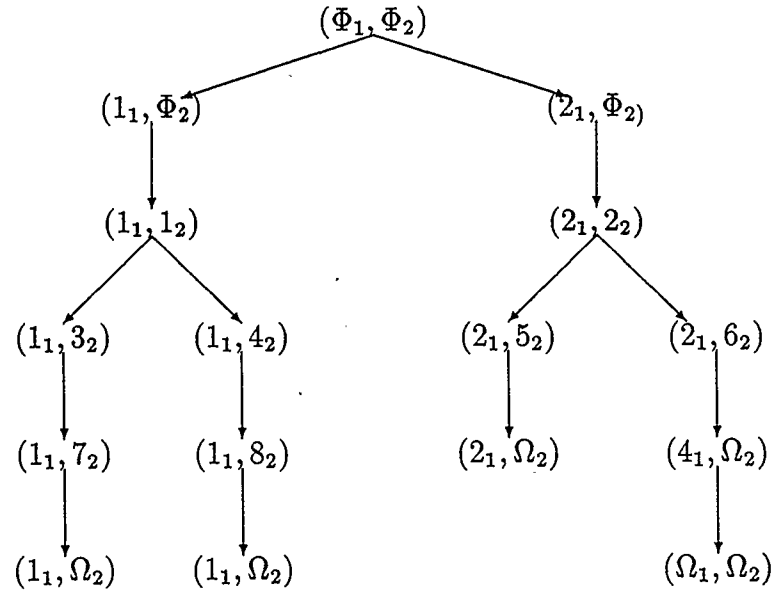


Figure 5.7: Transition-tree of the Example

4. otherwise no transition can be made for P_i , $l'_i = l_i$;

Thus, the transition implemented by the *FEA* fulfills the **Definition 5.3** exactly.

Definition 5.4 Let \mathcal{P} be a distributed logic program, G a goal and (P_1, \dots, P_n) the coordinated computation invoked by $\mathcal{P} \cup \{G\}$. If $(\Phi_1, \dots, \Phi_n) \Rightarrow \dots \Rightarrow (\Omega_1, \dots, \Omega_n)$ is a sequence of transitions of (P_1, \dots, P_n) via *FEA*, and θ_i is a sequence of mgu's computed in the transitions of P_i , $i = 1..n$, then $(P_1\theta_1 \wedge \dots \wedge P_n\theta_n)$ is a solution for $\mathcal{P} \cup \{G\}$.

Theorem 5.1 (soundness) Let \mathcal{P} be a distributed logic program, G a goal. Then every solution for $\mathcal{P} \cup \{G\}$ is a correct solution.

Proof: Let (P_1, \dots, P_n) be the coordinated computation of $\mathcal{P} \cup \{G\}$ and $(P_1\theta_1 \wedge \dots \wedge P_n\theta_n)$ a solution of $\mathcal{P} \cup \{G\}$. The solution is a correct solution for $\mathcal{P} \cup \{G\}$ if the computed answer substitution θ_i 's ($i = 1..n$) are correct answer substitutions. From Definition 5.4, this solution is generated by a transition sequence $(\Phi_1, \dots, \Phi_n) \Rightarrow \dots \Rightarrow (\Omega_1, \dots, \Omega_n)$ of (P_1, \dots, P_n) .

(1) If the transtion sequence of (P_1, \dots, P_n) do not involve any communications, then $(P_1\theta_1 \wedge \dots \wedge P_n\theta_n)$ is correct solution of $\mathcal{P} \cup \{G\}$ because:

1. P_i 's do not share any non-ground variables (from Definition 3.4);
2. the refutation of each P_i is implemented by a standard Prolog interpreter (*FEA*(5) and **ASM**(2));
3. therefore, the computed answer substitution θ_i is a correct answer substitution (for a proof see [Llo84]: soundness of *SLD-resolution*).

(2) If the transtions of (P_1, \dots, P_n) involve communications, we define θ_i as the composition of ρ_i and σ_i , where ρ_i represents a sequence of *mgu*'s in the derivations of normal Prolog goals while σ_i represents a sequence of *mgu*'s in the derivations of communication predicates. We assert that $(P_1\theta_1 \wedge \dots \wedge P_n\theta_n)$ is a correct solution of $\mathcal{P} \cup \{G\}$ because:

1. P_i 's do not share any non-ground variables (from Definition 3.4);
2. each ρ_i is generated by a standard Prolog interpreter (*FEA(5)* and *ASM(2)*), i.e., ρ_i is a correct answer substitution with respect to all Prolog goals in P_i 's refutation;
3. each σ_i is generated by *FEA(3)* which uses the standard Prolog's unification algorithm and implements the *Synchronization-Test* (Definition 3.9) with respect to the *temporal property* of P_i (*ASM(3)*), i.e., σ_i is a correct answer substitution with respect to all communication predicates in P_i 's refutation;
4. therefore, θ_i , the composition of ρ_i and σ_i , is a correct answer substitution.

The problem now is to prove the completeness defined as follows :

A distributed logic programming system is said to be complete if for every given goal and distributed logic program (following *ASM(1)*) the system guarantees to produce a solution provided it exists.

It is important to note that the completeness proof is based on *ASM(1)* and *ASM(2)*. In other words, we prove that the proposed system is as complete as standard Prolog.

Lemma 5.2 *Let \mathcal{P} be a distributed logic program, G a goal and (P_1, \dots, P_n) the coordinated computation invoked by $\mathcal{P} \cup \{G\}$. Then (P_1, \dots, P_n) has a solution if and only if the solution is “displayed” on its transition-tree.*

Proof: This Lemma follows immediately from the definition of *transition-tree* and Definition 5.4.

Based on Lemma 5.2, the completeness proof is going to show that in order to find a solution for a coordinated computation, the proposed global backtracking algorithm searches its *transition-tree* exhaustively. In the following discussion, we only reference states which are displayed on the transition-tree.

Definition 5.5 *Let (l_1, \dots, l_n) be a state of (P_1, \dots, P_n) . (l_1, \dots, l_n) is called a failure state if it can not be transited forward to any new state and it has at least one l_i which is not Ω .*

For example, states (l_1, Ω_2) , (Ω_1, Ω_2) and (Ω_1, Ω_2) in Figure 5.7 are *failure states*.

Lemma 5.3 *No solution exists under a failure state.*

Proof: From Definition 5.4, a solution is produced by a transition sequence from (Φ_1, \dots, Φ_n) to $(\Omega_1, \dots, \Omega_n)$. Since no transition can be made at a *failure state* and there exists at least a l which is not a Ω , so the lemma holds.

Lemma 5.4 *Let (l_1, \dots, l_n) be a state of (P_1, \dots, P_n) . If all branches under ξ_i^l are failed and there exists at least one failed receive predicate in evaluating these branches, then P_i makes an open objection at ξ_i^l .*

Proof: From $ASM(2)$, standard Prolog searches all branches under ξ_i^l exhaustively.

From FEA , an *open* sponsor is established in evaluating a receive predicate.

From LTA , P_i invokes *Objection Algorithm*. From the objection selection algorithm, P_i objects one of its *open* sponsors at ξ_i^l .

Definition 5.6 Let (l_1, \dots, l_n) be a state of (P_1, \dots, P_n) . If there exists a process P_i which is making an *open* objection at l_i and the objection competition time equals to GCT , then (l_1, \dots, l_n) is called an *open-backtrack* state and ξ_i^l is called an *open-backtrack* search position.

Thus, all *failure* states in a transition-tree are *open-backtrack* states, but not *vice versa*. For example, states $(l_1, 7_2)$ and $(l_1, 8_2)$ in Figure 5.7 are *open-backtrack* states because ξ_1^1 is making an *open* objection against the messages sent under ξ_2^1 and snapshot of GCT at these states equals to the objection competition time of ξ_1^1 (recall the definition of GCT , at these states, ξ_1^1 's objection competition time is 10:1 while the virtual time of ξ_2^7 and ξ_2^8 are both 10.2). In these two states, ξ_1^1 is the *open-backtrack* search position. However, these two states are not *failure* states yet because they still can transit to new states.

Lemma 5.5 State transitions from an *open-backtrack* state always lead to a *failure* state.

Proof: Let (l_1, \dots, l_n) be an *open-backtrack* state and ξ_i^l the *open-backtrack* search position. Since ξ_i^l is making an objection and no other processes will ever send messages to ξ_i^l before GCT to overrule the objection (by $ASM(3,4)$), ξ_i^l can not transit to any child ξ . Thus transitions from an *open-backtrack* state (if any) will eventually reach a *failed* state.

Lemma 5.6 *No solution exists under an open-backtrack state.*

Proof: The lemma follows immediately from Lemma 5.3 and Lemma 5.5.

Lemma 5.6 shows the major difference between our algorithm and the dead-lock detection algorithm proposed in [Fut88]. In our scheme, the transition-tree under a state is pruned as soon as the state has been determined as *open-backtrack*, and then the execution transits to a new state by recomposing the search path of a process through the objection committed by the *open-backtrack search position*. In the latter approach, transitions under a *open-backtrack state* will continue until reaching a *failure state* (all processes are blocked), and then a global “unlock” algorithm is called to transit the program into a new state. The advantages of our scheme are that a *open-backtrack state* can be detected locally by processes wrt *GCT* and that it offers potential speed up in nondeterministic computations.

Definition 5.7 *Let (l_1, \dots, l_n) be an open-backtrack state, ξ_i^l the open-backtrack search position of the state and $\xi_j^{l'}$ an open sponsor of ξ_i^l . Then open-backtrackable states of (l_1, \dots, l_n) with respect to $\xi_j^{l'}$ are $(\dots, l_i, \dots, l_j'', \dots)$'s such that each $\xi_j^{l''}$ is an unsearched child of $\xi_j^{l'}$; the open-closed state of (l_1, \dots, l_n) with respect to $\xi_j^{l'}$ is $(\dots, l_i, \dots, l_j', \dots)$ and $\xi_j^{l'}$ is the open-closed search position.*

Recall the example in Figure 5.7, state $(1_1, 7_2)$ and $(1_1, 8_2)$ are open-backtrack states. They have a same open sponsor ξ_2^1 and their open-closed state with respect to ξ_2^1 is $(1_1, 1_2)$. According to this open sponsor, state $(1_1, 7_2)$ has one open-backtrackable state $(1_1, 4_2)$ while state $(1_1, 8_2)$ has no open-backtrackable state.

Lemma 5.7 *Let (l_1, \dots, l_n) be an open-backtrack state and ξ_i^l the open-backtrack search position of the state. An open objection made by ξ_i^l changes (l_1, \dots, l_n) to one of its open-backtrackable states or to its open-closed state when all its open-backtrackable states have been exhausted.*

Proof: Suppose the open sponsor to be objected by ξ_i^l is $\xi_j^{l'}$. When the objection is sustained, the sponsor process P_j either transits to the next unsearched branch (if any) under $\xi_j^{l'}$ (GTA(a) and ASM(2)) or backtracks to $\xi_j^{l'}$ if all branches under $\xi_j^{l'}$ have been exhausted (LTA and ASM(2)).

Lemma 5.8 *No solution exists under an open-closed state.*

Proof: From Lemma 5.7, an open-closed state is caused by open objections made at a set of open-backtrack states. From Lemma 5.6, no solution exists under these states, so the lemma holds.

Since no solution exists under an open-closed state, we have to decide which state to backtrack. There are two situations. First, at the open-closed state, the open-closed search position has its own open sponsors, then it is going to chose an open sponsor and join the open objection competition. All discussion and lemmas above can be applied to this situation. Second, at the open-closed state, all sponsors of the open-closed search position are closed. In this case, we use a heuristic knowledge that earlier synchronizations may make subsequent synchronizations more likely to succeed. Thus, the selection algorithm returns a closed sponsor with maximum virtual time to object.

Definition 5.8 *Let (l_1, \dots, l_n) be a state and ξ_i^l a search position. If ξ_i^l objects itself, then the state is a self-backtrack state.*

Lemma 5.9 *A closed objection made by an open-closed search position either renames the current state to open-backtrack state or to self-backtrack state, or causes a subsequent closed objection.*

Proof: Let (l_1, \dots, l_n) be a state and ξ_i^l a search position. Suppose ξ_i^l makes a closed objection to ξ_j^l . From **GTA(b)**, when process P_j receives the objection, it invokes the *Objection Algorithm*. If there are open sponsors in ξ_j^l 's sponsor set, then ξ_j^l is going to make an open objection, i.e., the state becomes the open-backtrack state; if all sponsors of ξ_j^l are closed, then ξ_j^l either objects itself or makes a subsequent closed objection according to the sponsor selected. In the former case, the current state becomes the self-backtrack state. In the later case, the current state retains its original name.

Lemma 5.10 *Closed objections at an open-closed state will eventually rename the state to either open-backtrack state or self-backtrack state.*

Proof: This lemma immediately follows from Lemma 5.9 and the fact that closed objections made by any process are in decreasing virtual time order (see objection selection algorithm).

Lemma 5.11 *No solution exists under a self-backtrack state.*

Proof: This lemma immediately follows from Lemma 5.8 and 5.10.

Definition 5.9 *Let (l_1, \dots, l_n) be a self-backtrack state, ξ_i^l the self-backtrack search position of the state. Then self-backtrackable states of (l_1, \dots, l_n) with respect to ξ_i^l are (\dots, l'_i, \dots) 's in the transition-tree such that each $\xi_i^{l'}$ is an unsearched sibling of*

ξ_i^l ; the self-closed state of (l_1, \dots, l_n) with respect to ξ_i^l is (\dots, l_i'', \dots) such that $\xi_i^{l''}$ is the parent of ξ_i^l , and $\xi_i^{l''}$ is the self-closed search position.

Recall the example in Figure 5.7, state $(1_1, 1_2)$ is a self-backtrack state. It has no self-backtrackable state. Its self-closed state is $(1_1, \Phi_2)$. State $(1_1, \Phi_2)$ is another self-backtrack state. It has one self-backtrackable state $(2_1, \Phi_2)$. Its self-closed state is (Φ_1, Φ_2) .

Lemma 5.12 *Let (l_1, \dots, l_n) be an self-backtrack state and ξ_i^l the self-backtrack search position of the state. A self-backtrack made by ξ_i^l changes (l_1, \dots, l_n) to one of its self-backtrackable states or to its self-closed state when all its self-backtrackable states have been exhausted.*

Proof: This lemma immediately follows from ASM(2)).

Lemma 5.13 *No solution exists under a self-closed state.*

Proof: From Lemma 5.12, a self-closed state is caused by a sequence of self-backtracking from a set of self-backtrack states. From Lemma 5.11, no solution exists under these states, so the lemma holds.

Lemma 5.14 *Objections at a self-closed state will eventually rename the state to open-backtrack state or self-backtrack state.*

Proof: Similar to the proof of Lemma 5.10.

Theorem 5.2 (completeness) *Let \mathcal{P} be a distributed logic program, G a goal and (P_1, \dots, P_n) the coordinated computation invoked by $\mathcal{P} \cup \{G\}$. When the initial*

state of (P_1, \dots, P_n) becomes a self-backtrack state, the proposed global backtracking algorithm has searched the transition-tree of (P_1, \dots, P_n) exhaustively.

Proof: This theorem directly follows from Lemma 5.11.

Chapter 6

COMMUNICATING SEQUENTIAL PROLOG

(CSP*)

A practical language proposal, Communicating Sequential Prolog, abbreviated to CSP*, is presented in this chapter. CSP* is a distributed logic programming language for discrete event simulation.

This chapter has two sections. First, the basic style of programming in CSP* is introduced. The important concern is enhancing the expressiveness of the original logic programming model to include dynamic process creation and more built-in predicates. Second, examples are given to show how CSP* is used in distributed discrete event simulations.

6.1 Basic Constructs and Programming Style

CSP* is an extension of Prolog. It inherits most features of Prolog and provides a process-oriented programming environment to its users. The major feature of CSP* is that a CSP* program consists of a set of dynamic processes which act as autonomous simulation objects, cooperate through communications, and are synchronized by their simulation times.

Execution of a CSP* program relies on a set of logic process interpreters which evaluate logic processes in parallel and permit backtracking within processes to be combined with concurrent activities among processes.

6.1.1 Syntax and Semantics

According to the distributed logic programming model in Chapter 3, a CSP* program consists of two types of clauses: *procedure* clauses and *process* clauses.

However, the definitions of these clauses differ from the original ones in dynamic process creation. To illustrate this, we use symbol A to represent procedure literals, symbol P to represent process literals, and symbol R to indicate either procedure literals or process literals. We have the following syntactic definitions (a more detailed syntax for CSP* in Extended Backus-Naur Form can be found in Appendix):

a procedure clause: $A : -R_1, R_2, \dots, R_n.$

a process clause: $P(\text{Name}, \dots) :: -R_1, R_2, \dots, R_n.$

a query: $: -R_1, R_2, \dots, R_k?$

The procedural reading of the procedure clause is “ A is solvable if all procedures represented by procedure literals in R_1, \dots, R_n are solvable, and all process instances represented by process literals in R_1, \dots, R_n are successful”.

A process instance is a copy of a logic process. A logic process may have a number of instances which are evaluated by logic process interpreters in parallel. Therefore, the procedural reading of the process clause is “an instance of $P(\text{Name}, \dots)$ is successful if all procedures represented by procedure literals in R_1, \dots, R_n are solvable, and all process instances represented by process literals in R_1, \dots, R_n are successful”.

Finally, the procedural reading of the query is “show all procedures represented by procedural literals in R_1, \dots, R_k are solvable and all process instances represented by process literals in R_1, \dots, R_k are successful”.

It is important to note that all clauses in CSP* are temporal clauses. They

basically follow the temporal property discussed in Chapter 3. However, as CSP* introduces dynamic process creation, the body of a temporal clause may involve both procedure literals and process literals. For example, a clause in CSP* may be:

$$p(\text{Name}) :: -a, q1(\text{Name1}), b, q2(\text{Name2}), c.$$

where $p(\text{Name})$, $q1(\text{Name1})$ and $q2(\text{Name2})$ are process literals and a , b and c are procedural literals. The temporal property of the example is explained as follows:

$$\begin{aligned} p(T_0, T_{out}, \text{Name}) :: & -a(T_0, T_1), q1(T_1, T_{out1}, \text{Name1}), b(T_1, T_2), \\ & q2(T_2, T_{out2}, \text{Name2}), c(T_2, T_{out}). \end{aligned}$$

More detailed specification about process creation and creation time will be discussed in the next section.

6.1.2 Process Naming, Creation and Destruction

The first parameter in the head of a process clause always refers to the name of a process instance. A process name can be any meaningful term and must be instantiated by a unique ground term when the process is instanced. Thereby the process name can be used as the identifier of the process instance in communications. If a process clause contributes only one instance, its name can be defined as a constant. However, if a process clause is used to create a number of instances, its name must be a variable or a function term with variable arguments, so that each instance can bind the name with a different value.

In the evaluation of a goal $g(n1, n2, \dots, nl)$, a match with the goal is tried with the head of each clause in a program. If a match is found and the matched clause is

a process clause, CSP* creates a new logic process interpreter to evaluate a copy of the process clause. We say a new process instance (or a new process) with name $n1$ is created. If $n1$ is not a ground term, it results in a run time error. A newly created process executes concurrently with other existing processes and it sequentially evaluates the goals in the tail of the process clause. On the other hand, if the matched clause is a procedure clause, the goal is evaluated as a normal Prolog procedure call. An important rule in CSP* is that a goal which matches a process clause cannot share any non-ground variables with other goals in conjunction.

A CSP* program is started by a normal query

$$: -R_1, R_2, \dots, R_k?$$

which constitutes itself as the root process with system-designated name *main*. As the *main* process sequentially evaluates the goal sequence R_1, \dots, R_k , new processes may be spawned.

For example, if we want to create a network with B-tree structure, as shown in Figure 6.1, different methods can be adopted.

Solution 1:

```
:- node(n(1)), node(n(2)), ..., node(n(15))?
```

```
node(n(I)) :- /* definition of node */.
```

Solution 2:

```
:- create(15)?
```

```
create(0).
```

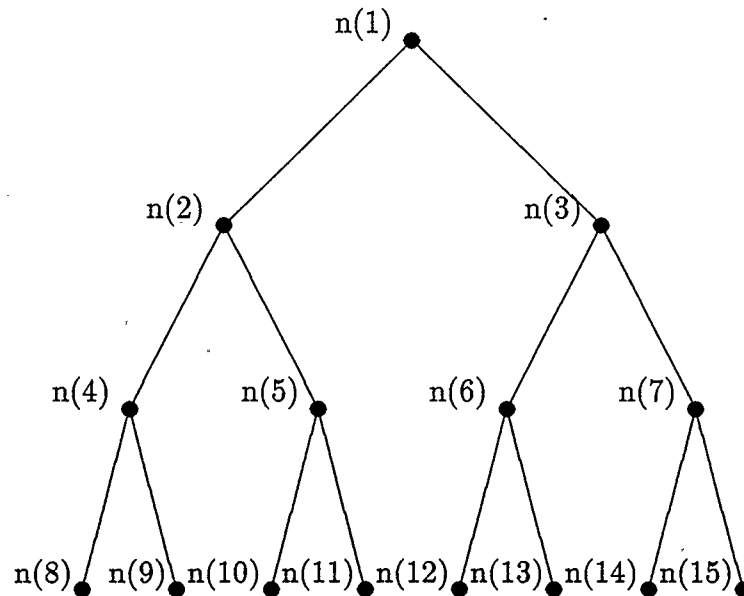


Figure 6.1: A B-tree Network

```

create(I) :-
    I > 0,
    node(n(I)),
    I1 is I-1,
    create(I1).

```

```

node(n(I)) :- /* definition of node */.

```

Solution 3:

```

:- node(n(1), 4)?

```

```

node(n(I), Layer) :- create(I, Layer),
    /* definition of node */.

```

```

create(_, 1).
create(I, L) :-
    L > 1,

```

```

Ln is I*2,
Rn is I*2+1,
L1 is L-1,
node(n(Ln), L1),
node(n(Rn), L1).

```

The main process creates 15 node processes, one by one, in **solution 1**, because each goal in the query matches the head of the process clause $node(n(I))$. Similarly, **solution 2** produces the same number of node processes in a reversed order by recursive procedure calls. In **solution 3**, the *main* process only creates process $n(1)$ and then, process $n(1)$ creates $n(2)$ and $n(3)$, process $n(2)$ creates $n(4)$ and $n(5)$, and so on; finally, fifteen processes are spawned dynamically. It is easy to see that the third solution takes less time, because node processes are spawned in parallel.

The parent of a process is its creator. A process can be destroyed by either its parent or itself. Suppose process *father* creates process *son* by evaluating goal $g(son, \dots)$, there are two possible situations: (1) some time later process *father* backtracks to $g(son, \dots)$; or (2) process *son* fails. No matter which situation happens, process *son* is killed and process *father* tries to match $g(son, \dots)$ with an alternative clause (if any) in the program. If a match is found, a new *son* will be created; otherwise, *father* continues backtracking and leaves the system with no knowledge that a *son* process had ever existed.

6.1.3 Built-in Predicates

The standard Prolog provides a set of built-in predicates that are essential to make Prolog a practical language. These predicates can be divided into two classes: *meta-logical* predicates and *extra-logical* predicates.

Meta-logical predicates are outside of the scope of first-order logic but do not cause any side effects in their evaluations. In general, they are used for type checking, term comparison, and data conversion.

CSP* inherits all *meta-logical* predicates from standard Prolog and provides a few new *meta-logical* predicates for handling time and communication.

A simulation program involves a description of the way in which a system state changes over time. Simulation time is fundamental to determining the order in which events occur. In a CSP* program, each process has its own view of simulation time, *i.e.*, it holds a read-only clock which denotes its progress in computation.

The initial simulation time of a process is defined to be the simulation time of its parent at which it is created. The initial simulation time of the *main* process is zero. By calling the built-in predicate

$$\text{time}(T),$$

a process obtains the value of its current simulation time.

An event is scheduled by evaluating a predicate

$$\text{send}(D, M, T)$$

where *D* is the event receiver's name, *M* is the event information and *T* is a non-negative delay interval of simulation time which indicates that the receiver must receive the event at the *receive time* which is defined as *T* plus the time the event is scheduled. If *T* is absent from the predicate, say *send(D, M)*, *T* is assumed to be zero. The parameters of a *send* predicate must be instantiated to ground terms when it is called.

Predicate *receive* is used for receiving an event. CSP* guarantees that events delivered to a process are in nondecreasing receive time order. The call

`receive(S, M)`

tries to unify its arguments with the event delivered by the CSP* system, that is, S is unified with the sender's name and M with the event information. If a unification is made, the event is consumed and the simulation clock of the receiver is advanced to the receive time specified by the event.

Predicates *self(P)* and *parent(P)* are used to get the caller's name and parent name respectively. Therefore, these names can be used to direct communications. For example, a process can advance its simulation clock by calling the following procedure:

```
advance(T) :-
    self(P),
    send(P, null, T),
    receive(P, null).
```

It is important to note that predicate *advance(T)* is deterministic. Therefore, if a process at simulation time 100 calls *advance(5)*, the call succeeds only if there are no events in between 100 and 105.

Extra-logical predicates in the standard Prolog not only violate first-order logic, but also achieve side effects in the course of being satisfied as logical goals. They are usually used for I/O operation and program manipulation.

CSP* adopts different policies to treat these predicates. Some of them are no longer available in CSP*; and some of them are implemented in different ways from Prolog, so that their side effects are removed on backtracking.

For example, predicates *assert* and *retract* are used to add or remove a clause from a program database in the standard Prolog. The major purpose of the predicates is to use the database as a medium to remember partially computed results for frequently used objects. A good example of this requirement is a random number generator. Commonly, whenever a random number procedure is called, it *retracts* an old *seed* from the database as input for generating the next random number and then *asserts* a new *seed* into the database. This is because standard Prolog has no global variables to remember partially computed results. Without *assert* and *retract* predicates, a program has to carry all partially computed results, *e.g.*, the *seed*, through subsequent procedure calls. This would be extremely inconvenient in writing a large program. However, CSP* can implement every object, such as a random number generator, as a logic process which hides all its intermediate state as well as partial results from others. Thus, *assert* and *retract* are not inherited by CSP*.

Predicate *repeat* in the standard Prolog is used to simulate repeat loops in conventional languages. These loops are useful only when used in conjunction with extra-logical predicates which cause side effects. Since the side effects of most system predicates are removed in CSP*, *repeat* is no longer useful.

In the CSP* system, input and output functions are implemented by I/O processes with each manipulating an input/output stream. Therefore, I/O operations are carried out through communications between logic processes and I/O processes. As a consequence, I/O predicates become sequences of communication predicates. The implementation details of I/O processes are not discussed in this thesis.

Another useful but dangerous feature in the standard Prolog is *cut*, which is usually written as “!”. For the sake of efficiency, *cut* can be used to control backtracking

by pruning the search space of a program. However, using a *cut* may destroy the correspondence between the declarative and procedural meaning of a logical relation. *Cut* is also allowed in CSP* programs, but the user should be aware that backtracking on a *cut* may invoke a global backtracking if communications or process creations are involved in the effective range of the *cut*. If this happens, the *cut* not only prunes the search space of the process in which it resides, but also possibly prunes the search spaces of other processes. Therefore, it is strongly suggested to use *cut* with care and not to use it without reason.

6.1.4 Pragmatics

The pragmatics of logic programming concern efficiency. Typical aspects of efficiency are execution time and memory space requirements of a program.

As we mentioned before, CSP* can be used to describe both deterministic and nondeterministic computations. We hope that speedup can be achieved for both these computations. However, for a distributed logic program written in CSP*, the first important factor which has a great influence upon the execution time is the degree of determinacy of the program.

If a deterministic model is evenly decomposed into n concurrent processes, by running each process on a different processor, then it would be possible to achieve an optimal *n-fold* speedup over the sequential case.

On the other hand, if a nondeterministic model (here we refer to the “don’t know” nondeterminism) is decomposed into a set of concurrent processes, although these processes are executed on different processors, it is possible that CSP* provides very little, or even no speedup as compared with sequential execution.

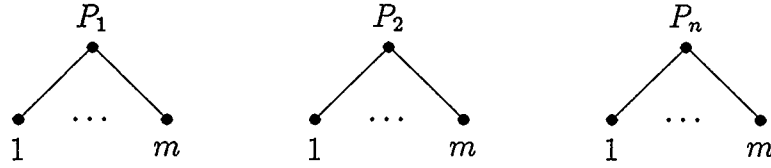


Figure 6.2: A Homogenous Nondeterministic Computation Model

Let us consider a simple homogeneous nondeterministic computation model. Suppose the model is decomposed into n processes each with m nondeterministic branches, as shown in Figure 6.2. The worst case for this model is that the only solution of “: $-P_1, \dots, P_n$?” is the combination of the rightmost branches of these processes.

If a unit time is taken on each branch and the interprocess communication time is assumed to be zero, then execution time on CSP* with n processors and on a standard Prolog system with single processor, respectively, is given by:

$$\begin{array}{ll} \text{Prolog:} & \sum_{i=1}^{i=n} m^i \\ \text{CSP*} & m^n \end{array}$$

and we have

$$2 * m^n \geq \sum_{i=1}^{i=n} m^i \geq m^n.$$

Though n processors are used to solve the nondeterministic model, the speedup is less than two.

This analysis tells us the important fact that for running a program on an AND-parallel system (such as CSP*), the more determinism the program model possesses, the more speedup the system could achieve. This is because a deterministic computa-

tion does not involve any global backtracking while a nondeterministic computation must try possible branch combinations of processes in order to find a solution.

The second influence on execution time of a distributed logic program in CSP* is caused by the asymmetry of the built-in communication predicates. We illustrate this by comparing two simple examples:

EX1:

```
p1(lp1) :- foo.
    foo :- send(lp2, a, 0).
    foo :- send(lp2, b, 0).
    foo :- send(lp2, c, 0).
p2(lp2) :- receive(lp1, c).
```

EX2:

```
p1(lp1) :- foo.
    foo :- receive(lp2, a, 0).
    foo :- receive(lp2, b, 0).
    foo :- receive(lp2, c, 0).
p2(lp2) :- send(lp1, c).
```

To be able to find a solution of “:- p1(lp1), p2(lp2)”, **EX1** needs five messages which include three normal messages passed from lp1 to lp2 and two objection messages transferred from lp2 to lp1 while **EX2** only needs one normal message. Therefore, the organization of message producers and consumers in a nondeterministic computation becomes critical in improving the efficiency of a program.

Space requirements of a program are also closely related to the determinism of the program model. It is clear that the execution states of a nondeterministic program must be saved to enable backtracking. If a simulation program runs a long time,

memory overflow may happen during execution. On the other hand, the execution states of a deterministic program can be collected as garbage and can be used to fulfill future memory requirements. Since most programs are mixed with deterministic and nondeterministic computation phrases and the CSP* system is not smart enough to figure out when any process reaches a deterministic state, CSP* provides a special predicate for garbage collection purpose.

Predicate *commit*, denoted by “!”, is used to prune the search space of a logical process. The precise semantics of *commit* are as follows:

Let us call the initial goal the goal that started the process. When a *commit* is encountered as a goal it succeeds immediately, but it commits the system to all choices made between the time the initial goal was invoked and the time the *commit* was encountered. All the remaining alternatives before the *commit* are discarded.

In other words, the evaluation of a *commit* divides the computation of a process into two parts: the computation before it becomes the deterministic part and the computation after it is the nondeterministic part. Once a process has progressed enough to determine that this is the only way to find a solution, a *commit* can be inserted. Therefore, backtracking is only allowed in the latter part and the garbage in the former part can be collected by the system. The difference between a “!” and a “!!” is that the former discards all remaining alternatives between its parent goal (the goal that matched the head of a clause containing the cut) and the “!”, the later discards all remaining alternatives between the initial goal and the “!!”.

6.2 Discrete Event Simulation in CSP*

Discrete event systems generally involve contention for scarce resources, with queues developing where system components must wait for resources to become available. Further, delays between state changes are usually determined statistically, with the exact interval selected according to some random number distribution. The objects for manipulating resources, queues, random numbers and other useful abstract data structures are usually called simulation facilities.

In this section, the implementation of simulation facilities in CSP* is discussed. We describe some of the facilities in depth because the others could be specified in similar ways. Furthermore, we use three examples to illustrate how to use these simulation facilities, how to decompose a simulation model into logic processes and how to describe deterministic and nondeterministic computations.

6.2.1 A Resource Allocation Process

Queueing systems are common components of discrete event simulation. Typically, a collection of demands for resources arise as time evolves. In general, there exist two kinds of resources. Resources which are used by requestors are *passive* resources. Resources which provide services are *active* resources.

For example, in a gas station, a customer who asks for service for his car needs an *active* resource - a worker, because they are going to exchange messages such as the type of service, the amount of payment, *etc.* On the other hand, the worker may need one or several *passive* resources - lift, lubricating gun or other tools - to finish his job.

Another way to distinguish these two kinds of resources is from the style in which their simulation time advances. The simulation time of a *passive* resource is scheduled only by its users while the simulation time of an *active* resource can be advanced both by its requestor and by itself. In the above example, the customer does not know in advance how long the service will take, but the server knows exactly how long it must hold a lubricating gun.

The following description defines a logic process which manipulates a collection of *passive* resources. It is an abstract data object and provides a deterministic interface to its users.

```

/* resource process interface */
(1) acquire(Rname, Num) :-
    send(Rname, acquire(Num)),
    receive(Rname, ok).

(2) release(Rname, Num) :-
    send(Rname, release(Num)).

/* resource process implementation */
(3) resource(Rname, Num) ::-
    resource(Num, Q, Q).

(4) resource(N, QH, QT) :-
    receive(Who, M), !!,
    resource(Who, M, N, QH, QT).

(5) resource(-, -, -).                                % terminate when no message

(6) resource(Who, acquire(Num), N, [H|QH], QT) :-
    var(H),
    Num ≤ N,
    N1 is N - Num,
```

- send(Who, ok),
resource(N1, [H|QH], QT).
- (7) resource(Who, acquire(Num), N, QH, [[Who, Num]|QT]) :-
resource(N, QH, QT).
- (8) resource(–, release(Num), N, [[Who, Req]|QH], QT) :-
nonvar(Who),
Req \leq N + Num,
N1 is N + Num - Req,
send(Who, ok),
resource(–, release(N1), 0, QH, QT).
- (9) resource(–, release(Num), N, QH, QT) :-
N1 is N + Num,
resource(N1, QH, QT).

In the above description, clauses (1) and (2) are used for *encapsulation* – hiding the implementation of the *resource* while exhibiting its interface to other processes. Clause (3) defines the resource process. An instance of the resource process must have a unique name and a fixed amount of resources.

By calling the procedure *resource(Num, Q, Q)*, a resource instance starts its life cycle. Its state is described by the current available amount of resources and a waiting queue with two pointers to the queue head and tail respectively. If there is no request, clause (5) terminates the process. Otherwise, clause (4) transfers the state of the process and the incoming request to a further procedure call.

If a requestor acquires a number of resources, clause (6) satisfies the request if there is no one in the waiting queue and the current available resources are enough to fulfill the requirements, otherwise, clause (7) appends the requestor to the waiting

queue. On the other hand, if a requestor releases a number of resources, clause (8) sets requestors in the waiting queue free if the total amount of resources (the amount just released plus the amount originally left) satisfies their demands. When clause 9 is chosen, the process will progress to next receive-service cycle with a changing state.

Now, suppose that a resource process has been created with a name “lub_gun”, a worker can call the following procedures to use a lubricating gun for five time units:

```
...
acquire(lub_gun, 1),
advance(5),
release(lub_gun, 1),
...
```

6.2.2 A Queue Process

A queue is used as an interface between a set of *active* resources (servers) and a set of resource users (customers). Different queue disciplines – the sequencing rules which determine which customer in a queue will be served next – can be implemented via different queue processes. In this section, we introduce a deterministic FIFO queue process written in CSP*.

In general, three operations are provided for manipulating a queue object. When a customer process executes an *enqueue*(*Qname*, *Server*, *Req*) operation, it is suspended in the queue referenced by *Qname* until a server process becomes available by a dequeue operation and the variable *Server* is bound to the server’s name. A similar behavior holds in the evaluation of a *dequeue*(*Qname*, *Customer*, *Req*) operation on

the server side. In addition, procedure *q_length* can be called by either customer or server processes which always returns the exact queue length at the simulation time the call is made.

```
/* FIFO queue process interface */
```

```
enqueue(Qname, Server, Req) :-  
    send(Qname, enqueue(Req)),  
    receive(Qname, Server).
```

```
dequeue(Qname, Customer, Req) :-  
    send(Qname, dequeue),  
    receive(Qname, [Customer, Req]).
```

```
q_length(Qname, Length) :-  
    send(Qname, length),  
    receive(Qname, Length).
```

```
/* FIFO queue process implementation */
```

```
queue(Qname):-  
    queue(EQ, EQ, DQ, DQ).
```

```
queue(EH, ET, DH, DT):-  
    receive(Who, M), !,  
    queue(Who, M, EH, ET, DH, DT).
```

```
queue(-, -, -, -). % terminate when no message
```

```
queue(Who, enqueue(Req), EH, [[Who, Req]|ET], [H|DH], DT):-  
    var(H), !,  
    queue(EH, ET, [H|DH], DT).
```

```
queue(Who, enqueue(Req), EH, ET, [H|DH], DT):-  
    send(H, [Who, Req]),  
    send(Who, H),  
    queue(EH, ET, DH, DT).
```

```

queue(Who, dequeue, [H|EH], ET, DH, [Who|DT]):-
    var(H), !,
    queue([H|EH], ET, DH, DT).

queue(Who, dequeue, [[H, Req]|EH], ET, DH, DT):-
    send(Who, [H, Req]),
    send(H, Who),
    queue(EH, ET, DH, DT).

queue(Who, length, EH, ET, DH, DT):-
    length(EH, N),
    send(Who, N),
    queue(EH, ET, DH, DT).

length([H|T], 0):- var(H), !.

length([H|T], N):-
    length(T, N1),
    N is N1+1.

```

The state of a queue instance is described by two inner queues (lists), one which delays customers that try to get service when all servers are busy and another which delays servers that try to grab customers from an empty queue.

From a closer observation of the queue process, we can find the following interesting properties:

1. it has an unbounded size;
2. it accepts requests from any number of customer/server processes;
3. it terminates automatically when there are no more requests (Note that Time Warp mechanism will roll a terminated queue process back whenever a new message arrives);

4. it not only synchronizes pairs of customer-server processes with respect to their simulation time, but also offers a message communication between each pair;
5. it provides a generic type of queue item, that is, the customers and the servers which access the queue can be any type of object (process), and the messages between customer-server processes can be any type of data structure.

With a slight modification, the queue process can be used between pairs of producer-consumer processes. In this case, a producer process does not suspend itself in the queue, instead, after adding an event into the queue it continues to generate the next event. On the other side, a consumer process is like a server, it dequeues an event if the queue is not empty, otherwise, it is blocked.

6.2.3 Random Number Generator

Simulation models usually contain random behavior. The purpose of random number generation is to provide a stream of numbers with specific statistical properties.

CSP* can be used to describe random number generators for different distributions in two ways: a distribution procedure or a distribution process. A distribution procedure generates random numbers according to the type of distribution and the arguments passed to the procedure. A distribution process has a common interface *sample(Rname, Next)* to return the next random number. The usage of these two methods depends on the application.

As an example, let us consider a simple, basic, random number distribution - the uniform distribution. We use the multiplicative congruential method which generates

random number n_{i+1} from the previous random number n_i by the equation

$$n_{i+1} := 8192 * n_i \bmod 67099547$$

such that the cycle length is 67099546 [Bir79].

```

/* uniform distribution procedure */
uniform(Seed, Next):-
    Next is (8192 * Seed) mod 67099547.

/* distribution process interface */
sample(Rname, Next):-
    send(Rname, next),
    receive(Rname, Next).

/* uniform distribution process */
uniform_process(Rname):-
    genunif(12345678).

genunif(Seed):-
    receive(Who, next), !!,
    uniform(Seed, Next),
    send(Who, Next),
    genunif(Next).

genunif(_).                                % terminate when no message

```

6.2.4 Single Server Queueing Model

Figure 6.3 shows a typical single server queueing simulation model. Suppose the model is applied to a bank system. Each arriving customer generates a successor and waits in a queue with a randomly selected request until the bank server is

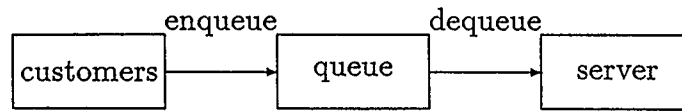


Figure 6.3: A Single Server Queueing Model

available. The server process handles different requests from customers, such as deposit, withdraw, *etc.*.

As soon as a customer receives a result from the server, it leaves the system immediately. On the other hand, the server stops service only if there are no more customers in the queue.

In the following program, service requests are uniformly distributed random integers from 1 to 5; inter-arrival times are negative exponentially distributed random numbers with arrival-rate 0.125; and service times are constant random numbers.

```
/* main process */
```

```
single_server(Startc, Starts, End):-
```

```
    negexp_process(nexp, 0.125),
```

```
% create a negexp process
```

```
    ranint_process(rint, 1, 5),
```

```
% create a ranint process
```

```
    queue(que),
```

```
% create a queue process
```

```
    server(server, Starts),
```

```
% create a server process
```

```
    customer(c(1), Startc, End),
```

```
% create the first customer process
```

```
    !!.
```

```
% and terminate
```

```
/* customer process */
```

```
customer(c(I), Start, End):-
```

```
    sample(nexp, Next),
```

```
% get inter-arrival time
```

```
    Start1 is Start + Next,
```

```

        generate_next(I, Start1, End),                % create next customer
        advance(Start),                               % start execution
        sample(rint, Req),                             % get a request
        enqueue(que, Server, Req),                    % wait for server
        receive(Server, Result),                       % wait for result
        !!.                                           % and terminate

generate_next(_, Start, End):-
    Start ≥ End.                                     % stop generating

generate_next(I, Start, End):-
    Start < End,
    I1 is I+1,
    customer(c(I1), Start, End).

/* server process */

server(Sname, Start):-
    advance(Start),
    service.

service:-
    dequeue(que, Customer, Req),                    % dequeue a customer with a request
    process_request(Req, Result),                   % serve the request
    send(Customer, Result), !!,                     % send the result back
    service.                                         % loop

service.                                           % terminate when dequeue fails

process_request(1, open_account):- advance(10).
process_request(2, deposit):- advance(5).
process_request(3, withdraw):- advance(5).
process_request(4, cash_check):- advance(7).
process_request(5, credit_bill):-advance(15).

```

Suppose customers arrive from 8:30 until 15:00, the bank server starts serving from 9:00 until the last customer is served, and the unit of simulation time is one minute, the program can be invoked by

```
:-single_server(510, 540, 900)?
```

Note that the number of customer processes is controlled by the inter-arrival rate and the model termination time.

With a slight modification, the above program can be used to simulate multi-server queuing models. For example, the clause

```
three_server(Startc, Starts, End):-
    negexp_process(nexp, 0.125),
    ranint_process(rint, 1, 5),
    queue(que),
    server(s(1), Starts),
    server(s(2), Starts),
    server(s(3), Starts),
    customer(c(1), Startc, End),
    !!.
```

and the query

```
:-three_server(510, 540, 900)?
```

can simulate a 3-server queuing model.

6.2.5 Bank Robbery

In order to illustrate how CSP* is used in nondeterministic computations, we take a simple but interesting example from [FS82]. Jim and Dick want to rob the Prolog

savings bank. Jim needs 5 minutes to climb into the bank. Dick waits outside and sends a tool to Jim when he hears a whistle from Jim. There are different safes in the bank and each safe takes a different time to be unlocked. If the robbery has to finish in 25 minutes, the question is which safe is to be chosen for a successful robbery.

```
/* process jim */
```

```
process_jim(jim):-
```

```
    advance(5),
```

```
% climb into bank
```

```
    send(dick, whistle),
```

```
    receive(dick, Safe),
```

```
    open(Safe),
```

```
    time(T),
```

```
    T ≤ 25.
```

```
open(milner):- advance(40).
```

```
open(wertheim):- advance(27).
```

```
open(chatwood):- advance(10).
```

```
/* process dick */
```

```
process_dick(dick):-
```

```
    receive(jim, whistle),
```

```
    has_tool(Safe),
```

```
    send(jim, Safe).
```

```
has_tool(milner).
```

```
has_tool(chatwood).
```

When we call

```
:- process_jim(jim), process_dick(dick)?
```

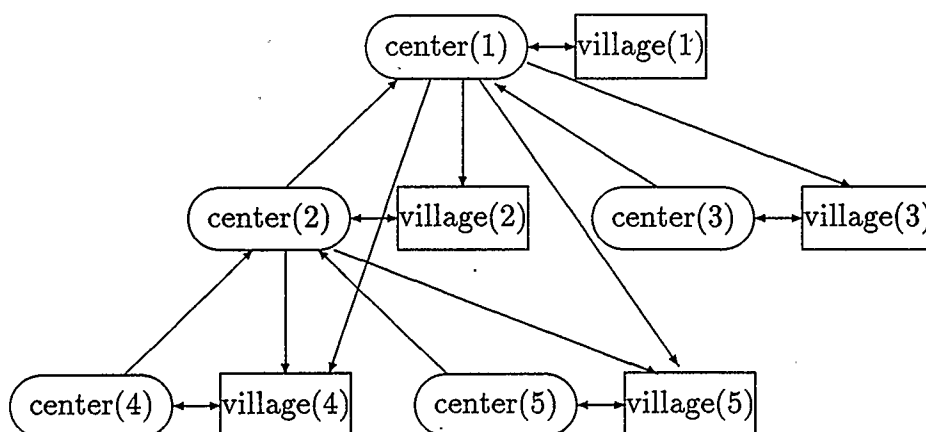


Figure 6.4: A Health Care Model

the simulation tries to find the correct safe and tool by backtracking and attempting different safes with different opening times until one is found that can be opened in the time available.

6.2.6 Hierarchical Health Care System

Figure 6.4 is a model of health care systems, typical of health delivery systems, in developing countries. There are three kinds of objects in the model: the villages, the reception nurses and doctors in health centers.

A village process is designed as an event generator. It periodically generates patient events, sends them to the corresponding health center and receives the treated people back. In between any two patient events, it is possible that a treated patient has been sent back. Therefore, the village process adopts a very interesting programming technique: it sends a message which indicates the next patient event to itself, then the process executes *receive*; if the message received is the message of the next

event, the process actually generates the event and schedules the second next event for itself, otherwise, the process saves the message which must represent a treated patient.

We assume that there is one reception nurse in each health center. A reception nurse process is implemented by a producer-consumer queue (see Section 6.2.2). It receives the incoming patient events and distributes them to available doctors. The interface of a reception nurse process now is defined as *enqueue(Name, Event)* and *dequeue(Name, Event)*.

There are a number of doctors in every health center. A doctor process gets a patient event from the nurse each time. If the patient is treatable, then after the treatment, the person is sent back to his (her) village, otherwise, the person is sent to a higher health center. A doctor process terminates its recursive processing only if there are no more patient events.

```
/* main process */
```

```
health_system(St, Et, DocList):-
```

```
    negexp_process(nexp, 0.25),
```

```
% neg-exp distribution
```

```
    unif_process(unif1, 0, 1),
```

```
% 0-1 uniform distribution
```

```
    unif_process(unif2, 10, 50),
```

```
% 10-50 uniform distribution
```

```
    creation(5, St, Et, DocList).
```

```
% create the others
```

```
creation(0, -, -, -):-
```

```
    !!.
```

```
% terminate
```

```
creation(I, St, Et, [Docs|DL]):-
```

```
    I > 0,
```

```
    center(c(I), St, Docs),
```

```
% create Ith health center
```

```
    village(v(I), St, Et, Q),
```

```
% create Ith village process
```

```
    Il is I - 1,
```

```

        creation(I1, St, Et, DL).

/* village process */
village(v(I), St, Et, Q):-
    advance(St),
    sample(nexp, Next),                                % next event time
    send(v(I), 1, Next),                                % schedule the first event to self
    village_operation(I, Et, Q).

village_operation(I, Et, Q):-
    receive(Who, M),
    operation(Who, M, I, Et, Q).                        % segregate the message
village_operation(-, -, -).                            % terminate if no message
operation(v(I), -, I, Et, Q):-                          % message from self
    time(T),
    T ≥ Et, !,                                          % stop generating
    village_operation(I, Et, Q).

operation(v(I), Pn, I, Et, Q):-                          % message from self
    enqueue(c(I), [I, Pn]),                            % Pn'th patient enters center
    sample(nexp, Next),
    Pn1 is Pn + 1,
    send(v(I), Pn1, Next),                              % schedule next event
    village_operation(I, Et, Q).

operation(D, Pn, I, Et, [[D,Pn]|T]):-                  % receive a treated patient
    village_operation(I, Et, T).

/* a health center consists of a queue process and a number of doctor
   processes */
center(c(I), St, Docs):-
    create_doctors(Docs, St, I),
    queue(c(I)).

```

```

create_doctors(0, -, -).
create_doctors(Docs, St, I):-
    Docs > 0,
    doctor(d(I, Docs), St),
    Docs1 is Docs - 1,
    create_doctors(Docs1, St, I).

/* doctor process */
doctor(d(I, J), St):-
    advance(St),
    diagnose(I).

diagnose(I):-
    dequeue(c(I), Patient), !!,                % receive a patient
    advance(2),                                % assessment
    sample(unif1, T),                          % diagnostic result
    treat(I, Patient, T).

diagnose(_).                                % terminate if no more patients

treat(I, [In, Pn], T):-
    treatable(I, T),                            % if the patient is treatable
    sample(unif2, TreatTime),                  % get treat time
    advance(TreatTime),
    send(v(In), Pn),                          % send the patient back
    diagnose(I).

treat(I, Patient, _):-
    parent_center(I1, I, T),                  % not locally treatable
    enqueue(c(I1), Patient),                  % transfer to a parent center
    diagnose(I).

parent_center(1, 2).
parent_center(1, 3).

```

```

parent_center(2, 4).
parent_center(2, 5).
treatable(1, _).                % all treatable in c(0)
treatable(2, T):- T≤0.75.        % 0.75 treatable in c(1), c(2)
treatable(3, T):- T≤0.75.
treatable(_, T):- T≤0.5.        % 0.50 treatable in c(3), c(4)

```

This program can be invoked by using

```
:- health_system(100, 1000, [5, 5, 10, 10, 20])?
```

which will create three random number generation processes, five village processes, five receptionist (queue) processes, and fifty doctor processes.

6.3 Summary

This chapter has described a practical distributed logic programming language, CSP*, for discrete event simulation. CSP* is an extension of standard Prolog and provides a process-oriented programming environment to users.

A CSP* program consists a set of procedure clauses and process clauses. A process clause defines a logic process which serves as a template for creating process instances while a procedure clause retains the same semantics as in Prolog. A process instance is created dynamically. It has a unique symbolic name and is evaluated sequentially by a logic process interpreter.

During execution, a process instance may communicate with other process instances through message passing. This is accomplished by calling built-in commu-

nication predicates. CSP* allows backtracking within processes to be coordinated with concurrent activities among processes.

Programming examples in this chapter reveal the simplicity and expressive power of CSP*. An experimental, deterministic version of CSP* has been built in a distributed programming environment Jade [XUC⁺86, UB CD86]. All the examples in this chapter, except the “Bank Robbery”, have been tested.

Chapter 7

CONCLUSIONS AND DIRECTIONS FOR FUTURE WORK

In this chapter, we review and summarize the contributions of this thesis, discuss the advantages and disadvantages of the proposed logic programming system, and suggest directions for future work.

7.1 Conclusions

Practical simulation work involves defining a problem and the goals of an experiment, specifying a model which represents enough detail to achieve goals, implementing the model as a working computer program, verifying the consistency of the model and the problem definition, validating the consistency of the program and the model, experimenting with the program, and producing documentation.

Since many simulation models are both large and complex, in many research areas, expensive computers and human resources are devoted exclusively to simulations. Therefore, two important research directions are to reduce the costs of simulation development and to speed up the execution of simulations. The objective of this thesis is to design a programming system that makes simulation easier, cheaper and faster.

Different proposals for distributed simulations were compared in Chapter 2 which suggested that a distributed logic programming system in conjunction with a run-

time kernel based on the Time Warp mechanism provides a basis for achieving our goal.

The first contribution of the thesis is a distributed logic programming model. In the model, a distributed logic program is represented by a set of logic processes with which the user can specify different components in a simulation model. The cooperations among logic processes are accomplished by explicit inter-process communication.

A very simple syntax and a well-defined semantics for the logic programming model were presented. We believe that the simpler the syntactical form and the better defined the semantics of the language framework, the easier the simulation task will be.

The second contribution of the thesis is the design of a practical implementation of the distributed logic programming system. Based on Jefferson's Time Warp mechanism and standard Prolog interpreter, a kernel and a logic process interpreter which provide functions to handle the cooperation of a logic process with other processes were presented.

The mechanism to deal with the failures on *virtual time* is the *rollback* facility; the mechanism to deal with the failures on *virtual space* is the *global backtracking* facility. Since the algorithms of these facilities utilize the built-in state-saving and local backtracking capabilities of the standard Prolog, we simplify the system implementation and overcome the non-knowledgeable state-saving problem in the original Time Warp proposal.

In addition, we proved that the global backtracking algorithm is sound and partial complete. The soundness and partial completeness results show that the system not

only provides a temporal coordinate system to measure computational progress and define synchronizations, but also provides a spatial coordinate system to support nondeterministic computations.

Finally, a distributed logic programming language proposal – Communication Sequential Prolog (CSP*) is presented. In addition to the features inherited from the standard Prolog, CSP* has new features to support distributed discrete event simulation and object-oriented programming. The major extensions are summerized as follows:

1. **Explicit, dynamic processes:** CSP* uses explicit processes to show the concurrency as well as the logic components of a simulation model. Processes can be created dynamically. The syntactic mechanism which support dynamic process creation is the concept of process instances represented by process literals in the bodies of clauses.
2. **Process naming and communication connection:** Each process instance of a logic program in CSP* has a unique symbolic name. Process names in a distributed logic program provide a global, user defined, flat naming space. They are used to direct communications among processes. CSP* offers arbitrary communication connections, that is, processes can communicate with each other provided they know their partner's names.
3. **Inter-process communication and synchronization:** CSP* provides an asynchronous communication mechanism. A process is free to send any number of messages to other processes without blocking. Synchronizations among processes are governed by the agreement of their simulation time as well as the

agreement of their search positions.

Interestingly, we find the origins of these features not from the declarative programming languages but from the procedural programming languages discussed in Section 2.1.1. This is because traditional logic programming techniques are not suitable for describing a changing world, while distributed discrete event simulation needs the facilities for specifying model dynamics, decomposing simulation components, and coordinating concurrent activities of these components.

On the other hand, based on the foundation of logic, CSP* has greater expressiveness than existing procedural programming languages. It provides a very simple syntax and well-defined semantics. It can be used both for model specification and model implementation. It can describe both deterministic and nondeterministic computations.

However, these achievements are not without costs. First, although CSP* is based on the theory of first order logic, the temporal construct and the evaluation order dilute the essential simplicity of pure logic programming, therefore making programmers pay more attention to procedural considerations. In other words, we can use CSP* as a specification language, but we must remember that it is only partially complete. The philosophy here is that it is better to have at least some declarative meaning rather than none, because a declaratively correct program (specification correct) makes it rather easy to become a procedurally correct program (implementation correct). For a large and complex simulation model, the proposed language framework minimizes the costs of developing, modifying and debugging the simulation program.

Secondly, since the *send* and *receive* predicates are treated asymmetrically, it is possible that different arrangements of these predicates in a nondeterministic computation will greatly influence the efficiency of the program (recall the example in Section 6.1.4). Thus the user has to consider pragmatic issues in implementing a simulation program.

Finally, CSP* does not provide security and protection for abstract data objects. For example, suppose we have created a queue process, other processes may communicate with the queue object directly without going through the specified interface.

To conclude, the major advantages of CSP* are the simplicity that comes from the distributed logic programming model, the flexibility that comes from dynamic process creation and a symbolic naming space, the concurrency that comes from asynchronous communication, the understandability that comes from the declarative meanings of programs, and the expressive power that supports concurrent activities, process synchronizations, message segregations and nondeterministic computations. Though there are disadvantages to CSP*, such as the need for the user to be aware of run-time efficiency issues (as described above), this language offers a potential tool for model specification and parallel execution.

7.2 Future Work

An efficient, complete CSP* system still needs to be implemented. Then performance of the system with a greater range of discrete event simulation applications can be tested and evaluated.

Problems left in the implementation of a practical CSP* system are how to specify

and implement I/O processes; how to allocate processes to different processors; how to debug a CSP* program; how to monitor the execution of a CSP* program; how to implement a compiler instead of an interpreter; what statistics facilities should be provided; and which simulation facilities can be standardized. All these problems are essential in building a usable simulation environment and should be carefully investigated.

When we measure the performance of a CSP* system, from what criteria can we compare the system performance with others? Of course, we can not only measure the performance at the stage of a program execution. As proposed both for model specification and implementation, CSP* should minimize the effort devoted to program development, debugging, testing and execution. The actual measurement should include the performance results on all stages of a simulation task. Therefore, we need a set of procedures to collect these results and a set of criteria to analyze the performance of the system.

Furthermore, it is still not clear what types of discrete event simulation models can be easily expressed and also achieve good performance in CSP*. Greater experimentation with different simulation models is required for analyzing the programming style, expressiveness, efficiency, theoretical considerations and practical implementation of CSP*.

Bibliography

- [AM87] M. Abadi and Z. Manna. Temporal logic programming. In *1987 Symposium on Logic Programming*, pages 4–16. IEEE, 1987.
- [BG84] K. Broda and S. Gregory. Parlog for discrete event simulation. Research Report DOC 84/5, University of London, March 1984.
- [Bir79] G. M. Birtwistle. *DEMOS a System for Discrete Event Modeling on Simula*. The Macmillan Press LTD., 1979.
- [CG86] K. Clark and S. Gregory. Parlog: parallel programming in logic. *ACM Tras. on programming languages and systems*, 8(1), January 1986.
- [CGU85] J. Cleary, K. Goh, and B. Unger. Distributed event simulation in prolog. In *AI, Graphics, and Simulation*, pages 8–13. SCS, 1985.
- [CHM79] K. M. Chandy, V. Holmes, and J. Misra. Distributed simulation of networks. *Computer Network*, 3(2), February 1979.
- [CM79] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, September 1979.
- [CM81] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11):198–206, April 1981.

- [Con87] J. S. Conery. *Parallel execution of logic programs*. KLUWER ACADEMIC PUBLISHERS, 1987.
- [Coo80] R. P. Cook. *mod - a language for distributed programming. *IEEE Software Eng.*, 6(6), November 1980.
- [CUL88] J. Cleary, B. Unger, and X. Li. A distributed and parallel backtracking algorithm using virtual time. In *Distributed Simulation*, pages 177–182. SCS, 1988.
- [DOD80] DOD. *Reference Manual for the Ada programming language*. United States DOD, 1980.
- [Fel79] J. A. Feldman. High level programming for distributed computing. *CACM*, 22, June 1979.
- [Fis78] G. S. Fishman. *Principles of Discrete event simulation*. Wiley Series on Systems Engineering and Analysis, 1978.
- [FS82] I. Futo and J. Szeredi. T-prolog: A very high level simulation system. Technical report, Computer Research Institute, H-1015 Budapest Donati u. 35-45, 1982.
- [Fut88] I. Futo. Distributed simulation on prolog basis. In *Distributed Simulation*, pages 160–165. SCS, 1988.
- [GLB85] M. P. Georgeff, A. L. Lansky, and P. Bessiere. A procedural logic. Research report, AI Center, SRI International, 1985.

- [Gol85] D. G. Golden. Software engineering considerations for the design of simulation languages. *SIMULATION*, pages 169–178, October 1985.
- [Han78] P. B. Hansen. Distributed processes: a concurrent programming concept. *CACM*, 21, November 1978.
- [HM84] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. In *The 3rd Annual ACM Symposium on Principles of Distributed Computing*, pages 50–61. ACM, August 1984.
- [Hoa78] C. A. R. Hoare. Communicating sequential processes. *CACM*, 21, August 1978.
- [Hoa82] C. A. R. Hoare. Specifications, programs and implementations. Technical report, Programming Research Group, Oxford University, 1982.
- [IBM77] IBM Corporation. *General Purpose Simulation System V User's Manual*. IBM Corporation, White, Plains, N. Y., 1977.
- [Jef85] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.
- [JS82] D. Jefferson and H. Sowizral. Fast concurrent simulation using the time warp mechanism, part i: Local control. Technique report, The Rand Corporation, December 1982.
- [Kes81] J. L. W. Kessels. The soma: a programming construct for distributed processing. *IEEE Software Eng.*, 7(5), September 1981.

- [KH85] W. H. Kaubisch and C. A. R. Hoare. Discrete event simulation based on communicating sequential processes. Technique Report CR 4.22, S.65, The Queen's University, 1985.
- [KTMB86] K. Kahn, E. D. Tribble, M. S. Miller, and D. G. Bobrow. Vulcan: Logical concurrent objects. Technical report, Xerox Palo Alto Research Center, 1986.
- [Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7), July 1978.
- [Lis79] B. Liskov. Primitives for distributed computing. In *Proc. of the 7th symposium on operating system principle*, 1979.
- [Llo84] J. W. Lloyd. *Programming in Logic*. Springer Verlag, 1984.
- [PWM79] J. K. Peacock, J. W. Wong, and E. Manning. Distributed simulation using a network of processors. *Computer Networks*, 3(1):44–56, February 1979.
- [Sha83] E.Y. Shapiro. A subset of concurrent prolog and its interpreter. Technique Report TR-003, ICOT, February 1983.
- [Sil81] A. Silberschatz. A note on the distributed program component cell. *ACM SIGPLAN NOTICES*, 16(7), July 1981.
- [SL87] Z. Sun and X. Li. Csm: A distributed programming language. *IEEE Software Eng.*, 13(4), April 1987.

- [SS86] L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, 1986.
- [SY85] R. Strom and S. Yemini. The nil distributed systems programming language: a status report. *ACM SIGPLAN Notices*, 20(5), May 1985.
- [UBCD86] B. Unger, G. Birtwistle, J. Cleary, and A. Dewar. A distributed software prototyping and simulation environment: Jade. In *SCS Conference on Intelligent simulation Environments*. SCS, 1986.
- [Ued85] K. Ueda. Guarded horn clauses. Technical Report TR-103, ICOT, 1985.
- [ULB84] B. Unger, G. Lomow, and G. Birtwistle. *Simulation software and Ada*. SCS, A publication of The Society for Computer Simulation, 1984.
- [VL87] J. Vaucher and G. Lapalme. Process-oriented simulation in prolog. In *SCS Multiconference on AI and Simulation*. SCS, 1987.
- [XUC+86] Z. Xiao, B. Unger, J. Cleary, G. Lomow, X. Li, and K. Slind. Jade virtual time implementation manual. Research Report 86/242/16, The University of Calgary, 2500 University Drive NW, Calgary, Alberta, Canada, T2N 1N4, October 1986.
- [YC87] K. Yoshida and T Chikayama. K11-u - parallel object-oriented language upon kl1. Technical report, ICOT, September 1987.

APPENDIX

Syntax of CSP*

$\langle \text{clause} \rangle ::= \langle \text{process clause} \rangle | \langle \text{procedure clause} \rangle | \langle \text{unit clause} \rangle$

$\langle \text{process clause} \rangle ::= \langle \text{process head} \rangle :: \langle \text{body} \rangle .$

$\langle \text{procedure clause} \rangle ::= \langle \text{procedure head} \rangle :- \langle \text{body} \rangle .$

$\langle \text{unit clause} \rangle ::= \langle \text{literal} \rangle .$

$\langle \text{process head} \rangle ::= \langle \text{functor} \rangle (\langle \text{process name} \rangle \{ , \langle \text{term} \rangle \})$

$\langle \text{procedure head} \rangle ::= \langle \text{literal} \rangle$

$\langle \text{body} \rangle ::= \langle \text{literal} \rangle \{ , \langle \text{literal} \rangle \}$

$\langle \text{process name} \rangle ::= \langle \text{literal} \rangle$

$\langle \text{literal} \rangle ::= \langle \text{functor} \rangle (\langle \text{term} \rangle \{ , \langle \text{term} \rangle \}) | \langle \text{functor} \rangle$

$\langle \text{functor} \rangle ::= \text{lower case identifier}$

$\langle \text{term} \rangle ::= \langle \text{constant} \rangle | \langle \text{variable} \rangle | \langle \text{list} \rangle | \langle \text{literal} \rangle$

$\langle \text{constant} \rangle ::= \text{integer} | \text{lower case identifier}$

$\langle \text{variable} \rangle ::= \text{identifier starting with an upper case letter or a “_”}$

$\langle \text{list} \rangle ::= [] | [\langle \text{term} \rangle ' \langle \text{list} \rangle]$