National Library of Canada
Canadian Theses Service
Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Service des thèses canadiennes

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

Canada

THE UNIVERSITY OF ALBERTA

Distributed Simulation of Performance Petri Nets

by

Kok-Lung Wong

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

FALL, 1988

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-45759-7

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Kok-Lung Wong

TITLE OF THESIS: Distributed Simulation of Performance Petri Nets

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: Fall 1988

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

Signed.....

Permanent Address:
Block 3, #05-103
Lorong 7, Toa Payoh
Singapore 1231

Date: October 13th, 1988

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Distributed Simulation of Performance Petri Nets** submitted by **Kok-Lung Wong** in partial fulfilment of the requirements for the degree of **Master of Science.**

.....

Supervisor

.....

.....

.....

Date: 13 October 1988

Abstract

This thesis is concerned with the design and implementation of a distributed simulation method for simulating a particular class of performance Petri nets called *Extended Place/Transition Nets* (EPTN). The method takes an EPTN, treats it as a set of smaller *subnets* and simulates these subnets in parallel over a network of processors. A distributed simulator based on the distributed simulation method has been implemented using a number of Sun Workstations, each running under the Unix operating system, connected by a 10-Mbit Ethernet local area network. Some preliminary experimental results are also given.

Acknowlegements

I would like to thank my supervisor Dr. M. Tamer Özsu for suggesting the topic of this research . I would also like to thank the members of my examining committee, Dr. E. Erkut, Prof. U. Maydell and Dr. P. Rudnicki, for their criticisms, comments and suggestions.

Thanks are also due to Dr. J. Schaeffer for his invaluable assistance in getting the implementation portion of this thesis going.

Finally, I wish to express my gratitude and appreciation to Flavia for her support and encouragement throughout the writing of this thesis.

Contents

Abstract iv

Acknowledgements v

List of Figures viii

List of Tables x

1 Introduction 1

1.1 Motivation 1

1.2 Thesis Objectives and Outline 3

2 Review of Distributed Simulation 6

2.1 Introduction 6

2.2 Sequential Simulation 6

2.3 Overview of Distributed Simulation 8

2.4 Design Issues in Distributed Simulation 9

2.4.1 Decomposition into Submodels 9

2.4.2 Allocation of Submodels 10

2.4.3 Synchronization of Local Simulators 11

2.5 A Survey of Distributed Simulation Methods 13

3 Petri Nets, Performance Petri Nets and the EPTN Formalism 17

3.1 Introduction 17

3.2 Petri Nets 17

3.3 Performance Petri Nets 23
3.3.1 An Analytic Modelling Example 27
3.4 Extended Place/Transition Nets 30
3.4.1 Token Definition 30
3.4.2 Place Definition 31
3.4.3 Transition Definition 32
3.4.4 Firing Rules 34

4 The Distributed Simulation Method **35**
4.1 Introduction 35
4.2 A Distributed System Model 35
4.3 Decomposition and Allocation 36
4.4 The Synchronization Algorithm 40
4.4.1 Overview of the Algorithm 44
4.4.2 Rollback Operation 45
4.4.3 Memory Management 52
4.4.4 Termination Detection 55
4.4.5 The Algorithm 55
4.5 Proof of Correctness 56

5 Implementation Details **60**
5.1 Introduction 60
5.2 Hardware and Software 60
5.3 Structure of the Distributed Simulator 62
5.3.1 The EPTN Subnet Layer 62
5.3.2 The Simulator Layer 65
5.3.3 The VTM Layer 66

6 Experimental Results **69**
6.1 Introduction 69
6.2 EPTN with Serial Structure 70

6.3 EPTN with Conflict Place 73
6.4 EPTN with OR-Postcondition Structure 75
6.5 EPTN with OR-Precondition Structure 77
6.6 EPTN with Feedback Structure 79
6.7 Remarks 82

7 Conclusions 84
7.1 Summary of Results 84
7.2 Future Research 86

Bibliography 87

List of Figures

2.1 Tandem queueing network 13
2.2 Feedforward queueing network 14
2.3 Feedback queueing network 14

3.1 A Petri net model of a computer system 18
3.2 Transition firing in a Petri net 21
3.3 Conflict in a Petri net 22
3.4 Concurrency in a Petri net 22
3.5 Analytic modelling example 27
3.6 The reachability graph 28

4.1 The heuristic allocation algorithm 37
4.2 An EPTN example for allocation 38
4.3 Weighted graph of the EPTN example 39
4.4 Allocation scheme for the EPTN example 39
4.5 The first counter-example 40
4.6 Allocation scheme of the first counter-example 41
4.7 The second counter-example 42
4.8 Allocation scheme of the second counter-example 43
4.9 Structure of a local simulator 47
4.10 Arrival of a token at a local simulator 48
4.11 Primary rollback at a local simulator 51
4.12 Secondary rollback at a local simulator 53

5.1 The distributed simulator 63
5.2 A process tree 67
5.3 Mapping of processor tree to hardware configuration 68

6.1 Allocation schemes for an EPTN with a serial structure 71
6.2 Allocation schemes for an EPTN with a conflict place 74
6.3 Allocation schemes for an EPTN with an or-postcondition structure 76
6.4 Allocation schemes for an EPTN with an or-precondition structure . 78
6.5 Allocation schemes for an EPTN with a feedback structure 81

List of Tables

3.1 The reachability set 29

4.1 Simulation of the first counter-example 41

4.2 Simulation of the second counter-example 43

6.1 Performance results for EPTN with serial structure 72

6.2 Performance results for EPTN with conflict place 73

6.3 Performance results for EPTN with or-postcondition structure . . . 77

6.4 Performance results for EPTN with or-precondition structure 79

6.5 Performance results for EPTN with feedback structure 80

Chapter 1

Introduction

1.1 Motivation

Petri nets originated from the early work of Carl Adam Petri [34]. In his doctoral dissertation titled "Kommunikation mit Automaten" [Communication with Automata], Petri developed a theory of communication among asynchronous components of a computer system. Based on his dissertation, the theoretic development of Petri nets evolved from the works of Holt [11], Dennis [7], Patil [29], Hack [8] and many others.

A Petri net is an "abstract, formal model of information flow" [33]. The properties, concepts and techniques of Petri nets are being developed in search for natural, simple and powerful methods for describing and analyzing asynchronous and concurrent activities in systems. Briefly, a Petri net consists of a set of *places*, a set of *transitions* and a number of *tokens* that reside at the places. *Directed arcs* are used to connect places to transitions and transitions to places. A transition can *fire* when all the input places to the transition have at least one token. Firing a transition involves removing a token from each of the input places to the transition and placing one token in each of the places output from the transition. The presence of a token at a place models the satisfaction of some *condition* and the firing of a transition models the occurrence of an *event* when certain conditions are satisfied. A more detailed discussion of Petri nets is presented in Chapter 2.

The major use of Petri nets has been in the modelling of systems which exhibit asynchronous and concurrent events. As a modelling tool, the Petri net formalism has the following advantages: (1) a graphical and precise representation scheme; (2) a structured representation scheme which facilitates the synthesis of a Petri net in a top-down and/or bottom-up fashion; and (3) the availability of a rich set of solution techniques for the analysis of Petri nets.

Petri nets do not include the time concept. As a result, it is not possible to model the behaviour of systems with respect to time. In the performance modelling of systems, the evolution of time plays a fundamental role in describing the dynamic behaviour of systems since most of the interesting parameters of system design, such as throughput and utilization, are time-dependent. *Performance Petri nets* are Petri nets extended with the time concept. The use of performance Petri nets in the performance modelling of systems is a relatively new approach that has attracted increasing interest in recent years. Performance Petri nets have been used successfully in the performance studies of computer communications protocols [21,36], distributed database systems [15,25] and multiprocessor architectures [9,19, 42].

The two approaches to system performance modelling using performance Petri nets are analytic solution and simulation. Analytic solution requires that restrictions be placed on the timing distributions and/or the structure of a performance Petri nets. For example, timing distributions are usually restricted to be Markovian and/or deterministic, so that Markov chain techniques can be used to compute steady state performance measures. Analytic solution techniques are defined for only performance Petri net models of closed systems. Analytic solution is, in general, mathematically intractable as the size and complexity of a performance Petri net increase. In practice, only a very small subset of performance Petri nets may be solvable using analytic techniques. A more detailed discussion of performance Petri nets is given in Chapter 3.

Simulation of performance Petri nets, on the other hand, removes the restrictions imposed by the use of analytic solution techniques. Using the simulation method, a

physical system can be modelled as a performance Petri net to any arbitrary level of detail. However, it is generally computationally costly to simulate performance Petri nets of non-trivial sizes and complexities.

Sequential simulation of performance Petri nets does not exploit the inherent parallelism available in performance Petri net simulation. Advances in VLSI and data communications technology have produced low cost microprocessors and high bandwidth communications networks. An alternative approach to performance Petri net simulation is to decomposed a performance Petri net model into a set of interacting submodels and simulate the submodels in parallel over a network of processors.

1.2 Thesis Objectives and Outline

This thesis is concerned with the design and implementation of a distributed simulation method for simulating a particular class of performance Petri nets called *Extended Place/Transition Nets* (EPTNs) [25]. Distributed simulation is a relatively new research area that has been attracting increasing interest in the last few years. Briefly, distributed simulation is concerned with techniques of decomposing a simulation model into a set interacting submodels and simulating the submodels in parallel over a network of processors. Most research in distributed simulation has concentrated on the development of algorithms for the synchronizing the interaction of the submodels. The synchronization of submodels is important for ensuring the processing of events across the submodels occurs only in correct simulation time sequences.

The distributed simulation method employed in this thesis takes an EPTN, treats it as a set of smaller *subnets* connected by common places and simulates these subnets in parallel by a set of *local simulators* over a network of processors. The local simulators interact by means of token exchanges and control messages for the purpose of synchronization. The distributed simulation method uses an *optimistic* synchronization algorithm where the local simulators are allowed to interact freely

without any explicit synchronization until an incorrect simulation time sequence is detected. When this happens, the affected local simulators are rolled back to a previous consistent state before the simulation proceeds forward again.

A distributed simulator based on the distributed simulation method was implemented using a number of Sun Workstations, each running under the UNIX operating system, connected by a 10-Mbit Ethernet local area network. The simulator was built in a hierarchical and modular way. It consists of three logical layers. Each layer is in turn decomposed internally to perform its function correctly. The top layer implements the user-interface component of the simulator. The middle layer forms the core of the simulator while the bottom layer provides a number of high-level facilities for inter-processor communication.

This thesis only considers the issues involved in the design and implementation of a distributed simulation method. A preliminary performance evaluation of the method is also given. Other simulation-related problems such as the construction of EPTN models, the validation and verification of EPTN models and the analysis of simulation output are not discussed.

The EPTN formalism was developed for the purpose of modelling and simulation of distributed systems and it has been been used successfully in the performance studies of distributed database concurrency control algorithms [25,15]. Distributed simulation of EPTNs was first proposed by Özsu [24]. However, the proposed method was erroneous: it stated that the interaction of the distributed simulation of the subnets of an EPTN need not be synchronized. It can be demonstrated by the use of counter-examples that the EPTN simulation rules are violated if the simulation of the subnets is not properly synchronized. Two examples are described in Chapter 4.

The outline of the thesis is as follows. In Chapter 2, an overview of sequential and distributed simulation techniques is given. Following that, the issues involved in the design of a distributed simulation method are discussed. A brief survey of distributed simulation methods is also included.

In Chapter 3, an overview of Petri net formalism and performance Petri nets is

presented. Next, the EPTN formalism is introduced.

In Chapter 4, the distributed simulation method is presented. The method describes how to decompose an EPTN into a set of interacting subnets, how to allocate the subnets to a set of processors and how to synchronize the execution of the local simulators simulating the subnets. Correctness proofs for the method are also provided.

Chapter 5 describes the implementation of a distributed simulator based on the distributed simulation method. The experimental environment in which the simulator was built is first described. Following that, the implementation details of the simulator are given.

Chapter 6 presents some preliminary empirical results on the performance of the distributed simulation method. Various representative EPTN models are used as test cases.

The last chapter summarizes the results of the thesis and gives some concluding remarks. Topics for future research are also presented.

Chapter 2

Review of Distributed Simulation

2.1 Introduction

This chapter surveys the current state of research in distributed simulation and identifies the important issues in the design of a distributed simulation method. The organization of the chapter is as follows. An introduction to traditional, sequential simulation is given in Section 2.2. Section 2.3 provides an overview of the distributed simulation problem and a classification of distributed simulation methods. Section 2.4 discusses the various issues involved in the design of a distributed simulation method. A survey of distributed simulation methods is presented in Section 2.5.

2.2 Sequential Simulation

Simulation is an experimental method for observing the behaviour of a system over time when the use of direct measurement methods is neither feasible nor possible. Two broad classes of simulation can be identified: *discrete* and *continuous*. In discrete simulation, the state changes in the systems take place instantaneously at discrete points in time. A continuous system, on the other hand, simulates a system where the state changes take place smoothly and continuously in time. A queue at a supermarket is an example of a discrete system while the atmospheric system is an example of a continuous system. Only discrete simulations are considered in this

thesis. A simulation study consists of the following steps:

1. Analysis of the system to be simulated.
2. Building of a simulation model from the system.
3. Verification of the simulation model.
4. Construction of a simulation program from the simulation model.
5. Conducting the simulation experiments.
6. Statistical analysis of simulation outputs.
7. Repeat steps (2) to (6) if necessary.

Discrete simulation is characterized by a *simulation clock* and a data structure called an *event list*. The simulation clock is used to maintain the progress of simulation time while the event list is used to maintain a set of event notices, with their event times, that are scheduled for the future. At each step in a simulation, the event notice with the smallest event time is removed from the event list and its effect is simulated. The simulation clock is then advanced to the event time of the event notice that was just simulated, The simulated event notice may cause either more event notices to be added to the event list or previously scheduled event notices in the event list to be cancelled and deleted from it.

The problems associated with simulations include:

1. Verification of the simulation model of a system.
2. The difficulty in constructing a simulation program from the simulation model.
3. The computational cost of running the simulation program.
4. Statistical analysis of simulation outputs.

One possible solution to problem (3) is to exploit the power and cost benefits of microprocessors and high-bandwidth communications networks by exploring a distributed approach to simulating some interesting classes of systems which exhibit a high degree of inherent parallelism.

2.3 Overview of Distributed Simulation

A natural way to speed up a simulation study is to decompose a simulation model into a set of interacting submodels that communicate solely by exchanging event messages, and simulate the submodels in parallel over a set of processors. Such a technique is known as *distributed simulation*.

Distributed simulation is characterized by the absence of a global simulation clock and a global event list. Each submodel is simulated by a local simulator and each local simulator has its own local event list and local clock for maintaining simulation time. The set of local simulators taken together form a *distributed simulator*. On simulating an event, a local simulator sends event messages to other local simulators that are affected by that event. The execution of the local simulators is carried out asynchronously, and any interaction among them is only via event messages. In order to ensure the processing of events across the local simulators occurs in a correct simulation time sequence, the simulation of the local simulators have to be synchronized. Synchronization of the local simulators is achieved through the encoding of time in event messages exchanged among the local simulators. Each event message is timestamped with the simulation time of occurrence of that event. Depending on the choice of synchronization algorithm, the execution of the local simulators may deadlock.

Distributed simulation methods can be classified according to the nature of the simulation model being simulated and the approach to synchronization. A simulation can be classified as either discrete or continuous. There are two approaches to synchronization in distributed simulation methods. In the *tight* approach, a global simulation clock is used to maintain simulation time so that at any instant in real time, simulation time is the same for all local simulators. The loose approach, on the other hand, maintains simulation time in an asynchronous manner where each local simulator maintains its own simulation time. At any moment, some local simulators may be far ahead in their simulation times while others lag behind. This chapter will only considers loose synchronization algorithms since they offer higher

potential for maximum parallelism.

A theoretical upper bound on the parallelism achievable is dictated by the number of local simulators. However, the actual lowest execution cost may not necessarily coincide with the largest degree of parallelism because of the amount of overhead introduced by distributing the simulation. The amount of overhead introduced is a function of the frequency of interaction among the submodels and the particular synchronization algorithm that is employed for coordinating the execution of the local simulators.

2.4 Design Issues in Distributed Simulation

There are a number of issues to be addressed in the design of a distributed simulation method. The first issue involves how to decompose a given simulation model into a set of submodels so that a high degree of parallelism can be obtained when the submodels are simulated in parallel. The second issue deals with how to optimally allocate the submodels to an available network of processors so that the execution time of simulation is minimized. The third issue addresses the problem of synchronizing the execution of the local simulators so that simulation occurs only in correct simulation time sequences.

2.4.1 Decomposition into Submodels

The decomposition of a simulation model into a set of submodels is essentially the division of the simulation event list into a set of smaller local event lists to be processed in parallel by a set of local simulators simulating the submodels. Ideally, the submodels should be relatively autonomous with minimum interaction so that a high degree of parallelism is achievable.

The basic decomposition scheme adopted in most distributed simulation methods assumes that a system to be simulated can be represented by the *entity-event* model [20]. Such a system consists of a set of entities. Events occur at these entities at discrete points in time. The occurrence of an entity may affect other

entities. Each entity is autonomous, except when an event involving it occurs. The decomposition of such systems is straightforward: each entity is treated as a submodel and the interaction between two entities is simulated by the interaction of the corresponding submodels.

For example, in a computer communications network, the processors and communications links can be viewed as entities and the transmission of a message can be viewed as an event.

2.4.2 Allocation of Submodels

The optimal allocation of the submodels of a simulation model to a network of processors is a non-trivial problem. It is equivalent to mapping the set of submodels to the topology of the network of processors in such a way that the execution time of simulation is minimized. Factors that must be taken into account include the topology of the network of processors, the frequency of interaction of the submodels, the cost of interprocessor communication and the cost of simulating the submodels on the processors. Finding such an optimal mapping is known to be NP-complete [6]. In practice, scheduling heuristics must be used; sub-optimal mappings are normally produced at considerably less computational cost.

The two approaches to allocating submodels to processors are *static submodel assignment* and *dynamic submodel assignment*. In static submodel assignment, the assignment of the submodels to the processors is fixed for the duration of the simulation. When the number of submodels equals the number of processors, each submodel is assigned to a separate processor. Otherwise, the submodels have to be clustered into several groups where the number of groups equals the number of processors so that each group can be assigned to a separate processor.

In dynamic submodel assignment, submodels are assigned during simulation. Dynamic submodel assignment attempts to balance the load across the network of processors so that the idling times of processors are minimized.

The problem of submodel allocation is a neglected topic in distributed simulation. In most cases, it is assumed that each submodel is assigned to a separate

processor using a static assignment approach.

2.4.3 Synchronization of Local Simulators

The most important aspect of a distributed simulation method is synchronizing local simulator interaction. Synchronization is required for preserving the dependencies among events across the local simulators so that the processing occurs only in correct simulation time sequences. That is, if the occurrence of event A at local simulator LS_1 affects the occurrence of event B at local simulator LS_2, then event B cannot occur before the occurrence of event A in real time. If A and B do not affect each other, then the occurrences of A and B can take place in any arbitrary order without affecting the correctness of the simulation. The dependencies among events form a partial order so that there may be more than one possible sequence of event occurrences that are correct.

Synchronization algorithms can be classified as either *pessimistic* or *optimistic*. In the pessimistic approach, the simulation time at each local simulator advances only when it is known that dependencies among events across local simulators are preserved. The optimistic approach, in contrast, assumes that the simulation time at each local simulator can be advanced until a violation in the dependencies among events is detected. When this happens, the local simulators roll back to a previous consistent state.

Pessimistic Approach

In pessimistic synchronization algorithms, if a local simulator LS_i can send event messages to local simulator LS_j, then there is *channel* from LS_i to LS_j. Each channel is assigned a *channel time*, which is defined to be the time component of the last event message received along that channel. Each local simulator can be simulated up to the minimum of all channel times of its incoming channels. This minimum is updated whenever a new event message arrives at a channel and simulation time is advanced. A problem arises when the channel time of a channel remains fixed at some value indefinitely, causing the local simulator to wait for

the arrival of an event message. Waiting can cause deadlocks if there is a cycle of waiting local simulators. Pessimistic synchronization algorithms differ basically in how the deadlock problem is handled.

Optimistic Approach

Optimistic synchronization algorithms do not require local simulators to wait until it is "safe" before proceeding. If a local simulator receives an event message that is in the simulation past of the local simulator, the local simulator has to undo what it has done, cancel all side effects at other local simulators and roll back to a state prior to the simulation time of the arrival of the event message before resuming normal simulation.

The core of optimistic synchronization algorithms lies in the rollback operation. During simulation, a local simulator has to save information about its interaction with other local simulators in a history log. The state of the local simulator is also saved at regular checkpoints. If an event message arrives in the simulation past of a local simulator, the local simulator rolls back to the latest checkpoint prior to the timestamp of the event message. All side effects in other local simulators resulting from event messages sent after the timestamp of the "late" event message have to be cancelled.

The frequency of state saving is directly proportional to how fast rollbacks can be carried out. However, frequent state saving is also associated with high run-time memory overhead. For more efficient memory utilization, a distributed algorithm is required to calculate the minimum of the simulation times at the local simulators from time to time. Since the local simulators never roll back to a point prior to the minimum of simulation time, this can be used to remove unnecessary entries from history and state logs. Optimistic synchronization algorithms assume that most programs obey a *temporal locality principle* where most event messages arrive in increasing chronological order at a local simulator so that the chances of rolling back are small, and those that arrive in the "past" tend to arrive in the "recent" past so that the amount of side effects to be cancelled is small [13]. The validity of the

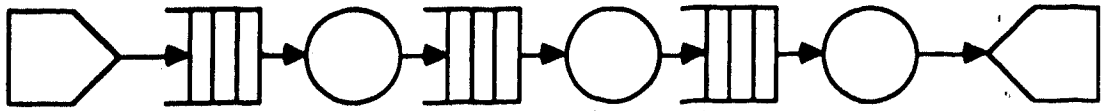


Figure 2.1: Tandem queueing network

principle is obviously application dependent, and can only be verified empirically.

2.5 A Survey of Distributed Simulation Methods

Distributed simulation was first proposed by Chandy in 1977 in a series of lectures at the University of Waterloo, and was also proposed independently by Bryant [2]. A refinement of these ideas is found in [4,5]. These methods are based on pessimistic synchronization algorithms. Deadlock is avoided by using overhead *NULL* messages. A *NULL* message with timestamp t sent from local simulator LS_i to local simulator LS_j informs local simulator LS_j that no further event message would be sent on channel from LS_i to LS_j up to time t. Lakshmi [16] studied the performance of these methods empirically using a simulation approach. The study indicated that the performance gains for simulating tandem queueing networks were very good, feedforward queueing networks fairly good, and feedback queueing networks very poor. Examples of these queueing networks are illustrated in Figure 2.1, Figure 2.2 and Figure 2.3.

Peacock *et a.* [30,31,32] and Holmes [10] have proposed methods based on pessimistic synchronization algorithms where deadlocks are avoided by using *probe* messages. Probe messages are sent out periodically to collect state information of

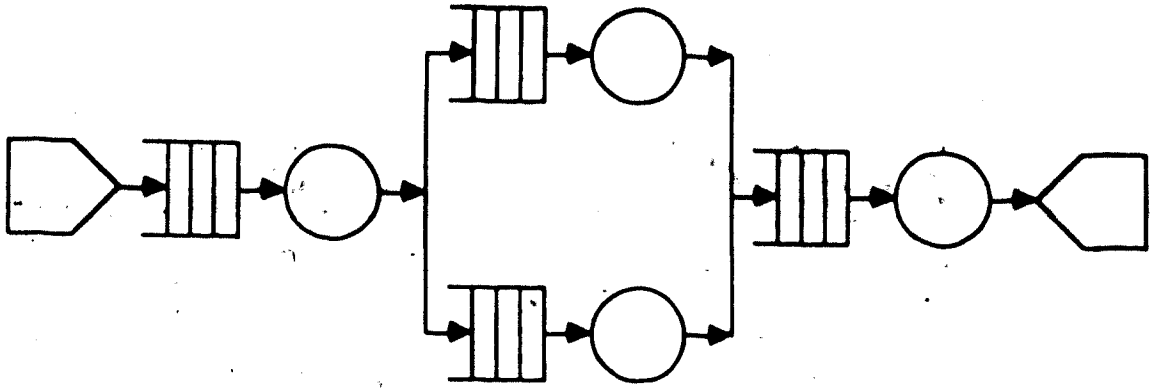


Figure 2.2: Feedforward queueing network

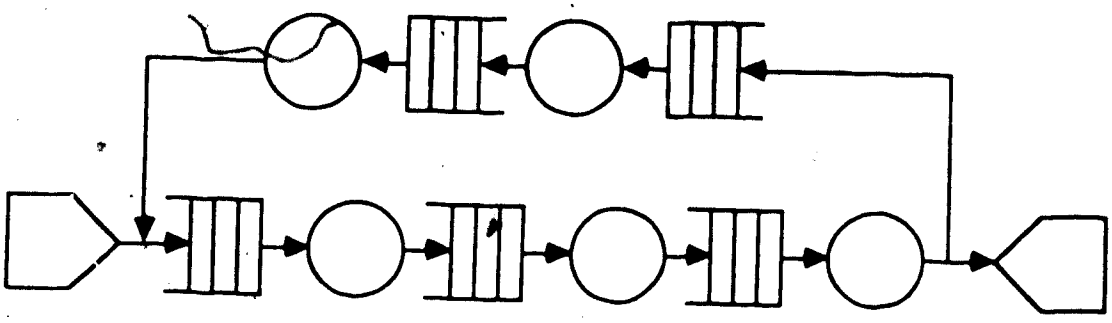


Figure 2.3: Feedback queueing network

local simulators.

Later, Chandy and Misra [3] proposed a method using a pessimistic synchronization algorithm where a distributed algorithm is used for deadlock detection and recovery. An empirical study of the performance of this method was carried out by Reed [37] using an actual implementation on a Sequent Balance 21000 shared-memory multiprocessor. Using queueing networks as the simulation applications, the study indicated that the method is not viable approach in queueing network simulation, particularly networks containing feedback loops. The study also indicated that a dynamic approach to the allocation of submodels showed better performance gains.

Jefferson and Sowizral [12,13,14] proposed a method using an optimistic approach to synchronization. At any moment, some local simulator may be far ahead in their simulation times, while others lag behind. The local simulators interact by exchanging timestamped event messages. An event message corresponds to an event notice in sequential simulation. There is a single input queue of event messages associated with each local simulator. The local simulators process event messages from their input queues until they are empty. They do not wait until conditions are "safe" before processing event messages. When an event message arrives at a local simulator where the timestamp of the event message is less than the simulation time of the local simulator, the local simulator has to be restored to a consistent state prior to the arrival of the event message. Any side effects caused at other local simulators have to be cancelled. For every event message sent that has to be nullified, an *anti-message* is sent. Lavenberg *et al.* [17] analyzed the effect of system parameters on the performance of this method for the simple case where a simulation consists of two interacting submodels. Empirical studies of the performance of this method has been investigated by Berry [1] using a simulated environment, and by Jefferson *et al.* [14] using an actual implementation on the Caltech Mark III Hypercube multiprocessor. Both studies suggest that this method can provide substantial performance gains for large, irregular simulations.

A more detailed and complete survey on distributed simulation can be found in

[20].

Chapter 3

Petri Nets, Performance Petri Nets and the EPTN Formalism

3.1 Introduction

This chapter provides an introductory tutorial to Petri nets and performance Petri nets, and an overview of the Extended Place/Transition Net (EPTN) formalism. A comprehensive treatment of Petri net theory can be found in texts by Peterson [33], and Reisig [38], while a survey of performance Petri nets is given in [27]. A formal, detailed definition of the EPTN formalism is described in [25].

The rest of this chapter is organized as follows. Section 3.2 and Section 3.3 present tutorial introductions to Petri nets and performance Petri nets respectively. Section 3.4 describes the EPTN formalism.

3.2 Petri Nets

Definition 3.1 A Petri net is a triple $PN = (P, T, A)$ where

1. $P = \{p_1, p_2, \ldots, p_m\}$ is a set of *places*, $(P \neq \emptyset)$;
2. $T = \{t_1, t_2, \ldots, t_n\}$ is a set of *transitions*, $(T \neq \emptyset)$; and
3. $A \in \{P \times T\} \cup \{T \times P\}$ is a set of *directed arcs*.

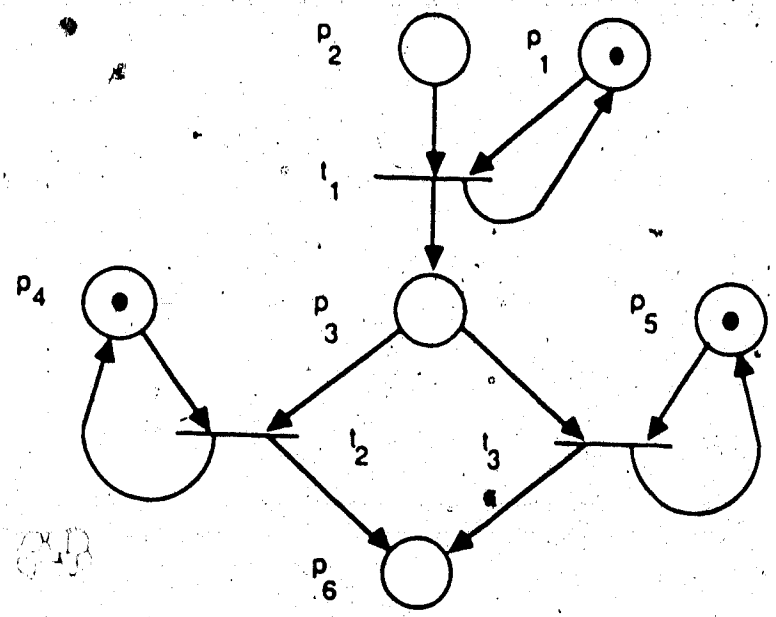


Figure 3.1: A Petri net model of a computer system

□

According to the above definition a Petri net is a *directed bipartite graph* where $\{T \cup P\}$ form the nodes of the graph and A is the set of arcs which connect these nodes. Note that $T \cap P = \emptyset$.

A Petri net model of a simple computer system with a single CPU and two I/O processors is illustrated in Figure 3.1. The places are represented by circles and the transitions are represented by bars.

From a modelling viewpoint, the places in a Petri net may be interpreted as corresponding to the conditions that have to be fulfilled for the events - corresponding to transitions - to take place. Thus, in Figure 3.1, the definition of places are as follows: p_1: CPU is idle; p_2: a job has arrived and is waiting to be processed; p_3: a job is waiting for I/O; p_4: I/O processor 1 is idle; p_5: I/O processor 2 is idle; and p_6: the processing of a job is completed. For the time being, ignore the small dots at places p_1, p_4 and p_5. The interpretation of transitions, on the other hand, is as

follows: t_1: the CPU is processing a job; t_2: the I/O processor 1 is processing a a job; and t_3: the I/O processor 2 is processing a job.

Definition 3.2 A *marking* M of a Petri net $PN = (P, T, A)$ is a mapping

$$M : P \rightarrow \mathbf{N}.$$

where $\mathbf{N}$ is the set of natural numbers.

□

According to this definition, the marking of a Petri net is defined in terms of the markings of its places. The notation $M(p_i)$ is used to denote the marking of place p_i. The marking of a place is indicated by the number of *tokens* (which are the small dots at the places p_1, p_4 and p_5 in Figure 3.1) that exist at that place. For example, $M(p_i) = 2$ indicates the existence of two tokens in place p_i.

Definition 3.3 For a Petri net $PN = (P, T, A)$ the initial placement of tokens is called the *initial marking* of PN.

□

The initial marking is an important concept in discussing the dynamic nature of Petri nets. In Figure 3.1, the initial marking of the net is $M(p_1) = M(p_4) = M(p_5) = 1$ and $M(p_2) = M(p_3) = M(p_6) = 0$, which can also be represented as $(1, 0, 0, 1, 1, 0)$ in vector notation.

At this point it is useful to point out that a marking of a Petri net indicates the *state* of the net at that moment. Thus a Petri net may change states by moving from one marking to another (starting with the initial marking). These state changes of the Petri net model the execution of the underlying system. Thus a Petri net models both the static behaviour of the underlying system (e.g., the model in Figure 3.1) as well as its dynamic behaviour.

The state changes in a Petri net obey certain rules. The following definitions set the formal framework within which such changes can occur.

Definition 3.4 Given a Petri net $PN = (P, T, A)$, the following are defined:

1. For each $t_j \in T$, $I(t_j)$ is the *set of input places* and $O(t_j)$ is the *set of output places* given by

$$I(t_j) = \{p_i \mid (p_i, t_j) \in A\},$$

$$O(t_j) = \{p_i \mid (t_j, p_i) \in A\}.$$

2. For each $p_i \in P$, $I(p_i)$ is the *set of input transitions* and $O(p_i)$ is the *set of output of transitions* given by

$$I(p_i) = \{t_j \mid (t_j, p_i) \in A\},$$

$$O(p_i) = \{t_j \mid (p_i, t_j) \in A\}.$$

□

Definition 3.5 A transition $t_j \in T$ of a Petri net $PN = (P, T, A)$ is said to be *enabled* iff $M(p_i) > 0$, $\forall p_i \in I(t_j)$.

□

In the example shown in Figure 3.1, no transition is enabled. If, however, a job arrives at the system (i.e., $M(p_2) = 1$), t_1 is enabled since there is at least one token in each of the places (p_1 and p_2) that are in the set of input places of t_1 (as shown in Figure 3.2).

Definition 3.6 An enabled transition $t_j \in T$ of a Petri net $PN = (P, T, A)$ can *fire* iff it is enabled. The firing causes a change in a state (i.e. a new marking M') that is defined by

$$M'(p_i) = \begin{cases} M(p_i) + 1 & \text{if } p_i \notin I(t_j) \wedge p_i \in O(t_j) \\ M(p_i) - 1 & \text{if } p_i \in I(t_j) \wedge p_i \notin O(t_j) \\ M(p_i) & \text{otherwise} \end{cases}$$

□

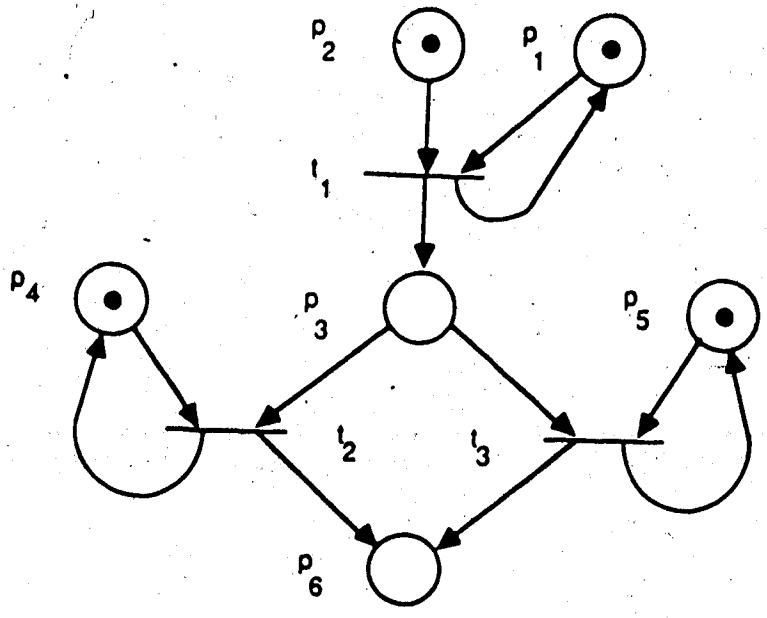


Figure 3.2: Transition firing in a Petri net

Following the discussion on the example, when t_1 is enabled, and thus fires, the following changes occur in markings : $M(p_1) = 1$, $M(p_2) = 0$, $M(p_3) = 1$ as shown in Figure 3.3. Besides depicting the effects of a transition firing, this new state also demonstrates a fundamental issue, namely the modelling of *conflict*: both t_2 and t_3 are enabled and can fire; however, the firing of either one disables the other since the token at place p_3 is removed. The decision as to which one fires is completely arbitrary.

Assume that a new job arrives at the system (i.e. $M(p_2) = 1$). This situation, depicted in Figure 3.4, represents a *concurrent* execution in Petri nets: both t_1 and either t_2 or t_3 are enabled and can fire concurrently. Assuming that I/O processor 1 is chosen for the service, both t_1 and t_2 can fire simultaneously.

The ability of Petri nets to naturally model concurrency and conflict in asynchronous computer systems is what makes them powerful modelling tools.

The execution of a Petri net, like the system it models, is viewed as a sequence of *discrete events* whose order of occurrence is one of possibly many allowed by the

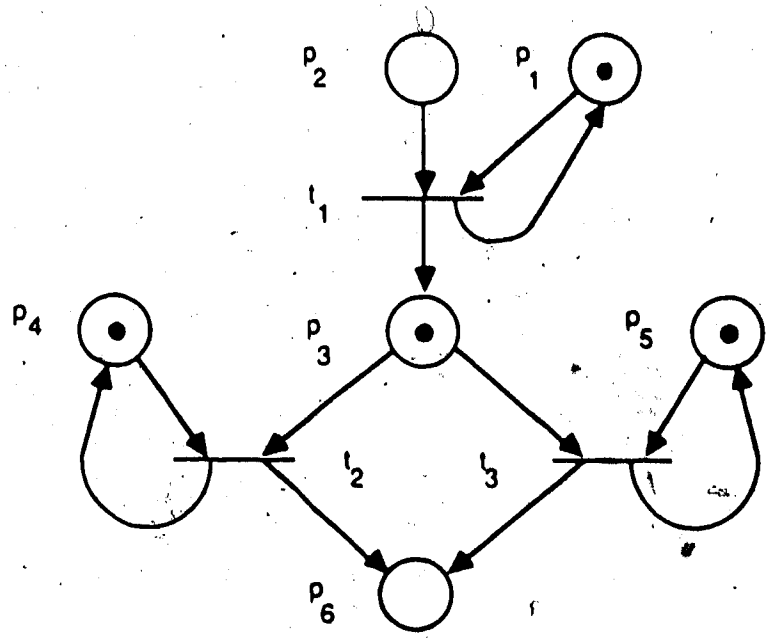


Figure 3.3: Conflict in a Petri net

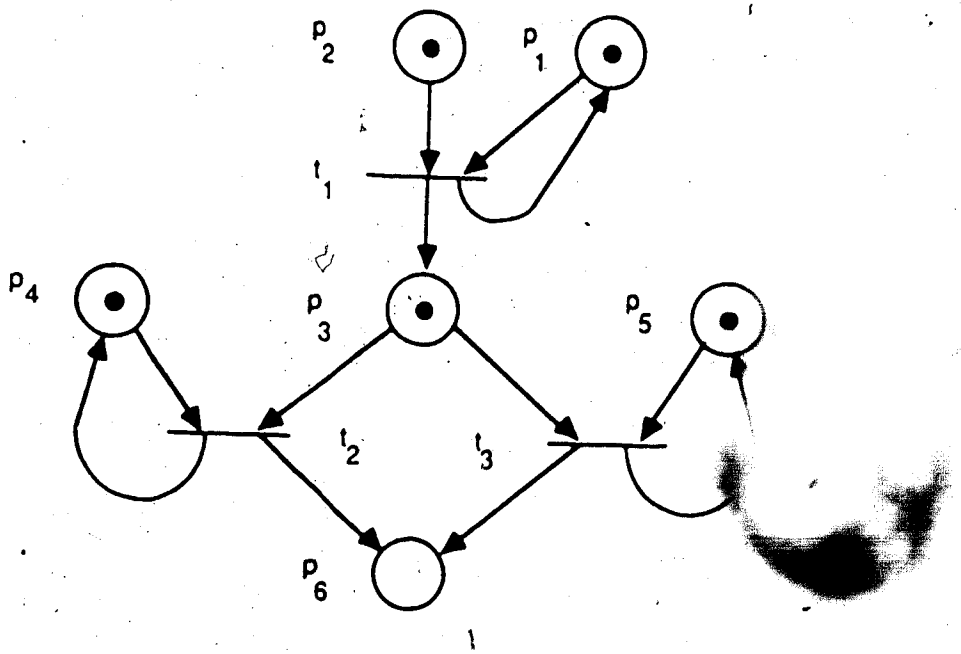


Figure 3.4: Concurrency in a Petri net

basic structure. If at any instance more than one transition is enabled, then any of the enabled transitions may fire. The choice as to which transition fires is made in a *nondeterministic* manner. The nondeterminism in the execution of Petri nets reflects in real life situations in which several things are happening concurrently, the apparent order of occurrence of events is not unique, but rather any of a set of sequences of events may occur. However, the partial ordering of event occurrences is unique.

The behaviour of a system modelled by a Petri net can be studied by analyzing the properties of the Petri net. The analysis of Petri nets uses the concept of *reachability sets*. The reachability set of a Petri net is the set of all possible markings that are reachable from some initial marking, and it can be finite or infinite.

Two important analysis problems of Petri nets are *boundedness* and *liveness*. A place in a Petri net is k-boundeded if and only if the number of tokens in that place does not exceed some integer k for all possible markings of that place in the reachability set of the Petri net. A Petri net is k-bounded if all of its places are also k-bounded. Boundedness is a very important property for the modelling of hardware devices. For example, a place which is 1-bounded can be used to model a flip-flop while a place which is k-bounded for $k > 1$ can be used to model a counter.

Another important analysis problem is *deadlock* which is defined to be a marking in the reachability set of the Petri net in which no transition is enabled. When a Petri net deadlocks, it remains in some marking forever. A Petri is *live* if it does not deadlock. Deadlock and liveness are important properties in the modelling of resource allocation problems.

3.3 Performance Petri Nets

Petri nets have some shortcomings when it comes to the performance modelling of systems. These include [27]:

1. The lack of the time concept. In Petri nets, transition firings are instantaneous, meaning that events do not take any time.

2. The insistence that the markings of all the input places of a transition must be greater than zero before the transition could fire; similarly, a transition firing implies that all the output places of the transition are affected by the firing (i.e., conjunctive logic on inputs and outputs).

3. The absence of a mechanism whereby tokens may be interpreted as representing jobs in the system are allowed to carry data.

Performance Petri nets are derivatives of Petri nets that aim to overcome the above limitations. Performance Petri nets based on analytic solutions can be grouped into two broad categories depending on the nature of the time. The first category consists of those where transition firing times are constant or deterministic. These are the *deterministic timed Petri nets*. The second category of performance Petri nets associate stochastic behaviour with the firing times. This class of performance Petri nets are called *stochastic timed Petri nets*. The analysis of performance Petri nets uses the reachability set concept. The markings of a *finite* reachability set can be represented as a *reachability graph* which is a labelled directed graph depicting the relationship of reachability between markings or states of the Petri net. The reachability graph of a performance Petri net has an additional "time" description associated with each state describing how long the performance Petri net is in a particular state.

The first deterministic timed Petri net was introduced by Ramchandani [35] in which deterministic firing times are assigned to transitions. In his seminal paper, he studied the steady state behaviour of deterministic timed Petri nets and provided methods for calculating the throughput rate of the underlying system modelled by the net.

Later, Razouk and Phelps [36] used the a modified deterministic timed Petri nets to model timeouts and processing delays of communication protocols. Their deterministic timed Petri net model extends Petri nets in three ways. Firstly, an *enabling* time is associated with each transition. Enabling times apply from the point when a transition first becomes enabled and the transition must remained

enabled for the period of the predefined enabling time before it can fire. Secondly, fixed firing times are associated with each transition to model processing delays. The third extension is the grouping of each set of potentially conflicting transitions into a *conflict* set. Every transition belongs to exactly one conflict set and conflict sets do not overlap. A firing frequency is assigned to each transition in a conflict set to indicate the priority of firing in the set. The performance analysis procedure consists of four steps. The first step involves transforming a deterministic timed Petri net into a reachability graph. The second step is to compress the reachability graph into a *decision* graph. The decision graph is basically a compressed version of the reachability graph with a smaller state space. The third step calculates the steady-state probabilities of the decision graph and the last step calculates throughput and utilization measures using the steady-state probabilities.

Holliday and Vernon [9] have developed a very general model of deterministic timed Petri nets for evaluating the performance of computer architectures. The model is essentially augmenting Petri nets with a set of firing durations, a set of firing frequencies and a set of named resources. A resource is associated with a transition and the resource is said to be in use if the transition is firing. A state is defined by a marking vector and a set of remaining firing times associated with each transition whose firing is in progress. The firing frequencies are used in assigning probabilities to next states when building the reachability graph. The performance analysis procedure consists of four steps. The first step is to obtain the reachability graph, which is an embedded, discrete parameter, finite state Markov chain. The second step involves aggregating the states of the Markov chain in order to cut down the size of the state space. The third step is the analysis of the reduced Markov chain to find the long run probabilities. The fourth and last step computes resource usage expectations.

Molloy [21], Natkin [22] and Symons [40] have independently proposed the concept of stochastic timed Petri nets. The stochastic timed Petri net model proposed by Molloy in [21] extends Petri nets by assigning an exponentially distributed firing rate to each transition for modelling continuous-time systems, or a geometrically

distributed firing rate to each transition for modelling discrete-time systems. He proves that the new model is isomorphic to homogeneous Markov processes. The proof is based on the memoryless property of the exponential and geometric distributions. A simple procedure for generating reachability graphs is given based on the proof of isomorphism. Reachability graphs obtained in the procedure are generated without taking into account the element of time. An additional token-flow analysis has to be performed on the reachability graphs in order to obtain performance measures.

Marsan *et al.* [19] generalizes the class of stochastic timed Petri nets described by Molloy [21] by partitioning the transitions into two sets: *timed* and *immediate* transitions. Immediate transitions fire in "zero" time when they are enabled while timed transitions obey the same firing rules as before. Several transitions may be enabled simultaneously, and if they do, immediate transitions always have priority over timed ones in firing. Probabilities are assigned to sets of transitions that are enabled simultaneously to resolve the contention in firing. The generalized stochastic timed model was shown to be equivalent to Markovian models [19]. In fact, the time behaviour of the model can be represented by a stochastic point process with a finite state space. There exists a one-to-one correspondence between the markings of the net and the states of the stochastic point process. Within the stochastic point process is an embedded Markov chain, whose analysis yields the steady-state information of the net, and performance measures can be calculated. The inclusion of immediate transitions was also shown to be efficient in reducing the complexity of the solution space.

It is clear that analytic solutions to performance Petri nets with general time distributions are not yet at hand. Therefore, a number of studies have concentrated on the simulation of performance Petri nets. Two of these studies are *Simulation Nets* [41] and *Function Nets* [39]. Simulation Nets extends Petri nets by the use of *inhibitor arcs*, *interrupt arcs* and *coloured tokens*. Inhibitor arcs allows the testing of zero marking. Interrupt arcs are used as a mechanism to "interrupt" firing transitions. Coloured tokens enable the classification of tokens into different

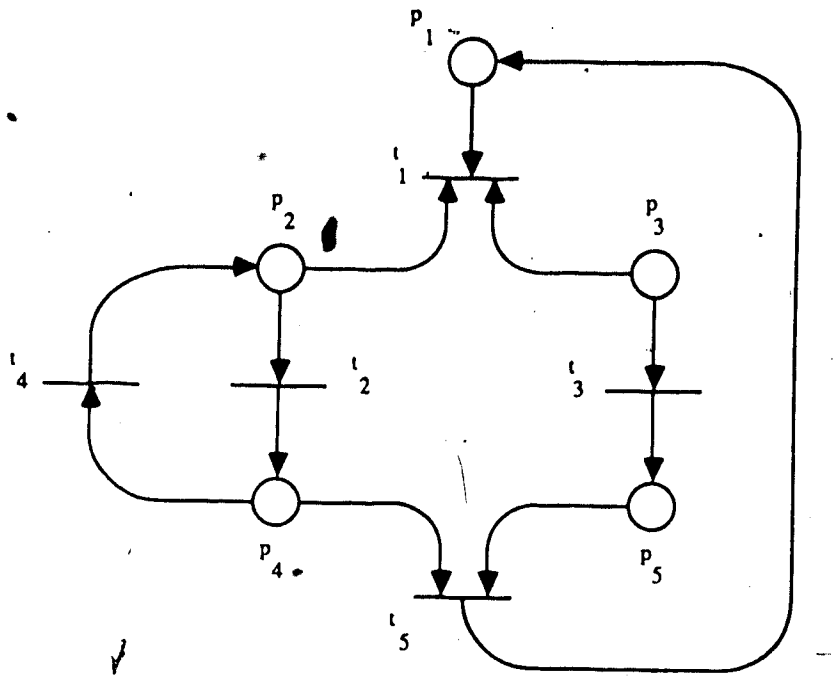


Figure 3.5: Analytic modelling example

classes, and therefore, conditioning the token movement within the net on the type of the token. In Function Nets, a transition is interpreted as an "agency" and a place is interpreted as a "channel". Token definition is extended so that a message, which can be interpreted and manipulated by transitions during their firings, is associated with each token. The treatment of a Function Net is as a queueing system, which is then simulated.

3.3.1 An Analytic Modelling Example

It is appropriate to provide an example illustrating the use of performance Petri nets based on analytic solution techniques. The example chosen borrowed from [27].

Figure 3.5 shows a continuous-time stochastic timed Petri net with five places and five transitions. The net models sequential operation (t_5, t_1) and parallel operation (t_2, t_3), forking (t_1) and contention (t_4, t_5). The firing times of the transitions are exponentially distributed with firing rates given by

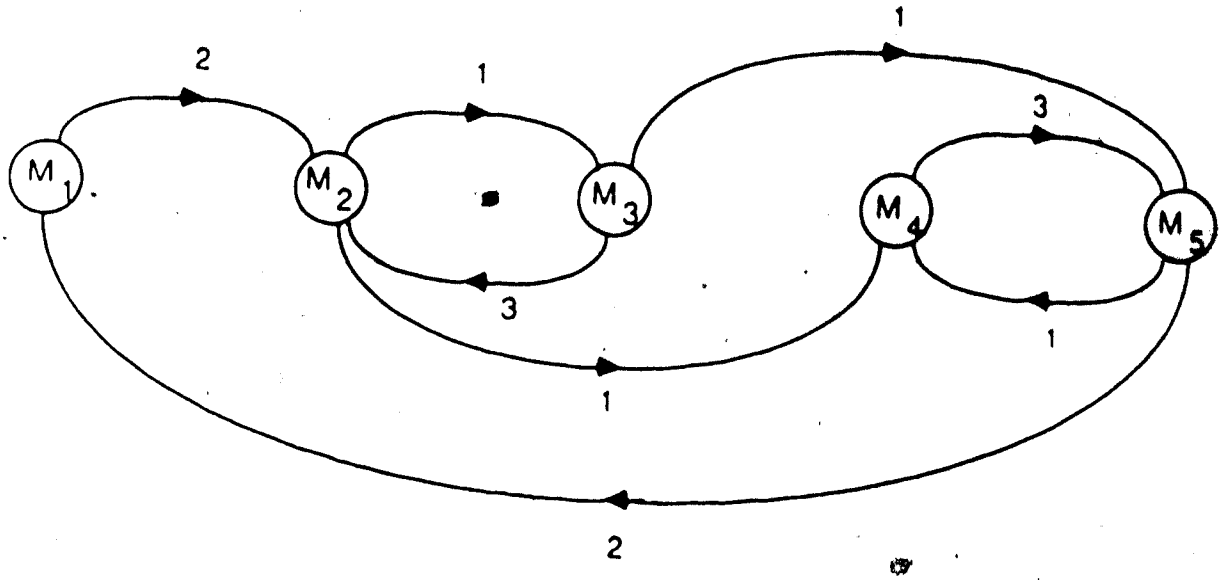


Figure 3.6: The reachability graph

$$\lambda(t_1) = 2,\ \lambda(t_2) = \lambda(t_3) = 1,\ \lambda(t_4) = 3,\ \lambda(t_5) = 2$$

Assume that the initial marking of the Petri net has one token in place p_1 and no tokens in the remaining places. The objective of the analysis is to determine the delay of the different components in the net.

The reachability graph is shown in Figure 3.6 and the reachability set is enumerated in Table 3.1. Each row in Table 3.1 represents a marking vector for that state. In other words, entry (i, j) indicates the number of tokens that exist at place p_j in marking M_i.

Treating the reachability graph as shown in Figure 3.6 as an ergodic Markov chain, the following steady-state marking probabilities are obtained:

	p_1	p_2	p_3	p_4	p_5
M_1	1	0	0	0	0
M_2	0	1	1	0	0
M_3	0	0	1	1	0
M_4	0	1	0	0	1
M_5	0	0	0	1	1

Table 3.1: The reachability set

$$P[M_1] = 0.1163$$
$$P[M_2] = 0.1860$$
$$P[M_3] = 0.0465$$
$$P[M_4] = 0.5349$$
$$P[M_5] = 0.1163$$

Based on these marking probabilities, the steady-state probability that there are μ_i tokens in each place for any marking M_i can be obtained. These steady-state probabilities form the token density function as shown below:

$$P[\mu_1 = 0] = 0.8837 \quad P[\mu_1 = 1] = 0.1163$$
$$P[\mu_2 = 0] = 0.2791 \quad P[\mu_2 = 1] = 0.7209$$
$$P[\mu_3 = 0] = 0.7675 \quad P[\mu_3 = 1] = 0.2325$$
$$P[\mu_4 = 0] = 0.8372 \quad P[\mu_4 = 1] = 0.1628$$
$$P[\mu_5 = 0] = 0.3488 \quad P[\mu_5 = 1] = 0.6512$$

The delay for the components can now be calculated using token-flow analysis based on Little's result [18] and flow balance. Consider the subsystem that is made up of the places p_2, p_3, p_4 and p_5 and transitions t_2, t_3, t_4 and t_5. Transition t_1 is enabled whenever there is a token in p_1 and so the frequency of usage of t_1 is 11.63%. Since

the firing rate of t_2 is 2, the average rate at which tokens flow (per unit time) through p_1 is

$$P[\mu_1 = 1] \times \lambda(t_1) = 0.1163 \times 2 = 0.2326$$

The number of tokens flowing into the subsystem is double the flow through t_1 because t_1 is a fork operation. Hence, the rate at which tokens enter the subsystem is $0.2326 \times 2 = 0.4652$ tokens per unit time. The inflow and outflow are balanced as t_5 is a join operation. Since the subsystem conserves tokens, Little's result can be applied to obtain the average number of tokens in the subsystem which is the sum of the average number of tokens in each place:

$$\sum_{i,j} jP[\mu_i = j] = 1.7674$$

On the average, the time a token takes to return to place p_1 is given by:

$$\frac{\textit{average number of tokens in subsystem}}{\textit{rate tokens enter subsystem}} = \frac{1.7674}{0.4652} = 3.799$$

3.4 Extended Place/Transition Nets

The EPTN formalism was formally proposed in [25] to allow representation and performance evaluation of distributed database concurrency algorithms. The extensions in the formalism are geared towards a performance evaluation perspective rather than that of a theoretical nature. Performance estimates for an asynchronous computation are derived by expressing the computation in terms of an EPTN model and then simulating the behaviour of the EPTN model.

The EPTN formalism extends Petri nets in basically four areas : (1) token definition, (2) place definition, (3) transition definition, and (4) state definition. A more complete treatment of the EPTN formalism is described in [25].

3.4.1 Token Definition

Definition 3.7 In an EPTN $E = (P, T, A)$, there are two types of tokens : *simple* and *data* tokens.

□

A data token is a token which has attributes associated with it. The inclusion of attributes in data tokens enables performance and other related data to be carried. A simple token, on the other hand, is a token which does not have any associated attributes, and therefore cannot carry any data.

A data token can be viewed as a tuple where each field in the tuple represents an attribute. The inclusion of data token is very convenient and useful in modelling computer systems. Messages and transactions which are used frequently in computer systems can easily be represented as data tokens where the movement of data tokens is associated with the movement of the corresponding messages or transactions.

3.4.2 Place Definition

Definition 3.8 In an EPTN $E = (P, T, A)$, each place $p_i \in P$ is defined as a 4-tuple $p_i = (To(p_i), Src(p_i), Snk(p_i), Re(p_i))$ where

1. $To : P \rightarrow \{simple, data\}$ is a *token type* function.
2. $Src : P \rightarrow \{true, false\}$ is a *source* function.
3. $Snk : P \rightarrow \{true, false\}$ is a *sink* function.
4. $Re : P \rightarrow \{true, false\}$ is a *resource* function.

□

As a result of different token types assigned to the places in an EPTN, simple and data tokens may take different routes during their movements within a net. The presence of a simple token at a place only indicates the satisfaction of some condition associated with the place. The presence of data tokens at a place, on the other hand, represent system inputs (e.g., jobs, transactions, etc.) so that the place can be regarded as a queue.

Sources and sinks are normally represented using transitions without input places and transitions without output places, respectively, in Petri net modelling. In

the EPTN formalism, special places associated with sources and sinks are adopted. Places associated with a source are called *source places* while places associated with a sink are called *sink places*. Data tokens are generated at a place associated with a source and consumed at a place associated with a sink.

The purpose in associating a place in an EPTN with a resource is to allow performance estimates to be collected when the movement of tokens in the EPTN is simulated.

3.4.3 Transition Definition

Definition 3.9 In an EPTN $E = (P, T, A)$, each transition $t_j \in T$ is defined to be a 4-tuple $t_j = (F(t_j), Pr(t_j), Tp(t_j), Po(t_j))$ where

1. $F: T \rightarrow \mathbf{R}^+ \cup \{0\}$ [1] is a *transition firing* function.
2. $Pr : T \rightarrow \mathbf{\Phi}$ is a *transition precondition* function where $\mathbf{\Phi}$ is a set of transition preconditions.
3. $Tp : T \rightarrow \mathbf{\Pi}$ is a *transition procedure* function where $\mathbf{\Pi}$ is a set of transition procedures.
4. $Po : T \rightarrow \mathbf{\Psi}$ is a *transition postcondition* function where $\mathbf{\Psi}$ is a set of transition postconditions.

○ □

Transition Firing Time

The transition firing function $F(t_j)$ associates each transition t_j with a firing time. The firing time may be constant for all firings of the transition or it may be a function of the specific tokens that enable the transition for that firing. For example, a transition representing the action of enqueuing an element may be specified to take a constant time, whereas one which represents the reading of a set of data items of

[1] Note that $\mathbf{R}^+$ is the set of positive real numbers.

a transaction (represented by the enabling token) normally takes time proportional to the size of the set.

Transition Precondition

The specification of a precondition for a transition t_j, defined by the transition precondition function $Pr(t_j)$, facilities the activation of only a subset of $I(t_j)$ to take part in the firing of t_j. Such a subset, denoted $e_{j,k}$, is called an *enabling set* of t_j, where $k = 1, 2, \ldots, n$ for some $n \in \mathbf{N}^+$. The set of all enabling sets of a transition t_j is denoted E_j. Note that the enabling sets in E_j are mutually exhaustive but not mutually exclusive.

The transition precondition of a transition t_j is expressed using concepts borrowed from propositional logic. Specifically, the transition precondition is specified as a disjunction of conjunctive terms. Each conjunctive term is associated with an enabling set in E_j. The Boolean variables in each conjunctive term correspond to whether the places in the associated enabling set are marked. Thus, if the places in an enabling set are marked, then the disjunction evaluates to be true, thereby enabling t_j and the enabling set of places are selected to participate in the firing of t_j. If the places in two or more enabling sets are marked, then the decision as to which enabling set takes part in the firing is arbitrary.

Formally, the transition precondition of t_j is given by

$$Pr(t_j) = \vee_{e_{j,k} \in E_j} \{ \wedge_{p_i \in e_{j,k}} (M(p_i) > 0) \}.$$

t_j is *enabled* if and only if $Pr(t_j) = true$. If $Pr(t_j) = true$, let e_j^{pr} denote the enabling set chosen to participate in the firing of t_j. e_j^{pr} is called the *selected enabling set*.

Transition Procedure

One of the most important features that the EPTN formalism has, since it is to be used in performance evaluation studies, is the ability to have tokens carry performance related data. In an EPTN, a data token represents a transaction or a message that the system receives and the data it carries relates to the arrival time

number of network messages that are transmitted to execute it, etc. In collecting data, transitions manipulate the attributes of these tokens. The transition procedure of a transition t_j, defined by the transition procedure function $Tp(t_j)$, specifies how transition t_j manipulates the token attributes.

Transition Postcondition

The postcondition of a transition t_j, defined by the transition postcondition function $Po(t_j)$, specifies the routing of tokens to a subset of output places $O(t_j)$ after t_j finishes firing. Such a subset of output places, denoted $r_{j,k}$, is called a *receiving set* of t_j, where $k = 1, 2, \ldots, m$ for some $m \in \mathbf{N}^+$. The set of all receiving sets of a transition t_j is denoted R_j. Note that the receiving sets in R_j are mutually exhaustive but not mutually exclusive.

The postcondition of a transition t_j is a function of the data tokens that participate in the firing of t_j and the marking of $O(t_j)$ prior to the end firing of t_j. The evaluation of the postcondition selects one of the receiving sets in R_j. That is, the range of $Po(t_j)$ is partitioned into exactly $\mid R_j \mid$ sets where there is a one-to-one correspondence between these $\mid R_j \mid$ sets and the receiving sets in R_j. The receiving set that is chosen in the routing of tokens, called *selected receiving set*, is denoted r_j^{po}.

3.4.4 Firing Rules

The firing of a transition $t_j \in T$ of an EPTN $E = (P, T, A)$ under a marking M in which it is enabled, results in a new marking M' defined as

$$M'(p_i) = \begin{cases} M(p_i) + 1 & \text{if } p_i \in r_j^{po} \wedge p_i \notin e_j^{pr} \\ M(p_i) - 1 & \text{if } p_i \in e_j^{pr} \wedge p_i \notin r_j^{po} \\ M(p_i) & \text{otherwise.} \end{cases}$$

Chapter 4

The Distributed Simulation Method

4.1 Introduction

This chapter presents the distributed EPTN simulation method. The method consists of two major parts. The first part deals with how to decompose a given EPTN into a set of interacting subnets (i.e., the submodels), and how the subnets are allocated to an available network of processors. The second part is concerned with how the interaction of the local simulators simulating the subnets are coordinated using a loose, optimistic synchronization algorithm.

The rest of the chapter is organized as follows. Section 4.2 describes the characteristics of the network of processors upon which the distributed simulation method is based. The schemes for decomposition and allocation are described in Section 4.3. Section 4.4 describes the synchronization algorithm. Section 4.5 provides correctness proofs for the distributed simulation method.

4.2 A Distributed System Model

The underlying distributed system is assumed to be a set of autonomous processors that communicate only by sending messages over a communications network. Each

processor has some memory and some form of I/O devices. Every pair of processors is able to communicate in both directions, perhaps not directly. The unit of communication is the message which conveys a bit string from one processor to another and there is no bound on the size of the message. A message takes an arbitrary but finite amount of time to reach its destination. Messages sent between two processors are assumed to be delivered in the sequence in which they are sent.

4.3 Decomposition and Allocation

An EPTN can be viewed as a collection of sub-EPTNs, or subnets. Subnets are connected to one another by places known as *global* places. Two subnets interact by exchanging tokens at the connecting global places. A natural way of decomposing the EPTN is to treat each subnet in the EPTN as a submodel, which are in turn allocated to a network of processors to be simulated in parallel.

The problems of decomposing an EPTN into a collection of subnets and allocating the subnets to a network of processors so that the execution time of the simulation is a minimum is a non-trivial problem. The factors that must be taken into consideration include the structure of an EPTN, the number of global places connecting the subnets, the frequency of token exchange among the subnets and the topology of the network of processors. As discussed in Chapter 2, optimal solutions to this problem have been shown to be NP-complete.

Complete and rigorous solutions to the EPTN decomposition and allocation problems are beyond the scope of this thesis. Instead, a heuristic approach is adopted where the number of subnets is equal to the number of processors and the number of global places connecting the subnets is a minimum. In this approach, a given EPTN is first transformed into a weighted graph. Each transition is interpreted as a vertex. Two vertices are connected by an edge if there is at least one place connecting the two corresponding transitions and the edge is weighted according to the number of places in common. The next step is to cluster the vertices together as described below.

Let G be a weighted graph where $G = (V, E)$.

The weight of an edge e is denoted $w(e)$.

Let k be some integer where $1 < k \leq |V|$.

Find a partition of the vertices $V = \{V_1, V_2, \ldots, V_k\}$ minimizing $\sum_{e \in E} f(e)$ where

$$f(e) = \begin{cases} 0 & \text{if } e = (u, v) \text{ for } u, v \in V_i \\ w(e) & \text{otherwise} \end{cases}$$

A cluster of vertices is a subset of the vertices in the weighted graph. Given a cluster and a vertex not in the cluster, the *attraction* between the vertex and the cluster is the sum of the weights of the edges the vertex shares with vertices in the cluster. The iterative part of the algorithm is to merge the cluster-vertex with the highest attraction.

```
find initial clusters;
calculate attraction with the other vertices;
WHILE there exists a vertex that is not assigned to a cluster DO
      find the cluster-vertex pair with highest attraction;
      assign the vertex to the cluster;
      calculate attraction of this cluster with unassigned vertices;
ENDWHILE
```

Figure 4.1: The heuristic allocation algorithm

The algorithm is illustrated in Figure 4.1 The first step in the algorithm is to have k appropriate seed vertices as the initial clusters. This is done in the following way. Find two vertices with the *lowest* attraction and compute the attraction of every other vertex with these two vertices. Next add to these two vertices the vertex with the lowest attraction successively until k vertices are obtained. The attraction of every other vertex with these k vertices is also calculated. These k vertices will be used as the initial k clusters for the algorithm. At an arbitrary step in the algorithm, the cluster-vertex pair with highest attraction is chosen, and the vertex is assigned to the cluster. The attraction of this cluster with the remaining unassigned vertices is also updated. The algorithm iterates until every vertex has been assigned to some cluster. When the algorithm terminates, there will be exactly k clusters of vertices

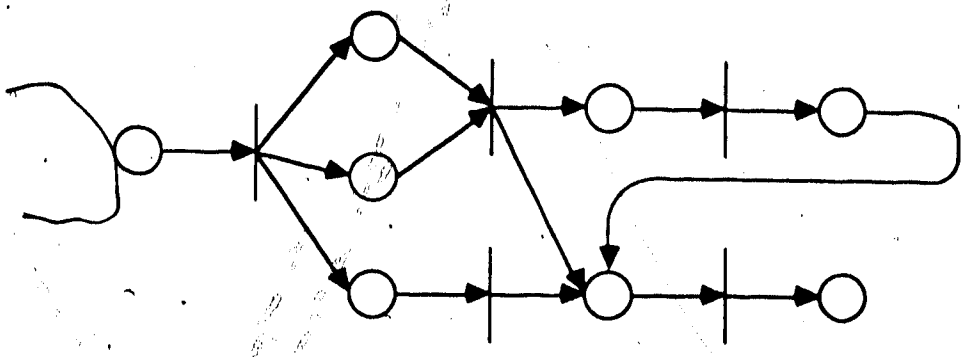


Figure 4.2: An EPTN example for allocation

corresponding to k subnets. These subnets are then each allocated statically to a processor.

The analysis of the algorithm is as follows. The selection of each of the first k vertices takes $O(ep)$ steps where e is the maximal degree in the weighted graph. As there are k vertices to be selected, the complexity of selecting the initial clusters is $O(ekp)$. After obtaining the initial k clusters, there are $p-k$ vertices left to be assigned so that the $p-k$ iterations are required. During each iteration, the cluster-vertex pair is found in $O(e(p-k))$ steps, so that the complexity of the iteration part of the algorithm is $O(e(p-k)^2)$. Thus, the complexity of the algorithm is $O(ekp + e(p-k)^2) = O(p^2)$, since $k \leq p$.

Consider the EPTN example in Figure 4.2. The weighted graph for the EPTN is shown in Figure 4.3. After applying the heuristic algorithm assuming that there are 3 processors, the allocation scheme for the EPTN is shown in Figure 4.4.

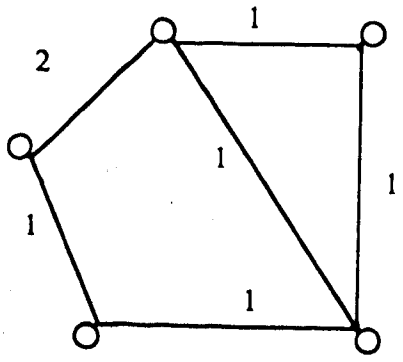


Figure 4.3: Weighted graph of the EPTN example

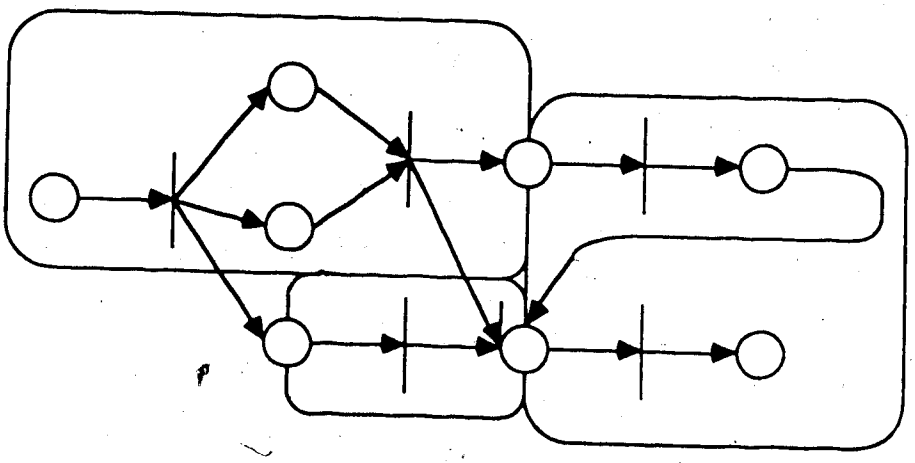


Figure 4.4: Allocation scheme for the EPTN example

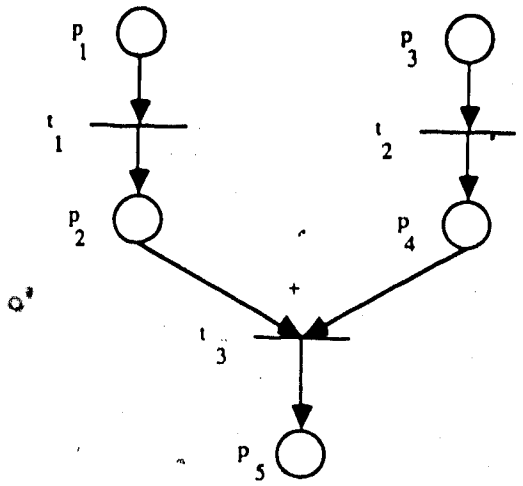


Figure 4.5: The first counter-example

4.4 The Synchronization Algorithm

Özsu [24] first proposed using a distributed approach for EPTN simulation. However, the approach was erroneous in that the synchronization of subnets was considered to be a non-issue. The simulation of an EPTN can be viewed as a sequence of events consisting of data token arrivals at source places and transition end firings. Owing to the asynchronous nature of EPTN simulation, the order of event occurrence is non-deterministic. However, the partial order in which events occur is unique as determined by the structure of an EPTN. In distributing the simulation of an EPTN, the occurrence of events has to be properly synchronized in order to maintain the partial order of event occurrence. The following two counter-examples illustrate why the synchronization of subnets is essential.

The first counter-example is shown in Figure 4.5. The "+" symbol at transition t_3 indicates that the precondition of t_3 is $\{M(p_2) > 0 \vee M(p_4) > 0\}$. That is, t_3 is enabled whenever either place p_2 or place p_4 is marked, or both places p_2 and p_4

time(sec)	$M(p_1)$	$M(p_2)$	$M(p_3)$	$M(p_4)$	$M(p_5)$	t_1	t_2	t_3
0	0	0	0	0	0	idle	idle	idle
1	0	0	0	0	0	idle	idle	idle
2	1	0	1	0	0	firing	firing	idle
3	0	1	0	1	0	idle	idle	firing
4	1	0	1	0	1	firing	firing	idle

Table 4.1: Simulation of the first counter-example

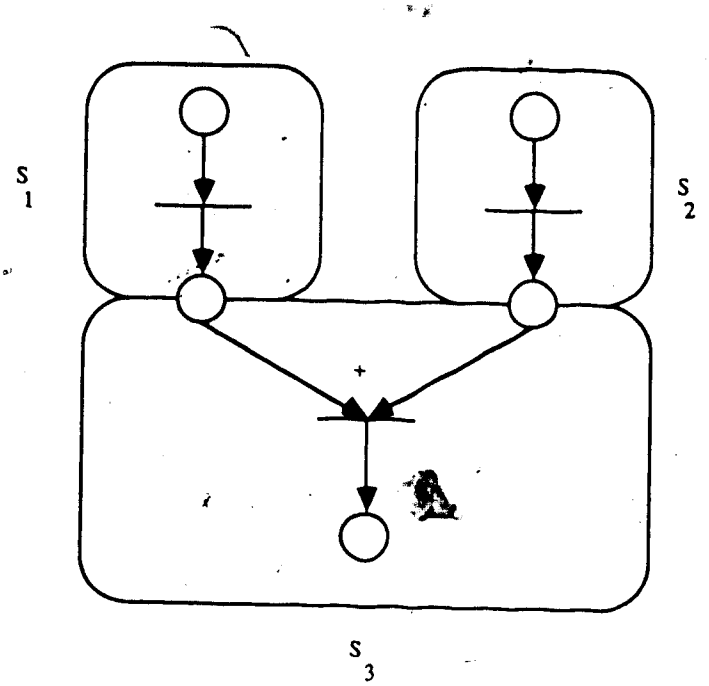


Figure 4.6: Allocation scheme of the first counter-example

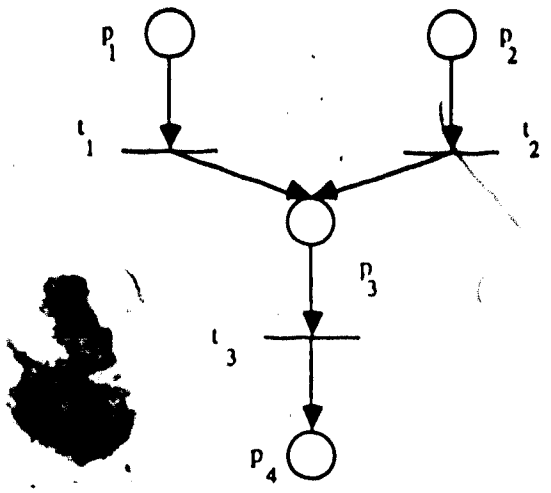


Figure 4.7: The second counter-example

are marked. p_1 and p_3 are source places while p_5 is a sink place. Assume that all transition firing times are fixed at 1 second, and the arrival rate of data token at both source places is 1 data token every two second. The simulation of the EPTN for the first 4 seconds is depicted in Table 4.1. Suppose the EPTN is decomposed into 3 subnets S_1, S_2 and S_3 to be simulated in parallel on three processors as shown in Figure 4.6. As indicated in Table 4.1, transition t_3 receives a token from both p_2 and p_4 when the simulation time is at 3 seconds. However, due to network delay and variation in the speed of the processes simulating the three subnets, there is no guarantee that subnet S_3 will always receive the two tokens from subnets S_1 and S_2 at the same instance in real time. Hence, if the simulation of the subnets is not synchronized in some fashion, then the entire simulation will no longer be valid.

The second counter-example is illustrated in Figure 4.7. As in the first counter-example, assume that all transition firing times are fixed at 1 second, and the arrival rate of data token at the source places is 1 data token every two second. Table 4.2 shows the simulation of the EPTN for the first 4 seconds. The EPTN is decomposed

time(sec)	$M(p_1)$	$M(p_2)$	$M(p_3)$	$M(p_4)$	t_1	t_2	t_3
0	0	0	0	0	idle	idle	idle
1	0	0	0	0	idle	idle	idle
2	1	1	0	0	firing	firing	idle
3	0	0	1	0	idle	idle	firing
4	1	1	0	1	firing	firing	idle

Table 4.2: Simulation of the second counter-example

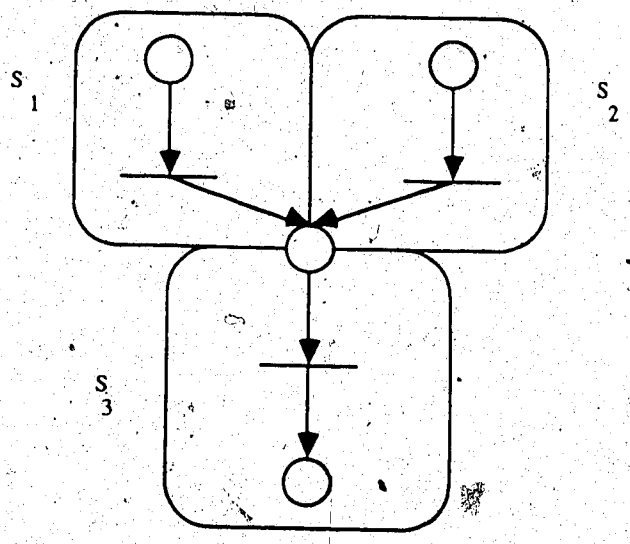


Figure 4.8: Allocation scheme of the second counter-example

into 3 subnets to be simulated by three processors as shown in Figure 4.8. Table 4.2 indicates that place p_3 receives two tokens from transitions t_1 and t_2 when the simulation time reaches 3 seconds. However, as in the first counter-example, S_3 will not always receive the two tokens from subnets S_1 and S_2 at the same moment in real time without proper synchronization.

4.4.1 Overview of the Algorithm

The synchronization algorithm that is developed in this thesis is based on an optimistic approach. Each subnet is simulated by a local simulator residing at a processor. Each local simulator maintains its own simulation time and event list. Subnets interact by token exchanges at the connecting global places. This is simulated by the exchange of messages among the corresponding local simulators. When a local simulator sends a token to another local simulator, the token is timestamped with the simulation time of the sending local simulator. The execution of the local simulators are carried out asynchronously, as long as incoming tokens at a local simulator are received in non-decreasing timestamp order. As a result some local simulators may be far ahead in their simulation while others lag behind.

The execution of a local simulator moves ahead in an optimistic way, blocking only if the local simulator has no events to process, or the simulation time of the local simulator has reached some user-specified simulation duration. That is, a local simulator assumes that no tokens will arrive from other local simulators with timestamps that are less than the simulation time of the local simulator. Note that a local simulator simulating a subnet with source places do not block since data tokens are always arriving at the source places, or the user-specified simulation duration is reached. As long as tokens arriving from other local simulators have timestamps that are greater than or equal to the simulation time of a receiving local simulator , the simulation of the receiving local simulator proceeds smoothly.

However, when a "late" token arrives at a local simulator, that is, the timestamp of the token is less than the simulation time of the local simulator, the local simulator has to be restored to a state with a simulation time that is strictly less than the

timestamp of the received token. Any side effects that may have been caused in other local simulators must be cancelled. When the simulation of the local simulator starts forward again, the "late" token will be received in the right timestamp order.

There are several reasons why an optimistic approach is chosen. Firstly, an optimistic approach allows the simulation of the local simulators to move ahead without blocking, maximizing the amount of parallelism achievable. Secondly, if a EPTN that is being simulated does not deadlock, the distributed simulation of the subnets of the EPTN also does not deadlock. This will be proven in Section 4.5. In addition, the operations of checkpointing and rollback are simple and straightforward as the state of a subnet at any moment is is described by its marking and transition firings that are in progress.

The synchronization algorithm consists of three major parts: rollback operation, memory management and termination detection. Each of these parts will be described in detail next.

4.4.2 Rollback Operation

The state of a local simulator at any instance is characterized by the markings at the places and the remaining firing times of the transitions whose firings are in progress. The synchronization algorithm requires the state of a local simulator to be saved from time to time, so that during a rollback, the local simulator can be restored to a previous saved state. The information about interaction with other local simulators also has to be saved so that side effects caused at other local simulators can be cancelled during a rollback.

Let E be an EPTN with subnets $S_1, S_2, \ldots, S_p$ which are simulated by local simulators $LS_1, LS_2, \ldots, LS_p$ respectively. The saved states of LS_k are given by $\{state_1^{LS_k}, state_2^{LS_k}, \ldots\}$ and the simulation time when state $state_u^{LS_k}$ was saved is denoted $st(state_u^{LS_k})$. The states of a local simulator are saved in a *state queue*. The saving of states has to be carried out every m events, where an event is one of transition end firing, data token arrival at a source place or token arrival at a global place. The size of m determines the speed of the rollback operation: the smaller

the value of m is, the faster the rollback operation can be carried out. However, setting a small value for m also means higher run-time memory overhead for saving states.

A local simulator interacts with other local simulators through the exchange of tokens at the connecting global places. The information about token departures is kept in a *history queue* for each global place. The saved histories for global place gp_i is given by $\{hist_1^{gp_i}, hist_2^{gp_i}, \ldots\}$ and the departure time of a history $hist_v^{gp_i}$ is denoted $dt(hist_v^{gp_i})$.

Figure 4.9 shows the basic structure of a local simulator. The five major components of a local simulator are:

1. A *simulation clock* which holds the simulation time for the local simulator.
2. A *current state* which holds the current state information of the subnet being simulated.
3. An *event list* which holds the event notices of the local simulator. Each event has a simulation time field and an event information field.
4. A *state queue* which holds the saved states of the local simulator. Each state consists of a simulation time field and a state information field.
5. A *history queue* which holds the saved histories for each global place. Each saved history consists of a departure time field and history information field.

Consider the situation of the local simulator in Figure 4.9. Assume that all time values are in seconds. The simulation time of the local simulator is at 200. If a token arrives at the local simulator with timestamps greater than or equal to 200, then the token will simply be enqueued to the event list and the execution of the local simulator continues as usual. Figure 4.10 depicts the situation when a token with timestamp 300 is received and enqueued to the event list.

There are two types of rollbacks: *primary* and *secondary*. A primary rollback at a local simulator is caused by the arrival of a token at the local simulator where

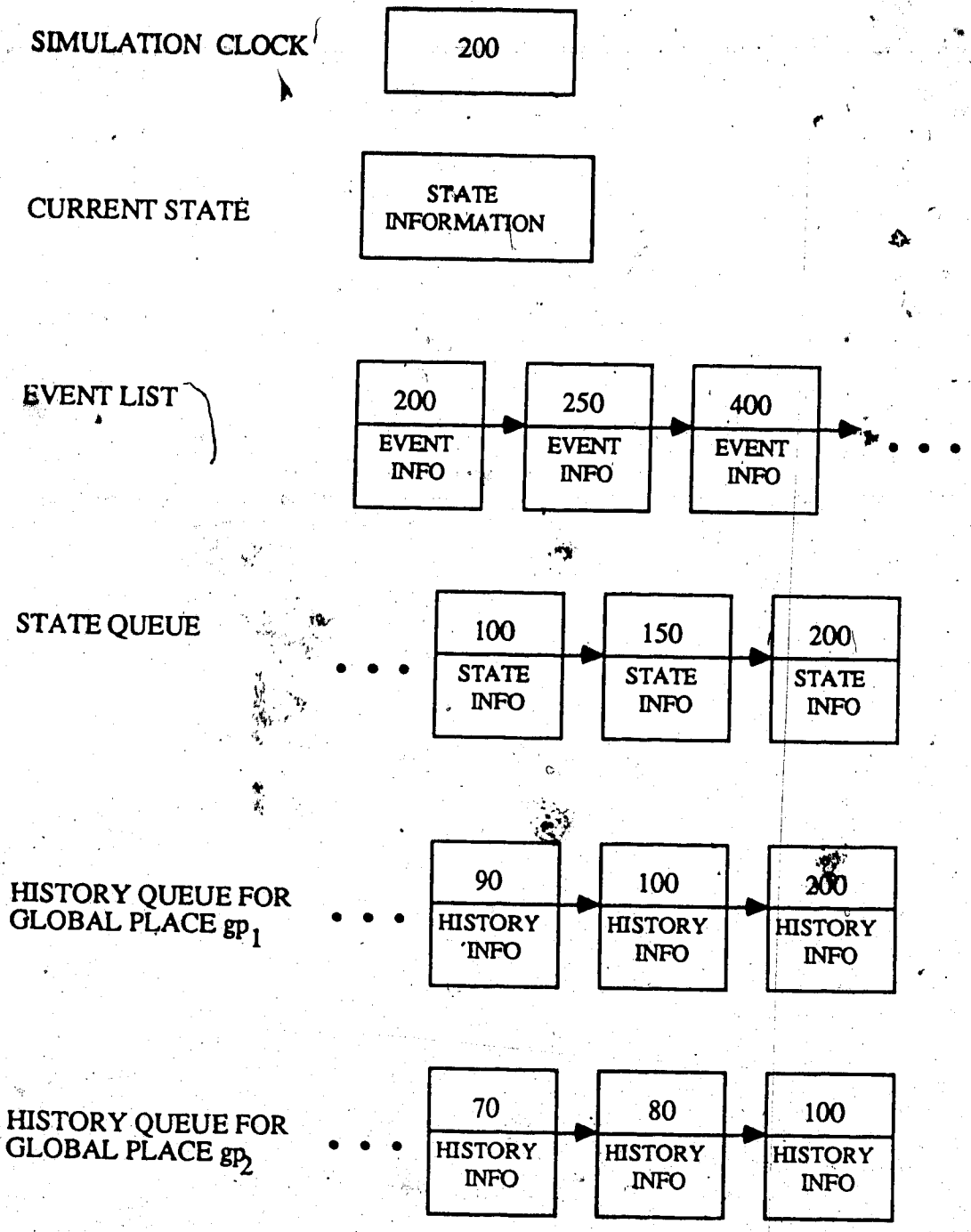


Figure 4.9: Structure of a local simulator

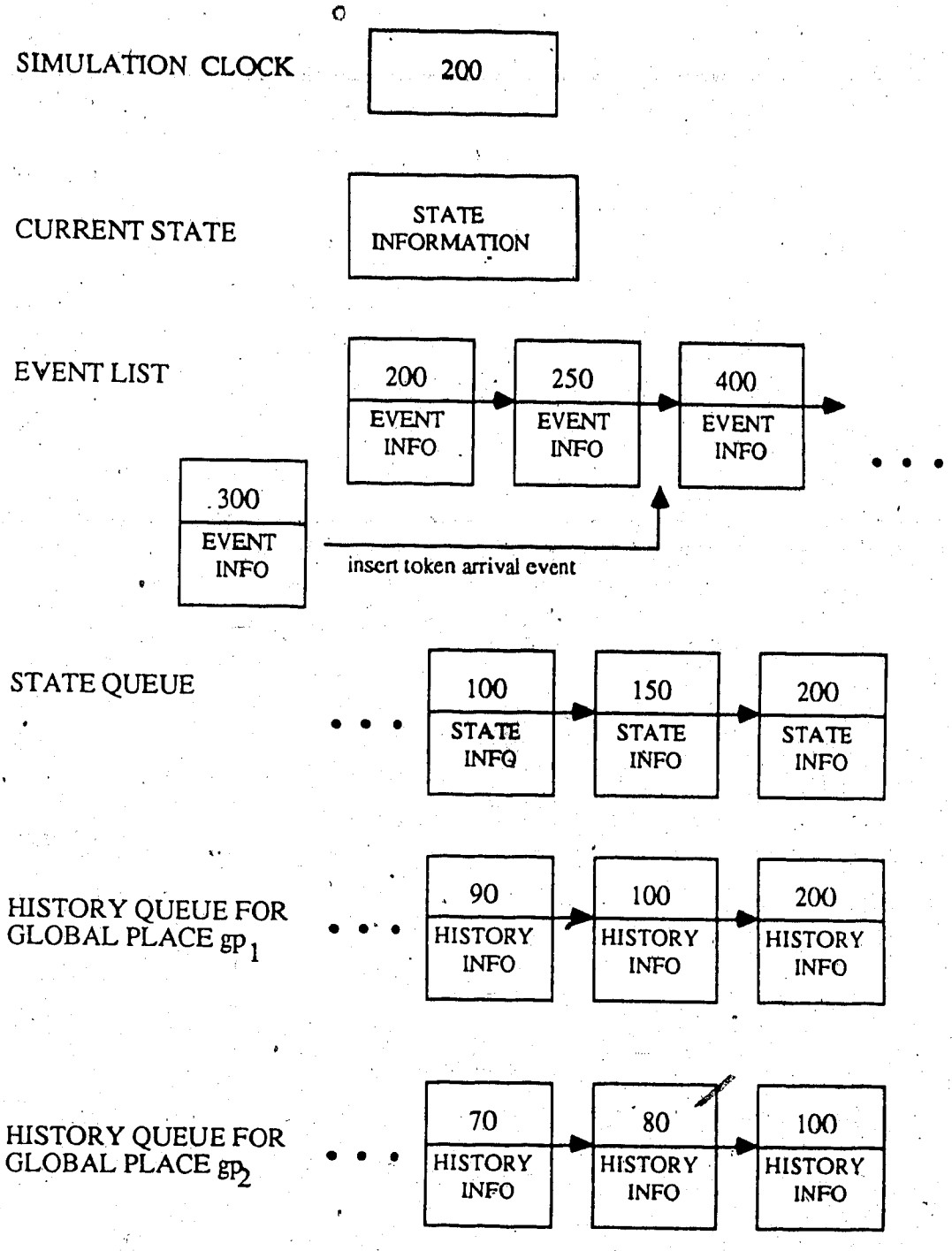


Figure 4.10: Arrival of a token at a local simulator

the timestamp of the token is less than the simulation time of the local simulator. During a primary rollback, all side effects at other local simulators must be undone. Local simulators affected by a primary rollback of a local simulator are sent *cancellation notices*, causing secondary rollbacks. The timestamp of a cancellation notice cn is denoted $ts(cn)$.

Primary Rollbacks

Consider a token to arriving at a local simulator LS_k. Let $clk(LS_k)$ denote the simulation clock of LS_k. Let $ts(to)$ denote the timestamp of token to. A primary rollback is initiated if $ts(to) < clk(LS_k)$. LS_k is first restored to a saved state $state_u^{LS_k}$ with simulation time that is strictly less than $ts(to)$ where

$$st(state_u^{LS_k}) = \max_w\{st(state_w^{LS_k})\} < ts(to)$$

That is, $state_u^{LS_k}$ is the most recent saved state. Since all saved states with simulation times greater than $ts(to)$ are made invalid by the arrival of token to, these saved states can be discarded.

If tokens have been sent out to other local simulators where the timestamps of the tokens are greater than $ts(to)$, then cancellation notices have to be sent to these subnets since these tokens are invalidated by the arrival of token to. Thus, for each global place gp_i, if there is a saved history with departure time that is greater than $ts(to)$, then a cancellation notice cn has to be sent to the subnets connected to gp_i where

$$ts(cn) = \min_w\{hist_w^{gp_i}\} > ts(to)$$

That is, the timestamp of cancellation notice cn is assigned the smallest departure time that is greater than $ts(to)$. Saved histories with departure times that are greater than $ts(to)$ are discarded since they are made invalid by the arrival of token to.

During the simulation time interval $[st(state_u^{LS_k}), ts(to)]$ some tokens might have been sent to other local simulators. These tokens are not invalidated because their

timestamps are less than $ts(to)$. Thus, when the execution of LS_k is started again from the restored state, the sending of tokens to other local simulators is inhibited until $clk(LS_k) > ts(to)$. Otherwise the sending of tokens in the simulation time interval $[st(state_u^{LS_k}), ts(to)]$ will be sent twice.

Consider the situation of a local simulator with simulation time 100 receiving a token with timestamp 90 as shown in Figure 4.11. As the timestamp of the token is less than the simulation time, a primary rollback is initiated. First, the local simulator is restored to the state with simulation time 80. Since those states with simulation time greater than 90 are no longer valid, they can be discarded. Next, a new event list is constructed from the restored state since the old event list is no longer valid. For each global place, if there is a saved history with departure time greater than 90, a cancellation notice has to be sent to the local simulators connected to the global place. In this case, a cancellation notice is sent with timestamp 95 to local simulators connected to global place gp_1. The execution of the local simulator is resumed without sending out tokens until its simulation time exceeds 90 so that tokens that were sent in the time interval [80, 90] will not be sent twice.

Secondary Rollbacks

Consider a cancellation notice cn arriving at a local simulator LS_k. If $clk(LS_k) \leq ts(cn)$, a secondary rollback is not initiated because the arrivals of the invalid tokens have not been processed yet. In this case, the arrivals of the invalid tokens are simply removed from the event list.

A secondary rollback at LS_k is initiated only if $clk(LS_k) > ts(cn)$. During a secondary rollback, local simulator LS_k has to be restored to a state $state_u^{LS_k}$ with simulation time that is strictly less than $ts(cn)$ where

$$st(state_u^{LS_k}) = \max_w \{st(state_w^{LS_k})\} < ts(cn)$$

That is, $state_u^{LS_k}$ is the most recent saved state prior to the timestamp of the cancellation notice. Saved states with simulation times that are greater than $ts(cn)$

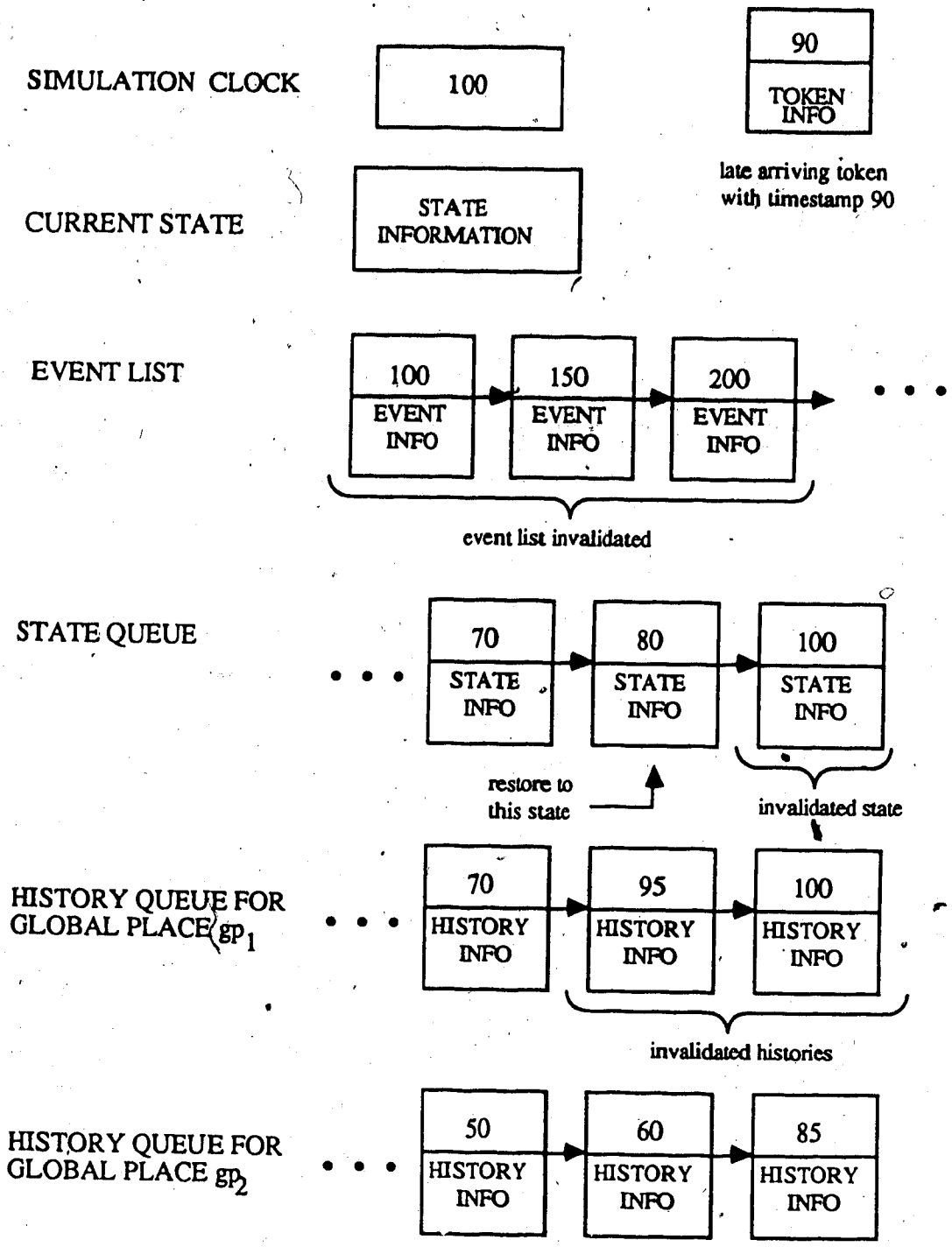


Figure 4.11: Primary rollback at a local simulator

and saved histories with departure times that are greater than $ts(cn)$ are discarded since they are made invalid by the arrival of cancellation notice cn.

If tokens have been sent to other local simulators where the timestamps of the tokens are greater than $ts(cn)$, then more cancellation notices have to be sent to the affected subnets. For each global place gp_i, if there is a saved history with departure time that is greater than $ts(cn)$, then a new cancellation notice cn' has to be sent to the subnets connected to gp_i where the

$$ts(cn') = \min_w\{hist_w^{gp_i}\} > ct(cn)$$

That is, the timestamp of the cancellation notice cn' is set to the smallest departure time that is greater than $ct(cn)$. LS_k is then simulated again starting from the restored state where the sending of tokens to other subnets is also inhibited, as in a primary rollback, until $clk(LS_k) > ts(cn)$.

Consider the situation of a local simulator with simulation time 100 receiving a cancellation notice with timestamp 80 as shown in Figure 4.12. Since the timestamp of the cancellation notice is less than the simulation time, a secondary rollback is initiated. The local simulator is first restored to the saved state with simulation time 70 and a new event list is constructed from the restored state. For each global place, if there is a saved history with departure time greater than 80, a cancellation notice has to be sent to the local simulators connected to the global place. In this case, a cancellation notice is sent with timestamp 90 to local simulators connected to global place gp_1. The execution of the local simulator is resumed without sending out tokens until its simulation time exceeds 80 so that tokens that were sent in the simulation time interval $[70, 80]$ will not be sent twice.

4.4.3 Memory Management

The frequency of state saving is a parameter that decides how fast rollbacks can be carried out. However, the more frequent the states of a local simulator is saved, the higher is the run-time memory overhead.

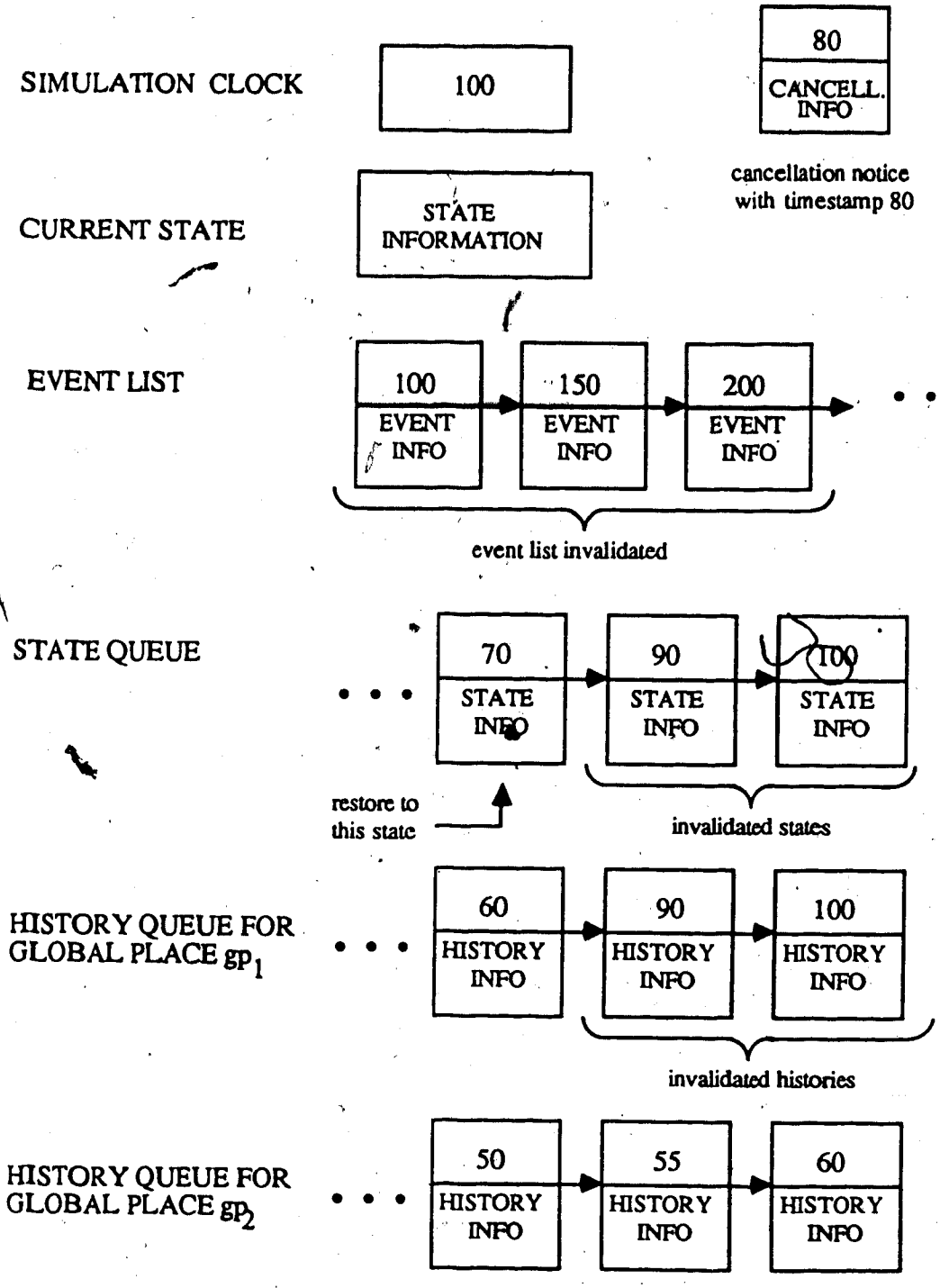


Figure 4.12: Secondary rollback at a local simulator

The states of a local simulator need not be saved all the way back to the beginning of simulation. Define *minimum simulation time* (MST) to be the minimum of:

1. The simulation times at the local simulators.
2. The timestamps of tokens in transit (i.e., tokens that have been sent but not received).
3. The cancellation times of cancellation notices in transit (i.e., cancellations that have been sent but not received).

As will be proven in Section 4.5, no subnet can roll back to a point prior to the MST. Thus, saved states with simulation times less than the MST can be safely discarded in order to free up memory.

The value of the MST is calculated in the following way. At every time interval of n seconds in real time, each local simulator calculates the minimum of:

1. its simulation time,
2. timestamps of tokens sent but not acknowledged, and
3. timestamps of cancellation notices sent but not acknowledged,

and broadcasts this value to the other local simulators. The MST is simply obtained by taking the minimum of these values. The value of MST obtained in this way will be slightly out of date due to the delay in the exchange of messages among the local simulators.

The frequency of the calculation of the MST is a trade-off: high frequency produces an up-to-date version of the MST and more efficient memory utilization, but, also at the same time, uses up processor time and network bandwidth, and thus slowing progress.

4.4.4 Termination Detection

It is assumed that an EPTN is required to be simulated for a time period $[0, SD]$, where SD is any positive real number representing the simulation duration. Simulation terminates whenever one of the two conditions hold:

1. The simulation times of *all* local simulators have reached SD.
2. The simulation times of *some* local simulators have reached SD; the other local simulators are blocked waiting to received tokens.

The simulation time of a local simulator is set to an arbitrary large number Z whenever the simulation time of a local simulator reaches SD, or the local simulator is blocked waiting to receive tokens. When MST reads Z, the simulation times of all local simulators must read Z and no tokens or cancellation notices can be in transit. Thus, termination is detected whenever MST reads Z.

4.4.5 The Algorithm

The overall synchronization algorithm for a local simulator LS_k is given next. Note that the periodic sending of simulation time messages to other local simulators is not included.

1. Initialize local simulator LS_k. $clk(LS_k) \leftarrow 0$.
2. If LS_k receives a token to where $ts(to) \geq clk(LS_k)$, schedule the arrival of to.
3. If LS_k receives a token to where $ts(to) < clk(LS_k)$, a primary rollback is initiated. Restore LS_k to a previous saved state. Schedule the arrival of to. Send out cancellation notices to invalidate tokens sent earlier. Discard invalid saved states and histories. Execute LS_k without sending tokens until $clk(LS_k)$ exceeds $ts(to)$.
4. If LS_k receives a cancellation notice cn where $ts(cn) \geq clk(LS_k)$, remove the

5. If LS_k receives a cancellation notice cn where $ts(cn) < clk(LS_k)$, a secondary rollback is initiated. Restore LS_k to a previous saved state. Send out more cancellation notices to invalidate tokens sent earlier. Discard invalid saved states and histories. Execute LS_k without sending tokens until $clk(LS_k)$ exceeds $ts(cn)$.

6. If LS_k receives a simulation time message, calculate the value of MST. Free up memory by discarding saved states and histories with simulation times less than MST.

7. Execute LS_k and update $clk(LS_k)$.

8. When termination criterion is reached, stop. Otherwise go back to step 2.

4.5 Proof of Correctness

In order to show that the proposed distributed simulation method correctly simulates an EPTN E with subnets $S_1, S_2, \ldots, S_p$ where E is assumed to be deadlock-free and each subnet S_k is simulated by a local simulator LS_k, for $k = 1, 2, \ldots, p$, it is required to prove that:

1. The partial order in which events occur in E is preserved across the local simulators.

2. The execution of the local simulators does not deadlock.

3. The execution of the local simulators terminates.

Theorem 4.1 (Partial Order Preservation): The execution of the local simulators $LS_1, LS_2, \ldots, LS_p$ preserves the partial order of event occurrence.

Proof: The partial order of event occurrence is preserved across the local simulators if the causality of event occurrence across the local simulators is maintained. Independent events across the local simulators can be simulated in parallel even if

they occur at different simulation times. The causality of event occurrence manifests itself in the exchange of tokens among the local simulators. For example, two local simulators that do not exchange tokens are independent of each other so that there is no causality of events between them. As long as tokens are received at each local simulator where the timestamps of the tokens are greater than or equal to the simulation time of the local simulator, the causality of events is maintained. The synchronization algorithm ensures that a token received at a local simulator always has a timestamp that is greater than or equal to the simulation time of the local simulator. Hence, the partial order of event occurrence is preserved.

□

Theorem 4.2 (Absence of Deadlock): The execution of the local simulators $LS_1, LS_2, \ldots, LS_p$ does not deadlock.

Proof: The execution of the local simulators $LS_1, LS_2, \ldots, LS_p$ deadlocks if the following two conditions hold:

1. All local simulators are blocked waiting to receive tokens where the simulation times of the local simulators are less than SD.
2. There are no tokens or cancellation notices in transit.

Suppose the execution of the local simulators $LS_1, LS_2, \ldots, LS_p$ is deadlocked. A local simulator only blocks if its simulation time has reached SD, or it is waiting to receive tokens. There are three cases in which all the local simulators can be blocked simultaneously.

The first case is when the simulation times of all local simulators have reached SD. This case cannot be true because it contradicts the assumption that simulation times of the local simulators are less than SD.

The second case is when the simulation times of all local simulators are less than SD and all local simulators are waiting to receive tokens. This case represents a deadlock in EPTN E which also cannot be true since the EPTN E is assumed to be deadlock-free.

The third case is when the simulation times of some local simulators have reached SD while the others are waiting to receive tokens. This case also cannot be a deadlock because it represents proper termination of the simulation.

Hence, the execution of the local simulators $LS_1, LS_2, \ldots, LS_p$ cannot deadlock.

□

Theorem 4.3 (Termination): The execution of local simulators $LS_1, LS_2, \ldots, LS_p$ is guaranteed to make progress and terminate.

Proof: Consider a snapshot in the execution of the local simulators. Let clk_k' denote the simulation time of local simulator LS_k during this snapshot, for $k = 1, 2, \ldots, p$. Let the set of tokens in transit be $\{to_1', to_2', \ldots\}$ and the set of cancellation notices in transit be $\{cn_1', cn_2', \ldots\}$. Let τ' denote the MST during the snapshot where

$$\tau' = min\{clk_k' \cup ts(to_x') \cup ts(cn_y')\}$$

and

$$\tau' \neq Z$$

for $k = 1, 2, \ldots, p$, $x = 1, 2, \ldots$ and $y = 1, 2, \ldots$. No local simulator can rollback to a time earlier than τ' because:

1. A local simulator cannot generate tokens with timestamps earlier than τ' since $clk_k' \geq \tau', k = 1, 2, \ldots, p$.
2. A token in transit cannot trigger primary rollbacks earlier than τ' since $ts(to_x') \geq \tau'$.
3. A cancellation notice in transit cannot trigger secondary rollbacks earlier than τ' since $ts(cn_y') \geq \tau'$.

Consider a next snapshot in the execution of the local simulators where the tokens and rollback notices in transit in the previous snapshot have been received by the local simulators. Let clk_k'' denote the simulation time of local simulator LS_k during this snapshot, for $k = 1, 2, \ldots, p$. Let the new set of tokens in transit be $\{to_1'', to_2'', \ldots\}$, and the new set of cancellation notices in transit be $\{cn_1'', cn_2'', \ldots\}$. Let τ'' denote the new MST where

$$\tau'' = min\{clk_k'' \cup ts(to_x'') \cup ts(cn_y'')\}$$

Between the snapshots, a local simulator either was simulated forward (possibly after a series of rollbacks due to the arrivals of "late" tokens and/or cancellation notices) or it was blocked waiting to receive tokens, so that the simulation time of the local simulator was set to Z between the two snapshots. The local simulators cannot be all blocked simultaneously since E is assumed to be deadlock-free. Thus, $\min\{clk_k''\} > \tau'$, for $k = 1, 2, \ldots, p$.

During a primary or secondary rollback at a local simulator , the sending of tokens to other local simulators is inhibited until the simulation time of the local simulator exceeds the time of the rollback. Hence, only tokens with timestamps greater than τ' can be generated by the local simulator between the snapshots because a local simulator cannot rollback to a point prior to τ'. Thus, $\min\{ts(to_x'')\} > \tau'$.

Similarly, only cancellation notices with timestamps greater than τ' can be sent between the snapshots. Thus, $\min\{ts(cn_y'')\} > \tau'$.

Since $\tau'' = \min\{clk_k'' \cup ts(to_x'') \cup ts(cn_y'')\}$, it follows that $\tau'' > \tau'$, and the MST has increased.

Hence, the value of MST will always increase and when either the simulation times at all the local simulators have reached SD, or the simulation times at some of the local simulators have reached SD while the other local simulators are blocked waiting to receive tokens, the value of MST reaches Z and the simulation terminates.

□

Chapter 5

Implementation Details

5.1 Introduction

A distributed simulator was built in order to evaluate the performance of the distributed simulation method described in Chapter 4. The simulator was implemented using a number of Sun Workstations connected by a 10-Mbit Ethernet local area network.

This chapter describes the simulator and the principles of software engineering applied in the construction of the simulator. The rest of the chapter is organized as follows. Section 5.2 outlines the hardware and software used in the construction of the simulator. Section 5.3 describes in detail the structure of the simulator.

5.2 Hardware and Software

The hardware configuration consists of eight Sun Workstations interconnected by a 10-Mbit Ethernet local area network. Each of the Sun Workstations has 4 Mbytes of memory and an execution rate of 1 MIPS. The 4.3BSD UNIX operating system provides the environment for programming, file system support, editing and compilation facilities. The native language of UNIX, the C programming language, is chosen for implementing the simulator because of its simplicity, power, speed and the excellent library provided under UNIX.

The capability for performing experiments in distributed processing is provided by the Virtual Tree Machine (VTM) environment [23] developed at the University of Alberta. VTM provides facilities for creating a *virtual* distributed executing environment by replacing physical processors with processes and communications links with virtual communications paths. Each process is provided with the full capabilities of the UNIX operating system. To create a virtual distributed system, a user simply has to create a configuration file describing the number of processors and the interconnections among the processors in the distributed system. VTM is built on top of the UNIX interprocess communications primitives which support the use of reliable two-way communication channels based on the client/server model. VTM is implemented as a collection of *servers*. Each server resides on a Sun Workstation. These servers create processes by recursively traversing an internal representation of the user-specified configuration. As soon as the processes are created, and communications channels established, the virtual distributed virtual distributed system can start running.

The advantages of using VTM are [23]:

1. The ability to design and implement parallel algorithms quickly and efficiently without resorting to specialized architectures.

2. Faster turnaround time for designing and testing parallel algorithms.

3. The availability of debugging and performance monitoring tools provided by UNIX.

The major disadvantage of VTM is the difficulty in obtaining accurate timing results due to operating system overhead and bus contention. In order to minimize the effect of bus contention with other users, experiments were conducted overnight and on weekends.

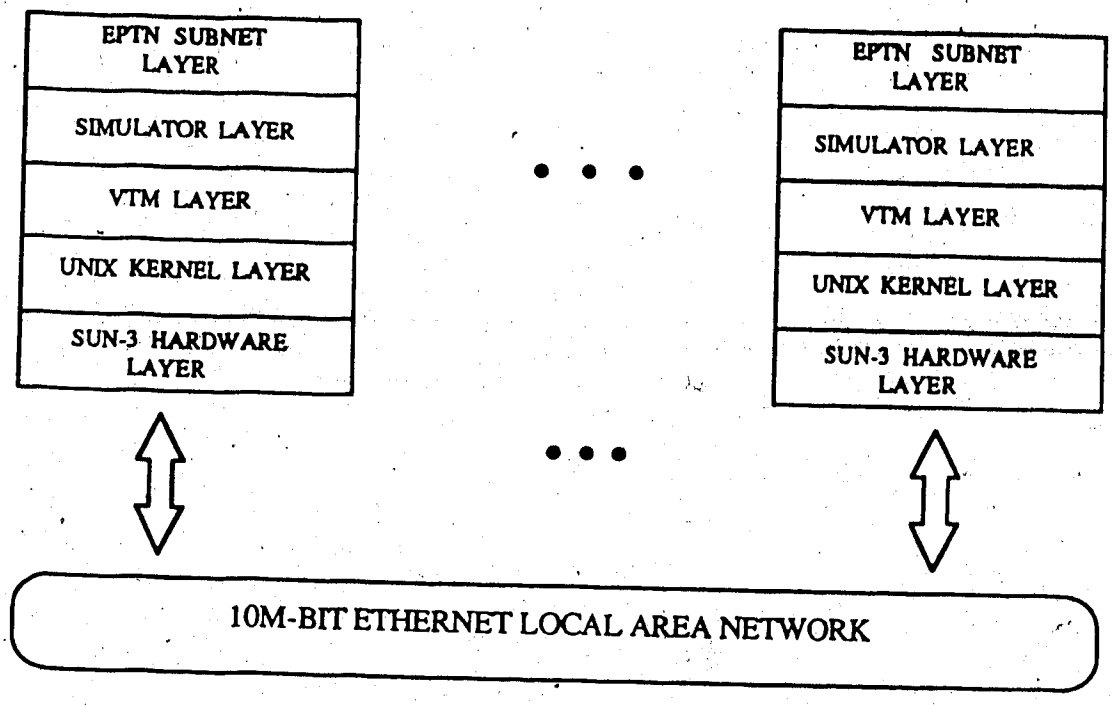


Figure 5.1: The distributed simulator

5.3 Structure of the Distributed Simulator

The first step in the design of the distributed simulator was to identify its subsystems, and to decompose it into corresponding modules. The second step was to define the interaction of the modules. The subsystems were identified as the *EPTN subnet layer*, the *simulator layer* and the *VTM layer*. Their interaction is hierarchical in that each layer provides software support to the layer above it. Any functions defined within a layer for its own use are private to it. The details about how a layer provides the support for the next layer are also private. Such a decomposition scheme obeys the principle of information hiding [28] and the interfaces between the layers are well-defined.

The overall structure of the simulator is illustrated in Figure 5.1. The top three layers are portable since they are completely written in the C language. The last two layers, which are the UNIX kernel layer and the Sun Workstation hardware layer, are not portable.

The EPTN subnet layer contains the code which describes a subnet of an EPTN to be simulated. Information about the connectivity of the subnets in the EPTN is also included.

The simulator layer implements the core of the distributed simulator. In addition to simulating the subnet described in the EPTN subnet layer, it is also responsible for managing the exchange of messages with other subnets, maintaining synchronization and collecting statistics.

The VTM layer is a collection of routines which implement the the VTM environment. This layer is responsible for handling the exchange of messages among the subnets during simulation.

The function of each of these three layers is described in detail next.

5.3.1 The EPTN Subnet Layer

The EPTN subnet layer constitutes the user-interface component of the simulator. The input to this layer is a C program describing a subnet of an EPTN

simulated. The C program consists of two parts: a declaration part and a simulation part. The declaration part contains the declaration of the places and transitions in the subnet, and the connectivity of the subnet with other subnets in the EPTN. The declaration of a place includes the following:

1. The input and output transitions of the place.
2. Specifying whether the place accepts simple or data tokens.
3. Specifying whether the place is global place.
4. Specifying whether the place is a source place.
5. Specifying whether the place is a sink place.
6. Specifying whether the place is associated with a resource.

The declaration of a transition includes the following:

1. The input and output places of the transition.
2. The firing function of the transition.
3. The precondition of the transition.
4. The procedure of the transition.
5. The postcondition of the transition.

The information about the connectivity of the subnets in the EPTN is coded in an adjacency matrix where the number of rows and the number of columns are equal to the number of subnets. The (i, j)-entry in the matrix is set to 1 if the i-th subnet is connected to the j-th subnet; otherwise it is set to zero.

The simulation part consists of simulation control statements. These statements provide facilities for:

1. Error checking in the construction of a subnet.
2. Setting the seed value for the random number generator in the simulator.

3. Executing a simulation.

4. Tracing a simulation.

5. Reporting of simulation outputs.

Owing to the distributed nature of the simulator, the tracing of simulation and the reporting of simulation outputs are applicable to each subnet simulating on a workstation only. A global trace facility of the entire simulation is not available.

The EPTN subnet layer was implemented as a simple means for the purpose of carrying out experiments. It can be easily connected to SIMLANG [26] for a better user-interface. SIMLANG is a PASCAL-like programming language developed for the purpose of describing an EPTN model.

5.3.2 The Simulator Layer

The simulator layer serves as a specialized operating system for simulation. It provides facilities for the instantiation of a subnet, scheduling of events and communication with other subnets. It also handles the rollback operation, garbage collection, and calculation of MST.

The simulator layer is made up of four components: a place/transition component, a scheduler component, an inter-subnet communications component and a synchronization component.

The place/transition component reads the subnet described in the EPTN subnet layer and sets up the necessary data structures for representing the subnet. It is responsible for simulating the movement of tokens between the places and transitions. Statistics collection and reporting for resource places and transition firings are also performed by this component.

The scheduler component maintains the simulation clock for the simulation. It also keeps an event list for the scheduling of transition end firings, data token arrivals at the source places and token arrivals at the global places.

The inter-subnet communications component is responsible for the exchange of messages with other subnets during simulation. The messages that are exchanged

with other subnets are tokens, cancellation notices and simulation times. The exchanging of messages with other subnets is achieved by calls to the VTM layer. The information about the connectivity of the subnets described in the EPTN subnet layer is read and kept in this component.

The synchronization component takes care of the tasks of state saving, history saving, rollback operation, garbage collection, and initiation of sending of cancellation notices and simulation time.

5.3.3 The VTM Layer

The VTM layer is essentially a collection of routines which implement the VTM environment. The interface to this layer from the simulator layer is made up of user procedures for handling connection establishment, connection initialization, exchange of messages, interrupt handling and providing information on the configuration and layout of the virtual environment being implemented. The VTM environment consists of virtual processors, which are in reality UNIX processes, with virtual interconnections, which are in reality UNIX stream-socket connections over Ethernet, upon initialization.

The basic VTM support routines include [23]:

1. A set of initialization routines for initializing the VTM environment. The interconnections among the virtual processors are contained in a configuration file. The virtual processors are assigned unique ids based on their appearance in the configuration file.

2. A set of communication routines for sending and receiving of messages to and from a specified virtual processor.

3. A set of polling routines for the the checking of the status of messages from the neighbouring virtual processors.

4. A set of interrupt handling routines for the enabling and disabling of interrupts, the holding and releasing of interrupts, and the specification of interrupt

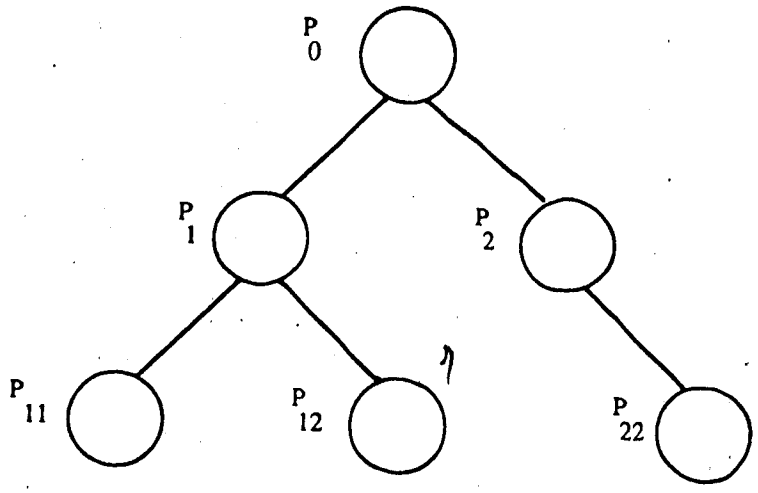


Figure 5.2: A process tree

handlers.

5. A set of reconfiguration routines allowing the virtual distributed system to be reconfigured dynamically at run time.

The following VTM example illustrates the use of the VTM environment. Consider the creation of a VTM to execute the configuration of processes depicted in Figure 5.2. This configuration is mapped onto hardware in the following way:

P_0 on *sun000*

P_1 on *sun001*

P_2 on *sun002*

P_{11} on *sun003*

P_{12} on *sun004*

P_{22} on *sun005*

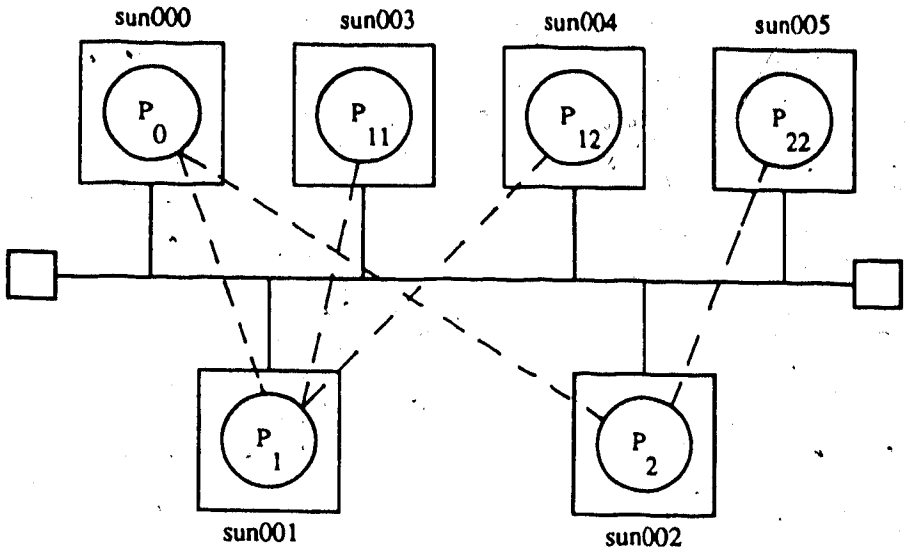


Figure 5.3: Mapping of processor tree to hardware configuration

The above configuration maps the processes on Sun Workstations (*sun000* to *sun005*). The mapping between the virtual machine and the physical hardware layer is described the configuration file.

Process P_0 is first created by *sun000* by reading the configuration file. Next, process P_0 sends a service request to *sun001* to start up the process P_1. Process P_1 then receives configuration information from process P_0 and it in turn sends service requests to *sun003* and *sun004* to start up processes P_{11} and P_{22}. When the left branch of the process tree is created, process P_0 sends a service request to *sun002* to start up process P_2. P_2 then receives configuration information from P and start up P_{22}. The VTM created is shown in Figure 5.3.

Chapter 6

Experimental Results

6.1 Introduction

In this chapter, the results of some preliminary experiments conducted for evaluating the performance of the distributed simulation method are presented. Various representative EPTNs were chosen as test cases:

1. An EPTN with a serial structure.
2. An EPTN with a conflict place.
3. An EPTN with an or-postcondition structure.
4. An EPTN with an or-precondition structure.
5. An EPTN with a feedback structure.

For each EPTN, a number of simulations were run using different allocation schemes of the EPTN to workstations, and measured the time taken for each simulation to execute. In order to compare the performance of the distributed simulation method to that of the sequential simulation method, each EPTN was also simulated on a single workstation.

The performance metric that is used to compare all experimental results is the *speedup* which is defined as,

$$speedup = \frac{execution\ time\ using\ 1\ workstation}{execution\ time\ using\ p\ workstations}$$

where $p = 2, 4, 8$.

In addition, the *rollback fraction* is also used as a performance measure. It is defined as,

$$rollback\ fraction = \frac{number\ of\ rollbacks}{number\ of\ message\ transmissions}$$

The rollback fraction measures the amount of "useful" computation performed by each simulation.

In order to minimize the occurrence of rollbacks, the state of an EPTN was saved after every event. The calculation of the MST was carried out every second in real time.

To give an accuracy of the results, 95% confidence intervals are computed for the mean execution time $\overline{E}$ as:

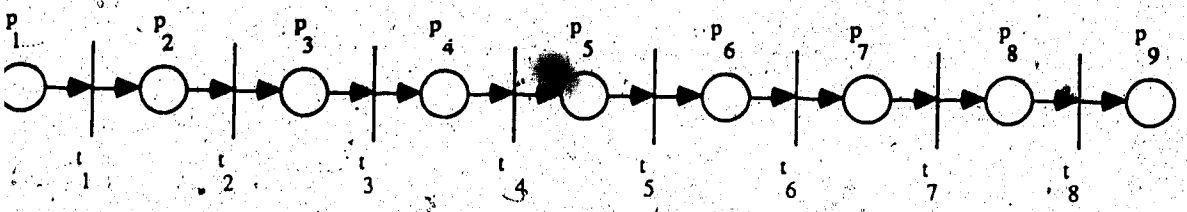
$$\overline{E} \pm 1.96\sqrt{\frac{s^2}{n}}$$

where s^2 is the sample variance obtained from the simulation runs and n is the total number of independent simulation runs.

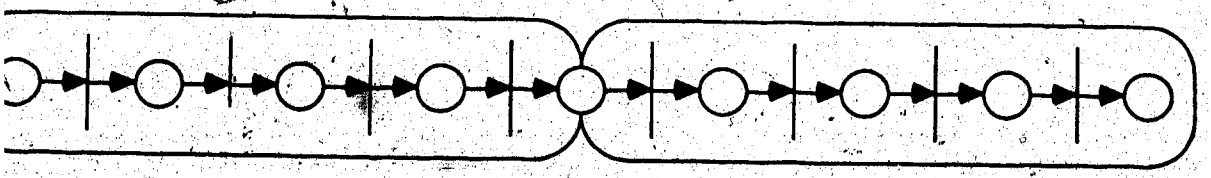
6.2 EPTN with Serial Structure

The first experiment involved an EPTN with a serial structure having 9 places and 8 transitions as shown in Figure 6.1. p_1 and p_9 are source and sink places respectively. The data token arrival rate at source place p_1 is 1 data token per second. All transition firing times were fixed at 1 second. The EPTN was simulated for 1000 seconds.

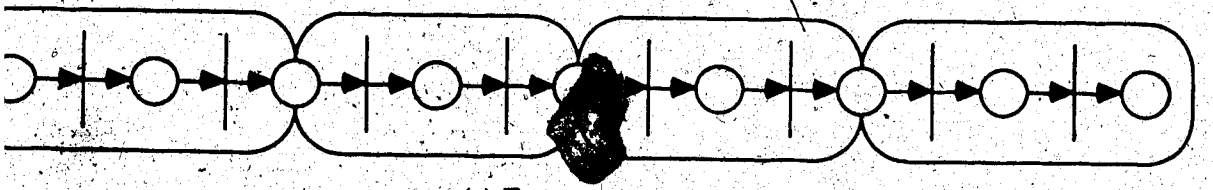
The different allocation schemes of the EPTN to workstations are also shown in Figure 6.1. Each enclosed group of places and transitions corresponds to a subnet to be simulated on a workstation. A place that is common to two enclosed groups (see Figure 6.1) represents a global place connecting the corresponding two subnets.



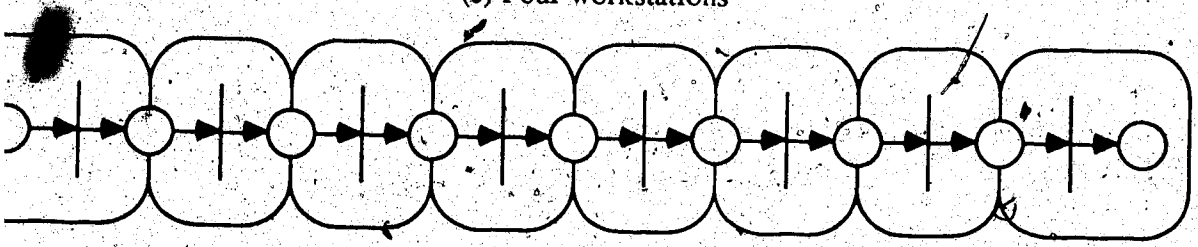
(a) Serial EPTN



(b) Two workstations



(c) Four workstations



(d) Eight workstations

Figure 6.1: Allocation schemes for an EPTN with a serial structure

Workstation(s)	*Exec. Time*	*Messages*	*Rollbacks*	*Cancellations*
1	84.4 ± 0.37	-	-	-
2	54.1 ± 0.34	996.1	4.8	0.0
4	29.6 ± 0.38	3004.1	28.8	0.0
8	26.4 ± 0.37	7056.0	101.6	0.0

Workstation(s)	*Speedup*	*Rollback Fraction*
1	1.00	-
2	1.56	0.00
4	2.85	0.01
8	3.20	0.01

Table 6.1: Performance results for EPTN with serial structure

Table 6.1 shows the performance results obtained. The *Messages* column represents the number of messages transmitted while the *Rollbacks* and *Cancellations* columns represent the total number of rollbacks and total number of cancellation notices sent respectively. The results obtained are encouraging since the speedup steadily increases as the number of workstations increases. The rate of increase of speedup shows a slight decrease in the cases when 4 and 8 workstations were used. Since the number of rollbacks were very small in both cases and there were no cancellation notices sent at all, the decrease in the speedup rate can be explained by the increase in bus contention due to the increase in the number of messages exchanged among the local simulators. Another possible source of performance degradation is the increase in synchronization overhead when the number of workstations increased. The synchronization overhead includes sending and receiving of messages, saving of states and histories, garbage collection of saved states and histories, and calculation of MST.

Workstation(s)	*Exec. Time*	*Messages*	*Rollbacks*	*Cancellations*
1	48.4 ± 0.31	-	-	-
2	38.1 ± 0.28	1002.0	0.0	0.0
4	45.6 ± 0.37	1037.6	89.6	0.0
8	46.1 ± 0.28	3156.4	80.4	0.0

Workstation(s)	*Speedup*	*Rollback Fraction*
1	1.00	-
2	1.27	0.00
4	1.06	0.09
8	1.05	0.03

Table 6.2: Performance results for EPTN with conflict place

6.3 EPTN with Conflict Place

The second experiment conducted involved an EPTN with a conflict place as shown in Figure 6.2. The EPTN has 9 places and 8 transitions. p_1 is a source place, p_5, p_7 and p_9 are sink places, and p_3 is a conflict place. The data token arrival rate at source place p_1 was set to be 1 data token per second. The transition firing times were fixed at 1 second. The EPTN was simulated for 1000 seconds.

The different allocation schemes for the EPTN to workstations is also shown in Figure 6.2. As before, each enclosed group of places and transitions represents a subnet to be simulated on a workstation.

Table 6.2 shows the performance results obtained. The results are not as good as those obtained in the previous experiment. The speedup decreases gradually as the number of workstations increases. This can be explained by the way the EPTN was allocated in the cases which used 4 and 8 workstations. Consider the case with 4 workstations. The three local simulators simulating the three subnets

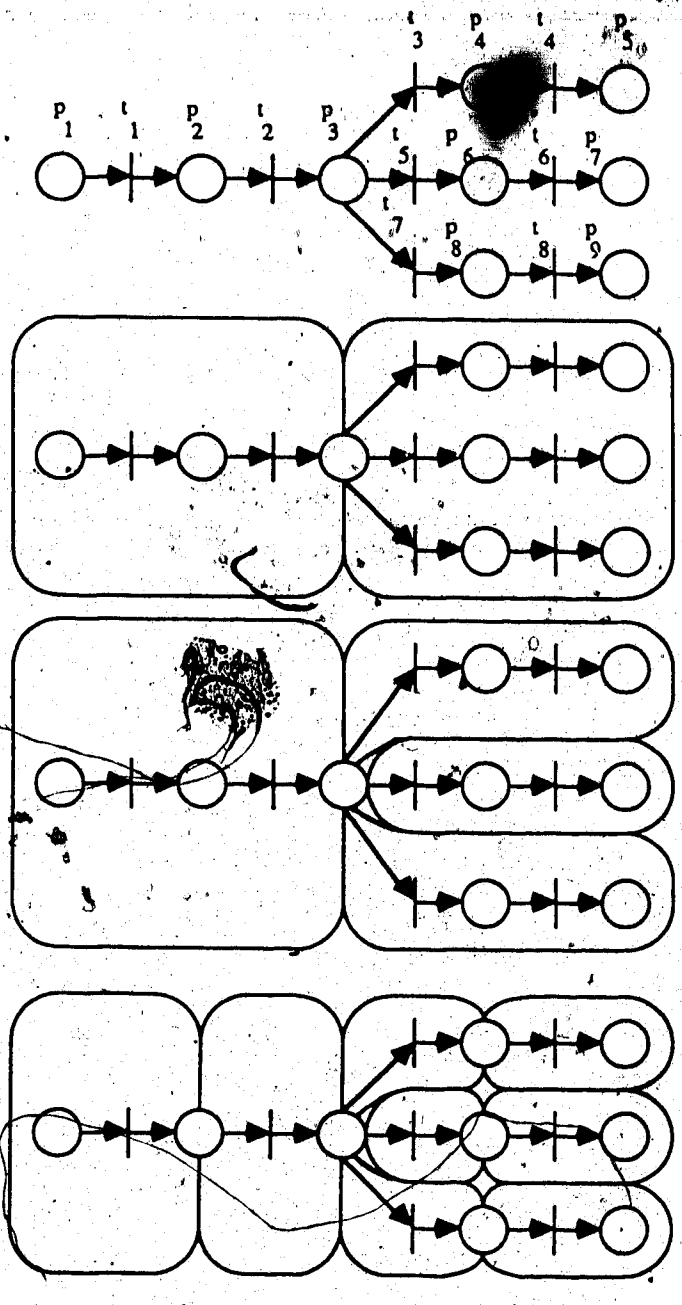


Figure 6.2: Allocation schemes for an EPTN with a conflict place

to the right of place p_3 (see Figure 6.2) were not always busy since p_3 is a conflict place. That is, whenever a token was sent from the local simulator simulating the subnet containing p_3, only one of the other three local simulators would receive it. As a result, two of the three local simulators would be waiting. Similar explanation can be offered for the low speedup for the case with 8 workstations. The waiting of the local simulators in the two cases created an unbalanced load across the workstations so that the execution times in the two cases are almost equal to that using 1 workstation.

6.4 EPTN with OR-Postcondition Structure

The third experiment consisted of an EPTN with an or-postcondition structure having 11 places and 8 transitions as shown in Figure 6.3. p_1 is a source place, and p_5, p_8 and p_{11} are sink places. The postcondition of transition t_2 is an "or" so that one of p_3, p_6, p_9 or any combination of them can receive tokens after t_2 finishes firing. The data token arrival rate at source place p_1 was 1 data token per second. All transition firing times were set to be 1 second. The EPTN was simulated for 1000 seconds. Figure 6.3 also shows the various allocation schemes of the EPTN to workstations where each enclosed group of places and transitions represents a subnet to be simulated on a workstation.

Table 6.3 shows the performance results obtained. The structure of the EPTN in this experiment is very similar to that in the previous one. In fact, the results obtained for both experiments are also close. The low speedup rate observed in this experiment can also be explained by the way the EPTN was allocated in the cases using 4 and 8 workstations. As a result, some local simulators are always waiting, creating an unbalanced load, as in the previous experiment. It is interesting to note that the speedup rate for this experiment is higher than that in the previous one. The reason for this is that place p_3 in the EPTN in the previous experiment can only send one token to one of transitions t_3, t_5 and t_7 while more than one of the places p_3, p_6 and p_9 can receive a token from transition t_2 in this experiment.

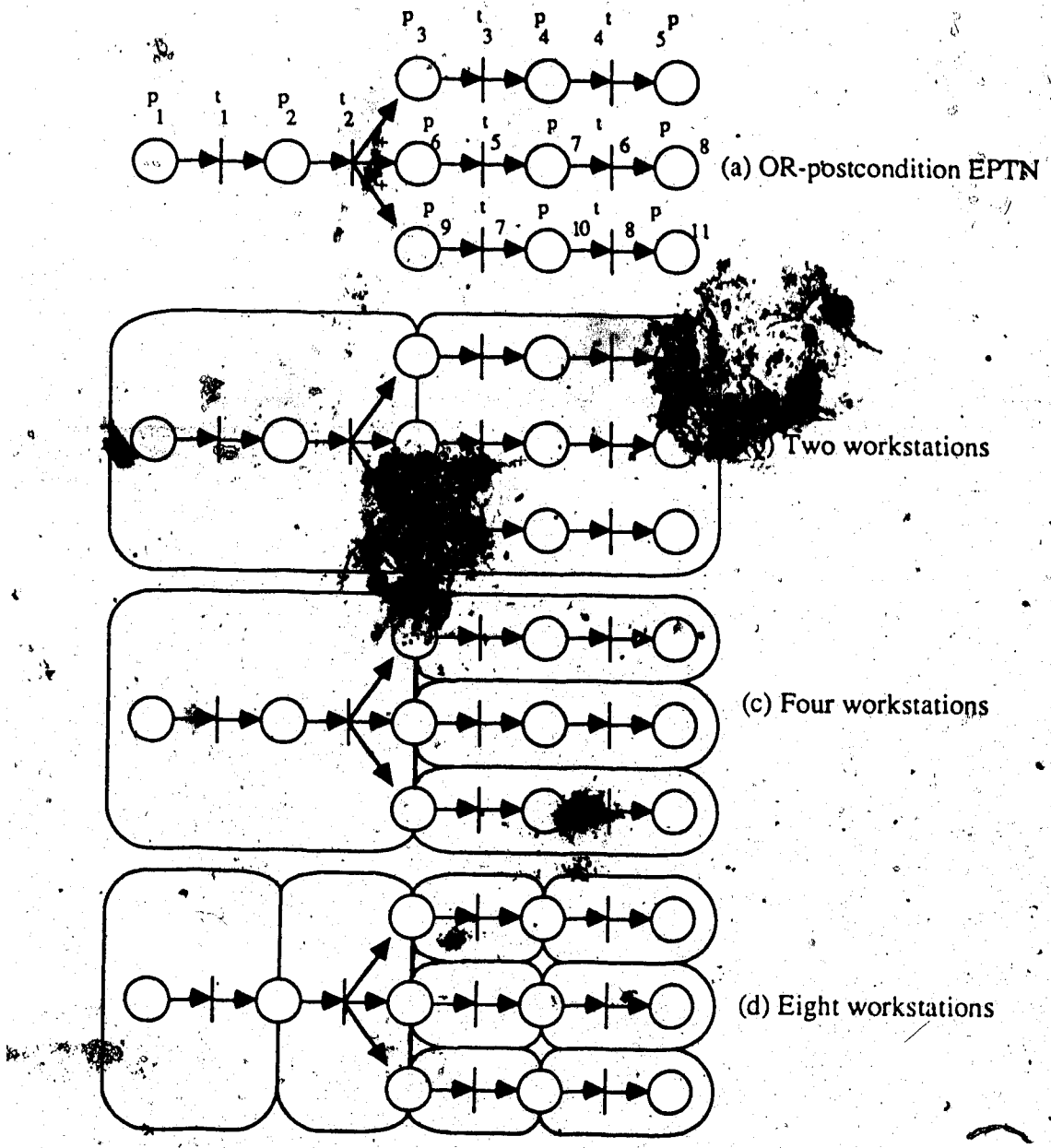


Figure 6.3: Allocation schemes for an EPTN with an or-postcondition structure

Workstation(s)	*Exec. Time*	*Messages*	*Rollbacks*	*Cancellations*
1	43.2 ± 0.26	-	-	-
2	30.2 ± 0.26	1002.0	15.8	0.0
4	31.4 ± 0.27	1012.4	81.4	0.0
8	27.0 ± 0.41	3063.6	66.8	0.0

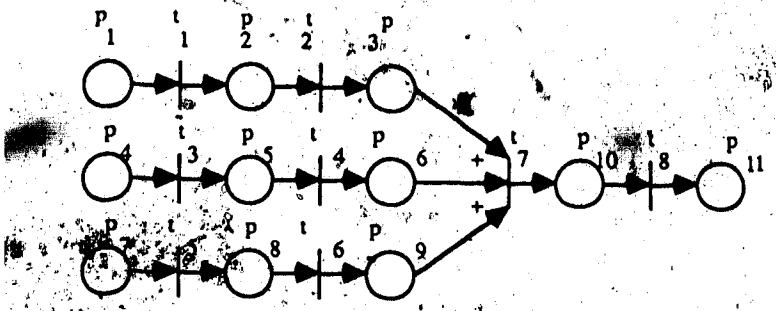
Workstation(s)	*Speedup*	*Rollback Fraction*
1	1.00	
2	1.43	0.02
4	1.38	0.08
8	1.60	0.02

Table 6.3: Performance results for EPTN with or-postcondition structure

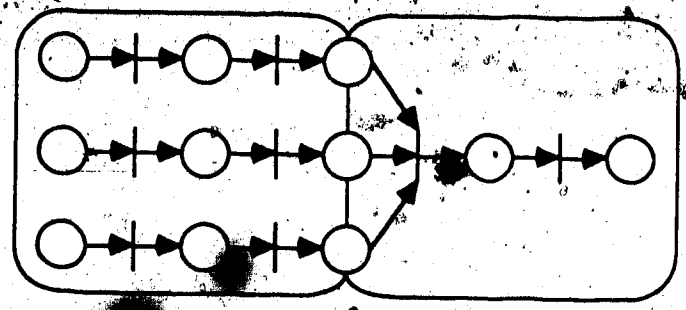
6.5 EPTN with OR-Precondition Structure

The fourth experiment conducted involved an EPTN with an or-precondition structure having 11 places and 8 transitions as shown in Figure 6.4. p_1, p_4 and p_7 are source places, while p_{11} is a sink place. The precondition of transition t_7 is given by $\{M(p_3) > 0 \wedge M(p_6) > 0 \wedge M(p_9) > 0\}$. The data token arrival rate at source places p_1, p_4 and p_7 were all set to be 1 data token per second. All transition firing times were fixed at 1 second. The EPTN was simulated for 1000 seconds. The allocation schemes for the EPTN to workstations are also depicted in Figure 6.4.

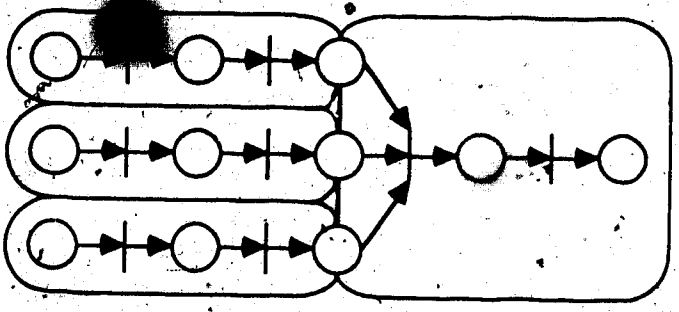
Table 6.4 shows the performance results obtained. The results are very discouraging as all speedups are below 1, although the speedup increases as the number of workstations increases. The reason for this poor performance is due to the high rollback fractions in all three cases. In the case with 2 workstations, about 65% of messages transmitted resulted in rollbacks. Similarly, about 35% and 13% of messages transmitted in the other cases, respectively, also resulted in rollbacks. The



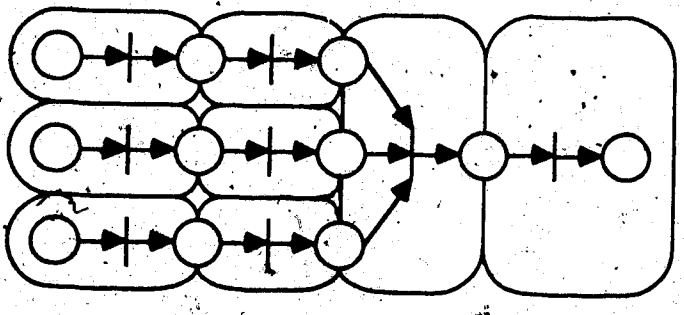
(a) OR-precondition EPTN



(b) Two workstations



(c) Four workstations



(d) Eight workstations

Figure 6.4: Allocation schemes for an EPTN with an or-precondition structure

Workstation(s)	*Exec. Time*	*Messages*	*Rollbacks*	*Cancellations*
1	41.6 ± 0.26	-	-	-
2	90.3 ± 0.29	882.5	576.7	0.0
4	66.4 ± 0.32	936.4	324.2	0.0
8	51.1 ± 0.19	3160.3	420.2	39.1

Workstation(s)	*Speedup*	*Rollback Fraction*
1	1.00	
2	0.46	0.65
4	0.63	0.35
8	0.81	0.13

Table 6.4: Performance results for EPTN with or-precondition structure

high frequency of rollbacks can be explained by the fact that transition t_7 (see Figure 6.7) is a bottleneck region where tokens arrive from places p_3, p_6 and p_9. The EPTN is defined such that tokens arrive at t_7 from the three places at the same rate. Owing to the delay in the network and the different speeds in the processes executing the local simulators, chances are small that t_7 will be able to receive the tokens from the places at the same instance in real time when the simulation of the EPTN is distributed. As a result, a rollback has to be initiated whenever tokens arriving from p_3, p_6 and p_9 are not received in the right timestamp order.

6.6 EPTN with Feedback Structure

The last experiment involved an EPTN with a feedback structure having 10 places and 8 transitions as shown in Figure 6.5. p_1 and p_5 are source and sink places, respectively. The feedback loop consists of $\{t_1, p_2, t_2, p_3, t_3, p_4, t_4, p_6, t_5, p_7, t_6, p_8, t_7, p_9, t_8, p_{10}, t_1\}$. The data token arrival rate at source place p_1 was set at 1

Workstation(s)	*Exec. Time*	*Messages*	*Rollbacks*	*Cancellations*
1	50.1 ± 0.19	-	-	-
2	150.2 ± 0.26	1780.8	587.5	390.4
4	4[illegible].4 ± 0.37	22780.3	12040.4	5040.9
8	570.3 ± 0.29	58700.2	10800.7	6800.2

Workstation(s)	*Speedup*	*Rollback Fraction*
1	1.00	-
2	0.33	0.33
4	0.10	0.53
8	0.09	0.18

Table 6.5: Performance results for EPTN with feedback structure

data token per second while all transition firing times were fixed at 1 second. The EPTN was simulated for 1000 seconds. The allocation schemes of the EPTN to workstations are depicted in Figure 6.5.

Table 6.5 shows the performance results obtained. As in the last experiment, all speedups are below 1. In fact, As the number of workstations increases, the speedup decreases. The low speedups are due to the high rollback fractions and high number of cancellation notices sent. In the case with 2 workstations, about 33% of messages transmitted resulted in rollbacks. About 66% of the rollbacks are triggered by cancellation notices. In the case with 4 workstations, 53% of messages transmitted also resulted in rollbacks and more than 40% of the rollbacks are triggered by cancellation notices. In the case with 8 workstations, 18% of the messages transmitted also resulted in rollbacks and more than 60% of the rollbacks are a result of cancellation notices. The high frequency of rollbacks and large number of cancellation notices sent can be explained by the feedback structure of the EPTN. A token that is circulated within the feedback loop of the EPTN

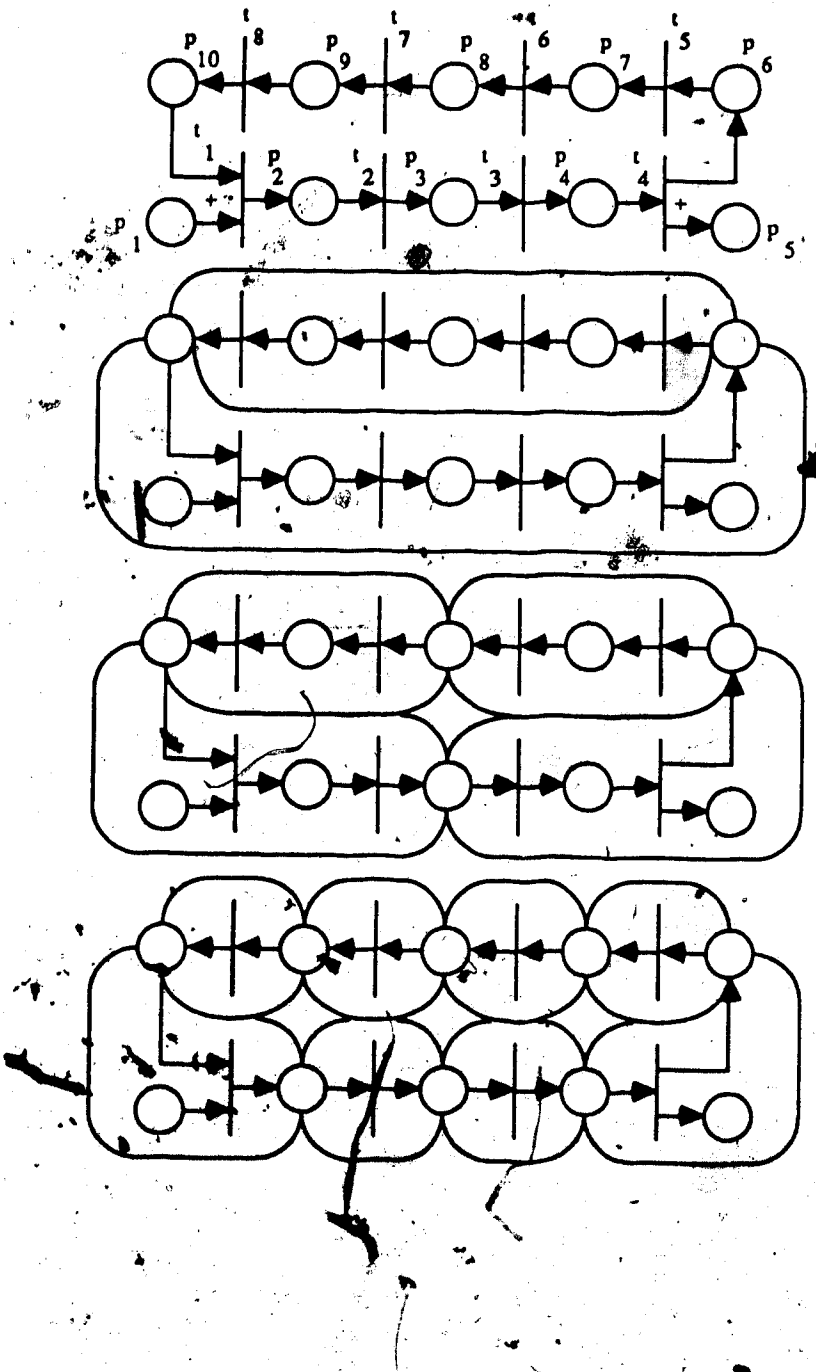


Figure 6.5: Allocation schemes for an EPTN with a feedback structure

when the simulation of the EPTN is distributed is very likely to trigger rollbacks. Consider the case with 2 workstations where the a token is sent from one local simulator to another and then back to the first. When the token returns to the first local simulator, there is no guarantee that the simulation time of the second local simulator is greater than or equal to the simulation time of the first local simulator. When a rollback is initiated at the first local simulator as a result of the returning token, a cancellation notice might also have to be sent to the second local simulator to cancel the effect of the token that was sent there the first time. As a result of these circulating tokens, cascading rollbacks are frequent, crippling the progress of the entire simulation.

6.7 Remarks

The set of experiments described in this chapter is only a preliminary study of performance of the distributed simulation method. They are by no means complete, nor are they indicative of the actual performance of the distributed simulation method.

However these experiments do provide some very important insights that are helpful in enhancing certain aspects of the distributed simulation method, in tuning certain parts of the distributed simulator and in understanding what sort of EPTN models lend themselves for efficient distributed simulation.

The results for the first experiment point out a limitation in the design of the distributed simulation method. The method fails to take into account the structure of the serial EPTN in the synchronization of the [illegible]ets. As shown in Figure 6.1, the firings of the transitions in the serial EPTN are independent of one another. Hence the arrival of a "late" token should not trigger a rollback. Similar observations can also be made about the second and third experiments. An enhancement of the method is to minimize rollbacks by taking advantage of the structure of an EPTN to be simulated. Such an enhancement requires some preprocessing of the EPTN to be simulated.

An important observation gained is the effect of network contention and synchronization overhead on the performance of the distributed simulation method. Increasing the number of workstation does not necessarily increases the amount of parallelism. In the first experiment, a degradation in performance was observed as number of workstations was increased due to the the increase in network contention and synchronization. More experiments have to be conducted to find out more about the degradation characteristics of the distributed simulation method. The distribution of the various components in synchronization overhead also has to be determined in order to tune the distributed simulator for better performance.

The results of the experiments also suggest that the heuristic allocation algorithm was too simplistic. The absence of a load balancing mechanism in the algorithm has a rather detrimental effect on performance of the distributed simulation method as observed in the second and third experiments. This points to the need for a dynamic approach to the allocation problem. Another deficiency of the algorithm is that it fails to take into account the characteristics of the interaction of the subnets of an EPTN. The characteristics of the rate of token flow among subnets were observed to be the reasons for the rather poor results obtained in the fourth experiment.

The extremely poor results obtained in the last experiment is an indication that certain EPTN structures are not suitable for distributed simulation. The set of such EPTN structures should be identified with further experimentation. The knowledge of such a set of "forbidden" EPTN structures also facilitates the use of certain heuristics for decomposition and allocation. For example, in the case of an EPTN with feedback structures, the decomposition of the EPTN should be done in such a way where each feedback loop in the EPTN is contained within each subnet so that during simulation, cascading rollbacks are minimized.

Chapter 7

Conclusions

7.1 Summary of Results

In this thesis, the design and implementation of a distributed method for EPTN simulation were studied. The distributed simulation method consists of two main parts. In the decomposition step, a given EPTN is transformed into a weighted graph where each vertex in the graph corresponds to a transition. Two vertices are connected by an edge if the corresponding transitions have places in common, and the edge is weighted according to the number of common places. Next, a simple heuristic allocation algorithm is used to merge vertices into clusters. The resultant clusters correspond to subnets of the EPTN which are then allocated to a set of processors. The second part is the development of an optimistic synchronization algorithm. The algorithm allows the local simulators simulating the subnets of an EPTN to interact freely until an incorrect simulation time sequence is detected. When this happens, the local simulators are rolled back to a previous consistent state. The major components of the synchronization algorithm are the rollback operation, memory management and termination detection. Correctness proofs (preservation of partial order, deadlock-freeness and termination) of the synchronization algorithm were also given.

A distributed simulator based on the method was implemented using 8 Sun Workstations connected by a 10-Mbit Ethernet local area network. It was con-

structed in a hierarchical and modular manner. The three levels in the simulator are the EPTN subnet layer, the simulator layer and the VTM layer. The EPTN subnet layer forms the user-interface component of the simulator while the simulator layer implements the actual simulator. The VTM layer provides a number of high level routines for inter-processor communication.

The preliminary performance evaluation of the method was carried out using various EPTNs as test cases. The performance metric used was the speedup which is defined as the ratio of the execution time of an EPTN simulated on a workstation to the execution time of the same EPTN simulated over several workstations using the distributed simulation method.

The performance of the distributed simulation method was found to be extremely sensitive to the structure of the EPTNs simulated and how the EPTNs were allocated to the workstations. For EPTNs with a serial structure, the performance of the method was found to be very encouraging as the speedup increases as the number of workstations increases. The number of rollbacks was also observed to be infrequent. The performance for simulating EPTNs with a conflict place was satisfactory. Similar results were obtained for EPTNs with a fork structure except the speedups were observed to be higher. The performance for simulating EPTNs with a merge structure was observed to be poor. This is due to the high rollback fractions. The late arriving tokens at the merging transition were the cause of the high number of rollbacks. The performance for simulating EPTNs with a feedback structure was also observed to be poor due to high rollback fractions. The reason for the high rollback fractions is a result of tokens that were circulated within the subnets, causing cascading rollbacks.

Overall, the experimental results are quite consistent with previous performance evaluation studies on other distributed simulation methods [1,14,16,37]. These studies found that while distributed simulation could provide substantial speedup for certain simulation models, it is not a viable approach for simulating simulation models having certain structures (e.g. feedback loops). It seems that the use of distributed simulation is only limited to certain classes of simulation models.

To conclude, this thesis has demonstrated that the distributed simulation of EPTNs could be a feasible approach. The challenge is to find better decomposition and allocation algorithms, understand better where distributed simulation spends most of its time and what kind of EPTN structures are most suitable for distributed simulation.

7.2 Future Research

The work reported in this research can be extended in a number of ways. More extensive experimental evaluation of the distributed simulation method has to be carried out. In particular, more experiments has to be carried out using a more comprehensive set of EPTN test cases to establish what sort of EPTN structures are suitable for distributed simulation. In addition, obtain secondary performance measures such as memory usage patterns, the effect of the rate of state saving has on the speedup rate, the effect of the rate of MST calculation has on the speedup rate, fraction of workstation time spent in performing useful work, and utilization of the workstations.

Another area that requires more work is to develop better heuristics for the EPTN decomposition and allocation problems, taking into account both the static structure and the dynamic behaviour of an EPTN to be simulated. It will also be interesting to compare static and dynamic approaches to allocation of local simulators to workstation and see how they will affect the performance of the distributed simulation method.

Last but not least is the undertaking of a detailed empirical study comparing the performance of the distributed simulation method proposed in this thesis with other proposed distributed simulation methods in the literature.

Bibliography

[1] O. Berry. A Testbed for the Time Warp Distriubted Simulation Mechanism: A Modular and Hierarchical Software System. *Transactions of The Society for the Computer Simulation*, 3(2):135–157, 1986.

[2] R.E. Bryant. *Simulation of Packet Communication Architecture Computer Systems*. Technical Report MIT-LCS-TR-188, M.I.T., 1977.

[3] K.M. Chandy and J. Misra. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM*, 24(4):198–205, 1981.

[4] 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, 1979.

[5] K.M. Chandy, J. Misra, and V. Holmes. Distributed Simulation of Networks. *Computer Networks*, 3(2):105–113, 1979.

[6] W.W. Chu, L.J. Holloway, M.T. Lan, and K. Efe. Task Allocation in Distributed Data Processing. *IEEE Computer*, 13(11):57–89, 1980.

[7] J. Dennis. Modular, Asynchronous Control Structures for a High Performance Processor. *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, 55–80, 1970.

[8] M. Hack. *Decidability Questions for Petri Nets*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1975.

[9] M.A. Holliday and M.K. Vernon. A Generalized Timed Petri Net Model for Performance Evaluation. In *Proceedings of the 1985 International Workshop on Timed Petri Nets*, pages 181–190, 1985.

[10] V. Holmes. *Parallel Algorithms on Multiple Processor Architectures.* PhD thesis, University of Texas, Austin, Texas, 1978.

[11] A. W. Holt and F. Commoner. Events and Conditions. *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, 1–52, 1970.

[12] D. Jefferson and H. Sowizral. Fast Concurrent Simulation Using the Time Warp Mechanism. In *Proceedings of the Conference on Distributed Simulation*, pages 63–69, 1985.

[13] D.R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.

[14] D.R. Jefferson et al. Distributed Simulation and the Time Warp Operating System. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 77–93, 1987.

[15] T.M. Koon and M.T. Ozsu. Performance Comparison of Resilient Concurrency Control Algorithms for Distributed Database Systems. In *Proceedings of the 2nd International Conference on Data Engineering*, pages 565–573, 1986.

[16] M. S. Lakshmi. *Performance Analysis of Distributed Simulation.* Master's thesis, University of Texas, Austin, Texas, 1979.

[17] S. Lavenberg, R. Munt, and B. Samadiz. Performance Analysis of a Rollback Method for Distributed Simulation. In *Performance '83*, pages 117–132, 1983.

[18] J.D.C. Little. A Proof of the Queueing Formula $L = \lambda W$. *Operations Research*, 9:383–387, 1961.

[19] M.A. Marsan, G. Conte, and G. Balbo. A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems. *ACM Transactions on Computer Systems*, 2(21):93–122, 1984.

[20] J. Misra. Distributed Discrete-Event Simulations. *ACM Computing Survey*, 18(1):39–65, 1986.

[21] M.K. Molloy. Performance Analysis Using Stochastic Petri Nets. *IEEE Transactions on Computers*, C-31(21):913–917, 1982.

[22] S. Natkin. *Reseaux de Petri stochastiques.* PhD thesis, CNAM-Paris, 1980.

[23] M. Olafsson and T. A. Marsland. A Unix Based Virtual Tree Machine. In *CIPS Congress 1985, Montreal, Quebec*, pages 176–181, 1985.

[24] M.T. Ozsu. Distributed Simulation Using Petri Nets. In *Proceedings of the Summer Computer Simulation Conference*, pages 1–6, 1987.

[25] M.T. Ozsu. Modeling and Analysis of Distributed Database Concurrency Control Algorithms Using an Extended Petri Net Formalism. *IEEE Transactions on Software Engineering*, SE-11(10):1225–1240, 1985.

[26] M.T. Ozsu, G. Miklos, S. Munro, and T.M. Koon. EPTN/SIMLANG *Reference Manual.* Technical Report, University of Alberta, 1986.

[27] M.T. Ozsu, K.L. Wong, and T.M. Koon. System Modeling and Analysis Using Petri Nets. *Journal of Systems Analysis-Modelling-Simulation*, 5(1):3–25, 1988.

[28] D.L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(2):1053–1058, 1729.

[29] S. Patil. *Coordination of Asynchronous Events.* PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1970.

[30] J.K. Peacock, J.W. Wong, and E.G. Manning. A Distributed Approach to Queueing Network Simulation. In *Proceedings of the 4th Berkeley Confer-*

ence on Distributed Data Management and Computer Networks, pages 237–259, 1979.

[31] J.K. Peacock, J.W. Wong, and E.G. Manning. Distributed Simulation Using a Network of Processors. *Computer Networks*, 3(1):44–56, 1979.

[32] J.K. Peacock, J.W. Wong, and E.G. Manning. Synchronization of Distributed Simulation Using Broadcast Algorithms. In *Proceedings of the IEEE Winter Simulation Conference*, pages 399–406, 1979.

[33] J.L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.

[34] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, Bonn, West Germany, 1962.

[35] C. Ramchandani. *Analysis of Asynchronous Concurrent Systems by Petri Nets*. Technical Report 120, MAC Project M.I.T., 1974.

[36] R.R. Razouk and C.V. Phelps. Performance Analysis Using Timed Petri Nets. In *Proceedings of the 1984 International Conference on Parallel Processing*, pages 126–129, 1984.

[37] D.A. Reed, A.D. Maloney, and B. D. McCredie. Parallel Discrete Event Simulation: A Shared Memory Approach. *IEEE Transactions on Software Engineering*, 14(4):541–553, April 1988.

[38] W. Reisig. *Petri Nets : An Introduction*. Springer-Verlag, 1985.

[39] G. Schiffner and H. Godbersen. Function Nets : A Comfortable Tool for Simulating Database System Architectures. *Simulation*, 46(5):201–210, 1986.

[40] F.J.W. Symons. Introduction to Numerical Petri Nets, A General Graphical Model of Concurrent Processing System. *A.T.R.*, 14, 1980.

[41] A. Torn. Simulation nets, a simulation modeling and validation tool. *Simulation*, 45(2):71–75, 1985.

42] W.M. Zuberek. Timed Petri Nets and Preliminary Performance Evaluation.
In *7th Annual Symposium on Computer Architecture*, pages 88–96, 1980.