DISTRIBUTED SIMULATION OF HIGH-LEVEL ALGEBRAIC PETRI NETS

Karim Djemame

Thesis Submitted in Accordance with the Requirements
for the Degree of Doctor of Philosophy

UNIVERSITY
of
GLASGOW

The University of Glasgow
Computing Science Department

July 1999

11696 (copy 1)

Abstract

In the field of Petri nets, simulation is an essential tool to validate and evaluate models. Conventional simulation techniques, designed for their use in sequential computers, are too slow if the system to simulate is large or complex. The aim of this work is to search for techniques to accelerate simulations exploiting the parallelism available in current, commercial multicomputers, and to use these techniques to study a class of Petri nets called high-level algebraic nets. These nets exploit the rich theory of algebraic specifications for high-level Petri nets: Petri nets gain a great deal of modelling power by representing dynamically changing items as *structured tokens* whereas algebraic specifications turned out to be an adequate and flexible instrument for handling *structured items.* In this work we focus on ECATNets (Extended Concurrent Algebraic Term Nets) whose most distinctive feature is their semantics which is defined in terms of rewriting logic. Nevertheless, ECATNets have two drawbacks: the occultation of the aspect of time and a bad exploitation of the parallelism inherent in the models.

Three distributed simulation techniques have been considered: asynchronous conservative, asynchronous optimistic and synchronous. These algorithms have been implemented in a multicomputer environment: a network of workstations. The influence that factors such as the characteristics of the simulated models, the organisation of the simulators and the characteristics of the target multicomputer have in the performance of the simulations have been measured and characterised.

It is concluded that synchronous distributed simulation techniques are not suitable for the considered kind of models, although they may provide good performance in other environments. Conservative and optimistic distributed simulation techniques perform well, specially if the model to simulate is complex or large - precisely the worst case for traditional, sequential simulators. This way, studies previously considered as unrealisable, due to their exceedingly high computational cost, can be performed in reasonable times. Additionally, the spectrum of possibilities of using multicomputers can be broadened to execute more than numeric applications.

To Chafia and Djamel

Acknowledgements

First and foremost I would like to thank my supervisors, Prof. Dennis Gilles and Prof. Mohamed Bettaz, for guiding my research over the course of my Ph.D. They have been a constant source of advice, encouragement and guidance during my research, and have always provided constructive criticism of my work. Prof. Bettaz's enthusiasm for the subject has helped to make this a very enjoyable period of study for me.

I would also like to thank Dr Lewis Mackenzie for his time, his advice and for helping me in avoiding the pitfalls on the way to obtaining a Ph.D. degree.

Also, I would like to express my gratitude to the Computing Science Department at the University of Glasgow for providing such a stimulating and friendly environment to work in.

My research was supported by the Algerian Ministry of Higher Education and Scientific Research, and the British Council in the form of a Split-Ph.D. Research Studentship during the period when I was visiting the University of Glasgow (September 1993 - September 1997). I am grateful to them.

Declaration

This thesis is submitted in accordance with the regulations for the degree of Doctor of Philosophy in the University of Glasgow. No part of it has been previously submitted by the author for a degree at any other university and all results contained within are claimed as original.

Section 5.4.2 contains ideas suggested by Nicol and Mao [NR91]. Section 5.6.4 contains an algorithm inspired from ideas suggested by Thomas and Zahorjan [TZ91]. Sections 5.6 and 6.3 are revised versions of material published in [DBGM96a] and [DBGM95], respectively. Sections 5.7 and 6.4 cover material published in [DBGM96b] and [DBGM98].

Acronyms

Some frequently used abbreviations appearing in this thesis are listed here, together with a brief explanation of their meaning.

ATNet	Algebraic Term Net
C	Capacity
CATNet	Concurrent Algebraic Term Net
CMB	Chandy-Misra Bryant
CMB-DA	Chandy-Misra Bryant with Deadlock Avoidance
CM-DDR	Chandy-Misra Bryant with Deadlock Detection and Recovery
CPNets	Coloured Petri Nets
CT	Created Tokens
DDES	Distributed Discrete Event Simulation
DT	Destroyed Tokens
ECATNet	Extended Concurrent Algebraic Term Net
EP	Efficient Partitioning
ES	Event Stack
EVL	Event List
GSPN	General Stochastic Petri Net
GVT	Global Virtual Time
IQ	Input Queue
IC	Input Condition
LP	Logical Process
LVT	Local Virtual Time
MPI	Message Passing Interface
NOW	Network of Workstations
OQ	Output Queue
PDES	Parallel Discrete Event Simulation
PE	Processing Element
PN	Petri Net
PP	Physical Process
PVM	Parallel Virtual Machine
QN	Queueing Network
SCS	Separation of Concern Strategy
SPN	Stochastic Petri Net
TC	Transition Condition
TW	Time Warp
TW-AC	Time Warp with Aggressive Cancellation
TW-LZ	Time Warp with Lazy Cancellation

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation of the Work

The objective of the work presented in this thesis is to propose efficient, parallel ways to simulate high-level algebraic Petri nets. The interest of the work is twofold:

1. Our research group's effort focuses on the study of distributed systems and their performance modelling. As a significant amount of simulation work is performed, we would like to find a way of accelerating these simulations because they are time consuming.

2. We are also interested in the use of currently available multicomputers. Simulation is an interesting application to parallelise in this context.

Next we elaborate these two ideas.

1.1.1 Modelling and Analysis of Real Systems

During the last few years, our research group has been working on the analysis and use of high-level algebraic Petri nets. The effectiveness of our proposals has to be validated somehow, and there are tools considered for this purpose: analytical models and computer simulation. This situation is common to many fields of science and engineering. In this thesis, we will focus and speak, in general, about the evaluation of any kind of real (or proposed) system specified using high-level algebraic nets.

Analytical tools can be described as cheap and fast to use. Of course the development of an analytical model of a system can be very complex, but once a set of equations has been developed, it is easy to extract information from it. Unfortunately, this approach commonly requires the assumption of simplifications in some (or many) of the characteristics of the system, for the researcher to be able to solve the analytical problem. These simplifications can lead to a model whose behaviour may be far from the behaviour of the real system: the results might not be accurate if some of the simplifying assumptions are not realistic.

Computer simulation offers an interesting alternative: the system can be described somehow (eg. using a simulation language) and then simulated using a computer. The description could include simplifying assumptions, like the analytical model, and then the simulation time would be short. In contrast, the description could be very detailed, containing as many elements as the real system, and then a highly accurate insight into the behaviour of the system would be obtained. But the accuracy comes at a price: simulation's drawback is its long execution time. Computer simulation is very flexible and can be used in many contexts:

- to validate an analytical model;
- to see how an existing system works, when it cannot be easily instrumented;
- to study a non existing system, without building it. There are many reasons not to build a system: it might be very expensive, or it might be simply impossible;
- to analyse the effect of different design parameters, in an existing or a non-existing system.

Those using simulators know that analysing large and/or detailed systems can be desperately slow. In this context, any possibility of increasing the execution speed of the simulation is welcome. The speed increments due to the advances in VLSI (Very Large Scale Integration) have been significant, but there always exists a demand for more. The introduction in the market of reasonably priced parallel systems has allowed researchers to accelerate many computations, and it seems logical to think that simulation may also benefit from this technology.

1.1.2 Parallel Implementation of High-Level Nets Simulation Applications

As researchers in the broad field of parallel computing, we are interested in making a good use of currently available parallel computers. During this research, we have had access to a parallel computing system: a Network of Workstations (NOW) with MPI (Message Passing Interface) and PVM (Parallel Virtual Machine) libraries at

Glasgow. It was our interest to see how well traditional high-level Petri nets sequential simulation applications could be adapted to run on these machines. A good deal of work can be found in the literature reporting parallel algorithms to solve many problems, mainly in scientific and engineering fields, but most of those problems have some characteristics that make them easy to parallelise: big, partitionable data structures, simple communication patterns, reduced data dependencies, ...

While simulation is a tool commonly used by scientists and engineers, the algorithms exhibit a behaviour that makes them difficult to parallelise: data structures are not always regular, communication among the parts of the model may follow arbitrary patterns, there are very strong data dependencies, and so on. But difficult does not mean impossible. As it will be explained in Chapter 3, simulation can be parallelised, provided that new algorithms are developed, instead of simply trying to make in parallel some of the operations of the sequential programs.

Being successful in the search of parallel and distributed simulators is very important, because it extends the domain of applications that can be run in an available parallel computer, increasing the usefulness of the investment, which is in general expensive.

Obviously, we are not the first research group working on the field of parallel and distributed simulation, as many work has been done in the last 10-15 years and many work is still being done. Most of the work discussed in the literature about Parallel Discrete Event Simulation (PDES) has been done using shared memory multiprocessors for two reasons: (1) many multiprocessors are available; (2) they allow a more optimised implementation of many algorithms compared to distributed memory systems. However, it is assumed that future massively parallel systems will be distributed memory systems. Many currently available machines are built this way using message passing for synchronisation and communication.

The use of a network of workstations as a fully distributed parallel system is also becoming very popular, because it is a very cost/effective alternative to a parallel computer [Tur96]. The most popular model of communication for these systems is also message passing, an approach followed in this work. While typically more loosely coupled than the 'single' box parallel architectures described as parallel computers, network of workstations can provide an invaluable route for producing parallel code. Further to this, they offer the opportunity for users without the resources to buy massively parallel machines to gain some of the benefits of parallelism on machines available locally, and perhaps not used constantly.

1.2 Tools

In the last years, a considerable effort has been devoted to the parallel and distributed implementation of discrete event simulators. The objectives were: (1) to exploit the parallelism available in current multicomputers and multiprocessors and, mainly, (2) to accelerate simulation runs.

For some simulations studies, it is necessary to run several simulations to study the influence of a certain set of parameters on the system under study. In these cases, the most convenient way of accelerating the job is simply running as many simulations as processors are available, each one with different input parameters. This technique is called *replication.* The achieved efficiency is very good, because the simulations are completely independent, and therefore there is no need of communication or synchronisation among the involved processors.

However, it is not always possible to replicate the simulator. In some studies it is necessary to have the results of one simulation before starting with the next one; this is the case when the aim is to tune a set of parameters. It is also possible that the memory available at each processor is not large enough to keep a complete copy of the simulator. These limitations of the replication approach justify the need of solutions to parallelise a single simulation run.

The most promising techniques to perform parallel simulation uses the *spatial model decomposition*: the system to simulate is decomposed into several subsystems, and each subsystem is going to be assigned to a Logical Process (LP). As explained in chapter 3, a synchronisation mechanism is needed to maintain causal relationships among the events in the simulation. In this work, we analyse three different synchronisation mechanisms: *conservative*, *optimistic* and *synchronous.* In all cases, each LP has its own local view of time, and the collection of LPs run concurrently. The difference between these mechanisms is how they deal with causality errors. In conservative and optimistic mechanisms, the parallel simulation is *asynchronous.* A *conservative* simulator never allows causality errors to occur. To do so, LPs block before executing an event, until it is totally safe to proceed. An *optimistic* simulator allows erroneous situations to arise (a new event might arrive from other LP, with a timestamp smaller than that of the last executed event), but those are detected and a *rollback* is done to jump to an error-free point in the (simulated) past. In a *synchronous* parallel simulation, all the LPs which form the simulator share the same vision of time, as if they had a global clock. Events are simulated in the same order a sequential simulator would choose, simulating in parallel only those events scheduled for the same time.

On the other hand, concurrent simulation uses the concept of *temporal model decomposition* principles which could be an alternative approach to the application of multiple processors to discrete event simulation models on shared-memory. In this case, simulation processes are servers, each of which repeatedly waits for a pending event to become available for simulation and then simulates it. The concurrent pending-event set is central to this approach, and pending events are organised in chronological order. In addition, it must prevent events from being removed for simulation until there is an assurance that no events will be scheduled at earlier times. Mutual exclusion for access to the state variables of the simulation model is required due to the fact that multiple processes may simulate events in parallel.

1.3 Objectives

Previous research in parallel simulation shows that (1) the dynamic nature of parallel simulation problems is the principal reason that a "general" solution has been elusive; (2) that its efficiency is highly dependent on the characteristics of the system under study. For this reason it is not feasible to characterise the performance of the different parallel simulation algorithms in a general context. It is possible, however, to select a set of related models and extract conclusions about how a given algorithm performs with that set of models. Our research will focus on the analysis of high-level algebraic Petri nets. We will consider ECATNets (Extended Concurrent Algebraic Term Nets) as models of our study. A detailed description of these models are found in Chapter 2.

Algebraic theories have proved to be of great use for the formal specification of abstract data types [EM85]. High-level algebraic nets have been introduced in order to exploit the rich theory of algebraic specifications for high-level Petri nets. To define classes of high-level Petri nets having structured individual tokens is a very fundamental goal for making nets actually usable in real concurrent system modelling. A promising approach is that of combining nets with algebraic specification techniques. This results in a formal specification language which supports both aspects of system modelling, namely data structure and control structure modelling, with suitable abstraction notions.

The practical significance of the high-level algebraic Petri net concept has been shown in previous works [Bet91, BC92, BM93a, BMSB94, BM95] through the specification of problems mainly from the fields of communication networks, communication software, hardware diagnosis, and software testing. These are good examples where high-level algebraic nets play fundamental roles.

These studies concluded that ECATNets are good candidates for qualitative/quantitative performance evaluation. Fine-grain and/or coarse-grain parallelism inherent to these models has to be detected, then simulated on a computer. The work presented in this thesis has been focused on the simulation of Petri nets with these characteristics using a collection of processors which might be able to collaborate to solve problems.

Throughout the resarch we will focus on a better exploitation of the parallelism inherent in high-level algebraic Petri net models and programming in multicomputer environments.

A parallel computer may provide one or more of these programming paradigms: *SIMD* (Single Instruction, Multiple Data), means that all the processes run the same program, instruction by instruction, at the same time, and *MIMD* (Multiple Instruction, Multiple Data), meaning that each process might run a completely different program. In particular, more restrictive case of MIMD is *SPMD* (Single Program, Multiple Data), where all the parallel processes run the same program.

Shared memory computers have multiple processors and provide a global shared memory. For efficiency reasons, each processor has also a local cache, which in turn creates the problem of maintaining cache coherence. Synchronisation might be provided by mechanisms such as semaphores. Message passing computers are connected by a message passing network. Each processor has its own memory space. Both communication and synchronisation are provided by means of messages sent between processors.

Our work has been developed in message passing environments. There are several reasons for making this decision. Firstly, the use of a network of workstations because no parallel computer is available at Glasgow. The fact that our simulation protocols are built on top of the MPI message passing library makes them *portable* on a parallel computer. Secondly, message passing is a paradigm widely used not only on NOWs but also on certain classes of parallel computers, especially those with distributed memory. Thirdly, parallel simulation algorithms based on spatial decomposition are described by means of a set of processes which interchange messages. Finally, the concept of Distributed Shared Memory (DSM, generally built on top of message passing) can be used to implement further concurrent simulation applications.

We are concerned by the hardware mechanism used for message passing because it greatly influences the performance of the communication functions, and thus, of the applications. In a network of workstations, the MPI message passing library uses TCP/IP over an Ethernet local area network. This medium provides a raw 10 Mb/s data rate, which must be shared among all the stations attached to it. The communication effort is performed by the workstations CPUs, with the aid of the Ethernet cards for accessing the medium.

Communication is different compared with a parallel computer. For example, in the Intel Paragon there is a good deal of hardware support for message passing: communication is separated from computation, by means of a collection of hardware message routers organised in a mesh topology. The communication links which join routers can move up to 1600Mb/s. Additionally, each node has a second processor specialised in communication, leaving the main processor free to spend its time performing computation.

In Chapter 4 we will give a deeper insight into the programming models provided by parallel systems, as well as particular descriptions of the environments used in this research.

1.4 Major Contributions

The major contributions of this work include:

- the introduction of the aspect of time in ECATNets. Since the introduction of ECATNets [Bet91], the behaviour of a modelled system was explained by formal reasoning. The aspect of time is implicitly specified and by transforming the rewriting logic into a rewriting system, rapid prototyping and automatic proving of the system is possible. ECATNets enriched with temporal specification are suitable for discrete event simulation;

- the implementation of a model of ECATNets which can be simulated using a discrete event simulator, along with four simulation engines able to work with that model: one is sequential, and the other three are parallel, testing three different synchronisation mechanisms in a multicomputing environment.

In this research we have characterised:

- which synchronisation mechanisms provide an adequate tool for our studies, and which others are not so good. The conservative and optimistic approaches perform well, while the synchronous approach does not seem to be the right one for our purposes;

- the effect of the parameters of the ECATNet model on the performance of the simulators. Models of large size whose components interact frequently constitute a challenge for sequential simulators, but simplify the synchronisation tasks of conservative and optimistic distributed simulators, offering good level of performance;
- the effect that the organisation of a simulator has in its performance. The partition of a ECATNet model into a set of submodels might be done in several ways such as a separation of concern partitioning leading to fine, medium or large grain LPs. Large grain LPs provide better performance than fine grain LPs but it is sometimes advantageous to assign several medium grain LPs to each processor;
- the influence of the target multicomputer on the efficiency of the simulation. Our simulators (sometimes) exploit a fine-grain parallelism which requires a fast message passing infrastructure. A network of workstations is not (always) efficient because communication is too costly compared to computation;

1.5 Overview of the Thesis

The previous sections have given an introduction to the work presented in this thesis. We summarise how it is organised.

Chapter 2 presents high-level algebraic Petri nets and focuses on ECATNets, a kind of high-level algebraic nets used in this thesis. We start with a general introduction to Petri nets then we describe how the aspect of time is handled in these nets. It is shown how the concept of time is used in high-level nets in general and how it is introduced in ECATNets.

In chapter 3 we provide a survey of parallel and distributed simulation techniques. We start with a description of the general process of studying a system by means of simulation. After introducing the main discrete event simulation concepts, two common sequential simulation techniques are presented: time-driven and event-driven. Then, a description of the ways of exploiting the parallelism available in current multiprocessors is given, focusing on those techniques based on model decomposition. After that, the conservative, optimistic and synchronous mechanisms are presented, along with a series of improvements and optimisations to the basic algorithms.

Chapter 4 gives an introduction to the programming environments available to parallel systems, with special attention to the network of workstations with the

MPI library. This parallel system is a distributed memory system which provides a message passing mechanism for synchronisation and communication.

Chapters 5, 6 and 7 describe a conservative, an optimistic (based on Time Warp) and a synchronous simulator respectively. Each chapter includes a detailed description of the partitioning of the ECATNet model into submodels (each submodel to be simulated by a LP), the LPs' communication interface and the simulation engine's algorithms. Performance results of preliminary experiments are presented. An ECATNet model from the area of communication networks is used to test how the three distributed simulators behave under the parameters of the model using different number of processors. It is observed that the partitioning of the ECATNet model into submodels has a great impact on the achieved performance for the case of the distributed simulators. Additionally, the performance of the optimistic simulator exceeds that of the conservative. The synchronous simulator exhibits poor performance. The conclusions of this preliminary experiment were tested by three later studies.

Chapter 8 describes the experiments performed on three case studies with the sequential and the distributed simulators. Each one is studied separately, presenting in first place the experiments, followed by the results and a series of partial conclusions. The availability of three different distributed simulation engines allows a characterisation of the obtained performance as a function of the synchronisation technique.

Finally, Chapter 9 summarises the contributions of this work, suggesting lines for further research.

Chapter 2

High-Level Algebraic Nets

This chapter presents high-level Petri nets and focuses on ECATNets, a kind of high-level algebraic nets used in this thesis. After a general introduction to Petri nets, it is shown how the concept of time is used in high-level nets in general and in ECATNets in particular.

2.1 Introduction

Petri nets is the oldest and perhaps the best established model of concurrent systems. In their various formats, they have been studied extensively since first proposed by Carl Adam Petri in the early 1960's [Pet62] and several algorithms exist to determine the functional properties of nets. The chief attraction of Petri nets is the way in which the basic aspects of concurrent systems are identified both conceptually and mathematically. Another paradigm which is aimed at testing for functional correctness is that of process algebras or calculi for communicating systems.

This chapter is structured as follows. Section 2.2 gives a definition of Petri nets and a summary of their properties, methods of analysis and semantics. A review of the types of Petri nets is done in §2.3 and focuses on timed amd high-level nets. Section 2.4 is devoted to a presentation of Extended Concurrent Algebraic Term Nets as models of our study. The introduction of the aspect of time in these nets is explained in §2.5. Finally, some conclusions are summarised in section 2.6.

2.2 Petri Net Definition

Petri nets [Pet81, Rei85, Mur89] are a graphical and mathematical tool applicable to many systems. They are a promising tool for describing and studying informa-

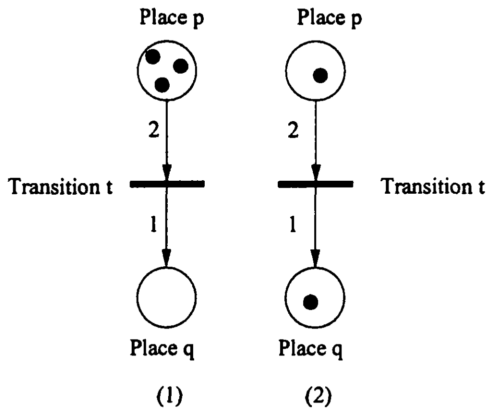


Figure 2.1: A Usual Petri Net (1) Before Transition t Fires; (2) After Transition t Fires.

tion processing systems that are characterised as being concurrent, asynchronous, distributed, parallel, nondeterministic and/or stochastic. A Petri net is a five-tuple PN = (P, T, F, W, M_0) (sometimes noted (N, M_0)) where :
P = $\{p_1, p_2, ..., p_n\}$ is a finite set of places called P-elements,
T = $\{t_1, t_2, ..., t_m\}$ is a finite set of transitions called T-elements,
F $\subseteq$ (P $\times$ T) $\cup$ (T $\times$ P) is a set of arcs (flow relation),
W : F $\rightarrow$ {1, 2, ...} is a weight function denoting the multiplicity of unary arcs between the connected nodes,
P $\cap$ T = $\emptyset$ and P $\cup$ T $\neq \emptyset$
M_0 is the initial marking of P-elements (initial state of PN).

Place/Transition nets (P/T) [Rei86] which were just called *Petri Nets*, are certainly the most common and the most extensively studied class of nets. Tokens are used in the net to simulate the dynamic and concurrent activities of systems. *Conditions* are modelled by places, and *events* are modelled by transitions. The places and transitions are represented by circles and bars, respectively. The conditions for the occurrence of an event are represented by the *input places* of a transition t. The *output places* designate the conditions after the occurrence of an event. The occurrence of an event is signaled by the *firing* of a transition and a transition fires only when it is *enabled.* In order to study the dynamic behaviour of a system, the enabling conditions of the transitions are expressed by the presence of *tokens.* The tokens, which reside in places, are represented by black dots. A transition is enabled if each input place p of t is marked with at least w(p,t) tokens, where w(p,t) is the weight of the arc from p to t. A firing of the transition removes w(p,t) tokens from each input place p of t, and adds w(t,q) tokens to each output place q of t, where w(t,q) is the weight of the arc from t to q (Figure 2.1). The number of tokens *M(p)* at each place represents a *marking* of the graph (state). A token can be thought of as representing some condition or holding some data items associated with that place. When an interpretation is given to the entities of a PN to represent a system, the tokens' movements will reflect the dynamic behaviour of the system.

Several examples can be given to introduce some concepts of Petri nets as useful modelling tools : finite-state machines, concurrency, dataflow computation, communication protocol, synchronisation control, multiprocessors systems, ... Areas of applications reported in the literature include cache coherence protocols, computer aided software engineering, telecommunication system, database system, fault tolerant system, production system, real-time control system, communication protocols, computer architecture, formal methods ... This list is by no means exhaustive, there are often multiple references in an area, and there are many areas other than those listed above. Further examples of applications are found in [Pet81, Rei85, Mur89].

2.2.1 Behavioural Properties

A major strength of Petri nets is their support for analysis of many properties and problems associated with concurrent systems. Two types of properties can be studied with a Petri net model: those which depend on the initial marking, and those which are independent of the initial marking. The former type of properties is referred to as *marking-dependent* or *behavioural properties*, whereas the latter type of properties is called *structural properties.* Behavioural properties include :

- *Reachability*: is the fundamental basis for studying the dynamic properties of any system. The firing of an enabled transition will change the token distribution in the net according to the transition rule. A marking M_n is said to be reachable from a marking M_0 if there exists a sequence of firings that transforms M_0 to M_n. The set of all possible markings reachable from M_0 is denoted by R(M_0).

- *Boundedness*: a place is k-safe or k-bounded if the number of tokens in that place cannot exceed an integer k. *Safeness* is a special case of boundedness, a place is safe if the number of tokens in that place never exceeds one.

- *Liveness*: a Petri net is said to be live if it is possible to ultimately fire any transition of the net by processing through some further firing sequence. This means that a live net guarantees deadlock-free operation.

- *Coverability*: a marking M is said to be coverable if there exists a marking M' in R(M_0) such that M'(p) $\geq$ M(p) for each p in the net.
- *Persistence*: a Petri net is said to be persistent if for any two enabled transitions, the firing of one transition will not disable the other.

A *conflict resolution strategy* is needed if the net is not persistent. A *decision place* is a place which is a source for more than one arc. Whenever it contains a token, its output transitions are in conflict because the firing of one disables the other.

2.2.2 Analysis Methods

Methods of analysis of Petri nets may be classified into the following three groups:

1. *Coverability tree*: given a Petri net (N, M_0), from the initial marking M_0 we can obtain as many new markings as the number of enabled transitions. From each new marking, we can again reach more markings. This process results in a tree representation of the markings.
2. *Incidence matrix and state equations*: is based on a matrix view of Petri nets. Two matrices D^- and D^+ are defined to represent the *input* (to the transitions) and *output* (from the transitions) respectively. Each matrix equation is m rows (one for each transition) and n columns (one for each place). The solvability of these equations is somewhat limited, partly because of the non deterministic nature inherent in Petri net models and because of the constraint that solutions must be found as non-negative integers.
3. *Reduction method*: it reduces nets to simpler nets while preserving properties such as boundedness or liveness by applying transformations which preserve these properties. The resulting nets might then be simple enough to be analysed by one of the standard techniques.

2.2.3 Semantics

Petri nets are a formalism that possess most of the desirable features :

- modelling and analysing concurrent systems;
- simplicity of the model;

- formality of the model;
- immediate graphical representation;
- easy representation of asynchronous aspects;
- possibility of reasoning about important properties (reachability, liveness, boundedness).

Many different Petri nets semantics have been proposed in the literature. At the most basic operational level we have of course the "token game". To account for computations involving many different transitions and for the causal connections between transition events, various notions of *process* have been proposed, but process models do not provide a satisfactory semantics denotation for a net as a whole. In fact, they specify only the meaning of single, deterministic computations, while the accurate description of the interplay between concurrency and nondeterminism is one of the most valuable features of nets [1].

Some semantics investigations, particularly those capitalising on the algebraic structure of Place/Transition nets, and the unification of the process-oriented and algebraic views are discussed in [Mes92b].

2.3 Types of Petri Nets

2.3.1 Background

Any developer of discrete event systems knows that the most important quality of the final system is that it must be functionally correct by exhibiting certain functional, or qualitative properties decided upon as being important. Once assured that the system behaves correctly, it is also important that it is efficient in that its running cost is minimal or that it executes in optimum time or whatever performance measure is chosen. While functional correctness is taken for granted, the latter quantitative properties will often decide the success (or otherwise) of the system.

Ideally the developer must be able to specify, design and implement his system and test it for both functional correctness and performance using only one formalism. Petri nets, although graphical in format are somewhat tedious for specifying large complex systems but, on the other hand were developed exactly to test discrete, distributed systems for functional correctness. With a Petri net specification one can test, eg., for deadlock, liveness and boundedness of the specified system.

[1]CSP (Communicating Sequential Processes) [Hoa85] does allow nondeterminism.

The major drawback of Petri nets, as originally proposed and process algebras (amongst others) is that quantitative analyses are not catered for. As a consequence, the developer who needs to know about these properties in his system has to devise a different model of the system which, apart from the overhead concerned provides no guarantee of consistency across the different models. Because of the latter, computer scientists added time, in various forms, to ordinary Petri nets to create *Stochastic Petri Nets* (SPNs) [Ajm89] for performance modelling and a great deal of theory has developed around SPNs as these are generically known.

2.3.2 Time Association with Petri Nets

Another aspect which also contributed significantly to the development of Stochastic Petri nets is the fact that their performance analysis is based upon Markov theory. Since the description of a Markov process is a "tedious task", abstract models have been devised for their specification. Of these, *Queueing Networks* (QNs) were originally the most popular, especially since the analysis of a large class of QNs (product-form QNs) can be done very efficiently. QNs cannot, however, describe system behaviours like blocking and forking and with the growing importance of distributed systems this inability to describe synchronisation naturally turned the focus to Petri nets as well.

Stochastic Petri nets are therefore a natural development from the original Petri nets because of (1) the advantage of their graphical format for system design and specification; (2) the possibility and existing rich theory for functional analysis with Petri nets; (3) the facility to describe synchronisation, and (4) the natural way in which time can be added to determine quantitative properties of the specified system.

The disappointing thing about Stochastic Petri nets is that the integration of time changes the behaviour of the Petri net significantly. So properties proven for the Petri net might not hold for the corresponding time-augmented Petri net. e.g., a live Petri net might become deadlocked or a non-live Petri net might become live. Thus, analysis techniques developed for Petri nets are not always applicable to SPNs. Also, using Stochastic Petri nets to specify the sharing of resources controlled by specific scheduling strategies is difficult. So certain concepts from queueing theory have been introduced to *Queueing Petri Nets* (QPNs) which offer the benefits of both worlds, Petri nets and queueing networks.

Adding Time to Petri Nets

Time has been added as an extra feature to Petri nets in three different ways which are sketched here:

1. Each transition is associated with a time interval [MF76]. The lower (respectively upper) bound of such interval gives the minimum (respectively the maximum) delay, computed with respect to the time instant at which the transition becomes enabled, from which (respectively to which) the transition fires, if not disabled by another transition firing in the meanwhile. When the maximum delay is reached and the transition has been continuously enabled from the minimum delay, the transition must fire.

2. Each transition is associated with a duration [Ram74, RH80]. When the transition is enabled, it immediately fires and removes the enabling tokens from the places of its preset. The tokens disappear and new tokens are created in the postset of the transition when the duration associated with the transition is elapsed.

3. Each place is associated with a duration [CR83]. A token created by a transition firing in a place becomes ready, i.e., it can participate in enabling a transition only after the delay associated with the place is elapsed. A transition fires instantaneously as soon as it becomes enabled.

2.3.3 Timed Petri Nets

Petri nets we described in §2.2 have a limited modelling power. To remedy this, a number of extensions have been proposed. A lot of properties of systems involving time, particularly issues of performance evaluation and simulation, can be covered by decorating Petri nets with requirements of timing: occurrence of transitions or residence of tokens in places are assumed to take a distinguished amount of time. These extensions have been widely adopted, either for necessity (eg. time is sometimes essential for performance evaluation) or for convenience purpose. Petri illustrates several semantics difficulties engendered by the introduction of time to nets [Pet86]. We show in this section how to consider the concept of time in Petri nets and present some kinds of timed Petri nets reported in the literature.

The introduction of time changes the semantics of firing. As an example, instead of tokens being deposited in transition's output places at the instant of firing, the tokens are deposited after a delay chosen from the firing time distribution of the transition. There are various types of timed Petri nets reported in the literature. *Timed Petri Nets* (TPNs) are nets having deterministic delays [Mol85]. *Stochastic Petri Nets* (SPNs) are obtained by associating a nondeterministic delay (which represents the enabling time) with each transition. When a transition is enabled by a marking, a value is randomly chosen from the associated variable. This value reflects the duration of the transition enabling period after which the transition is fired. As a matter of fact, in a SPN, the transition fires after an exponentially distributed amount of time [Ajm89]. Since the first definition of the SPN several extensions have been made. *Generalized Stochastic Petri Nets* [ACB84, ABC+91, ABC+95] are stochastic Petri nets which allow transitions with zero firing time (called *immediate* transitions) and exponentially distributed firing time, inhibitor arcs and random switches. Timed transitions are assumed to have the lowest priority level, whereas transitions at other priority levels are said to be n-immediate, where n is the priority level. Subsequently, *Extended Stochastic Petri Nets* (ESPNs) are proposed in which the most additional feature is represented by the presence of probabilistic arcs that upon firing of a transition may deposit tokens on subsets of its output set depending on a probability distribution [DTGN84]. *Deterministic Stochastic Petri Nets* (DSPNs) [AC87] allow transitions with zero firing time or exponentially firing time or deterministic firing time.

Stochastic Petri nets represent a formalism that is particularly interesting for its peculiar feature of being a useful modelling language for studying the performance of parallel systems [Bal92, WH94]. For example, they represent a formalism that is capable of representing both the characteristics of the architecture (hardware) and the pecularities of the program (software) of a parallel computer in such a way that both validation and performance evaluation can be performed using basically the same model [BDF92]. They are also useful for studying the correctness of parallel programs, and for performance oriented parallel program design [BBCC92, Fer92].

It has been shown that Stochastic Petri nets are isomorphic to continuous time Markov chains due to the memoryless property of the exponential distribution of firing times. A stationary embedded Markov chain can be recognised [Ajm89]. The system is represented by a Stochastic Petri net, and the reachability graph is constructed. Some analytical performance results are obtained if the firing time distribution functions associated with the transitions are exponential. If they are completely arbitrary, it is necessary to resort to simulation. From the steady-state distribution, performances such as the sojourn time in a state, steady state probabilities of marking, flow of tokens through a transition, the expected value of the number of tokens, the mean number of firings in unit time can be computed [Mol85, Pag86].

Modelling with Petri nets in general has to be supported by computer tools [Fel93]. Analytical evaluation and discrete-event simulation of Petri net models allow researchers to perform qualitative as well as quantitative analysis of the systems they model. A number of simulators are available for different classes of Petri nets. *GreatSPN* (GRaphical Editor and Analyzer for Timed and Stochastic Petri Nets) [CFGR95] is a software package for the modelling, validation, and performance evaluation of distributed systems using GSPNs. *TimeNET* [Kel95] is a software package for the modelling and evaluation of SPNs in which the firing times of the transitions may be exponentially distributed, deterministic, or more generally distributed.

According to [CF93b] a *Timed Transition Petri Net* is a tuple TTPN = (PN, Π, λ) where :
PN is a Petri Net,
$\Pi : T \mapsto N$ assigns priorities to T-elements,
$\lambda : T \mapsto R$ assigns firing delays to T-elements.

A complete list of references on the association of time with nodes of the models described with the Petri net formalism is found in [BBB+94]. The aspect of time in Petri nets is still an active area of research. Recently, processes have been defined and successfully utilised for some net classes with time [VdFC95, AL97].

2.3.4 High-Level Petri Nets

Ordinary Petri nets fail to represent complex functional aspects. Due to that, high-level Petri nets [JE91] have been proposed as a different class of extensions of Petri nets, and allow the representation of functional aspects in full details. They address the problem of dealing with data, their flow and transformations. Several high-level nets can be found in the literature. We briefly review the best known and applied net models.

Predicate-Transition Nets [Gen86] are defined as formal objects that can be interpreted and manipulated in a mathematical way that is comparable to working with logical formulae and algebraic expressions. In a *Coloured Petri Net* [Jen92], an information is attached to each token. The information can be inspected and modified when transitions fire. Coloured nets and Predicate-nets are very closely related to each other.

Stochastic High-Level Petri Nets are based upon high-level nets augmented with exponentially distributed transition rates. They allow tokens with multiple attributes, and the predicates associated with transitions may be expressed in terms of the attributes of the tokens present in the input places of the transitions [LM88]. *Generalized Stochastic High-Level Petri Nets* are obtained from stochastic high-level Petri nets by the introduction of immediate transitions with priorities, inhibitor arcs and cases [Car89].

A *Regular Net* (RN) is a Coloured Petri Net in which the colour domains of places and transitions are made of any cartesian product of basic object classes, each class appearing no more than once in the product [DH90]. A *Regular Stochastic Petri Net* (RSPN) is a timed extension of a Regular Net, in which transitions are either immediate or have exponentially distributed firing delays. *Well Formed (Coloured) Nets* (WN) are formally defined as an extension of Regular Nets and have the same modelling power as general Coloured Petri Nets, i.e., any CPN can be translated into an equivalent WN model with the same underlying structure; only the expression of the colour functions and of the composition of colour classes is rewritten in a more explicit (and parametric) form, in terms of the basic constructs provided by the WN formalism. *Stochastic Well-Formed Nets (SWN)* [CDFH93] are an extension of Regular Stochastic Petri Nets and Well-Formed (Coloured) Nets.

Environment/Relationship (ER) Nets are high-level nets where tokens are *environments*, i.e., functions associating values to variables and an *action* is associated with each transition, describing which input tokens can participate in a firing and which possible tokens are produced by the firing [GMMP91]. An *Interval Timed Coloured Petri Nets* (ITCPN) is a coloured Petri net extended with time which models large and complex real-time systems [Van93]. Time is in tokens and transitions determine a delay (specified by an upper and lower bound, i.e., an interval) for each produced token.

Object Petri Nets [Lak95] support a complete integration of object-oriented concepts into Petri Nets, including inheritance and the associated polymorphism and dynamic binding. In particular, Object Petri nets have a single class hierarchy which includes both token types and subnet types. Interaction between subnets can be either synchronous or asynchronous depending on whether the subnet is defined as a super place or a super transition. The single class hierarchy readily supports multiple levels of activity in the net and the generation and removal of token has been defined so that all subcomponents are simultaneously generated or removed, thus simplifying memory management.

A new class of timed Petri nets for the specification of temporal constraints and description of logical behaviour in distributed hypermedia systems is proposed in [SdSSW95].

2.3.5 High-Level Algebraic Nets

High-level algebraic nets have been introduced in order to exploit the rich theory of algebraic specifications for high-level Petri nets: Petri nets gain a great deal of modelling power by representing dynamically changing items as *structured tokens* whereas algebraic specifications turned out to be an adequate and flexible instrument for handling *structured items*. Multisets over any domain can be specified by sorts, operations and equations. The concept of algebra semantics, relating terms to interpretations, appear to be directly applicable to high-level Petri nets.

Vautherin [Vau87] redefines the domains of coloured nets in algebraic terms and gives a number of interesting results for deriving properties of the modelled system through the application of standard analysis techniques to the underlying Place/Transition net. Reisig [Rei91] uses the algebraic formalism to construct Petri nets with structured tokens which turn out to be representable by established concepts of algebraic specifications. A class of high-level nets (called OBJSA) which use algebraic techniques instead of (multi) set theory for specifying the individual tokens flowing into the net is proposed by Battiston et al. [BCM88]. *OBJSA Net Systems* are a class of modular high-level algebraic Petri nets in which : (1) the net can be decomposed in state-machine components; (2) the domains to which individual tokens belong are defined as abstract data types using the language OBJ [GKK$^+$88]. Briefly, an OBJSA Net System is a couple C = (N, A) where N is an extended SA (Superposed Automata) net and A is an OBJ algebraic specification. The application of OBJSA nets to a realistic case study for what concerns dimensions and complexity is found in [BBCC95]. The algebraic framework and Petri nets appear suitable to the study of properties of systems (eg. safety properties) which can be expressed by equations over the set of reachable states. Other properties like liveness for instance are more difficult to treat in general because they do not only depend on the set of reachable states, but also on the executions of these systems. Recently Schmidt proposed a symbolic approach for the verification of siphons and traps for algebraic Petri nets based on structural induction on the terms [Sch97].

2.4 Extended Concurrent Algebraic Term Nets

2.4.1 Introduction

ECATNets (Extended Concurrent Algebraic Term Nets) are a kind of high-level algebraic nets which combine high-level Petri nets with algebraic data types. They are used to model and simulate various aspects of distributed and parallel systems, communication networks [BMSB93b], concurrent programming [CD97], manufacturing systems [MBBP97, BCD98] ...

ECATNets are given semantics in terms of a rewriting logic that differentiates them from other algebraic nets and makes them suitable to handle true concurrency. They are built around a combination of three formalisms. The first two formalisms constitute a net/data model, and are used for defining the syntax of the system, in other terms to capture its structure. The net model, which is an ordinary Petri net [Mur89], is used to describe the process architecture of the system; the data model, which is an algebraic formalism [EM85], is used for specifying the data structures of the system. The third formalism, which is a rewriting logic [Mes92a], is used for defining the semantics of the system, or in other words to describe its behaviour. According to this logic, the system behaviour may be explained by formal reasoning. Transforming this logic into a rewriting system [BM93b] may be used for rapid prototyping and automatic proving of a system under design. More details about ECATNets, their motivation and relation to other works are found in [Bet91, BMSB92, BC92, BM93a].

2.4.2 From ATNets to ECATnets

Motivating ECATNets leads to motivating Petri nets, abstract data types, as well as their association into a unified framework. Petri nets are used for their foundation in concurrency and dynamics, while abstract data types are used for their data abstraction power and solid theoretical foundation. Their association into a unified framework is motivated by the need to explicitly specify process behaviour and complex data structures in real systems.

ECATNets are an extension of CATNets (Concurrent Algebraic Term Nets), which themselves evolved from ATNets (Algebraic Term Nets), introduced for the first time by Bettaz in [Bet91]. The main difference between ATNets and CATNets is a lack of semantics for the first ones disabling them to handle truly concurrent systems. The formal definition of CATNets (syntax and semantics) is given in [BM93a].

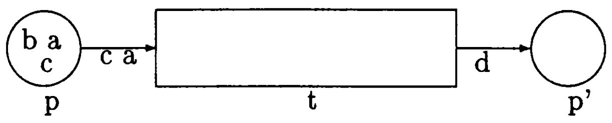


Figure 2.2: A Simplified CATNet.

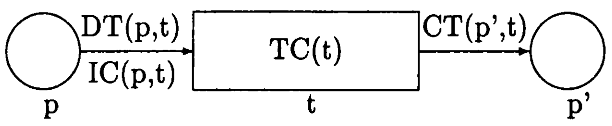


Figure 2.3: A CATNet.

Definitions

In ordinary Petri nets [Mur89], places and arcs are annotated by multisets of black dots, called tokens. From a syntactical point of view, the only difference between usual Petri nets and simplified CATNets is that places and arcs in simplified CATNets are annotated by multisets of algebraic terms (Figure 2.2), the syntax and semantics of which are given by abstract formal specifications called algebraic specifications [EM85]. Building of highly compact model often necessitates the use of powerful syntactic notations. For CATNets notations inspired from [WH87] are used and consist mainly in (Figure 2.3):

- Distinguishing the multiset of enabling tokens (Input Conditions: IC(p,t)) from the multiset of tokens which have to be removed when a transition t is actually fired. The removed tokens are called Destroyed Tokens and denoted by DT(p,t). The deposited tokens are called Created Tokens and denoted by CT(p',t).

- Annotating not only places and arcs but also transitions. However the transitions are annotated not by multisets of algebraic terms, but by boolean expressions, called Transition Conditions (TC(t)).

In some situations we are interested in firing a transition when its input place is empty. Bettaz et al. suggested in this situation to use the notation empty at the place of the multiset IC(p,t) [BMSB93a, BMSB93b]). In some other situations we would like to fire a transition if its input place does not contain a given multiset M of precised tokens. For this situation to use the notation $\tilde{}$M instead of the multiset IC(p,t) is also suggested in [BMSB93b].

Let CATNas(X) be a CATNet syntactic structure. The ECATNet syntactic structure denoted by $\text{CATNas}(X)^+$ is defined inductively as follows:
$\text{CATNas}(X) \subset \text{CATNas}(X)^+$
empty $\in \text{CATNas}(X)^+$
if $[m]_+ \in \text{CATNas}(X)$ **then** $\tilde{}[m]_+ \in \text{CATNas}(X)^+$
CATNas(X) will be called the ECATNet syntactic substructure.

An ECATNet is a structure (P, T, s, IC, DT, CT, C, TC) [BMSB92] where:
P is a set of places and T is a set of transitions;
s: $P \rightarrow S$ is a function that associates a sort with each place;
IC (Input Condition): $(P \times T) \rightarrow \text{CATNas}(X)^+$;
DT (Destroy Tokens) : $(P \times T) \rightarrow \text{CATNas}(X)$;
CT (Created Tokens): $(P \times T) \rightarrow \text{CATNas}(X)$;
C (Capacity): $P \rightarrow \text{CATNas}(\emptyset)$ is a partial function such that for every $p \in$ domain(C), $C(p) \in \text{CATNas}(\emptyset)$;
TC (Transition Condition): $T \rightarrow \text{CATNas}(X)_{bool}$ is a function such that for every $t \in T$, $TC(t) \in \text{CATNas}(X(t))_{bool}$ where X(t) is the set of variables occurring in IC(p,t) (when defined), DT(p,t) and CT(p,t) for every $p \in P$. X(t) will be called the transition context.

A marked ECATNet is an ECATNet with a function M: $P \rightarrow \text{CATNas}(\emptyset)$ such that for every $p \in P$, $M(p) \in \text{CATNas}(\emptyset)$ and $M(p) \subset C(p)$ **if** $p \in$ domain(C).

In a generic ECATNet, IC, DT and CT are multisets of (equivalence classes of) terms, with $\oplus$, $\cap$, $\subset$, $\setminus$ being respectively the multiset union, intersection, inclusion and difference, and ϕM the identity element. We let $[x]\oplus$ denote the equivalence class of x, w.r.t. the ACI (Associative, Commutative and with Identity element) axioms for $\oplus$. The terms are defined by an algebraic specification of an abstract data type given by the user [EM85]. We let [x]E (or just [x]) denote the equivalence class of x, w.r.t. the axioms (equations) given by the user in his (her) specification. TC (Transition Condition) is a boolean expression which may contain variables occuring in IC (Input Condition), DT (Destroyed Tokens) and CT (Created Tokens). Each place is associated with a capacity C(p) defined as a multiset of closed (equivalence classes of) terms. The marking M(p) of a place p of the net, which is itself a multiset of closed terms, is defined w.r.t. the capacity (which may be infinite). The extensions are related only to IC, and may be considered as an equivalent of the inhibitor arc concept as defined in [Bil89].

Transition firing and its conditions are expressed by rewrite rules which are strongly depending on the form of the syntactic notation used for representing IC. Those rewrite rules together with a set of deduction rules define a rewriting logic [Mes92a] which gives the semantics of the net. The left-hand and right-hand sides of the rewrite rules are multisets of pairs of the form (p,[m]⊕), where p is a place of the net and [m]⊕ a multiset of algebraic terms. The multiset union on the pairs (p,[m]⊕) is noted ⊗, and ϕB is the identity element for this case. Let us recall in the following part of this section the forms of the rewrite rules (metarules) to associate with the transitions of a given ECATNet [BM93a, BMSB92]. These metarules act as a parallelising compiler which tries to find sequences of "code" which may be executed in parallel. Examples on concrete instantiations and practical use of these metarules are found in [BMSB93a, BMSB93b, Bet93, BMSB94].

1 IC(p,t) is of the form [m]⊕

Case 1*[IC(p,t)]⊕ = [DT(p,t)]⊕*
The form of the rule is given by:
t: (p,[IC(p,t)]⊕) → (p',[CT(p',t)]⊕)
where t is the involved transition, p its input place, and p' its output place.

Case 2*[IC(p,t)]⊕ ∩ [DT(p,t)]⊕ = ϕM*
t: (p,[IC(p,t)]⊕) ⊗ (p,[DT(p,t)]⊕) ∩ [M(p)]⊕ →
(p,[IC(p,t)]⊕) ⊗ (p',[CT(p',t)]⊕)

Case 3*[IC(p,t)]⊕ ∩ [DT(p,t)]⊕ ≠ ϕM*
This case may be solved in an elegant way by remarking that it could be brought to the two already treated cases [BM93a].

2 IC(p,t) is of the form ~[m]⊕

t: (p,[DT(p,t)]⊕) ∩ [M(p)⊕] →
(p',[CT(p',t)]⊕) if ([IC(p,t)]⊕ \ ([IC(p,t)]⊕ ∩ [M(p)]⊕) = ϕM) → [false]

3 IC(p,t) = empty

t: (p,[DT(p,t)]⊕) ∩ [M(p)⊕]) →
(p',[CT(p',t)]⊕) if ([M(p)]⊕) → ϕM)

When the place capacity C(p) is finite, the conditional part of the rewrite rule includes the following component:

([CT(p,t)]⊕ ⊕ [M(p)]⊕) ∩ [C(p)]⊕) → [CT(p,t)]⊕⊕ [M(p)]⊕ **(Cap)**

In the case where there is a transition condition TC(t) the conditional part of our rewrite rule must contain the following component:[TC(t)] → [true].

Note that if one or more output place(s) has (have) a finite capacity, the conditional part of the rewrite rule must contain a component of the form denoted by **(Cap)** for each one of these places.

2.4.3 Rewriting Logic

A logic is understood as a method of correct reasoning about some class of entities. Rewriting logic is a logic of becoming or change, not a logic of equality, where a sequent: [t]→[t'] should be read as "[t] becomes [t']". The rules of rewriting logic are rules to reason about change in a concurrent system. They allow us to draw valid conclusions about the evolution of the system from certain basic types of change. For rewriting logic, the entities in question are concurrent systems having states and evolving by means of transitions. The rewrite rules in the theory describe which elementary local transitions are possible in the distributed state by concurrent local transformations. The distributed state of a concurrent system is represented as a term whose subterms represent the different components of the concurrent state. What the rules of rewriting logic allow us to reason correctly about is which general concurrent transitions are possible in a system satisfying such a description. Research has been carried out extensively on rewriting logic since it was introduced in the beginning of the nineties. A deeper presentation of this logic and its use as a semantic framework for concurrency is given in [Mes96].

2.4.4 ECATNets Semantics

It is worth to mention that it is not easy to explain the behaviour of ECATNets merely by giving the equivalent of a firing-like rule. This is because of their level of abstraction as well as their concurrent behaviour. We may however informally comment on this behaviour in the following way (see Figure 2.3). A transition t is fireable when various conditions are simultaneously true. The first condition is that every IC(p,t) for each input place p is enabled. The second condition is that TC(t) is true. Finally the addition of CT(p',t) to each output place p' must not result in p' exceeding its capacity when this capacity is finite. When t is fired DT(p,t) is removed from the input place p and simultaneously CT(p',t) is added to the output place p'.

ECATNets semantics enables the handling of truly concurrent systems. The rewrite rules describe Petri net transitions effects as elementary types of change. Such rules act in reality as the axioms of the higher mentioned rewriting logic. The axioms are in reality conditional rewriting rules describing transitions effects as elementary types of changes. The deduction rules allow us to draw valid conclusions about the evolution of the ECATNet from these changes. A rewrite rule is a structure of the form "t: u $\rightarrow$ v if boolexp"; where u and v are respectively the left and the right-hand sides of the rule, t is the transition associated with this rule, and boolexp is a Boolean term. More precisely u and v are multisets of pairs of the form (p, $[m]_\oplus$), where p is a place of the net, $[m]_\oplus$ a multiset of algebraic terms, and the multiset union on these terms, when the terms are considered as singletons. The multiset union on the pairs (p, $[m]_\oplus$) will be denoted $\otimes$. We let $[x]_\otimes$ denote the equivalence class of x, w.r.t. the ACI axioms for $\otimes$. An ECATNet state is itself represented by a multiset of such pairs where each place p is found at least once. Given a set R of rewriting rules (defining all the elementary types of changes), we say that R entails a sequent s $\rightarrow$ s' (defining a global change from a state s to a state s') iff s $\rightarrow$ s' can be obtained by finite and concurrent applications of the following rules of deduction: *Reflexivity, Congruence, Replacement, Splitting, Recombination and Identity* [BM93a].

The reflexivity rule says that everything may be transformed into itself. The congruence rule says that elementary changes have to be correctly propagated. The replacement rule is used when variable instantiations are necessary. The splitting and recombination rules allow us, by "judiciously" splitting and recombining different multisets of equivalence classes of terms, to detect ECATNet computations exhibiting a maximum of parallelism. The identity rule allows to relate Φ_M (i.e., the

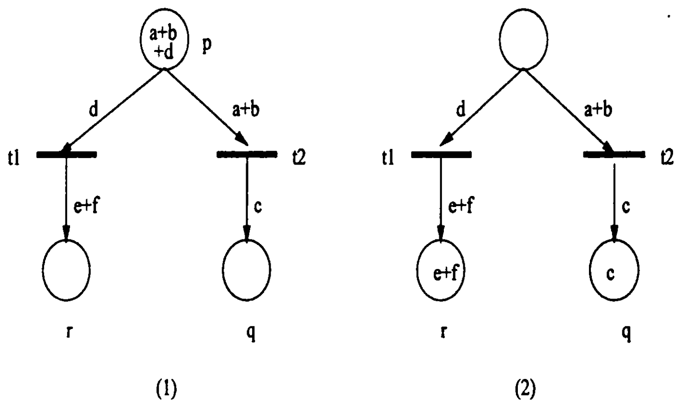


Figure 2.4: Firing in Parallel in ECATNets.

identity element of ⊕) with Φ_B (i.e., the identity element of ⊗). Once reviewed the basic notions about rewriting logic and its use for describing the semantics of ECATNets, let us now recall the forms of the rewrite rules (i.e., the metarules) to associate with the transitions of a given ECATNet. Examples on concrete instantiations and practical use of these rules are found in [BMSB93a, BMSB93b].

An ECATNet can be viewed as a model of concurrent system: if a certain number of transitions are enabled simultaneously and if they are not in conflict, they can be fired in parallel (Figure 2.4).

The rewrite rules associated with transitions t_1 and t_2 are:
t_1: (p,d) → (r,e ⊕ f),
t_2: (p,a ⊕ b) → (q,c)

and the initial state is given by: (p,a ⊕ b ⊕ c) ⊗ (q,∅) ⊗ (r,∅)

The deduction of the final state from the initial state may be performed using the rewriting logic associated with the net. The decomposition rules are:
(p,a ⊕ b ⊕ c) ⊗ (q,∅) ⊗ (r,∅) →
(p,a ⊕ b) ⊗ (p,d) ⊗ (q,∅) ⊗ (r,∅) →
(q,c) ⊗ (r,e ⊕ f) ⊗ (q,∅) ⊗ (r,∅) (* application of the rewrite rules in parallel *)
(q,c) ⊗ (q,∅) ⊗ (r,e ⊕ f) ⊗ (r,∅) → (* application of the structural axiom of commutativity *)
(q,c) ⊗ (r,e ⊕ f) ⊗ (p,∅) (* application in parallel of the structural axiom of identity *)

2.4.5 Example of an ECATNet

The objective of this example, borrowed from [BM93a], is not only to illustrate the use of ECATNets, but also to present their deficiences, in order to show the motivation of the proposed solution (§2.5).

The specification, from the area of computer networks [Tan96], deals with the behaviour of the Ethernet transmitting station. It comprises four modules (Figures 2.5 .. 2.8), each module is specified by an ECATNet model. The first module deals with the functions of formatting and transmitting starting. The second module is relative to the functions of transmission with success and acknowledgment. The third module treats essentially the functions of collision handling and acknowledgement. The fourth module is relative to the retransmission function.

The transmitter station transmits one frame at a time. The user is not allowed to request the transmission of a new frame before receiving the ackowledgement of the previous frame. The formatting function starts when a token of type "d,s,data" is deposited in place FROM_USER. This token is considered as a primitive transferred from the user layer to the MAC layer for requesting the transmission of data "data", from a source "s" to a destination "d". The place FROM_USER is an interface between the two layers. The frame "d.s.data.fcs" is then deposited as soon as it is composed in a transmission register (TRANS_REG). The formatting function is consisting of the concatenation of sequences of bits corresponding to the addresses "d" and "s", to the data "data", and to the error control sequence "fcs" previously computed. On the other hand, the MAC layer is listening to the medium (CARRIER_SENSE) in order to avoid any collision occurring with a current transmission. The place CARRIER_SENSE is an interface between the MAC layer and the physical layer. When the medium becomes free (a token "false" is present in place CARRIER_SENSE), it waits a certain amount of time corresponding to the inter-frame spacing delay. Then, considering that the transmission may terminate with success (deposit of a token "false" in place BUSY_CHANNEL and a token "true" in place SUC_TRANS), it takes possession of the medium (CHANNEL_ACCESS) and the transmission starts (deposit of a token "true" in INIT_TRANS).

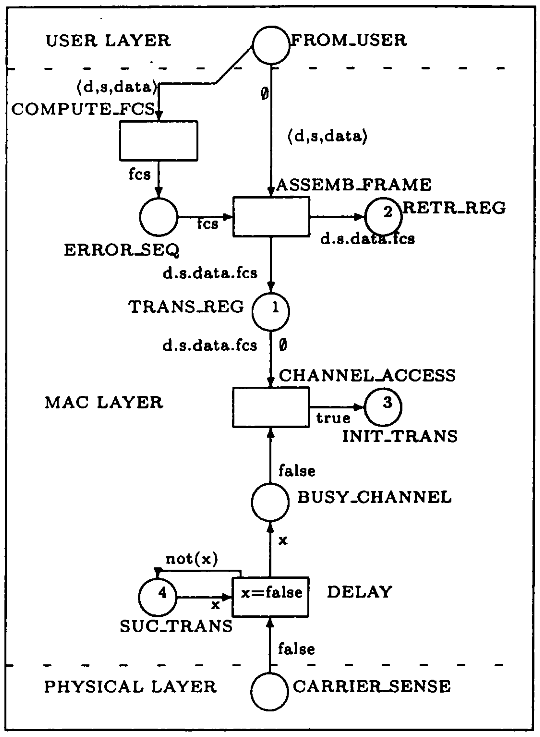


Figure 2.5: Starting of Transmission

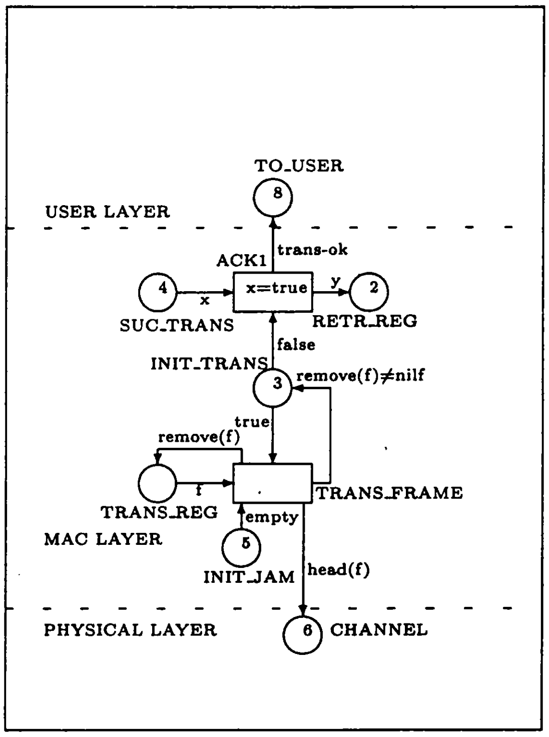


Figure 2.6: Transmission with Success

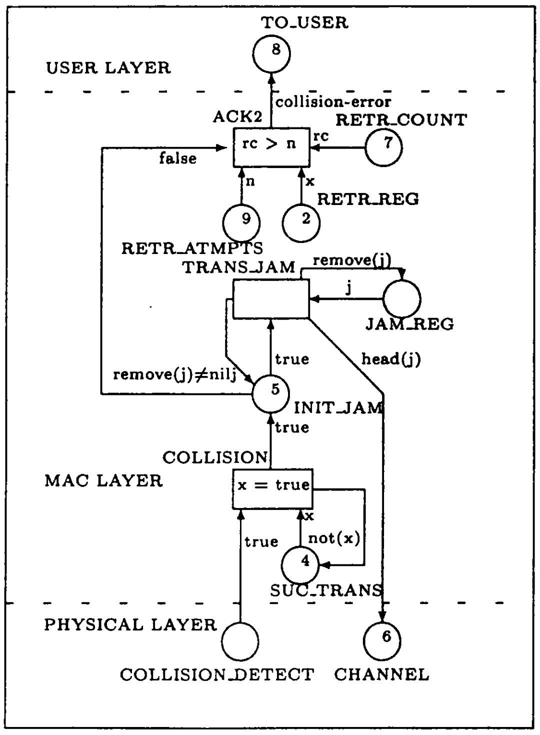


Figure 2.7: Collision Handling

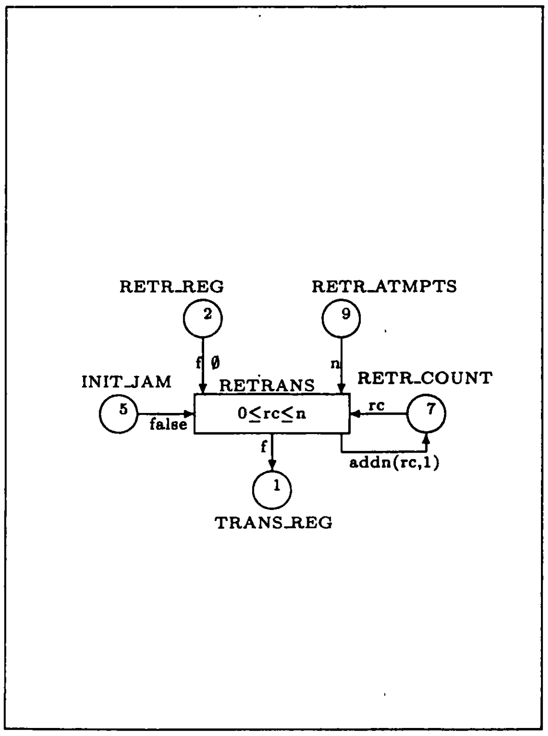


Figure 2.8: Retransmission

When we started this research we found out that there was no explicit specification of time in ECATNets. For instance, time is specified implicitely in module "Starting of transmission" (inter-frame spacing delay (action DELAY)). In module "Retransmission", the random waiting time before retransmission is also implicitely specified. If a collision occurs during the frame transmission (several nodes start transmission more or less simultaneously), the transmission is aborted and has to be repeated from the beginning (this state is called *backoff*). The backoff time must be random to avoid repetitive collisions and prevent a deadlock. It is computed anew on every collision. A variety of methods are used to determine this time:

1. make the backoff proportional to the nodes'address (ID): it is an efficient method but gives a certain type of priority to nodes with lower addresses. The address is defined so that every Ethernet station will have a unique address which is built into the hardware;

2. make the backoff time random: it does not have the implied priority property, but consecutive collisions may still happen. It is a uniformly random number of slot times chosen in the interval [0..2**min(rc,10)], where rc is the number of failed attempts.

2.5 Introducing Time in ECATNets

Previous works [BMSB93a, BM93b, BMSB93b, Bet93, BMSB94] showed how ECATNets are used for specifying and validating applications from the area of distributed and parallel systems. The achieved models have two drawbacks: the occultation of the problem of time and a bad exploitation of the parallelism inherent in the studied models. The objective of introducing time in ECATNets is twofold. The first objective is the need to specify practical applications where the explicit specification of time is "missing". The second objective is the need to turn to simulation because the formal specification of ECATNets is based on implementation concepts rather on theoretical ones.

ECATNets simulation is attractive because it can not only perform the validation of these models, but can evaluate their performances as well. Intuition suggests that simulation of these models may be amenable to parallel execution in order to exploit the inherent parallelism. It is worth mentioning that we are dealing with the parallelism at two levels: the inter-module level where parallelism is achieved by partitioning the "initial" models w.r.t. a "separation of concern" strategy, and the intra-module level where the detection of parallelism is permitted by the use of rewriting logic, since this logic may act as a "parallelising compiler" which tries to find sequences of "code" that can be executed in parallel.

2.5.1 Aspect of Time in ECATNets

The literature shows that Petri nets have been extended in different ways in order to incorporate the concept of time: associating time values with transitions or associating time values with places. We propose extensions for ECATNets which will permit to take into consideration the management of the aspect of time [DB94]:

1. the introduction of the notion of time in the token itself [MPT91]. A timestamp is attached to each individual token in order to represent the time it was created. Tokens, which are algebraic terms, can carry as much information as needed, including time information. This will lead to the firing of a transition depending on the replication of the token in input places. Created tokens are then defined by a multiset of the form CT(q, t, ts) where *ts* is a variable representing the timestamp (firing time of t);

2. in ECATNets, transitions are not labelled using multisets of algebraic terms, but using boolean expressions (transitions conditions (TC)). We may introduce in each transition a (marking) related rate. This will lead to take into account a firing time to perform the operations "remove/deposit tokens".

Research has been done to put time in high-level Petri nets. The relationships between high-level Petri nets and timed Petri nets are investigated by Morasca et al. [MPT91], where the generality of time representation in Environment/Relationship nets is assessed. It is shown how the mechanism for time representation introduced in ER nets is extended to both Coloured Petri nets and Predicate/Transition nets. The authors use the definition of *high-level nets simulation*, which can be *local*, *state-behaviour* or *local state-behaviour*. A unifying Petri net based model for time representation using ER nets is proposed by Ghezzi et al. to generalise most time Petri net based formalisms which appeared in the literature [GMMP91]. We have chosen to inscribe ECATNet transitions for the following reasons:

1. transitions are used to model the active parts of a system that can be assigned to a timed behaviour in a natural manner;

2. this choice allows to preserve the incremental approach used for defining ECATNets;

3. this choice preserves the semantic framework defined in terms of rewriting logic. Representing the concept of time at the level of places leads to replicating the corresponding token in several places;

4. this choice does not obscure the state of the system modelled by the ECATNet during the time that a process is in execution.

In our approach the form of the rewrite rule associated with a transition must guarantee the following constraint: the timestamp of the created token (CT) should be equal to the time the transition starts firing augmented with its firing time (right hand side of the rule). In some situations it is necessary to specify an activity duration or an action which must be performed before its deadline. At the rewrite rules level the concurrent execution of the rewrite rules associated with the transitions must be controlled and monitored. The rewrite rule associated with a transition is marked when the firing starts, and unmarked when firing ends.

Inscribing transitions to integrate timing aspects in ECATNets is achieved by introducing a new syntactic notation. Each transition t_i is specified with a firing rate λ_i. Let λ : T$\rightarrow$ $\mathcal{R}$ assigns firing delays λ_i to T-elements $t_i \in$ T, three major types of transitions are specified:

1. zero delays are associated with transitions that are called *immediate*;

2. *deterministic* timed transitions are annotated with a firing rate λ_i where $\lambda_i \in \mathcal{R}$ is a deterministic value;

3. *stochastic* timed transitions where λ_i is an instance of a random variable.

Markings that enable timed transitions only are said to be *tangible*, whereas markings that enable immediate transitions are said to be *vanishing*. If T contains stochastic timed transitions, the firing-delay random variable is *exponentially* distributed.

2.5.2 Firing Semantics

ECATNets integrate two different kinds of timing aspects and both relate to transitions. ECATNets offer *delay times* as defined for Stochastic Petri Nets [Ajm89]. They also provide *firing times* as defined for Timed Petri Nets [RH80], which are usually preferred for modelling an activity duration while delay times are more suitable to represent a waiting period or a preparation time. Two firing policies can be defined for ECATNets, leading different semantics and intents for different modelling domains:

1. A transition firing is *atomic*, in that removing tokens from input places (DT) and depositing tokens in output places (CT) are a single indivisible operation (Figure 2.9). A delay elapses between the enabling and the firing of the transition t_i, during which the enabling tokens reside in the input places (t_i must be continuously enabled during the time λ_i, and must fire after that time; t_i can also become disabled by the firing of another transition). Considering the example of Figure 2.5, when transition DELAY is enabled to fire at time *Tsim* and a firing delay *d* is associated with this transition, then :

   - at time *Tsim* the transition is enabled to fire;
   - a firing delay is elapsing from *Tsim* to *Tsim*+*d*;
   - at time *Tsim* +*d* the transition fires; appropriate tokens are removed from its input places (CARRIER_SENSE and SUC_TRANS) and deposited in its output places (SUC_TRANS and BUSY_CHANNEL).

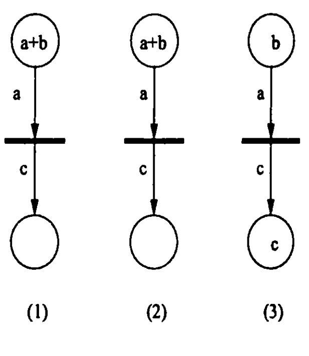


Figure 2.9: Firing Behaviour in ECATNets (Atomic Firing).

2. When the transition is enabled, it fires in three phases. It immediately removes the enabling tokens (DT) from the places of its preset. The tokens disappear and new tokens (CT) are created in the postset of the transition when the duration associated with the transition is elapsed (Figure 2.10). Examples of such firing semantics are found in chapter 8.

These two different firing semantics affect the construction of the ECATNets simulation engine (sequential or parallel) because of the event-list management.

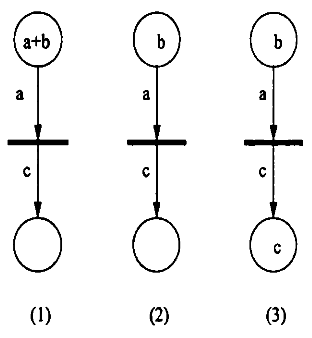


Figure 2.10: Firing Behaviour in ECATNets (Three Phase Firing).

2.6 Conclusion

In this chapter we have reviewed a series of concepts related to Petri nets in general and to high-level algebraic nets in particular. The presentation has been purposefully biased towards Extended Concurrent Algebraic Term Nets which are built around a combination of three formalisms: the net structure (a P/T net), the data model (an algebraic formalism) and the rewriting logic (to describe the system's behaviour). This decision has been motivated by the use of ECATNets as a powerful modelling tool for research.

This chapter has also served to introduce the reasons of our interest in adding time to ECATNets. There was no explicit specification of time in ECATNets when we started this research. These nets enriched with temporal specification are suitable to discrete simulation. This is an important step in their quantitative performance evaluation. In this sense, the use of transition timed Petri net formalism provides a substantial contribution to the implementation of efficient, general purpose discrete event simulation techniques. The reachability set of a timed ECATNet is identical to the one of the underlying P/T net model with inhibitor arcs. Therefore some of the structural properties valid for the basic underlying Petri net are retained by the ECATNet model. At present, firing times of the transitions are immediate, deterministic or exponentially distributed. Obviously there is a great scope for further work in tailoring Petri net analysis techniques to ECATNets.

Modelling with ECATNets in general has to be supported by a computer tool. A main goal of our research project is to develop a user-friendly and efficient tool for modelling with ECATNets. Due to the concepts of token types which are algebraic terms and timing concepts of ECATNets, a formal analysis of ECATNet models is not an easy task: an ECATNet model must be simulated in order to get information on its features. Simulation can not only perform the validation of ECATNet models, but evaluate their performances as well. A problem with the simulation of any kind of model is that it takes enormous amount of time to execute, if large or complex systems have to be treated in a detailed manner. In this case simulation of these models could be amenable to parallel execution in order to exploit the inherent parallelism.

Chapter 3

Distributed Discrete Event Simulation

This chapter surveys the literature about parallel and distributed discrete event simulation, with the purpose of introducing the terminology and algorithms used in the remainder of the thesis. After a general introduction to discrete event simulation, it is shown how the concept of causal order is the key element which allows the parallelisation of simulations, when used instead of temporal order. Three distributed simulation algorithms used in this research are then introduced: conservative (CMB), optimistic (TW) and synchronous (SYNC).

3.1 Introduction

Most of the fields of science and technology require the modelling and analysis of the behaviour of systems. Common to realistic models of time dynamic systems is their complexity, very often prohibiting numerical or analytical evaluation. Prototyping is a complementary tool to use, but in many cases it is very expensive and in others it is absolutely infeasible. For those cases, simulation remains the only tractable methodology. Also, simulation is an easily controlled and guided methodology.

As computer simulation is a wide field, our domain of study will focus on discrete event systems simulation. We describe the sequential approaches of this class of simulation problems, then a survey of the currently available techniques to realise parallel and distributed discrete event simulation is made. The methods and terminology described in this chapter will be repeatedly used in the remainder of the thesis.

In this chapter we start considering the general topic of analysing systems by means of computer simulation (§3.2), and then the sequential simulation algorithms for discrete event systems are introduced (§3.3). As these algorithms become very expensive in memory demands and execution time when the problem being simulated is large, the need of parallel processing to perform the simulations become evident. Several approaches to the parallelisation of discrete event simulators are presented, with special attention to event-driven simulations based on model decomposition (§3.4). Sections 3.5, 3.6 and 3.7 are devoted to the description of three important families of parallel simulators: asynchronous conservative, asynchronous optimistic and synchronous. The basic algorithms are introduced, with their most important variations or optimisations. In §3.8 a review of the literature on the application of DDES techniques to modelling tools is done, summarising the main results of studies similar to ours. Finally, some conclusions and some directions for an interest in a deeper insight into this field are given in §3.9 .

3.2 Modelling and Simulation

The simulation of real systems using computers needs, at least, four steps:

1. study of the real system in order to understand its characteristics;
2. modelling the system;
3. simulation of the model;
4. analysis of the simulator's output.

The systems to study can be separated into two categories: discrete or continuous. A system is *discrete* when its *state* changes only at discrete times, whereas a system is *continuous* when its state varies continuously in time.

A model being simulated can be classified as static or dynamic, deterministic or stochastic, and discrete or continuous. A model is *static* when it tries to capture snapshot of a system, at a particular instant of time, and it is *dynamic* if it tries to represent the evolution of the system along a certain interval of time. A model is *deterministic* when it generates, for a given set of input values, a single set of output values; it is *stochastic* when random variables are part of the input and, therefore, the output can only be considered as an estimate of the actual behaviour of the system.

3.3 Discrete Event Simulation

A system to be simulated is defined to be a collection of *entities* that interact and operate to accomplish some logical end. These entities are discrete objects, each being separate from all the others. Entities possess certain properties called *attributes* that affect the behaviour of the entities within the model. The *system's state* is the collection of attributes or *state variables* that represent the entities of the system. An *activity* represents a time period of a specified length. Entities may be in one of two states: either they are busy, engaged in some activity, or they are idle, doing nothing but waiting for the arrival of an event. An *event* is defined as an instantaneous occurrence that may alter the state of the system.

In discrete event simulation, changes in state of the model being simulated occur at discrete points in time. Fundamental to every simulation study is the mechanism to model the passage of time. Thus, every model contains a state variable called the *simulation clock.* Major *world views*, the lens through which the underlying modelling paradigm views the model, include *Event Scheduling, Activity Scanning,* and *Process Interaction* [LK91]. The differences between the world views lie in the way in which manner the model processes the events. The differentiation among world views is best captured using the concept of locality:

- *Event scheduling* provides locality in time: each event routine in a model specification describes related actions that may all occur in a single instant.
- *Activity scanning* provides locality of state: each activity routine in a model specification describes all actions that must occur due to the model assuming a particular state.
- *Process interaction* provides locality of object: each process routine in a model specification describes the entire action sequence of a particular model object.

  The object-oriented approach provides powerful modelling concepts to support computer-based tools for complex system design. Discrete-event simulation has a long history of association with the object-oriented paradigm and provides the ability to study the dynamic behaviour of models that are defined with object-oriented means [Zei91].

We consider that a real system, or *physical system* is modelled as a *physical process* (PP) which evolves in time. It is assumed that a *global clock* exists which can be used as a reference of the advance of time in the system. A *process* is a sequence of events that may contain several activities. The model also maintains a list of events that have been scheduled but that have not occurred yet, called the *future event list*, ordered by increasing occurrence time.

Events contain two fields of information: the event they represent, and the time where that event should happen (its *time_of_occurrence*). We assume that PP has a certain ability to predict the events that will occur in a next future. When PP knows that at time t ($t \geq clock$), it will schedule an event of type e, this scheduling action is modelled as an insertion of an event $<e,t>$ in the *future event list* (also called *event calendar* or *event queue*). The restriction of t belonging to the future is self-explanatory: the past cannot be affected by a present event. The event $<e,t>$ will be consumed in PP when the clock reaches the value t. As a result, PP's state will change accordingly to the class of interaction modelled by e. This state change can trigger the scheduling of new events for the future and, therefore, their insertion in the future event list.

Sometimes an event previously scheduled for the future needs to be cancelled before it actually happens (i.e., before the clock reaches the event's time of occurrence). The time of occurrence of an event says when the event *should* happen. An already scheduled event for time t can be cancelled by means of another event timestamped less than t.

Once we have a system modelled the way just described, and expressed in an executable using either a simulation language or a general-purpose programming language, the model can be simulated in a computer. Simulation clock is advanced using one of the two approaches: fixed-increment time advance (*time-driven*), or next-event time advance (*event-driven*) [Gar90].

3.3.1 Time Driven Approach

In this approach, in each step of the algorithm the clock advances one time unit. After doing so, all the state variables are examined, to check which events must occur at that particular time: those whose time of occurrence equals the value of the clock. Then, those events are *consumed*. Consuming an event produces the following effects:

- a change in the state of the system, i.e., in its state variables;
- new events might be scheduled for the future;
- some previously scheduled events might be cancelled.

These two steps (clock advance, event consumption) are repeated until the simulation finishes. Usually this happens when the clock reaches a given *end_of_simulation* value, or when the system reaches a particular state. As it can be seen from the description, the advance of the clock determines the advance in the simulation, and in each step exactly one time unit is simulated. However, in many systems events occur with a large time difference between each other, in such a way that, in most of the iterations of the algorithm, there are none (or just a few) events to consume. In these cases we have a low *event density* where the event density is defined as the (average) number of events consumed per unit of simulated time.

3.3.2 Event Driven Approach

Low event density scenarios led to an *event-driven* approach, where the clock can advance faster than it does in a time-driven simulator. The main elements of an event-driven simulator are, as in the previous case, a clock, a set of state variables, and an *event list.*

The first message in the event list is the one with the minimum *time_of_occurrence.* In each step of the algorithm this message is removed and the clock advanced to reach that simulation time_of_occurrence. The event is consumed with the effects already described: change of the PP's state, scheduling new events, cancellation of old messages. The way of advancing the simulation clock determines the difference between time-driven and event-driven simulation: in the last case, after consuming an event, the clock advances to reach the value of the next event's time_of_occurrence, with time jumps which might be larger than one unit of time.

3.3.3 Exploiting Parallelism

The sequential discrete event simulation algorithms become very expensive in memory demands and execution time when the problem being simulated is large. The need of parallel computers to perform the simulation becomes evident.

Since case studies may be given in a modular way using discrete systems, it seems reasonable that the inherent parallelism in these systems can be exploited by simulation. In discrete simulation, the inherently sequential nature of the global event list manipulation limits the potential parallelism of simulation models. By eliminating the global event list, additional parallelism can be obtained. Using multiple processors for this simulation appears to be a promising approach for a better modelling. The use of multiple processors can also improve the simulation execution time, because simulation of complex (discrete event) systems is usually exceedingly slow. There are five ways of decomposing a simulation for processing on multiple processors [RW89]:

- use of parallelising compilers: such compilers try to find sequences of the code that can be done in parallel and schedule them on separate processors. Such compilers ignore the structure of the problem and may exploit a small portion of the available parallelism;
- do separate runs on separate processors: the simulation is replicated on N processors and an average of the results is done in the end. There is no coordination between the processors, but a long run simulation might be preferred to N short runs. Heidelberg considers the statistical properties of estimators obtained by running parallel independent replications of a discrete event simulation on a multiple processor computing system [Hei88];
- put different subroutines on separate processors: a set of processors is dedicated to some functions like random variable generation, statistics and file manipulation. This approach does not exploit any of the parallelism in the system being modelled;
- maintain a global event list and process the next event in the list by a processor as soon as it becomes available. A protocol for consistency is required for this approach since the next event in the list may be affected by events currently being processed;
- simulate different system components by different processors. This approach shows the greatest potential in terms of exploiting the inherent parallelism of the system.

In the next section, we introduce parallel discrete event simulation and show that it falls into two categories: conservative and optimistic. The main difference between these two mechanisms is how they deal with causality errors.

3.4 Distributed Discrete Event Simulation

A simulation model may be used to predict the behaviour of a physical system under a variety of operating conditions. In the process-interaction approach to simulation, a physical system is assumed to consist of a set of Physical Processes (PP) that interact with each other at discrete points in time. In its simulation model, a Logical Process (LP) is used to model one or more Physical Processes. The events in the physical system are modelled by message exchanges among the corresponding Logical Processes in the model.

Parallel Discrete Event Simulation (PDES) refers to the execution of a single discrete event simulation program on a parallel computer. The system being modelled is viewed as being composed of some number of PPs that interact at various points in simulated time. The simulator is constructed as a set of LPs, one per physical process. An event is represented by a timestamped message, LPs exchange timestamped event messages to interact. However, relationship between events may exist, so concurrent execution of these events must be synchronised, otherwise *causality errors* can occur. So a certain sequencing constraints must be maintained in order for the computation to be correct. Parallel Discrete Event Simulation falls into two categories [Fuj90]: conservative and optimistic. The conservative mechanisms strictly avoid the possibility of any causality error ever occurring [Mis86]. The optimistic mechanisms use a detection and recovery approach, this means that causality errors are detected and a rollback mechanism is invoked to recover. The most well-known optimistic protocol is the Time-Warp mechanism based on the virtual time paradigm [Jef85][1]. Notions such as causality, virtual time, clock synchronisation, organisation and exploitation of timestamps, and a lot of related concepts are reported in [CM79, CM81, Mis86, Jef85, Fuj90]. The idea in [Lam78] is meanwhile a standard concept in many models of concurrent computation. Parallel Discrete Event Simulation (PDES) refers to an implementation for a shared memory machine (tightly coupled multiprocessor) whereas Distributed Discrete Event Simulation (DDES) refers to an implementation for a machine with communication based on message passing (loosely coupled multiprocessor). A survey of the literature on parallel simulation has been reported by Kaudel [Kau87], Fujimoto [Fuj90], Ayani [Aya93] and Ferscha [Fer96].

3.4.1 Event Dependencies

As mentioned in the previous section, the main challenge of DDES techniques is to guarantee that the causal dependencies among events are respected. The simulation of an event cannot be allowed to affect previously simulated events, otherwise the simulation would be incorrect. In a sequential event-driven simulation, events are processed in the right order, because in each iteration the event with the minimum

[1]Protocols such as the *probabilistic* one [CF95], a performance efficient compromise between the two classical approaches are found in the literature.

timestamp is selected, and this choice guarantees that the event dependencies are observed. A formal proof of the correctness of the sequential simulation is found in [Mis86]. In this section we will formally define the classes of event dependencies that must be observed in any event-driven simulation, sequential or parallel.

Definition 1: we say that event e_i *affects* the execution of e_j if at least one of these situations arise:

- the execution of e_i creates or cancels e_j;
- the execution of e_j reads or updates state information that was created or altered by the execution of e_i.

In any case, it is assumed that the timestamp of e_i is strictly less than the timestamp of e_j, because in a real system an event cannot influence past events.

Definition 2: we say that event *a causally affects* event *b* (or that b *causally depends* on a) if there is some chain of events a = e_0, e_1, e_2, ..., e_n = b such that, for each pair e_i and e_{i+1}, the execution of e_i affects the execution of e_{i+1}.

Definition 3: given two events *a* and *b*, if neither *a* causally affects *b* nor *b* causally affects *a*, then we say that *a* and *b* are *causally independent.* In particular, note that any two events with exactly the same timestamp are causally independent by assumption. The "causally affects" relation defines a partial order on the events in a simulation.

In a sequential simulation events are executed in non-decreasing timestamp order. As several events might have the same timestamp, they can be consumed in any order, even concurrently. This gives us the idea that *some* actions can be parallelised in the simulator. However, if it is not common to have equally timestamped events, this does not mean that no parallelism is available.

The objective is, then, to concurrently execute events with different timestamps. To do so, we need to relax the requirements of executing events in temporal order, using instead the defined causal order. Given a traditional sequential event-driven simulator, and considering the previous definitions, a parallel simulator that executes all the pairs of causally dependent events in causal order satisfies these properties:

- exactly the same events are executed in the parallel simulator and in the sequential one;

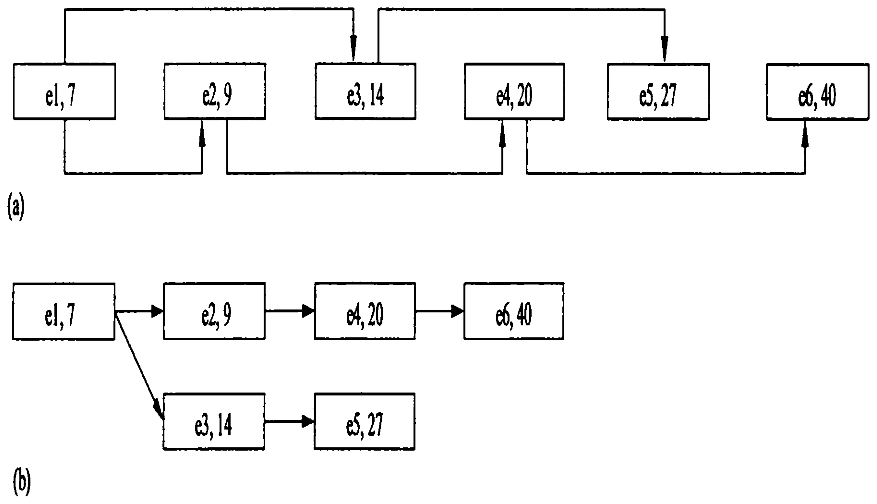


Figure 3.1: (a) List of Scheduled Events in Timestamp Order. (b) Sequence Ordered by Causal Dependencies.

- when a given event is executed, the portion of the state of the system that affects the simulation of that event is exactly the same in the parallel simulator and in the sequantial one.

In other words, a simulation that executes events in any order consistent with the causal order is indistinguishable from a simulation that executes events in temporal order. Obviously, the temporal order imposed by a sequential simulator is consistent with the causal order, but the opposite is not always true. For this reason, imposing a temporal order is unnecessarily restrictive. The most important asynchronous DDES methods precisely try to take advantage of the more relaxed causal ordering to simultaneously execute events with (potentially) different timestamps. Figure 3.1 depicts an example of restrictions imposed by causal dependencies, and shows how the sequence of events (e_2, e_4, e_6) can be executed in parallel with the sequence (e_3, e_5). However, if any event were simulated in parallel with e_1, the causal dependencies would be violated.

3.4.2 Model Decomposition

We describe in this section a set of common characteristics of the most important families of model decomposition-based DDES techniques. We consider, as defined in the beginning of the chapter, that the physical system to be simulated is composed of a set of physical processes which only interact at discrete times by means of messages. The message has two fields: the event to occur and the timestamp or time when the event should occur. The LPs do not share any kind of information among them, synchronisation and information interchange is done by message passing. Events are encapsulated into messages sent to other LPs. Each LP has its own *Local Virtual Time* (LVT) which indicates up to what point in simulated time the evolution of the corresponding PP has been simulated. The timestamp of a message scheduled by a LP must be greater (or equal) than the LVT of the LP: this is essential to maintain the causal relationship in the system. Each LP has one or several *input queues*, where incoming messages with events awaiting to be executed are stored in timestamped order. The LP selects, as the candidate to be executed, the message with minimum timestamp among those waiting in the input queues. As it happens in the sequential simulator, the effect of executing an event includes the advance of the LVT to reach the timestamp of the message. Additionally, the state of the PP (and thus, of the LP) might be changed and new messages can be sent to other LPs. It is important to remark that there is no global information shared by the set of LPs. In particular, there is not a global clock but a collection of local clocks, which might not have the same value at a given instant of real time; similarly, there is not a central event list, but a collection of input queues (and a local event list) which play the same role. In order to have a correct simulation, it is sufficient (although not necessary) to obey what it is known as the *local causality constraint* [Mis86]:

If each LP consumes messages in non-decreasing timestamp order, then the execution of the simlation is correct.

Here *correct* means that there are no causal errors in the simulation of events. *Conservative* simulators guarantee that the constraint is *always* obeyed, stopping a LP when it does not have enough information from the other LPs to continue safely. *Optimistic* simulators allow an aggressive execution of events, with the effect that situations may arise where the constraint is violated in some LPs, but these situations are detected and then the affected LPs *rollback* to the past, undo the erroneous computation, to reach a point where all the events were consumed in a correct causal order.

3.4.3 Mapping

For parallel execution, once we have the set of LPs which form the simulator, those processes must be mapped onto a set of processors. N LPs in the model have to be distributed among the available processors P. Each process has its own event list which stores the events for the entities that are mapped onto its processor.

A parallel program can be represented by a task graph, whose nodes represent program modules and edges indicate modules needing to communicate. The weights assigned to the nodes and edges denote the computation and communication times, respectively. Mapping a task graph to a parallel architecture requires partitioning the task graph into a number of partitions equal to the (available) number of processors, and assigning each partition to a processor. Algorithms for optimally mapping chain structured computations onto different models of parallel architectures that have a linear array interconnection network are found in [CN93].

In DDES, we can have a task graph to which we can apply a heuristic algorithm to allocate LPs to processors. This task graph sometimes has precedence information about the order of execution of tasks [WM93]. Nandy and Loucks developed an implementation of a parallel partitioning algorithm which is suitable for use in a conservative simulation and showed through an example that both the inter-processor communication traffic and the computation load balance have an impact on the simulation performance [NL93]. Boukerche and Tropper [BT94] addressed the problem of partitioning a conservative simulation on a parallel computer making use of a simulated annealing algorithm with an adaptive search schedule to find good partitions. A criterion for assignment of LPs to processors during an optimistic simulation is proposed by Som and Sargent [SS93]. The criterion aims at reducing the number of rollbacks by assigning to the same processor LPs which may have rollbacks caused by a common LP.

3.4.4 Real World DDES Applications

DDES has been used with different degrees of success in many real world applications. Some of the domains are very specific such as VLSI circuit simulation [CH94], parallel computing [ACLS94, FW94], communication networks [CGU$^+$94, CT96a], ... Other domains such as computing systems, combat scenarios, health care system, and road traffic are reported in [Fuj90]. Some other refer to the simulation of models described using a kind of "specification language", such as queueing networks or Petri nets, while the model itself can be anything from a supermarket to a factory.

3.4.5 Performance Measures

DDES performance measures include:

- *Execution time* of the simulation;
- *Speedup*: is defined to be the time it takes a single processor to perform a simulation divided by the time it takes the multiprocessor system to perform the same simulation [Ert94]. The speedup can be thought of as the effective number of processors used for the simulation. Obviously, the ideal speedup with N processors is N;
- *Efficiency*: is defined to be the speedup divided by the number of processors used, and measures the effective utilisation of the processors.

Other measures may be the number of rollbacks, their distance, the number of messages (positive and negative), the static and dynamic lookahead, and the memory requirements. In all mechanisms (conservative, optimistic and synchronous), the number of processors, their speed, the cost of operating system overhead can be manipulated to achieve the best speedup.

3.4.6 The Time-Division Approach

In DDES, most of the algorithms presented are based on the *space-division* approach, which means that the system to be simulated is viewed as a set of LPs communicating by sending messages. Parallelism can also be exploited using the *time division* approach where the simulation model is partitioned in the time domain [LL91b]. This means that :

1. the simulation time is partitioned into N subintervals, a processor is assigned to each subinterval. Knowledge concerning the initial state for each interval is needed;
2. at the end of the simulation, a comparison between the final state of the n^{th} subinterval and the initial state of the $(n+1)^{th}$ subinterval is made to see if they match.

A survey of three time-division algorithms by Chandy & Sherman, Greenberg, Mitrani & Lubachevsky, and Heidelberger & Stone is found in [LL91b]. The time-division approach is out of the scope of this thesis and will not be considered.

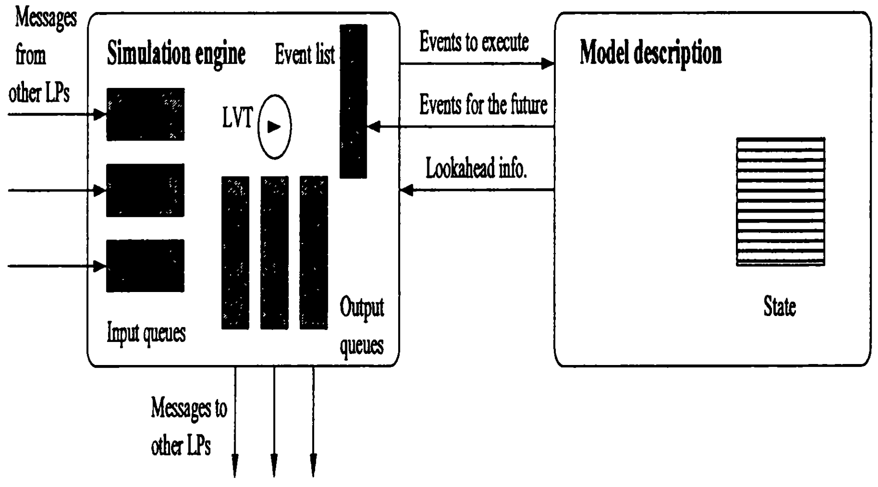


Figure 3.2: A LP in a CMB Simulator.

3.5 The Conservative Mechanisms

In works by Bryant [Bry77] and, independently by Chandy and Misra [CM79], a method called *conservative* is proposed to realise DDES. The name conservative comes from the way the local causality constraint is enforced: an LP must wait, before consuming an event, until it is absolutely sure that no new message will arrive with smaller timestamp. To behave this way, some restrictions are imposed to the LPs:

- each LP maintains one input queue for each possible source of messages. The interconnection topology of the LPs must be static, and known since the beginning of the simulation;
- each LP must send messages through each of its output channels in a non-decreasing timestamp order.

In the conservative mechanisms, each LP is an execution model which contains a section of code, a portion of the system modelled state, a local clock (which denotes how far the process has progressed) and input/output links (Figure 3.2). Input links are characterised by queues of timestamped messages received from other LPs, sorted on time of occurrence. The LP will not process an event before it is sure it will not receive an event with a smaller timestamp. Each input queue has a clock denoting the timestamp of the first message in the queue if it is not empty, or denoting the timestamp of the last message extracted if it is empty. The LP always selects the input queue with the smallest clock value. If the queue is not empty, the Logical Process extracts the first message from the queue, processes it and advances the local clock time. If the queue is empty, the process blocks itself and conversely, waits for other interactions to resume its execution . The simulation engine deals with the messages (events) scheduled for/by the LP. The other part includes the description of the PP simulated by the LP. An analytical study of the performance of a conservative parallel discrete event simulation protocol is found in [Nic93].

A C-like language is used to express the algorithm which is as follows:

```
$C_j = 0$;
for (each i) $cc_{ij} = 0$;
while (not end_of_simulation) {
  while (input queues are empty) await message arrival;
  $m_j$ = message with minimum timestamp;
  $H_j = min_i\{cc_{ij}\}$
  while ($m_j$.timestamp > $H_j$) {
    await message arrival;
    $m_j$ = message with minimum timestamp;
    $H_j = min_i\{cc_{ij}\}$;
  }
  remove($m_j$)
  $C_j = m_j$.timestamp;
  execute($m_j$.event);
}
```

3.5.1 The Deadlock Problem

A LP cannot advance its simulation clock before it has received a message in each input queue. Since a process must block when its input queue with the smallest clock value is empty, a deadlock situation may occur. In Figure 3.3, all three processes are blocked even though there are event messages in other queues that are waiting to be processed.

It is worth mentioning that deadlock in a Petri net is a transition (or set of transitions) which cannot fire whereas in DDES it is the simulation program that deadlocks. In a Petri net a transition is *live* if it is not deadlocked. This does not mean that the transition is enabled but rather that it can be enabled.

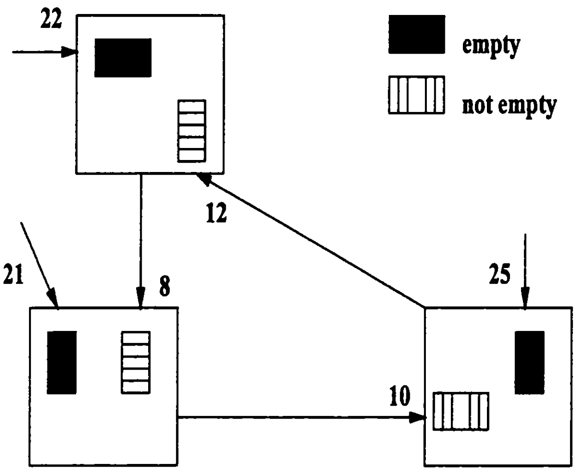


Figure 3.3: Deadlock Situation. Each process is waiting on the incoming link containing the smallest link clock value because the corresponding queue is empty.

Two deadlock resolutions have been proposed in DDES: *deadlock avoidance* [CM79] and *deadlock detection and recovery* [CM81, Mis86]. It has been shown that the cost of deadlock detection and recovery is much higher than deadlock avoidance [Fuj90].

Deadlock Detection and Recovery

One solution to the deadlock problem is to allow the simulation to deadlock, detect it and recover. Thus, the simulation consists of a sequence of phases performing useful computation in parallel separated by phase interfaces, where computation takes place to break the deadlock and allows various LPs to proceed. Two drawbacks of this approach are apparent: the simulation is making no progress during the phase interfaces, and nothing is done to reduce the amount of blocking.

Deadlock Avoidance

An alternative to deadlock detection and recovery is deadlock avoidance which uses *null messages.* A null message does not represent any event in the simulated system. Instead, a null message (*t,null*) sent from process p_1 to process p_2 is a control message which tells p_2 that there will be no more messages from process p_1 with timestamps less that *t*. The message (*t,null*) is determined by adding the minimum clock value of all input queues and the minimum time increment of any message passing through this process. Whenever a process finishes processing an event, it sends a null message on each of its output ports indicating its bound. Null messages are used only for synchronisation purposes, and do not correspond to any activity in the physical system. Another technique that yields substantial improvements in conservative parallel simulation include the use of *appointments* [Nic88]. An appointment is a promise not to send a message before a certain time; thus, it is equivalent to a null message. The difference is that the scheduling of appointments is demand-driven: when a LP is unable to receive a message because the timestamp of the message exceeds the appointment time of one or more of the LPs, the LP requests new appointments from those sources.

3.5.2 Lookahead

Lookahead is the process' ability to predict what will happen, or more importantly, what will not happen in the simulated time as regards to its behaviour and when next it may affect other processes. If a process at simulated time *Clock* can predict with complete certainty all events it will generate up to simulated time *Clock* + L, the process is said to have lookahead L [LL90]. The lookahead information is carried by null messages which are used to break deadlock as well to improve the progress of a conservative simulation.

Experimental studies have indicated that the larger the lookahead values, the better the performance of the conservative simulation. Several techniques for lookahead exploration are proposed in [Nic88, WL89, LL90]. The effectiveness of null messages depends greatly on the amount of lookahead available and is in general application dependent. The number of null messages may become quite large during a simulation. Feedforward and feedback networks investigation for reducing their number is found in [Vri90].

3.6 The Optimistic Mechanisms

Optimistic mechanisms detect and recover from causality errors, they do not strictly avoid them. Each process has a single input queue, all arriving messages are stored in the input queue in order of increasing (virtual) receive time. When a LP determines that an error has occured, a procedure to recover is invoked (a rollback) [Jef85]. A causality error is detected whenever an event message is received that contains a timestamp smaller than that of the LP's local time clock. This event is called *a straggler.* Recovery is accomplished by undoing the effect of all events that have been processed prematurely by the process receiving the straggler. Each process must maintain a state queue containing copies of its previous state. Whenever a (positive) message is sent to another LP, its (virtual) send time is copied from the sender's virtual clock. Whenever a process rolls back to time *t*, antimessages are immediately sent for any previously sent positive messages with a timestamp larger than *t* to undo their effect. This is called *aggressive cancellation.* In *lazy cancellation*, when a process resumes executing from its new logical virtual time (LVT), only messages that are different from previously sent messages are transmitted. A kernel of Time Warp, known as JPLTW (Jet Propulsion Laboratory Time Warp) has been developed by Jefferson's team [JBW+87]. The performance of rollback is investigated by Lin and Lazowska [LL91a], and Lubachevsky et al. [LWS91]. Recently, Das and Fujimoto proposed an adaptive protocol which reduces unnecessary optimism by economising memory usage and without undergoing any significant protocol related overheads [DF97].

3.6.1 Logical Processes in Time Warp

In Time Warp, an event is represented by a message, and contains the name of the sender process; the virtual send time; the name of the receiver process; the virtual receive time; a sign of the message (positive or negative). A process is defined by its name; its LVT; a state queue containing copies of the process's recent states, ordered by LVT; an input queue containing all recently arrived messages ordered by receive time; an output queue containing all the negative copies of the messages recently sent, ordered by send time (antimessages for unsending positive ones) (Figure 3.4). The *Global Virtual Time* (GVT) is the smallest timestamp among all unprocessed event messages (both positive and negative). The sequence of actions that each LP executes is as follows:

- If no unprocessed message is awaiting in the input queue, wait for new arrivals and then go to the next step;
- make a copy of the current state and save it in the state queue;
- consume the message pointed by *next_event*, i.e., advance LVT, change the status according to the class of event, and send new messages to other LPs;
- add an antimessage to the output queue per each message sent in the previous step;
- advance the *next_event* pointer. Go to step 1.

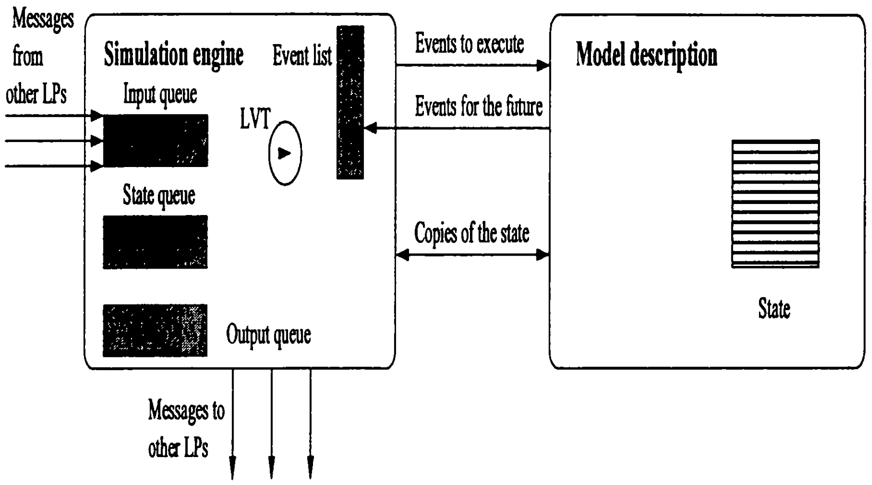


Figure 3.4: A LP in a TW Simulator.

This algorithm can be interrupted each time a message arrives. The received message can be positive or negative, and can belong to the past (if its timestamp is smaller than the local virtual time) or to the future (if its timestamp is larger than LVT). Depending upon the circumstances, one of these four actions must be taken (Figure 3.5):

Positive message for the future. This is the common case in any event-driven simulator. The message simply carries an event scheduled for the LP's future. It is stored in the input queue, in the right position according to its timestamp.
Antimessage for the future. This is a kind of cancellation. The positive message must be in the input queue (if the communication system delivers messages in order). After locating the positive message, both messages (positive and negative) are annihilated.
Positive message for the past. This is a straggler. A rollback is needed because the local causality constraint has not been obeyed. All the effects of simulating messages with timestamp greater than the straggler must be undone, to be re-executed after consuming the straggler. The straggler is inserted in the input queue. The state is restored to the copy saved just before consuming the message that now follows the straggler in the input queue. All the copies of the state following the restored one are destroyed. All the antimessages generated during the erroneous computation are sent. The *next_event* pointer is set to point the straggler. After all these steps, the simulation can resume.

```
(* $LP_i$: arrival of a message with timestamp TT from $LP_j$) *)
if TT > $LVT_i$
then if TypeMessage == '+'
        then Insert(Message) into Buffer-In
        else (* TypeMessage == '-', AntiMessage *)
           Cancel Message in InputQueue
else (* arrival of a straggler message, rollback *)
        (* Restoration phase *)
        insert (Message) into Buffer-In
        Fetch in StateQueue for $State_k$ / $LVT_k \leq$ TT
        Restore $State_k$ with $LVT_k$
        Discard states (L) in StateQueue / $LVT_L >$ TT
        (* Cancellation phase *)
        Fetch in Buffer-Out for SendingTime $ST_m$ / $ST_m >$ T
        Send AntiMessages
        (* Coast forward phase *)
```

Figure 3.5: Message Execution in TW.

Antimessage for the past. The corresponding positive message is searched for and located in the input queue, and both messages are annihilated. A rollback must be done, recovering the state associated to the destroyed positive message, destroying other copies of the state and sending the necessary antimessages. The *next_event* pointer is set to point the message just after the annihilated one. Normal computation can resume.

3.6.2 Messages Cancellation Phase

When a process receives an antimessage that corresponds to a positive message that it has already processed, then that process must also be rolled back to undo the effect of processing the soon-to-be annihilated positive message. Whenever a message is sent, its virtual send time is copied from the sender's virtual clock. Each process has a single input queue which all arriving messages are stored in order of increasing virtual time. Some attempts have been made to reduce rollbacks in optimistic distributed simulation. Prakash and Subramanian presents an algorithm that limits the propagation of erroneous computations by keeping track of knowledge like the assumptions made in the generation of a message and the straggler events that have occurred in the simulation [PS91]. The algorithm presented by Som and

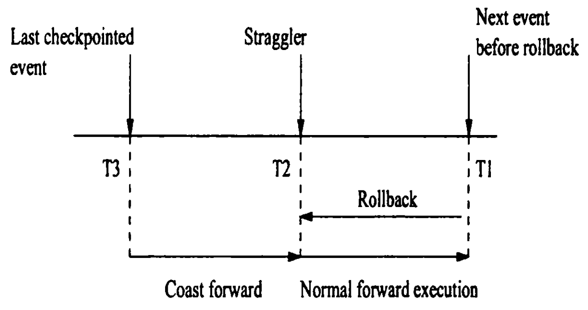


Figure 3.6: Rollback with Infrequent Checkpointing.

Sargent in [SS93] uses the assignment of processes to processors and shows a gain in performance and a reduction of overall completion time. However, it can only be used when the connectivity among the LPs in the simulation model is known.

3.6.3 Global Control

Although most of the operations of the TW algorithm are done in a distributed fashion, with the LPs evolving autonomously, the system cannot work unless a series of global operations are done, satisfying these requirements:

- Guarantee that simulation advances, even taking rollbacks into account. The LVT at a LP is not an accurate estimation at the actual situation of the simulation: an unexpected straggler might arrive, making the LP jump back to the past. A mechanism is needed to establish a fixed point in time, in such a way that no jumps before that time will ever happen.

- Detect the end of the simulation. When a LP reaches the *end_of_simulation* time, it does not mean that it can finish: again, the possibility of a rollback exists, and some work might need to be re-done.

- An important problem to solve is memory management, a complex part in TW. From the descriptions of the data structures managed by the LPs, it can be deduced that those structures grow unboundedly while simulation advances: messages are stored in an input queue, copies of the state must be saved and an antimessage is stored for each sent message. All this information is stored because it might be needed to realise a rollback. However, the amount of memory available to the LP is finite (sometimes it is quite small), and this limits the growth of the data structures. Figure 3.6 shows an example of a rollback where not every event is checkpointed.

To help solving these problems, a TW simulator needs a *global control* mechanism whose purpose is to keep an up-to-date measurement of the *Global Virtual Time* (GVT). This global time indicates up to what point of simulated time the simulation has been done, with a global rather than a local point of view. It is computed as the minimum of all non-executed messages in the simulator. A review of the most well known algorithms for GVT computation are found in [Fer96].

Taking as a restriction that the consumption of a message can *never* affect the past, it can be guaranteed that it is not possible to do a rollback to a time before the GVT. Therefore, all the memory space associated with events timestamped less than the GVT can be safely retrieved, because it will not be needed. This includes past messages in the input buffer, copies of the state stored before the execution of those messages and the antimessages stored as an effect of the execution of those messages. This process of retrieving memory space is known as *fossil collection.* The problem of signaling the end of simulation can also be solved when the GVT reaches the *end_of_simulation* value.

The complexity of the memory management in the LPs, and the need of a global control, makes implementations of TW quite tricky. In comparison, CMB and SYNC algorithms are much simpler. Additionally, TW needs much more memory space to work properly. Although some researchers demonstrated that a TW simulator can work with a very reduced memory space, this does not mean that it will work efficiently. On the other hand, TW does not require the LPs to have a knowledge of the model being simulated to work properly as it was the case with CMB.

3.6.4 Variations of the Basic Time Warp

In [Fuj89], Fujimoto characterises four sources of overhead which appear when TW is used to do parallel simulations, in comparison with an equivalent sequential simulation. Those are:

- Keeping a log of the history of the LPs. That is, keeping the input queue, the state queue and the output queue;
- Message passing. This is common to all DDES techniques based on a distribution of the model among a collection of LPs. The overhead is not only the effort of passing messages, which can be very costly depending on the computer and the message passing software being used, but also the time to prepare them and extract information from them;

- Cancellation, rollbacks. One rollback does not impose a big overhead, but in general rollbacks do not appear alone: one straggler might cause an avalanche of rollbacks, and this in turn means the movement of an important number of antimessages;
- Erroneous computations. All the (real) time that an LP devotes to execute events whose effects are undone afterwards is lost time.

Once the problems have been characterised, solutions might be searched. In the literature, several proposals can be found which try to improve TW by reducing its sources of overhead [LP91, GT93]. Next we discuss four variations of TW.

Lazy Cancellation

In the TW algorithm previously described, during a rollback a set of antimessages is immediately sent, one per positive message generated during the erroneous computation. This policy of sending antimessages is known as aggressive cancellation. An alternative to aggressive cancellation has been proposed, known as lazy cancellation [Fuj90]. This approach tries to minimise the overhead imposed by the treatment of antimessages and, at the same time, to reduce the chain reaction effect of the rollbacks. The optimisation is based on temporarily holding the antimessages to be sent as a consequence of the rollback. Instead of sending them immediately, the LP monitors the positive messages it sends during the normal advance phase which follows the rollback. If it sees that a newly generated message is identical to another generated during the erroneous phase, then the first message can be considered as correct, the antimessage need not be sent and the new positive message can be destroyed. If the described situation is common, i.e. many of the messages generated by a LP are correct even when the LP is violating the local causality constraint, the advantages of lazy cancellation are obvious: less antimessages are sent, and less rollbacks are triggered. However, in some cases lazy cancellation can be worse than aggressive cancellation. It requires additional overhead, and may allow erroneous computations to spread further than they would under aggressive cancellation.

Lazy re-evaluation

Basic TW also performs *aggressive re-evaluation*, which means that past copies of the LP state are immediately removed during the rollback procedure. A lazy re-evaluation approach also exists; in this case, copies of the state are not destroyed so promptly. After the straggler has been executed, the LP compares the copies of the state before and after that execution. If they are identical, then no further action is needed (no antimessages need to be sent, no copies of the state need to be removed), because the re-evaluation will produce exactly the same result as the original evaluation. Thus, simulation may resume at the point where it was before the reception of the straggler, without any re-evaluation of events. This is true unless new stragglers are received. The advantages of this technique are evident, provided that stragglers that do not modify the state are a majority. If this is not the case, the overhead imposed by state comparisons does not compensate the possible advantages. Both lazy cancellation and lazy re-evaluation have an additional negative effect: antimessages or state copies are retained longer than in basic TW. On average, the data structures kept for logging purposes are longer than they would under the aggressive alternatives, so a larger amount of memory is needed to store them.

Conservative Time Windows

In many TW simulations, it has been observed that, when a LP runs its part of the simulation faster than the others (because it runs in a faster processing element or because is less loaded), it produces the apparition of cascades of rollbacks: some straggler can roll back the fast processor, which has generated many messages which are now cancelled. While the slower LPs are busy annihilating message/antimessage pairs, some of them rolling back and generating additional antimessages, the fast LP may progress forward again. To avoid this scenario, the optimism of the LPs must be somehow controlled. A usual way of doing so is the imposition of *time windows.* For example, if the GVT is t, a LP is allowed to advance optimistically until time $t + \delta t$. If all the messages in this windows are consumed, and the remaining ones are timestamped more than $t + \delta\text{t}$, the LP must block and wait until the window is advanced. The size of the window may be fixed, but then is a parameter difficult to tune: if the window is too wide, it is not effective; if it is too narrow, no optimism is allowed, and a synchronous simulation is performed. Instead of using a fixed window size, it is possible to tune it dynamically, i.e., to use an initial value and then make it vary according to the behaviour of the LP. The common approach is to increase the current window size if the LP is mainly doing useful work (i.e., if there is a significant advance without many rollbacks) and to narrow the window if the LP is rolling back too often. This approach is known as *adaptive time windows.*

Periodic and Incremental State Saving

Basic TW saves a copy of the state just before the execution of each message. This usually means that a huge amount of memory is consumed, specially if the size of the state to save is large. Using an optimisation called *periodic state saving*, copies of the state are saved every N message executions, instead of after every message execution. This way memory demands are reduced considerably. However, if this optimisation is included, the rollback procedure is more complex: the LP must recover a copy of the state saved before the one actually needed, and the right state must be reconstructed by means of a re-execution of already executed messages (this is called the *coast-forward* phase of the rollback). During this phase no messages are sent to other LPs. The practical effect is that less memory is needed, but more CPU time is consumed, compared to basic TW. However, a state saving is also a time consuming operation, its reduction can compensate the cost of the coast-forward phases. Experience seems to demonstrate that this optimisation actually improves the performance of the simulator, reducing execution time and memory demands [LPLL93].

An alternative, but similar approach to periodic state saving is *incremental state saving*. With this optimisation the complete state of the LP is again saved every N message executions. In the rest of the cases only incremental changes in state are saved. The coast-forward phase is then simpler: it is enough to find a full, old copy of the state and then update it by applying a sequence of increments to re-construct the required state value. It seems that, in general, this approach is more efficient than the previous one, specially when the cost of executing events is high and the amount of memory needed for an incremental state saving is low.

In either technique, we find again the problem of tuning the value of a parameter, in this case the interval between two full state copies. If this interval is too wide, the time spent saving copies of the state is reduced, but the coast-forward phase is very costly; if it is too narrow, no advantage is obtained over basic TW. As happened with the conservative time window optimisation, this interval can be dynamically tuned to optimise its width, and the same tuning procedure can be used: reduce the interval if the LP is suffering from too many rollbacks, extend it otherwise.

3.7 The Synchronous Mechanisms

In this section we describe a design for a synchronous, distributed event-driven simulator (SYNC), assessing its correctness and its performance potential. The

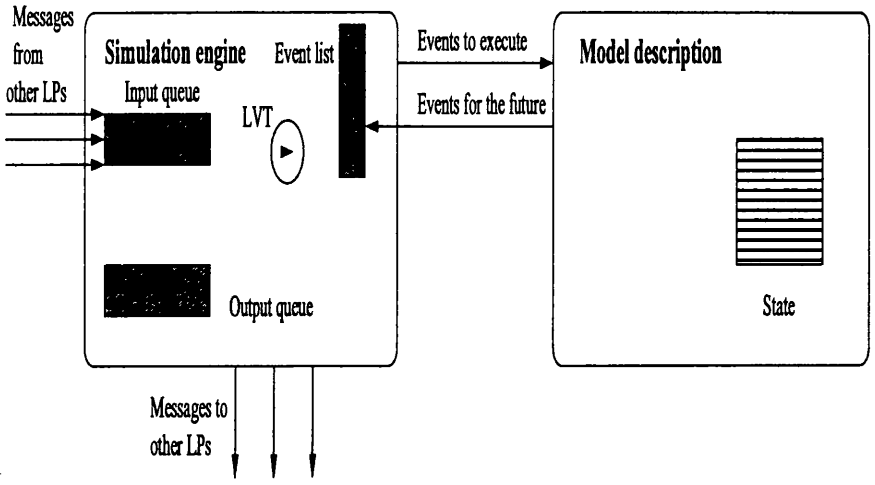


Figure 3.7: A LP in a SYNC Simulator.

description is aligned with the definitions given in the previous section, although different algorithms could be given using different assumptions. We assume that the model to simulate is distributed among a collection of LPs.

A synchronous simulator processes events with the same timestamp in parallel. This protocol is often used in VLSI circuit simulation. Several studies investigated the potential speedup of this approach and showed limited potential. Each LP of the SYNC simulator keeps the same data structures of a single, sequential event-driven simulator (Figure 3.7): clock, state variables, statistics, input queue and event list. A global clock is shared among all LPs and always keep the same value. The rest of the data structures are private. A single input queue of incoming messages is needed, where all received messages are stored in timestamp order.
Each LP performs the following algorithm:

```
clock = 0;
while(clock ≤ end_of_simulation) {
    t = minimum_timestamp();
    clock = global_minimum(t);
    simulate_events(clock);
    synchronise();
}
```

The algorithm works as follows: in the first step each LP obtains the minimum of the timestamp of (first message of its input queue, first message of its event list). Then a global operation is performed to compute the minimum among those timestamps. This value is assigned to the (shared) clock of all LPs. In the third step each LP consumes all the events whose timestamp equals the new value of the clock. The last step is needed to make the LPs start the next iteration at the same time. This synchronisation must be done after all the messages generated in the previous step have been delivered and safely stored in the corresponding input queues.

From this description, it is clear that the simulation performed by a SYNC simulator is correct: events are consumed in timestamp order preventing causality errors to occur. Only those events with the same timestamp are executed concurrently and they are causally independent. The design of the LPs and the barrier synchronisation ensures that the local causality constraint is always obeyed.

Regarding the performance of the SYNC simulator, it is guaranteed that at least one LP will consume one event in each iteration: the one that was used to compute the new clock. In other LPs this step might be void if the event density is very low or the events are not evenly distributed among LPs. In the worst case, the SYNC simulator behaves like the sequential one. But in case of a well balanced scenario, it efficiently exploits the avalaible parallelism with a moderate synchronisation cost. Two positive aspects can be found in this method: the simplicity of the design (which makes the simulator easy to build and to maintain) and the possibility of an efficient implementation on SIMD computers, while other approaches to model-distribution simulation are best suited for MIMD or SPMD systems.

3.8 Related work

In this section we will review the literature to introduce significant domains of application of DDES techniques to modelling tools. These domains refer to the simulation of models described using a kind of "specification language", such as queueing networks, finite state machines and Petri nets, while the model itself can be anything from a computer system to a factory. It should be clear that there are many other studies of parallel simulation algorithms. The ones presented here have been selected because of their similarity to our work.

3.8.1 Queueing Networks

A significant effort has been devoted to efficiently simulate queueing networks, as many real world applications can be modelled using this approach. Reed et al.

analysed the performance of CMB algorithms when simulating several queueing networks in a Sequent Balance 21000 with 20 processors, a shared memory multiprocessor [RMM88]. The tests included both CMB-DA and CM-DDR variants of the conservative algorithm. No effort was made to exploit the lookahead of the studied models, mainly networks of FCFS (First Come, First Served) queues. Poor performances were reported. Many other researchers concentrated on methods to exploit the lookahead of this and other networks disciplines. Maybe the most interesting works are those by Fujimoto [Fuj88], Wagner and Lazowska [WL89] and those by Nicol [Nic88, Nic92, NH93]. These works offer methods to efficiently exploit the lookahead of queueing systems to achieve good speedups when the CMB-DA is used. The techniques to exploit the lookahead are different for each queue discipline. Considered disciplines are FCFS, PS (Processor Sharing) and RR (Round Robin), with or without priorities, and with or without preemption. Other interesting works in the field include [RM91, MR94], where a workbench for queueing systems simulation over a network of transputers using the CMB-DA algorithm is presented. Characteristics about an object oriented conservative parallel simulator for simulating queueing networks designed for running under Windows NT in multiprocessor environment is found in [PSHH97].

3.8.2 Finite State Machines

Attention has been given to communicating finite state machines by parallel simulation researchers. Tropper and Boukerche [TB93] described a synchronisation/deadlock resolution mechanism for a network of communicating finite state machines implemented on an iPSC/2 hypercube. Good performance was reported.

3.8.3 Petri Nets

The simulation protocols we are proposing for ECATNets parallel simulation differ significantly from the protocols developed in [Tau88, BEM90, LKP92, KGS93]. The work by Taubner in [Tau88] was performed in the context of "Petri net driven execution" of distributed programs, where the firing of a transition causes the invocation of a procedure, with the net itself (a Place/Transition net) used to determine the flow of control. Each transition firing results in a procedure execution, which yields an amount of overhead less important for execution of distributed programs than in the simulation context. In addition to that, the nets assumed are untimed, so there is no notion of simulated time. In [LKP92], Lakos describes how the algorithms summarised in [Tau88] have been extended to handle object oriented nets. Butler et al. describe a distributed simulator of high order Petri nets, showing how the inherent parallelism can be used to obtain a fast simulator [BEM90]. The simulator is a component of a suite of tools which allow the construction of specifications of embedded systems. An overview of SYSTEMSPECS, an integrated graphic based software tool for the design and simulation of complex systems is given in [KGS93]. SYSTEMPSPECS allows the graphically animated execution of high order Petri nets and provides a parallel distributed simulation algorithm running on Transputer based parallel systems which proved to be highly suited to simulate complex nets in real time.

The contributions of parallel and distributed discrete event simulation in the area of Petri nets and reported in the literature include [TZ91, AD91, NR91, CF93a, CF93b, NM95, CT96b]. They all deal with Timed and/or Stochastic nets. Thomas and Zahorjan [TZ91] proposed a *conservative* simulation protocol of performance Petri nets. The decomposition of the "initial" net into subnets is node-based, each place and each transition are simulated by a LP in order to maximise the potential parallelism. The technique used, called "selective receive" is based on a communication protocol between a transition and each of its input places. To fire a transition, four messages are exchanged between LPs. The hardware platform is a Sequent Symmetri S81 shared memory with 20 processors. Nicol and Roy [NR91] introduced another *conservative* approach. The "initial" subnet is partitioned so that transitions in conflict are assigned together to the same LP with their input places. The simulator handles three kinds of events, exploits lookahead and is implemented on an Intel iPSC/2 distributed memory multiprocessor. Ammar and Deng proposed in [AD91] an *optimistic* simulator of stochastic Petri nets based on *Time Warp*, allowing completely general decompositions with a redundant representation of places. Five messages are exchanged for LPs synchronisation and to ensure that the marking in a place in one subnet is consistent with its image in the other subnets. The simulator was tested using an Encore Multimax with 18 processors. No speedup figures were given. Chiola and Ferscha [2] exploited Petri net structural analysis for the efficient implementation of DDES techniques using both approaches: *conservative* and *optimistic* [CF93b]. Tests were done using a Sequent Balance, an Intel iPSC/860 and a T805-based multicomputer. The authors state that efficient distributed simulations of timed Petri nets can be done, but real speedups can only be obtained after identifying the model's intrinsic parallelism and causality, and using this information to optimise LPs. Communication overhead seems to be the main obstacle to achieve

[2]The authors have published some articles on the same topic. See for example [FC95].

good performance, as some methods to reduce the number of interchanged messages are proposed. A typical advice in this direction is to make a LP have a load big enough to keep the computation/communication ratio properly balanced. Although a set of rules for partitioning networks based on Petri net topological properties are proposed in [CF93a], no large scale models were considered, and performance results were limited to very small number of processors. Nicol and Mao [NM95] describe a new heuristic technique for automated mapping, both static and dynamic, of the timed Petri net to the parallel architecture. The simulations were conducted on the YAWNS (Yet Another Windowing Network Simulator) parallel simulation testbed [Nic93] implemented on the Intel family multiprocessors (iPSC/860 and Touchstone Delta). Cui and Turner [CT96b] propose a new partitioning technique assuming that transitions have been assigned priorities in the model. The partitioning is one in which each transition, together with its input places, is assigned to a separate LP. A decision place is assigned to the LP containing the transition with the highest priority among the output transitions. An example of the approach using the dining philosophers example shows that it can give a better speedup than that of some other known approaches (eg. the one in [CF93a]). The conservative simulator was tested using a Transputer network with 16 processors. We concentrate in [Dje98] on the development of distributed simulation mechanisms based on the two classical approaches (conservative and optimistic) for queueing networks and timed Petri nets. The overlap in these simulation models in the domain of distributed simulation is addressed.

An alternative to discrete event simulation methods called *recurrence equations* approach is reported in [BC93]. Equations are used to express the evolution of the stochastic Petri net when certain events occur. The algorithm described allows the generation of a simulation program for a SIMD machine.

To the best of our knowledge, little attention has been given to high-level nets distributed discrete event simulation. It is worth mentioning that concerning simulation techniques for high-level nets with arc inscriptions, the enabling test and the firing operations are substantially more complex. The work on THOR (Timed Hierarchical Object-Related) Nets by Schöf et al. was reported in the literature [SSW95]. THOR nets are a kind of high-level Petri nets well suited to real-time systems simulation. They allow complex objects for token values and provide different kinds of timing aspects as well as an appropriate structuring mechanism for nets. The optimistic distributed simulator developed runs on a workstation cluster as well as on Transputer network. A THORN model of the Idle RQ communication protocol with implicit retransmission (also known as send-and-wait or stop-and-wait protocol) is presented as a case study. No speedup figures were reported.

3.9 Conclusion

In this chapter we have introduced a series of basic ideas about simulation of discrete event systems, including two sequential algorithms to realise this kind of simulation: a time-driven and an event-driven one. It has been shown that it is not trivial to implement a parallel simulation by simply modifying a sequential one, so new approaches to the problem have been developed, based on the model decomposition concept. The simulation of a physical system is distributed among a set of cooperating logical processes, which execute the events that affect its part of the system. The collection of LPs must be synchronised somehow, in order to prevent the violation of the cause-effect relationships among events.

Two asynchronous approaches have been presented: conservative and optimistic. The former totally avoids the violation of causal restrictions. The latter allows errors to happen, but recovers from them by means of a rollback procedure. Both kinds of synchronisation have been studied, and some modifications which can be done to improve their performance have been also presented. The synchronous strategy consists of making all the LPs progress at the same time at each step of the simulation, executing in parallel only those events with the same timestamp.

Many additional surveys about DDES can be found in the literature. Some of those concentrate on a particular technique, and many others try to cover a complete range of alternatives. Two main sources of information about conservative algorithms are, in addition to the seminal work [Bry77, CM79], a survey by Misra [Mis86]. For optimistic methods, the work by Fujimoto [Fuj89] is a complement to the work by Jefferson, the author of Time Warp [Jef85]. In the group of general surveys, recommended readings are [RW89, Fuj90, Lin90, Aya93, Fer96]. The work by Ferscha [Fer96] is an interesting qualitative comparison of conservative and optimistic methods.

It is interesting to note that after more than fifteen years of research in DDES, with successful applications in many fields, big effort is still devoted to study DDES algorithms, analysing its behaviour and proposing improvements. But still much work must be done to simplify the development of models, i.e., the work of researchers that use simulation as a tool, not as a research object. In this direction, further research lines are identified, including the following ones [Fuj93, Lin93]: application specific library packages, new simulation languages [BL94], support for shared memory [ACLS94], and automatic parallelisation of models [NM95].

Chapter 4

Environments for Distributed Computing

In this chapter we introduce a series of concepts related to the architecture of parallel systems, and to the different programming models which can be used to develop applications in those systems.

4.1 Introduction

In chapter 1, we stated our interest in multicomputers from a software point of view: we want to make an efficient use of currently available multicomputers, extending the spectrum of applications (simulation of high-level algebraic Petri nets) that can use these architectures.

In this chapter we will study parallel computers in general, and multicomputers in particular, as platforms for the design and execution of parallel applications. Careful decisions must be made to select the appropriate parallel programming model before starting with the design and implementation of an application. However, in some cases, the available computer and programming tools impose a given model, reducing the spectrum of design choices.

We make an introduction to parallel programming from a software point of view, i.e., how a programmer perceives and uses a parallel computer. After discussing a series of concept as MIMD versus SIMD (§4.2), message passing versus shared memory (§4.3), parallel programming languages and tools (§4.4) and two parallel programming environments (PVM and MPI in §4.5), we give a brief introduction to some hardware issues involved in parallel computer design, again focusing on multicomputers (§4.6). The hardware and software configurations of the network of workstations used in this work are presented in §4.7. The chapter finishes with a series of conclusions in §4.8.

4.2 MIMD versus SIMD Computers

Although it might be considered more a hardware than a software issue, the organisation of a parallel computer often has a definite impact on the way applications are programmed [Dun90, Bräunl93]. From a software point of view, a *MIMD* (Multiple Instruction Multiple Data) system allows a set of processes to execute *separate* streams of instructions, each one on its own data. The memory space might be shared among all processes, or might be separate for each process.

In contrast, a *SIMD* (Single Instruction Multiple Data) system allows a collection of processes to execute the *same* instruction stream, each process working on a different piece of data. This second model of parallelism is appropriate for specialised applications characterised by a high degree of regularity, while MIMD might work for both regular and irregular applications.

Somewhere in between MIMD and SIMD, applications might follow *SPMD* (Single Program Multiple Data) paradigm, which means that all the processes run exactly the same program, although not necessarily the same instruction at the same time, on separate data. SPMD is, in fact, a restricted class of MIMD.

In this research we only consider MIMD (or SPMD) applications. This restriction comes from the higher flexibility of this paradigm, and from the programming tools we have available. We consider a parallel application as a set of concurrent communication processes. A pair of those processes might run in parallel, if assigned to different processors of a physical computer, or might time-share one processing element. Each process runs a sequential flow of instructions and is able to communicate with other processes.

4.3 Message Passing versus Shared Memory

Communication and synchronisation are two operations needed in any concurrent programming environment, parallel or not. Two concurrent processes, even being totally unrelated, might need to compete for a shared resource, and they must synchronise before accessing that resource in order to guarantee that one waits while the other uses the resource without interferences. If the processes are cooperating to perform a common task, they might need to interchange information (communicate) in addition to synchronise. There are two basic paradigms for communication and synchronisation among concurrent processes: *shared memory* and *message passing*. We consider them separately.

If two or more processes share a common memory space, one easy way to communicate is by means of a shared variable: one process writes the variable while others can read it. Communication is achieved in a fast and efficient way. However, problems might arise when more than one process try to update a variable without any kinds of synchronisation. The variable used for communication has to be considered as a shared resource, and accesses to it must be somehow restricted to avoid inconsistent updates. Processes must synchronise to access that resource. Many synchronisation mechanisms for shared memory environments might be found in the literature; two common ones are *test & set* locks and *semaphores*.

An alternate paradigm is message passing. In this case each process might have a separate memory space. Explicit communication functions are provided to copy one set of data (a message) from a sender process to a receiver process. Both the sender and the receiver must collaborate to actually perform the data movement: the sender performs a *send* (also called *write*) operation and the receiver performs a *receive* (also called a *read*) operation. Send and receive operations may also provide synchronisation capabilities, depending on its actual semantics.

In some cases, it is possible to mix both paradigms in the same application. A common approach is to allow shared memory communication between processes running in the same processor (or, in general, multicomputer node) while messages are required if processes are in different nodes.

MIMD computers with shared memory are known as *tightly coupled* whereas MIMD computers without shared memory are known as *loosely coupled.*

4.4 Parallel Programming Languages and Tools

In order to implement a parallel application, a programmer needs a language able to express parallelism. Focusing on the design of applications where parallelism is explicit, we can identify at least three alternatives to do so: (1) parallel programming languages; (2) conventional programming languages enhanced with extensions to express parallelism; (3) conventional programming languages with libraries of functions to deal with parallel operations.

In the first group, we can find OCCAM [Inm89], developed by Inmos as the preferred programming language for the transputer family of processors. A collection of processes run in parallel (or concurrently, if several of those are mapped onto the same processor) and communicate by interchanging messages through *channels* using a blocking, synchronous communication model. Languages such as Ada provides support for concurrency. Ada was the first programming language to incorporate structured concurrent programming which is achieved with task notion. An Ada program is a static object whereas a process is the dynamic activity of obeing a program. In Ada terminology, a process is known as a task. A task unit is an Ada program unit which executes concurrently with the rest of the program. Therefore a concurrent Ada program consists of one process representing the execution of the main program, and one or more tasks representing the execution of task units which communicate in a RPC-like fashion [1].

In the second group, we can find tools such as CC++ and Fortran M [Fos95]. CC++ is an extensions to C++ for compositional parallel programming. It is a powerful tool which allows the programmer to use many paradigms of concurrency and communication. Six new keywords have been added to the language to allow to express concurrency, communicate via shared memory, synchronise access to shared data, copy data from one process to another, ... Fortran M is a parallel extension to Fortran with concurrent processes and communication channels.

In the third group, we find libraries of functions which allow conventional languages like C or Fortran to work in a parallel environment, but without modifying the language itself. The alternatives that can be found are either commercial (tailored for specific environment) or in the public domain with implementations for many host computers. The advantage of this approach is the use of a familiar programming language along with an available compiler. In this group we can find the set of libraries which form part of the Inmos ANSI C Toolset for transputer-based environments [Inm90], and several publicly available implementations of PVM (Parallel Virtual Machine) [GBD$^+$94] and MPI (Message Passing Interface) [Mes95]. MPI is able to work in many environments (multicomputers such as IBM SP1 and SP2, Paragon, IPSC860, Meiko CS-2, Sun multiprocessors; network of workstations from Sun, HP, DEC, IBM; networks of personal computers with Linux).

Several parallel simulation languages have also appeared in the last decade. Maisie is a C-based language for distributed simulation [BL94] that was designed to cleanly separate the simulation model from the underlying algorithm (sequential or parallel) that may be used to execute the model. A program written in Maisie is independent of any synchronisation algorithm. Therefore, when it is compiled, the analyst can indicate the specific simulation algorithm that is to be used to synchro-

[1]RPC stands for Remote Procedure Call.

nise execution of the model: sequential, parallel conservative, or parallel optimistic. Maisie has been implemented on a variety of sequential workstations and laptop machines, on networks of workstations, on platforms like the distributed memory IBM SP2 and the shared memory Sparc station 1000.

Another approach has been followed by other researchers that decided to implement the parallel simulation system as a run-time library written in C++: examples include SPEEDES (Synchronous Parallel Environment for Emulation and Discrete-Event Simulation) [Ste92].

4.5 Parallel Programming Environments

The research presented in this thesis has been done using a message passing paradigm. There are several reasons to justify these choices:

1. Message passing is a paradigm widely used in certain classes of parallel machines, specially those with distributed memory. Although there exist many variations some of those discussed in this chapter, the basic concept of processes communicating through messages is well understood. Additionally, a message passing system might be efficiently and portably implemented in most parallel environments [Mes95].

2. The parallel and distributed simulation algorithms used in this research, based on model decomposition, are described by means of message interchange. The implementation is more direct this way.

3. In the absence of a parallel computer, our parallel programs have been implemented in a network of workstations. A NOW can be considered as a special case of multicomputer, where each node is a complete workstation and the interconnection network is typically a LAN (Local Area Network).

4. The NOW that has been available to perform this research provides not only message passing for communication among processors, but distributed shared memory as well. Of course if it was possible to select between shared memory and message passing in a NOW, message passing would be the choice for portability reasons: porting an application from a NOW to a (parallel) machine is easier if both use the same communication paradigm.

To choose among the parallel programming languages and tools, we have used libraries and functions. This decision has been firstly forced by the available tools;

System	Clock	Protocol	Bandwidth	Latency
Cray T3D	151 Mhz	SHMEM_PUT	120 MB/s	6 μs
Cray T3D	151 Mhz	MPI	50 MB/s	40 μs
IBM SP2	66.6 Mhz	MPI/MPL	33 MB/s	143 μs
IBM SP2	66.6 Mhz	MPICH	35 MB/s	114 μs
Hitachi SR2201	150 Mhz	MPI	200 MB/s	45 μs
NEC SX-4		MPICH	1.9 GB/s	72 μs
NEC SX-4		MPISX	6.1 GB/s	35 μs
Cray J90		MPI/MPT	318 MB/s	95 μs
Unix on Ethernet			slow	large

Table 4.1: Some High-Performance Parallel Computers Parameters.

secondly, using the same programming language (C in this case) eases portability among platforms. Table 4.1 summarises the parameters of some high-performance parallel computers. The characteristics of the programming environment used in this work are summarised in Table 4.2.

4.5.1 Parallel Virtual Machine

PVM (Parallel Virtual Machine) [GBD$^+$94] is a software package that permits a heterogeneous collection of Unix computers hooked together by a network to be used as a single large parallel computer. Thus large computational problems can be solved more cost effectively by using the aggregate power and memory of many computers. The software is very portable. The source, which is available free through netlib, has been compiled on many computers from laptops to CRAYs. PVM enables users to exploit their existing computer hardware to solve much larger problems at minimal additional cost. Hundreds of sites around the world are using PVM to solve important scientific, industrial, and medical problems in addition to PVM's use as an educational tool to teach parallel programming.

4.5.2 Message Passing Interface

In the beginning of the nineties, whilst PVM had its adherents, MPI was for many a revelation. It contains a huge range of subroutines including the widely used blocking and non-blocking point-to-point communications, but also global reduction operations, groups and communicators within contexts, timing and profiling

Aspect	Distributed simulator
Programming tool	ANSI C with MPI library
Model of parallelism	MIMD (SPMD preferred)
Communication paradigm	Message passing
Communication models	Blocking, nonblocking
Communication models	Basic, buffered, synchronous
Partner	Explicit (addresses)

Table 4.2: Characteristics of the Programming Environments Used in this Work.

routines. It gives power and the ability for manufacturers to provide fast hardware for the higher level operations and also facilitates writing numerical libraries (necessary for applications programming).

In this research we used C plus MPI [Mes95]. Such environment provides the opportunity of designing MIMD as well as SPMD programs with message passing communication. MPI is efficiently and portably implemented in most parallel environments. As the distributed simulation algorithms are described by means of message interchange, the implementation is more direct this way.

MPI communication primitives may be *blocking* or *nonblocking*, and provide the following communication modes: *basic, buffered* and *synchronous.* The collection of processes collaborating in the distributed application can be depicted as a graph, where nodes represent processes and arcs represent communication *channels.* Each process in the distributed application is identified from 0 to N-1, where N is the number of processes. Explicit communication functions are provided to copy one message from a sender process (performing a *send*) to a receiver process (performing a *receive*).

Geist et al. compare PVM and MPI features, pointing out the situations where one may be favored over the other [GKP96]. For example, MPI has a richer set of communication functions and has the advantage of expected higher communication performance if an application is going to be developed and executed on a single Massively Parallel Processor (MPP). PVM has the advantage when the application is going to run over a networked collection of heterogeneous hosts. Also, the larger the cluster of hosts, the more important PVM's fault tolerant features become.

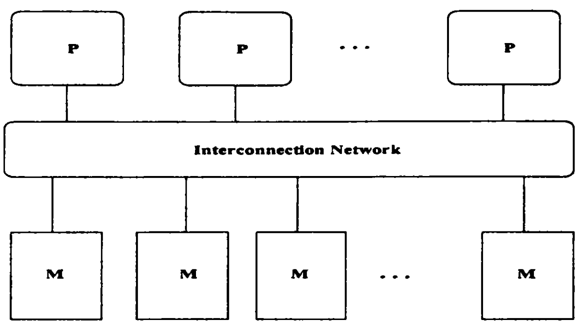


Figure 4.1: Model of a Multiprocessor.

4.6 Parallel Computer Design

In general terms, a parallel computer consists of a set of processing elements interconnected by means of a communication network. Two groups of parallel systems might be characterised: *multiprocessors* and *multicomputers.*

The term multiprocessor is used to refer to a parallel system with shared memory, where synchronisation and information exchange occur via m memory modules which can be accessed by p processors in a coordinated manner by means of an interconnection network (Figure 4.1). The design of the network is a critical issue, because memory access times should be minimised. Buses and multistage interconnection networks are normally used as a common class of network in this design. Another important issue is the cache memory: a local cache is needed at each process to obtain a reasonable performance, and some cache coherency mechanism must be added, because a memory word might be simultaneously in several local caches. This issue, among other things, limits the scalability of multiprocessors. The bandwidth and latency still make algorithms efficient or doomed to failure.

Multicomputers have local memory in each processor and correspond more closely to a group of loosely bound, independent computers interconnected by a network which provides the infrastructure for communication (Figure 4.2). <processor, memory> is referred as a *node.* The communication and synchronisation mechanisms are implemented by means of messages interchanges through the network. The main issues in multicomputer design are the structure of the node as well as the organisation of the interconnection network.

The preferred communication paradigm in multiprocessor environments is shared memory, while in multicomputer environments is message passing. It is possible, however, to have the memory modules physically distributed along the nodes of a

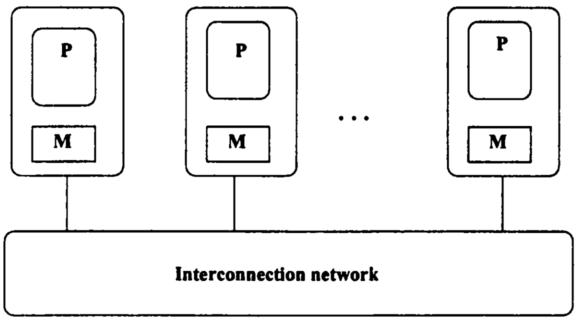


Figure 4.2: Model of a Multicomputer.

multicomputer while the programmer sees a shared memory place; a good deal of hardware/software support is needed to achieve this. In the same context, it is possible to simulate message passing over the shared memory space provided by multiprocessors.

4.6.1 Multicomputer's Node

Each node of multicomputer consists of a CPU plus a certain amount of memory. Some multicomputer manufacturers use custom designs for the CPU, although those used in workstations are in most cases general purpose multiprocessors. In some cases, the nodes of a multicomputer are actually small multiprocessors, with several CPUs and memory modules constituting a computing cluster; the interconnection network communicates clusters, instead of individual CPUs.

In addition to processing tasks, a node must provide some communication management functions. The kind of networks typically used in multicomputers are direct networks like hypercubes and meshes. In those networks, each node must perform certain message functions to allow a message to flow from its origin to its destination, traversing intermediate nodes if necessary (Figure 4.3).

Multicomputers such as the CM-5, the CRAY T3D and the Intel Paragon separate computation and communication tasks, providing hardware support to implement message passing functions, in such a way that these functions are assigned to a collection of hardware *routers*, while the CPUs can concentrate on computation tasks (Figure 4.4).

As mentioned earlier, a NOW can be considered as a special case of multicomputer. CPUs have to devote a certain amount of time to perform communication functions. Message passing has a series of *overheads* which might be reduced with

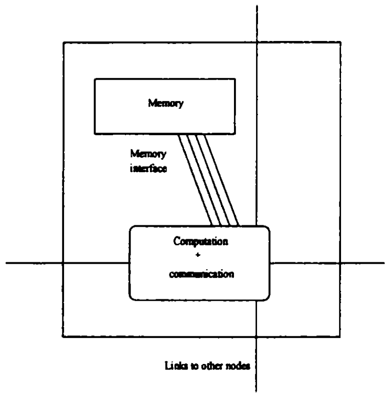


Figure 4.3: Node Where Communication and Computation Functions are Integrated.

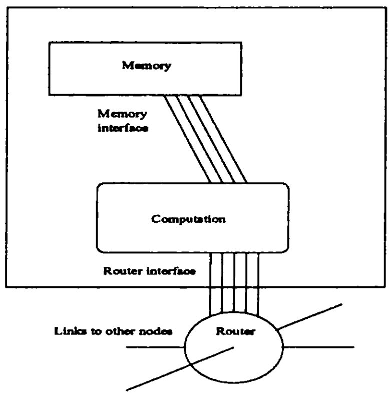


Figure 4.4: Node Where Communication is Separated from Computation.

appropriate hardware support, but which are very difficult to eliminate. Sending a message from one node to another requires a series of operations, summarised in Table 4.3.

In a NOW, message passing functions are not implemented directly over the LAN harware, but pass over several layers of protocols. As an example, the message passing system used in this work requires messages to pass through three high level protocol layers, in addition to the LAN layer (in this case, an Ethernet): the MPI library, TCP and IP. This software overhead can be minimised if messages are long, but this is not a common situation when the objective is to achieve massive, fine-grain parallelism.

	Sources of overhead
At the sender CPU	- Send system call - Argument processing - Allocate buffer - Prepare message - Initiation of send
At the network of routers (software or hardware)	- Transfer message via network interface at origin - Transfer message over the network - Transfer message via network interface, at destination
At the receiver CPU	- Interrupt service - Buffer management - Message dispatch - Copy data to user space - Receive system call

Table 4.3: Overheads Involved in a Pair of Send/Receive Operations.

4.6.2 Multicomputers's Interconnection Network

Message passing support must be provided by the interconnection network in a multicomputer with :

1. low *latency*: messages must cross the network connection from sender to receiver a fast as possible;
2. high *throughput*: the network must be able to manage all the messages generated by the computing elements; it must not be a bottleneck.

Other desirable characteristics are low cost, fault tolerance, expandability (not necessarily in this order). There are many issues to consider in order to design a network with the desirable characteristics. Some of those are:

1. *Topology* or shape of the network. Common topologies are: bus, ring, hypercube, mesh (2D and 3D), and torus (2D and 3D);
2. *Switching technique*: circuit switching or packet switching;
3. *Message flow control*: store-and-forward, wormhole, cut-through;

	NOW
Node	Sun Sparcstation
Implementation of message functions	-
Network topology	Bus (Ethernet)
Switching technique	Packet switching
Message flow control	-
Routing	-
Deadlock management	-

Table 4.4: Hardware Characteristics of the NOW.

4. *Routing strategy*: static, adaptive, with many other alternatives for both cases;

5. *Deadlock management*: necessary for some combinations of topology and routing strategy.

Commercially available machines offer many combinations of these parameters. Table 4.4 summarises the characteristics of the NOW used in this research.

4.7 Characteristics of the Network of Workstations Used in this Work

4.7.1 Hardware Configuration

Over an Ethernet local area network, the (homogeneous) workstations used in this work share the medium which provides a raw 10 Mb/s data rate. The type of workstation used is a Sun Sparc Classic ELC (4/15) with the following characteristics:

- Processor: microSPARC - 50MHz
- 32 bits registers
- 24 Mb (physical) memory
- 96 Mb (virtual) memory.

4.7.2 Software Configuration

The characteristics of the programming environment are:

- Operating system: SunOS version 5.5.1, Solaris 2.1
- C Compiler: gcc version 2.7
- Debugger: dbx 3.2
- CHIMP MPI from the Edinburgh Parallel Computer Center.

4.8 Conclusion

In this chapter we have reviewed a series of concepts related to the view a programmer has of a parallel programming system, and to the different architectural organisations that can be used to actually build such a system. The presentation has been purposefully focused on multicomputer systems, where a set of computing nodes, comprising a CPU and a certain amount of local memory, are connected by means of a message passing network. This decision has been motivated by the computing system available for this research.

The description of the hardware/software issues involved in parallel programming has served to introduce the main characteristics of the network of workstations with MPI and PVM libraries.

Any parallel computer that provides the SPMD or the MIMD models of computing allows the implementation of a parallel simulator with the described characteristics. If the communication model is message passing, as happens with the machines used in this research, the interchange of messages among LPs is implemented in the obvious way. If the system provides communication via shared memory, a library of functions to emulate message passing can be built. The simulation algorithms developed for a message passing environment trivially adapt without performance loss to shared memory by emulating *message exchange* via *shared variables.*

The communication infrastrucure of the parallel computer must be able to support the interconnection topology of the LPs in distributed simulation. In general, it is assumed that the communication is reliable: no message is lost, modified, duplicated, or delivered out of order.

Chapter 5

Conservative Simulation of ECATNets

In this chapter we present the first ECATNet distributed simulator based on a conservative approach and implemented in a distributed memory environment. Two partitioning techniques are proposed in order to spatially decompose the ECATNet into subnets, each subnet to be simulated by a LP. The objective of the study is to select an appropriate conservative distributed algorithm for the analysis of ECATNet models.

5.1 Introduction

ECATNets conservative algorithms do not permit any causality error. The set of LPs (represented as objects) in the simulation process an incoming message only when the underlying synchronisation algorithm can guarantee that they will not subsequently receive a message with a smaller timestamp. These algorithms, by definition, block until a LP can ensure that it will not violate causality by processing the next event.

The chapter is structured as follows. First, a description of the implemented ECATNet simulators (sequential and distributed) is done in §5.2. §5.3 introduces the characteristics of a conservative ECATNet LP. We see in §5.4 how to decompose an ECATNet model into submodels to be simulated by LPs, assessing the impact this decomposition may have on the simulator's performance. Details about the implementation of the LP's communication interface are given in §5.5. A description of the CMB-DA simulation engine is done in §5.6. The Ethernet transmitting station ECATNet model presented in §2.4.5 is chosen to carry out the experiments to evaluate the CMB-DA simulator in 5.7. Finally some conclusions are summarised in §5.8.

5.2 The Simulators

We present the simulators used in this study. Four different ECATNet simulators have been implemented and tested:

- SEQ: sequential event-driven, able to run in any of the parallel systems described in Chapter 4;
- CMB-DA: Chandy-Misra-Bryant with Deadlock Avoidance via null messages;
- TW (Time Warp), with Lazy Cancellation (LZ) as a message cancellation technique, and
- SYNC: synchronous distributed event-driven.

All the distributed simulators work with the same description of the model. SEQ works with a slightly different description of the same model. For our parallel programming environment (NOW) and a distributed simulator, the main performance figures to be considered are the execution time and the speedup. LPs profiles will also be studied.

All the simulators share as much code as possible, to be fair when making comparisons and, obviously, to reduce the development effort. In particular, in all the cases a set of functions to manipulate event lists has been used. We first implemented the event list using a linked linear list for the sake of simplicity, but later had to re-implement it using a splay tree data structure as recommended in [CSR93]. Although the difference between both data structures is less noticeable in the distributed simulators (because events are distributed among all the LPs and, therefore, event lists are shorter), it results in performance improvement for the sequential simulator when the density of events is high.

5.2.1 Input Parameters for the Simulators

In addition to selecting the parameters of the simulated model, a user running the simulators has to facilitate a series of additional parameters. These are enumerated in table 5.1.

Parameter	Meaning
Cycles	Simulated amount of time while the behaviour of the ECATNet model is studied
Seed	Seed for the random number generators
Number of PEs	Number of processing elements used in the simulation
ECATNet subnet	Subnet assigned to each LP of the distributed simulator: - net model (P, T, F) - data model (IC, DT, CT, TC, C) - set of rewrite rules

Table 5.1: Parameters of the Simulators.

The first two parameters are needed for all the simulators, sequential and distributed. The number of Processing Elements (PEs) must be given for any distributed simulator. A mapping of the simulated ECATNet model onto the actual network of PEs in the network of workstations must be done. The distributed simulator always consists of a collection of collaborating LPs, where each LP is, in fact, a Unix process.

5.2.2 Components of the Simulators

For each ECATNet LP, we identify three components of the distributed simulation framework:

- the work partition assigned to it according to the model decomposition;
- its communication interface which is required to preserve its behavioural semantics, and
- its simulation engine which implements the simulation strategy: CMB-DA, TW and SYNC.

There are fixed FCFS communication channels between LPs, timestamped messages are exchanged via these channels for their synchronisation. The division of the LP into three different components allows to decouple the activities of event consumption and message interchange.

5.2.3 Types of Events

In the following, we assume a three phase transition firing. Although an atomic transition firing affects the management of the event list, the proposed solutions can easily be modified to accomodate it.

There are two categories of events. An *internal* event is scheduled and executed at the same LP, and an *external* event is scheduled by one LP and is executed by another LP. The events which may occur when constructing a discrete event simulator of timed ECATNets are:

- **Start_firing** : if transition *t* is enabled to fire at *Tsim*, for every *t*'s input place remove appropriate tokens. DT tokens are destroyed thanks to the execution of event **Destroy_tokens**. The event **End_firing** with timestamp *Tsim* + `FiringTime(t)` is inserted into the event list. The event **Create_tokens** is also inserted into the event list at each *t*'s output place *p* with timestamp *Tsim* + `FiringTime(t)` (CT tokens are created), or scheduled as an external event and sent as a timestamp message to the LP *p* is assigned to.

- **End_firing** : when *t* ends firing at *Tsim*, it checks its condition (TC), its input places (IC) and its output places (CT, M(p) and C(p)). If the enabled conditions are satisfied, *t* is refired and an event **Start_firing** is inserted into the event list at time *Tsim*.

- **Create_tokens** : when tokens arrive at place *p* at *Tsim*, its marking is updated. This deposit may enable any of *p*'s output transitions. If *p*'s output transition is enabled, the event **Start_firing** at *Tsim* is inserted into the event list.

The events **Start_firing** and **End_firing** are always inserted into the LP's event list. However, when the event **Start_firing** is processed, if *t* is the firing transition and *p* its off-LP output place, the event **Create_tokens** needs to be encapsulated into a timestamp message carrying CT(t,p) sent to the LP *p* is assigned to.

5.2.4 The Sequential Simulator

In the following, we present some essential concepts of the design of an optimised sequential simulator for ECATNets.

ECATNet models can be executed using sequential or distributed simulation algorithms. A single processor, event list simulator was developed to allow comparison of distributed simulation programs with sequential event list implementations. In order to obtain a fair comparison, the simulators share most of the code. Both implementations maintain the same overall structure, organisation, programming style, and conventions.

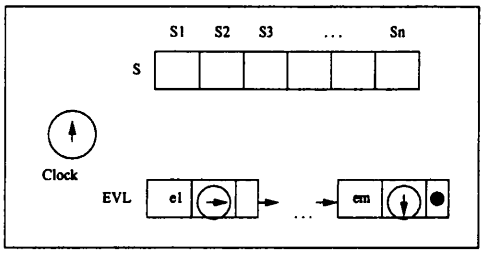


Figure 5.1: Sequential Simulation Engine.

The ECATNet sequential simulator repeatedly processes the occurrence of events **Start_firing**, **End_firing** and **Deposit_tokens** by maintaining: (1) an ordered data structure called the *global event list* (EVL) which stores all events that are generated in the system in their timestamp order *time of occurrence*; (2) a *global clock* indicating the current time; (3) *state variables* $S = (s_1, s_2, ..., s_n)$ defining the current state of the system (Figure 5.1).

The simulation engine drives the simulation by continuously taking the first event out of EVL, simulating the effect of the event by changing the state variables and/or scheduling new events in EVL (possibly removing obsolete events). This is performed until some pre-defined *end_of_simulation* time is reached.

The concept to improve run time efficiency of the simulator relates to the determination of enabled transitions. In a straightforward implementation a net simulator determines all enabled transitions (by invoking function `Enabled()`) and selects one of them to fire. On the resulting marking it repeats the same procedure. A better strategy is to store the knowledge about enabled transitions and to determine only the activation of those transitions that may be enabled by the last fired transitions after invoking function `Rewriting()` which checks the right-hand side of the rewrite rule associated with the transition to fire and the execution of a **Create_Tokens** event. With this procedure the simulator does not need to check all transitions of the net in every step but only those in the pre- and postset of a firing transition.

The event list has been implemented using a splay tree. However, the linked linear list yields performance comparable to the splay tree for simulations with low events density.

5.3 The Conservative Simulator

5.3.1 Logical Processes

The simulation of events is performed in *virtual time* according to their causality. The data structures according to the conservative approach are: (1) a Local Virtual Time (LVT) representing an accumulated value of firing times in a LP; (2) an event list (EVL) ordered by time of occurrence, used when there are internal events posted within the LP itself; (3) input queues (IQ) (one queue per each input channel), which collect recently arrived messages ordered by time; and (4) output queues (OQ) (one queue per output channel) which keep messages to send, ordered by time.

The attributes and functions of the LPs are classified into four categories:

- the *clock* mechanism. The `UpdateLVT()` function updates LVT to advance LP's clock;
- the *event list* mechanism to process the *internal* events in the LP with the following functions: `Enqueue()`: inserts a timestamped event into EVL; `Dequeue()`: deletes the event with the minimum timestamp in EVL; `Cancel()`: deletes the event with a specified timestamp in EVL; `ExecuteEvent()`: executes events in EVL and is also part of the synchronisation mechanism;
- the *synchronisation* mechanism interacts with other LPs to coordinate the execution of the simulation with the following functions: `ReceiveMessage()`: receives messages from other LPs. These messages will be inserted into the input queues for processing; `ExecuteMessage()`: executes incoming messages; `SendMessage()`: sends output messages generated by the execution of events to their destination LPs;
- the ECATNet simulation mechanism based on the transitions enabling conditions, their firing times and the application of rewrite rules.

5.4 Partitioning

A natural decomposition of the "initial" ECATNet model into LPs is a spatial partitioning into different subnets. In the following, we present two different partitioning techniques for ECATNets distributed simulation. The first one is based on a "separation of concern" strategy (SCS), the second one is called "efficient" partitioning

(EP) and is basically an assignment of the transitions sharing input places to the same LP. We recall that the partitioning of the "initial" ECATNet model has a strong impact on the DDES performance.

5.4.1 Separation of Concern Partitioning

The "separation of concern" partitioning is not only suitable for representing complex systems ECATNets models but also for the inter-module parallelism achievement: the Ethernet transmitting station in §2.4.5 is a good example which shows how *modularity* is achieved using this strategy.

Each module obtained from the partitioning corresponds to a subnet to be simulated by a LP. Because partitioning with SCS leads to a (possible) redundant representation of places and the corresponding communication arcs in adjacent LPs, we remove this redundancy and represent the places which are relevant for each subnet only once. If P, T and F are respectively the set of places, the set of transitions and the set of arcs of the ECATNet model, the partition is a set of n subnets such that :

$ECATNet_i = (P_i, T_i, F_i, \lambda_i)$ where $\cup P_i = \mathrm{P}$, $\cup T_i = \mathrm{T}$, $F_i \subset (P_i \times T_i) \cup (T_i \times P_i)$, $\mathrm{i} = 1..n$.

The inscriptions of arcs F_i by the multisets of terms IC, DT, CT and the association of TC to transitions are defined as in the ECATNet model and appear in the graphical representation of the subnets. The transitions remain in the same subnets, the duplicated places have to be assigned to the relevant ones.

However, the "separation of concern" strategy does not necessarily lead to a good parallel simulation partitioning. Its drawback is related to the firing of a transition. A transition t fires when all the enabling conditions are satisfied, by checking first its input places in the same LP. This transition may have one or more input places in different LPs. Since a LP only has information on its local marking, it needs to exchange messages with other LPs to obtain information concerning the marking of t's input places. Another difficult case is when several transitions share a *decision place.* Therefore, it is necessary to implement a distributed conflict resolution algorithm to decide which transition is going to fire among the enabled transitions. In such case, the synchronisation and communication needed to implement the simulation properly are rather more complex. Obviously, this will lead to substantial overhead in the distributed simulation and the amount of messages inherent to this protocol can prevent efficiency.

5.4.2 Efficient Partitioning

An efficient partitioning technique, first used in [NR91] and exploited later in [CF93b], is related to the firing of a transition: the transitions sharing places are assigned to the same LP with their input places, thereby making the enabling conditions *local* in the LP and avoiding the exchange of messages to decide which transition is going to fire among the enabled transitions. Thus, this partitioning technique improves the distributed simulation performance by avoiding the overhead of a distributed conflict resolution algorithm.

5.5 The Communication Interface

A communication interface is required to preserve the behavioural semantics of an ECATNet LP. Such interface has to be implemented by an appropriate protocol among the partitions according to the simulation strategy. We can map the set of arcs $(T_k \times P_l) \cup (P_l \times T_k)$ interconnecting different subnets to the channels of the communication interface. We define I_k = $(CHANNELS, \mathrm{m})$ of $subnet_k$ to be the communication interface with $CHANNELS = \bigcup_{i,j} ch_{i,j}$ where $ch_{i,j}$ = $(\mathrm{LP}_i, \mathrm{LP}_j)$ is a set of directed channels from LP_i to LP_j corresponding to the arcs $(t_k, p_l) \in (T_k \times P_l)$ and to the arcs $(p_l, t_k) \in (P_l \times T_k)$ carrying messages of type m. In the following, we give some basic definitions and explain how the partitioning of the ECATNet affects the LP's behaviour.

Definitions A place $p_i \in P_i$ in LP_i is said to be a member of the set of *LP-output places* (OP_i) of LP_i if there exists a transition $t_j \notin LP_i$ which p_i is an input place. A transition $t_i \in T_i$ in LP_i is said to be a member of the set of *LP-input transitions* (IT_i) of LP_i if there exists a place $p_j \notin LP_i$ for which t_i is an output transition. We define in exactly the same manner *input places* and *output transitions* by interchanging places and transitions.

For each *LP-input transition* t in subnet i, define :
$P_{t_{in}}$ = list of places to indicate which *output* places are related to t and the kind of relation that exists. $P_{t_{out}}$ is defined for each *output transition* in the same manner by interchanging output by input.

For each *LP-output place* p in subnet i, define :
$T_{p_{out}}$ = list of transitions to indicate which *input* transitions are related to p and the

kind of relation that exists. $T_{p_{in}}$ is defined for each *input place* in the same manner by interchanging input by output.

A *communication arc* is an arc connecting a place (transition) in LP_i to a transition (place) in LP_j. If there is a communication arc from an *LP-output transition* t to an *LP-input place* p, there exists a unidirectional channel between them. This is motivated by *LP-output* transitions which have to interact with their *LP-input* places for sending tokens when these transitions fire. If there is a communication arc from an *LP-output place* p to an *LP-input transition* t, there exists also an additional communication arc from t to p. These unidirectional channels are motivated by *decision places* which have to interact with their output transitions before choosing the transition to fire.

The syntax of the rewrite rules has to be modified according to the partitioning technique. If p, t and q are assigned to LP$_i$, LP$_j$ and LP$_k$ respectively, a rule of the form:

$$t: (p,a) \rightarrow (q,b)$$

will have the following syntax:

$$t_{LP_j}: (p_{LP_i},a) \rightarrow (q_{LP_k},b)$$

Thus, a **Create_tokens** external event is sent as a message from LP$_j$ to LP$_k$ when t fires.

Example: Consider the example of section 2.4.5, the modular specification of the Ethernet transmitting station. Partitioning with SCS leads to a set of four subnets, each subnet is simulated by a LP. The set of duplicated places (labelled 1..9) is partitioned among the subnets as follows (Figure 5.1):
Subnet 1: TRANS_REG(1), SUC_TRANS(4)
Subnet 2: INIT_TRANS(3), CHANNEL(6), TO_USER(8)
Subnet 3: INIT_JAM(5)
Subnet 4: RETR_REG(2), RETR_COUNTER(7), RETR_ATTEMPTS (9)

RETR_REG is an LP-input place in $subnet_4$, SUC_TRANS is an LP-output place in $subnet_1$. ACK1 is an LP-input transition in $subnet_2$, ASSEMB_FRAME is an LP-output transition in $subnet_1$. (ASSEMB_FRAME, RETR_REG), (SUC_TRANS, ACK1) are communication arcs, (ACK1, SUC_TRANS) is an additional one. The set of LP_i, i=1..4 exchange messages via communication channels. A **Create_tokens** external event is sent as message from LP_1 to LP_4 when transition ASSEMB_FRAME fires. In LP_2, transition ACK1 has local informations about its input place INIT_TRANS, and has to obtain informations about the marking of places SUC_TRANS and RE-TRANS_REG from LP_1 and LP_4 respectively.

The syntax of the rewrite rule associated with transition ASSEMB_FRAME in LP_1 is now:
ASSEMB_FRAME$_{LP_1}$: (ERROR_SEQ$_{LP_1}$, fcs) ⊗ (FROM_USER$_{LP_1}$, <d, s, data>)
→ (TRANS_REG$_{LP_1}$, d.s.data.fcs) ⊗ (RETR_REG$_{LP_4}$, d.s.data.fcs)

When ASSEMB_FRAME fires, this will lead to: (1) the insertion of events **End_firing** (ASSEMB_FRAME) and **Create_tokens** (d.s.data.fcs in TRANS_REG) in EVL; and (2) the sending as an external event of a message **Create_tokens** from LP_1 to LP_4 carrying tokens d.c.data.fcs for place RETR_REG.

However, when the initial ECATNet model is partitioned using EP, the partition is a set of three subnets only. Each subnet has a set of transitions and places in the output and input borders respectively. There is one type of communication arcs: those connecting transitions to their off-LP output places. Places and transitions are partitioned among the LPs as follows (Figure 5.3):
Subnet$_1$: 2 places, 2 transitions;
Subnet$_2$: 11 places, 8 transitions;
Subnet$_3$: 2 places, 1 transition.

The syntax of the rewrite rule associated with transition ASSEMB_FRAME in LP_1 is now:
ASSEMB_FRAME$_{LP_1}$: (ERROR_SEQ$_{LP_1}$, fcs) ⊗ (FROM_USER$_{LP_1}$, <d, s, data>)
→ (TRANS_REG$_{LP_2}$, d.s.data.fcs) ⊗ (RETR_REG$_{LP_2}$, d.s.data.fcs)

When ASSEMB_FRAME fires, this will lead to the sending as external events of two messages **Create_tokens** from LP_1 to LP_2 carrying tokens d.c.data.fcs for places TRANS_REG and RETR_REG.

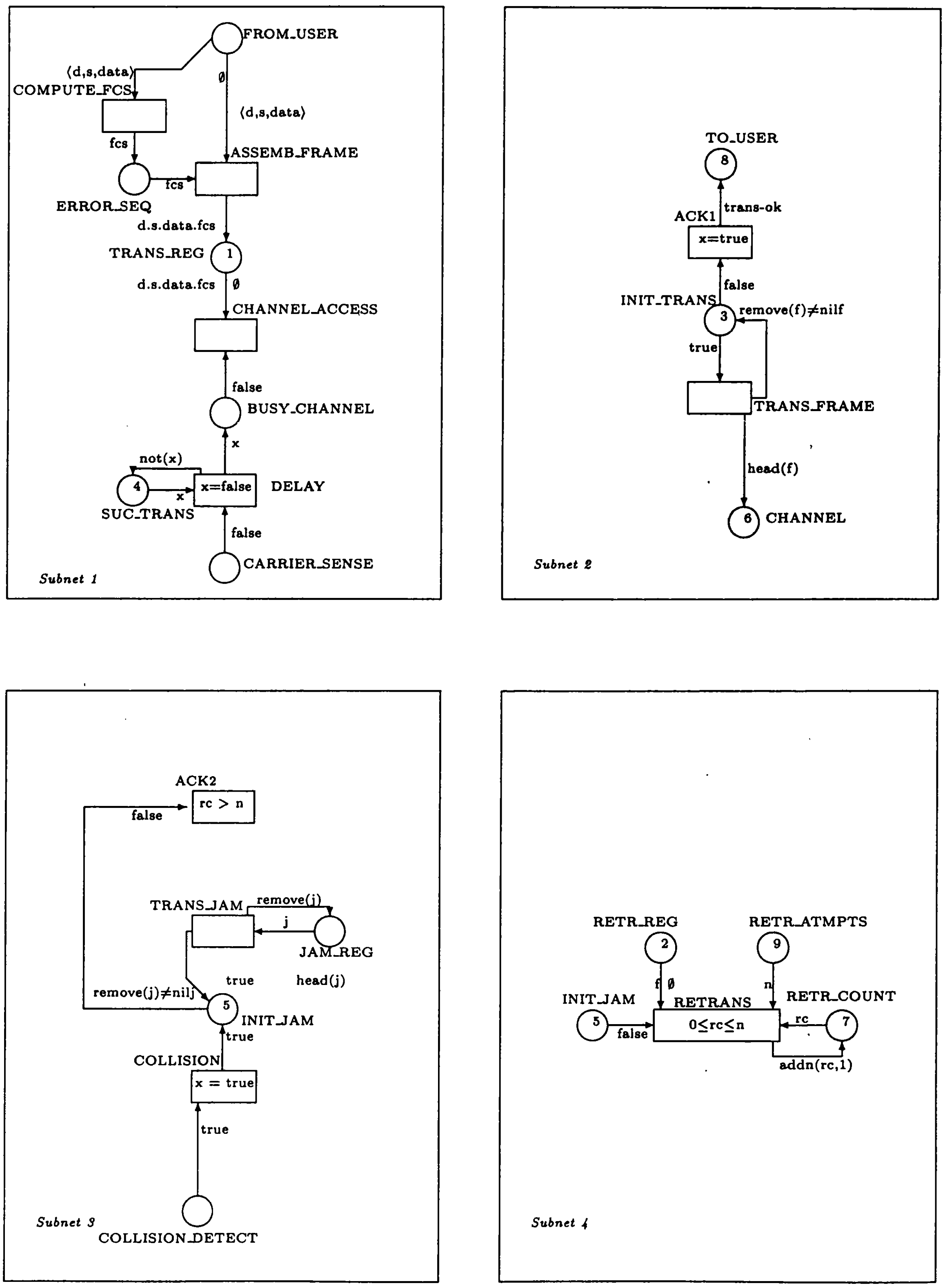


Fig. 5.2 Separation of Concern ECATNet Partition

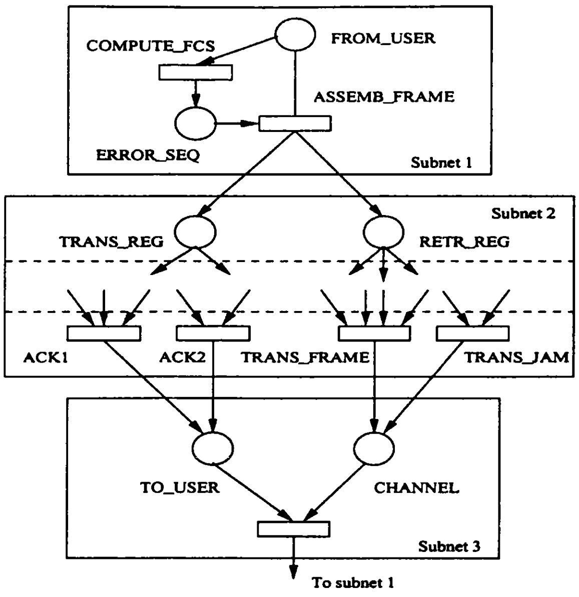


Figure 5.3: Efficient ECATNet Partition.

5.6 Simulation Engine

The conservative approach allows only the processing of safe events, firing of transitions up to LVT for which the LP has been guaranteed not to receive messages with smaller timestamps. In the following details about the CMB-DA simulation engine are given.

5.6.1 Types of Messages

The (general) format of the messages exchanged between LPs is

Type_message (source, destination, timestamp, type_token)

where *source* and *destination* are either a place (in LP_i) or a transition (in LP_j) depending on the communication arc, *timestamp* represents an accumulated firing time of transitions (it may have different meanings in the different types of messages), and *type_token* is an algebraic term token moved among subnets. Messages are also labelled with a port number that clearly states which channel they must be sent through.

The causality of events is preserved over all LPs by sending timestamped token messages of type **Create_tokens**(t,p,TT,CT) in non-decreasing order. This message is carrying Created Tokens when *t* in LP_i fires leading to a deposit of tokens in place *p* in LP_j. **Create_tokens** (t,p,TT, null) is a null message which is sent for synchronisation purpose. A null message is a timestamped signal sent by a LP to indicate to other LPs a lower bound of the timestamp of its future **Create_tokens** messages.

Each message of the simulated ECATNet model is represented in the obvious way: a record (struct) with elements representing its type, sending process, sending/receiving transition, receiving process, sending/receiving place, token time. An additional information *GenProc* (Generating Process) is used in the simulation of cyclic models to avoid an overflow of null messages in the message passing system.

5.6.2 Exploiting Lookahead

Our protocol provides a set of constructs to specify the lookahead of an ECATNet model and thus improves its performance with conservative implementations. As the causality constraint may introduce deadlocks, they are typically avoided by using null messages. Their efficient implementation is also facilitated because each LP maintains the set of its source and/or destination LPs.

The structure of the ECATNet simulated model has to be analysed to see where some lookahead can be extracted, and to tailor the simulator to exploit it. If this can be effectively done, timestamps of null messages will have higher values and the overall number of required null messages will be reduced, while a faster clock advance of the LPs will be allowed.

To highlight lookahead that exists in Petri nets simulation, if a transition *t* starts firing at *Tsim*, a LP can predict exactly when the tokens created by this firing are deposited: *Tsim* + `FiringTime(t)`. In order to compute the timestamp of the null message that will be sent from LP_i to LP_j, the information about timed transition among the succeeding transitions up to LP's output border has to be kept [CF93b]. To do so, it is necessary to analyse the structure of each ECATNet subnet. Lookahead is then the accumulated firing time of these succeeding timed transitions. It can be established for a pair of transitions in each subnet by a static analysis of the subnet's structure.

However, as suggested in [DBGM96a], a *time window* can be used when partitioning with SCS rather than null messages to prevent incorrect computations from propagating too far ahead into the simulated time. Lookahead provides a window [*Tsim*, *W(Tsim)*] such that all events with timestamps in the window can be executed safely and without further communication between LPs. We refer to [NR91] to compute it.

Lookahead is computed at the time the transition to fire is known. First, we find δ_{min}, the minimum firing time among all the LP's enabled transitions. Second, we find E_{min}, the value of the least timestamp on any event in EVL. As EVL is sorted on the time of occurrence of events, E_{min} is the event's timestamp in the head of the list. If the list is empty, we take $E_{min} = \infty$. Third, we compute $E_{min} + \delta_{min}$ which provides the desired upper bound.

We suppose that each LP call a routine **Bound_NextMsgTime()** which returns the value $E_{min} + \delta_{min}$. We then compute W(Tsim) = Min[Bound_NextMsgTime()] among all LPs. The LPs synchronise *globally* to make W(Tsim) known to each one.

5.6.3 Algorithms

Every ECATNet LP repeats the following steps:

Step 1 LP waits to select an input message m from its input communication channels by invoking `ReceiveMessage()` and inserts m into the relevant IQ_i. Each input queue IQ_i has a clock CC_i associated with it that is equal to either the timestamp of the message at the head of the queue if the queue contains a message, or the timestamp of the last received message if the queue is empty.

Step 2 LP processes the first event of EVL if there is no token message in one of the IQ_is with smaller timestamp, or to process the token message with the minimum token time in IQs. The execution of `ExecuteEvent()` or `ExecuteMessage()` may invoke `SendMessage()` to send output messages:

```
/* TokenTime(m) returns the timestamp of message (or event) m */
 if(TokenTime(First(EVL)) < $CC_{is}$
     ExecuteEvent(First(EVL));
else ExecuteMessage(First($CC_{is}$));
```

The execution of the event e by a LP (after invoking `Dequeue(e)`) is described as follows:

```
  ExecuteEvent(e){
      UpdateLVT(TokenTime(e));
      /* invoke function according to the type of event e */
          Case Start_firing: invoke Start_firing();
          Case End_firing: invoke End_firing();
          Case Deposit_tokens: Deposit_tokens();
}
```

The execution of the message *m* by a LP is described as follows:

```
ExecuteMessage(m){
      UpdateLVT(m.timestamp);
      /* invoke function according to the type of message m */
          Case Null: invoke ComputeLookahead();
                     invoke GenerateNullMessage();
          Case Create_tokens: Create_tokens();
}
```

If a new message arrives from an input channel *i* then the corresponding CC_i is advanced and, if it is not a null message, it is inserted in the associated IQ_i. Null messages need not be stored, because their only interest is the advance they produce in the channel clocks. This advance may increase the message-acceptance horizon and thus may allow any awaiting **Create_tokens** message to be consumed.

When the `ExecuteMessage()` function processes a null message *m* in an LP, it invokes a function `ComputeLookahead()` to compute the timestamp of the output (null) messages. The `ComputeLookahead()` function implements the lookahead exploiting the following technique. First, we find δ_{min}, the minimum firing time among all the LP's transitions. Second, we find E_{min}, the value of the least timestamp on any event in EVL. If the list is empty, we take $E_{min} = \infty$. Third, we compute Min(`TokenTime(m)`, E_{min}) + δ_{min} which provides the desired uppon bound.

The null message is then sent to some or all output channels by invoking `SendMessage()`. As the number of null messages may become quite large during the simulation, a function `ReduceNull()` to reduce their number is invoked before sending them [Vri90]. This saves communication time (in the sender) as well as processing time (in the receiver).

A LP does not process any input message until it has received at least one message from each of its input channels. The input message with the smallest timestamp is selected for processing. The LP blocks as soon as the minimum timestamp of messages in IQs is not larger than the occurrence time of the first event in EVL (if IQ_i becomes empty, the value of CC_i is changed to 0).

To obtain the set of transitions that can fire in parallel within a LP, function Enabled() is invoked. Basically, it takes the left-hand side of the rewrite rules associated with the transitions and checks the multisets of pairs (p,[m]⊕), where p is an input place.

The firing of a transition *t* is as follows. The right-hand side of the rewrite rule associated with *t* is checked by invoking a function Rewriting(). If $t \in OT_i$ in LP_i, then a message carrying CT tokens is is generated and inserted in the corresponding output queue (OQ). If *t* has an output place in LP_i, it schedules an event **End_Firing** of *t*. A null message is also deposited for every output border transition in the corresponding OQ.

```
Start_firing(t){
    Destroy_tokens();
    /* invoke function according to the type of transition t in $LP_i$ */
      Case 1: /* $t \in OT_i$ */
        invoke GenerateTokenMessage(t, LVT + FiringTime(t));
      Case 2: /* t has an output place p in $LP_i$ */
        invoke Enqueue(End_firing(t, LVT + FiringTime(t)));
        invoke Enqueue(Create_tokens(p, LVT + FiringTime(t)));
      Case 3: /* $t \notin OT_i$ */
        invoke ComputeLookahead();
        invoke GenerateNullMessage();
    SendMessage();
}
```

When transition *t* is not in LP's output border ($t \notin IT_i$), the function Start_firing generates a null message in an LP by invoking function ComputeLookahead() to compute its timestamp. The ComputeLookahead() function implements the lookahead exploiting the technique described in §5.6.2. Then the null message is sent to some or all output channels by invoking a SendMessage() function.

5.6.4 Distributed Conflict Resolution Algorithm

A distributed conflict resolution algorithm is needed when partitioning with SCS. The LP first checks whether the local firing is possible. If the local enabling conditions of transition t are satisfied because t's input places are assigned to the same LP, no communication with other LPs is necesssary. Otherwise, t has to wait for any **Tokens_available** messages sent from $p \in P_{t_{in}}$.

As soon as the marking of place p is updated because of the process of an event **Create_tokens**, p has to inform $t \in T_{p_{out}}$ so that t can compete for the available tokens. To solve the conflict, the LPs synchronise and communicate via the following messages using a four steps algorithm:

Step 1 a **Tokens_available**(p,t,T0,M(p)) message is sent from place p in LP$_i$ at time T0 to inform transition $t \in T_{p_{out}}$ in LP$_j$ that the marking of place p is M(p)[1]. If p and t are assigned to the same LP, t's enabling conditions are available *locally*;

Step 2 the transition t waits for **Tokens_available** messages from all the places $p \in P_{t_{in}}$, then processes T1 = MAX(T0), the latest timestamp among these messages. It sends a **Tokens_requested**(t,p,T1,IC(p,t)) message as a reply if its enabling conditions are satisfied (TC(t) is true and IC(p,t) is enabled). Otherwise, it sends a message to p to inform that it does not need the token;

Step 3 p waits for a reply for each **Tokens_available** message previously sent, collects **Tokens_requested** messages and then processes T2 = MIN(T1). If the set of transitions requesting tokens cannot fire in parallel, p executes a conflict resolution algorithm for choosing the transition t to be fired. A **Tokens_allowed**(p,t,T2,DT(p,-t)) message is sent to such transition;

Step 4 the transition t collects **Tokens_allowed** message from all its places $p \in P_{t_{in}}$ and then sends a **Tokens_consumed**(t,p,T2,DT(p,t)) message with timestamp T2 to such places. It processes T3, the time of the end of its firing (T3 = T2 + `FiringTime()`).

[1]In case of an inhibitor arc, p sends a Tokens_available(p, t, T0, empty).

5.6.5 Conflict Resolution Strategy

When multiple transitions sharing input places become enabled at the same time and compete for tokens, a decision must be made about which transition (or set of transitions) to fire in case they cannot fire *in parallel.* Two functions exist to solve this conflict:

- priorities may be specified for transitions: $\Pi : \mathrm{T} \to \mathrm{N}$ assigns priorities π_i to T-elements $t_i \in \mathrm{T}$.
- the decision place p selects randomly according to a probability distribution defined by the user one or more transitions that it would like to fire and offers them tokens via a **Tokens_allowed** message if these transitions $\in T_{p_{out}}$. If a transition t is lucky enough to receive **Tokens_allowed** messages from all places $p \in P_{t_{in}}$, it then fires. Otherwise, it replies by sending a message that it cannot use any of the tokens. Its input places try again later in simulation time by sending another **Tokens_available** message.

Note that the marking's update of place p may lead to the enabling and the firing of several output transitions at the same time.

5.6.6 Places with Limited Capacity

As explained in §2.4.2, in addition to TC and IC to be true, the third transition enabling condition in ECATNets is related to the capacity of the output place where the created tokens CT have to be deposited.

If transition t and its output place p are assigned to the same LP, then information regarding the capacity of p and its actual marking are available *locally.* In case they are not assigned to the same LP ($t \in \mathrm{OT}_i$ and $p \in P_{t_{out}}$), synchronisation and communication are necessary via the following messages using a three steps algorithm:

Step 1 Deposit_request(t, p, T0, CT). This message is related to the firing of transition t in LP_i after checking that its first two enabling conditions are both satisfied (TC and IC are true). This message is sent from t in LP_i at time T0 to request firing from place $p \in P_{t_{out}}$ in LP_j leading to a deposit of CT tokens in p.

Step 2 Deposit_request_ACK(p, t, T1, CT): the place p in LP_j waits for **Deposit_request** messages from all the transitions $t \in T_{p_{out}}$. Then it proceeds to the sorting in increasing timestamp order of TO_i, i = 1..n, where n is the number of transitions requesting firing. Starting with transition with the least timestamp TO_i, p checks whether the enabling condition $M(p) \oplus CT_i \leq Cap(p)$ is satisfied. The firing is allowed at either the time specified by t or by LP_j. If the deposit of tokens is not possible at $T0_i$ because $M(p) = Cap(p)$ (the place is full) or $M(p) \oplus CT_i > Cap(p)$ (overflow), LP_j sends a **Deposit_request_ACK** (acknowledgement) message as a reply to t specifying when exactly t can fire. The value of T1 is *local* to LP_j and is related to the time of p's output transition(s) is (are) going to fire. To exploit such information about T1, LP_j has to check EVL, IQs or even wait for future messages from its neighbours.

Step 3 t waits for a reply for each **Deposit_request** message previously sent, collects **Deposit_request_ACK** and processes T2 = MAX(T1). It then fires at T2 (if still enabled) and sends a **Create_tokens** for each $p \in P_{t_{out}}$.

5.7 Results of the Experiments

There are several parameters to take into account to explore their influence on the performance of the distributed simulation algorithms:

- the size of the ECATNet (number of places and transitions);
- the scenario of the simulation (eg. the distribution of the *firing times*);
- the number of *processors* used.

Output Data

The description of the ECATNet model is detailed enough to allow to obtain a good deal of insight into the behaviour LPs. In addition to the performance measures of the simulated ECATNet model such as the maximum and average number of algebraic terms in places, the number of transitions fired, ... the simulators can measure and give information about:

- the number of generated (positive and null) and consumed messages;
- the number of generated and consumed events;
- time statistics: time spent processing event (*Event Proc.*), sending positive and null messages (*Comm. Send*), receiving positive and null messages (*Comm.*

Recv), awaiting increments in the acceptance horizon (causality) and awaiting for messages to be received (communication) (*Blocking*), awaiting at the end barrier to terminate the protocol (*Term. Protoc*).

When LVT reaches the *end_of_simulation* value, each LP collects statistics, summarises them and sends the final results to be shown on the screen or saved in a file.

Performance Results

We have run our initial tests for distributed simulation on discrete-event ECATNet model of the Ethernet transmitting station. All the code is written in C, the run time system is made up on top of MPI and tested on a network of Sun Sparc workstations. In the remaining, a *processor* (or *processing element (PE)*) refers to a Sun Sparc machine with a single processor.

Some parameters of the model can be varied to evaluate their effects in the performance of the simulators such as the collision *probability* and the distribution of the *delay time.* Experimental work was also carried out with the purpose of evaluating the performance of the event-driven sequential simulator (§5.2.4).

We have run the conservative simulation (Chandy-Misra's approach with deadlock avoidance, CMB-DA) of the ECATNet efficient partitioning in a parametrisation: λ_{DELAY} is exponentially distributed with mean 1.0 (transition's delay time), probability of occurrence of a collision = 0.5, duration of the simulation = 10,000 cycles and N = 1, 2, 3 processors respectively. Each LP is assigned to a dedicated processor and reside there for the whole real simulation time.

In the simulation a total of about 100,000 transitions were fired. It took the CMB-DA simulator 5 minutes 41 seconds to execute on 3 processors whereas SEQ needed only 48 seconds (all timings are average time for execution in seconds). The speedup of 1.4 using 3 processors observed in Figure 5.4 is reported by comparison with the distributed simulation code running on a single processor.

We faced two primary problems with the conservative approach [DBGM96b]. The first problem is related to cyclic models (the Ethernet transmitting station is a cyclic one). A null message sent out by one LP could possibly circulate through a series of other LPs and arrive back at the original sender at the time it was sent (eg. a null message generated after transition DELAY in LP_2 fires). In some cases, the system is modelled using exponential firing times, and because these have a minimum delay of zero, they must be modified for use in the distributed simulation to avoid a deadlock situation. To cope with this situation, a firing transition identifier was

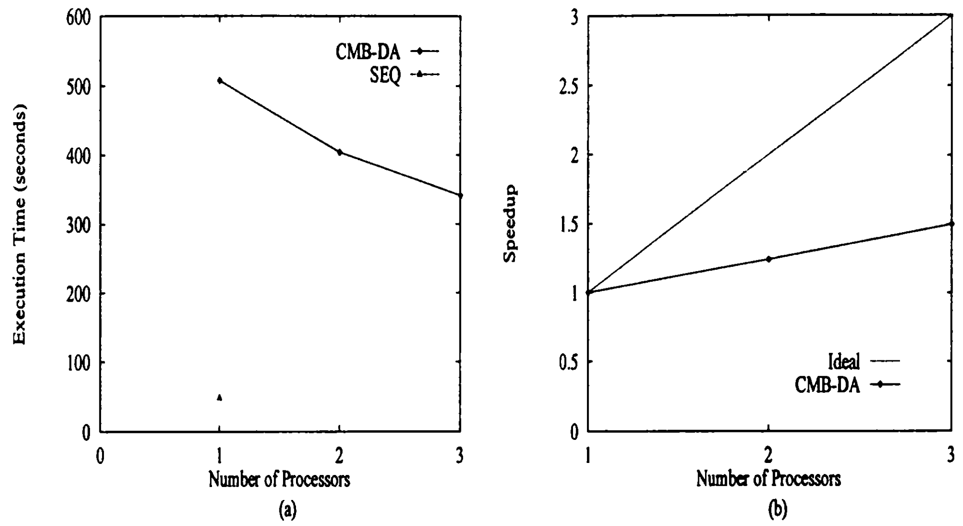


Figure 5.4: (a) Execution Time of CMB-DA; (b) Speedup over its One Processor Execution.

introduced in each null message generated by a timed transition in LP_i. If the null message arrives back at the original sender LP_i, it is simply discarded. This approach, similar to the "carrier null message" approach proposed by Cai and Turner [CT95] will permit LP_i to identify a null message initiated by itself. Deadlock is avoided because there are no cycles in which the collective timestamp increment of messages traversing those cycles is 0.

The second problem is related to the large number of null messages exchanged between LPs leading to considerable overhead. Since a large number of transitions in LP_2 are immediate, there is no need to generate new null messages when these transitions fire if there is no timed transition among the succeeding transitions up to the output border, i.e., the accumulated firing time is 0, and this does not change lookahead. A reduction of the number of Null messages improves the distributed simulation performance.

The results show that communication time between LPs is quite important. A LP does not block when invoking `SendMessage()`. However, it might block when invoking `ReceiveMessage()`, but only if no suitable event is ready to be consumed. Meanwhile, all the incoming messages can be received and stored in IQs by the LP which is a greedy receiver.

LP_1 and LP_3 contain one single transition in the output border, whereas LP_2 contains four (Figure 5.3). LP_2 is the process with the largest event processing time because of its large number of transitions (8) and places (11) leading to an important number of events to schedule. Using this partitioning technique, we note that:

- the load could be unbalanced;
- only large ECATNet models lead to a large number of LPs.

ECATNet models show enough parallelism to make good use of multiple processors. In part, even when identifying the set of LPs that can execute independently and concurrently on separate processors, we were not surprised to see that the events **Start_firing** and **End_firing** corresponding to the operations "Destroy Tokens" and "Create Tokens" that could execute in parallel were actually slowing execution in the NOW as the parallel system.

Using this model, an important aspect to be taken into account is that the simulation of an *activity* consumes a negligible amount of CPU time: only some increments of counters or movements of small amounts of data are needed. For this reason, when a simulator is running an ECATNet model, the execution time is mainly due to the simulation algorithm itself (management of events, synchronisation of LPs, interprocess communication, ...) and not to the simulated model (actual simulation of events). Another point to be considered is the high communication time needed to pass a message between two workstations (because the latency is high). These considerations are specially relevant when evaluating the distributed simulators.

From these considerations, and as the time to simulate an event is negligible, we tested what happens when the actual cost of processing an event is high. To do so, we made some experiments to artificially increase this cost. Chiola and Ferscha [CF93b] suggest the insertion of *additional transitions* in the various LPs to increase the amount of local simulation work (thus increasing the computation/communication ratio). Another method used in parallel computing suggests the insertion of various amounts of time of the order of micro- or milli-seconds. The method we use simply inserts a loop in the form "**for** i=1 **to** *W* **do** nothing" in the code of the simulator at the point where an event is simulated. The parameter *W* is a form of *synthetic workload*, which can be varied to evaluate its effect on the execution time of the simulator.

The aim of an efficient code is to keep all the processors busy (load-balancing) while minimising the amount of communication (usually the bottleneck in parallel processing), through often there is a *trade off* between communication and processing.

In order to investigate the influence of the synthetic workload on the CMB-DA simulator's performance, the value of *W* was varied to correspond to the values: 1, 10, 100, 1000 and 10,000 (therefore varying the grain size of event processing).

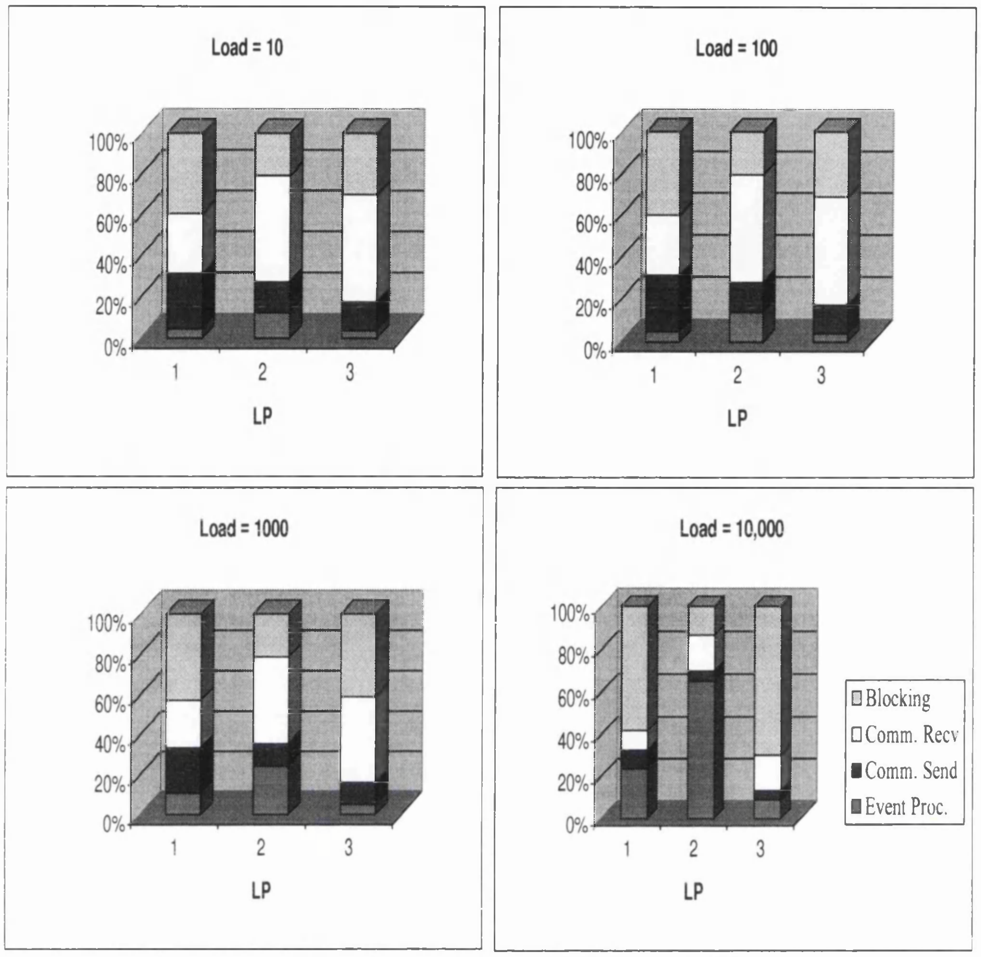


Figure 5.5: CMB-DA: Impact of the Workload onto the Execution Profile (3 PEs).

In Figure 5.5, we see that experiments with significant synthetic workload exhibit a more balanced communication/computation ratio (the amount of CPU time to simulate an activity increases with W). Figure 5.6 shows: (a) the execution times for various values of *W*; (b) the speedup obtained by comparison with the sequential simulation code running on a single processor. It can be seen that a moderate amount of speedup is obtained with a large value of *W* on 3 processors.

To summarise the results of these experiments on the Ethernet transmitting station ECATNet model, it can be concluded that the conservative approach to distributed simulation exhibits a poor performance (Figure 5.7). Nevertheless, other models could successfully be simulated using CMB-DA, if they belong to any (or even better, several) of these groups (see chapter 8):

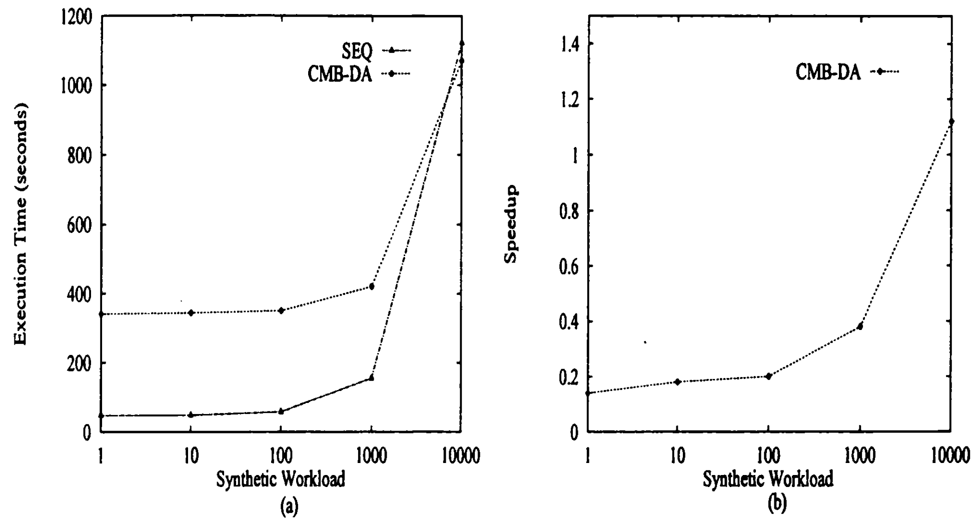


Figure 5.6: Influence of the Synthetic Workload in the Simulation. (a) Execution Times for SEQ and CMB-DA (3 PEs); (b) Achieved Speedup over SEQ.

- models that synchronise in a natural way by means of useful messages. In this case, the need of null messages is low, which is specially convenient when a message passing architecture is used;
- models with high levels of lookahead, which allows progress in the simulation when the LPs do not have events to process. If it is not possible to process useful messages, at least null messages are generated less often and with larger timestamps, and the channel clocks (and then the acceptance horizon, the LVT and the simulation) advance faster;
- models where the simulation of an event requires a high communication effort, which is distributed among the processors.

5.8 Conclusion

In this chapter we have shown how DDES has been successfully used to study a variety of real-world systems, including the study of different aspects of parallel computing. The conservative algorithms we proposed in this chapter are expected to improve the ability of efficiently simulate the behaviour of systems modelled by ECATNets over a period of time.

The decomposition of the "initial" ECATNet model into disjoint partitions representing smaller sized models has a strong impact on DDES performance. The

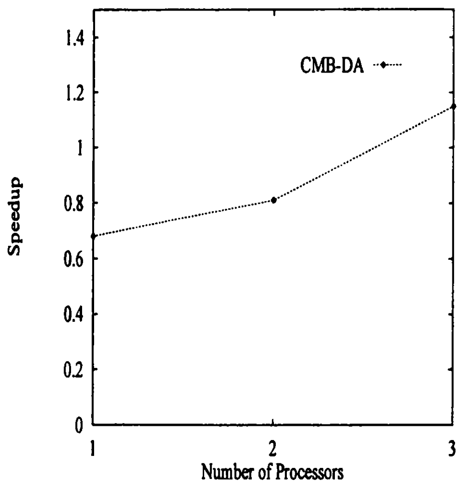


Figure 5.7: Speedup of CMB-DA over SEQ (W=10,000).

"separation of concern" strategy does not necessarily lead to a good partitioning for two reasons. The first reason is the duplication of places in the different subnets, the second one is related to the assignment of the transitions and the places to the LPs, and consequently to the case where several transitions share input places. Because a transition and its input places are not always assigned to the same LP, it is necessary to exchange messages between LPs before the transition firing. We developed a proper communication protocol among LPs in order to implement a distributed conflict resolution strategy for transitions sharing input places and thus competing for tokens. Obviously, such conflict resolution strategy may induce substantial overhead in the distributed simulation in a message-passing environment and may prevent efficiency.

A more efficient partitioning technique is used to implement the conservative simulator based on message passing and is related to the firing rule [NR91, CF93b]: a LP is a set of transitions along with their input places such that local information is sufficient to decide upon the enabling and firing of any transition. The simulator guarantees a certain performance speedup with respect to the sequential simulator, even though might be affected by load balancing problems on a wide number of ECATNet structures.

Compared with other works on parallel and distributed simulation of Petri nets [TZ91, AD91, NR91, CF93b, Tur96], the protocols of distributed simulation of ECATNets differ in at least one of the following points:

1. ECATNets are high-level algebraic Petri nets;

2. The simulation protocols have to respect rewriting logic;
3. The simulation protocols have to respect time, timed transitions and places with limited capacity;
4. The state of the ECATNet model is distributed;
5. The proposed simulation protocols use either a "separation of concern" or an efficient partitioning.

Chapter 6

Optimistic Simulation of ECATNets

We present in this chapter an ECATNet distributed simulator based on Time Warp. As for the conservative one, two partitioning techniques (SCS and EP) are considered.

6.1 Introduction

In ECATNet optimistic protocols, a LP is allowed to process events in any order. However, the underlying synchronisation protocol must detect and correct violations of the causality constraint. The simplest mechanism for this is to have each LP periodically save (or checkpoint) its state. Subsequently, if it is discovered that the LP processed messages in an incorrect order, it can be rolled back to an appropriate checkpointed state, following which the events are processed in their correct order. The rollback may also require that the LP unsends or cancels the messages that it had itself sent to other LPs during the simulation. An algorithm is also required to periodically compute a lower bound on the timestamp of the earliest global event (GVT). As the model is guaranteed to not contain any events with a timestamp smaller, token time messages timestamped earlier than GVT can be discarded.

This chapter is structured as follows. §6.2 introduces the characteristics of an optimistic ECATNet LP, and a description of the simulation engine based on TW-LZ is done in §6.3. The Ethernet transmitting station ECATNet model presented in §2.4.5 is chosen to carry out the experiments to evaluate the TW-LZ simulator in §6.4. Some conclusions are summarised in §6.5.

6.2 The Optimistic Simulator

6.2.1 Logical Processes

In order to simulate an ECATNet partition according to Time Warp, the data structures a LP maintains are: (1) a local virtual time (LVT) representing the LP's simulation time; (2) a single input queue (IQ) which collects recently arrived messages (*positive* and *negative*) ordered by time; (3) an output queue (OQ) which contains the *positive* messages to send; (4) an output queue (OQN) which contains the *negative* copies of the messages recently sent, ordered by time (*antimessages* for unsending the originals); (5) an event list (EVL); and (6) an event stack (ES) which records all state variables such that a past state can be reconstructed on occasion.

The Time Warp ECATNet protocol provides a set of facilities that can be used to control the various parameters that affect the performance of an optimistic implementation. These include setting the frequency of checkpointing and GVT computation among others.

6.3 Simulation Engine

The simulation engine's main task is not only to synchronise the LPs simulating the various subnets by controlling the timestamp of each message and LVT, but also to implement the functions of communication arcs, state saving and GVT management.

6.3.1 Types of Messages

LPs communicate by sending two types of timestamped messages: (1) **Create_tokens**(t,p,TT,CT,' is a message carrying CT when t ∈ OT_i fires leading to a deposit of tokens in place p in LP_j. The timestamp of this message is the accumulated firing time of transition t; (2) **Create_tokens**(t,p,TT,CT,'-') is used in the rollback mechanism needed for synchonisation to indicate which previously sent message should be cancelled.

6.3.2 Separation of Concern Partitioning

Since a transition $t \in IT_i$ may have one or more input places in different LPs (say LP_j), whether this transition is enabled will depend on the marking in all of its input places. In such case, LP_i will check first t's *local* input places. When all the local enabling conditions are satisfied, LP_i will assume that t is enabled and will determine the next transition to be fired among all local enabled transitions. If the next transition to be fired is t, two messages are exchanged for LPs synchronisation and to ensure that the marking in a place $p \in \mathrm{OP}_j$ is consistent [DBGM95]:

- **Tokens_requested**(t,p,TT,IC): is used for requesting tokens from p[1]. This message timestamp (received by LP_j from LP_i) represents the next local clock value (LVT) of LP_i if and only if LP_i can get its required tokens from LP_j and fire t;
- **Tokens_requested_ACK**(p,t,TT,DT,flag): is sent by LP_j and is used as a response to a **Tokens_requested** message. The response is either positive (flag=true) or negative (flag=false), and TT = LVT_j.

LP_i sends a **Tokens_requested** message to LP_j to ask for tokens. As the protocol is optimistic, LP_i does not wait for the response from LP_j, the local simulation in LP_i can be continued under the current local marking. When LP_j inputs and processes the **Tokens_requested** message, it checks t's input place(s). When t's enabling conditions are satisfied (TC is true and IC is enabled), LP_j sends a **Tokens_Requested_ACK** message with a positive response to LP_i, then updates its local marking (flag=true, `Destroy_tokens()` is invoked). In case tokens are requested from a decision place by several transitions, the strategy explained in the previous chapter (§5.6.5) is used to solve it. If t's enabling conditions are not satisfied, LP_j sends a **Tokens_requested_ACK** message with a negative response to LP_i (flag=false). When LP_i receives all response messages from other LP's, it checks the answers of the **Tokens_requested_ACK** messages. If all these messages are positive, t will be fired as assumed, otherwise t is not ready to fire before the time indicated by the maximum TT of the **Tokens_requested_ACK** messages with the negative response. In such case, LP_i must return all tokens carried by the positive **Tokens_requested_ACK** messages (by sending back **Create_tokens** messages to the senders). At that LVT, t may or may not be enabled depending on local marking conditions. If t is still enabled, new **Tokens_requested_ACK** messages are sent and the above process is repeated.

6.3.3 Checkpointing

In [LL91a], Lin and Lazowska indicated that the efficiency of state saving and restoration may have a significant effect on the performance of Time Warp. In

[1]Note the inhibiting conditions in case of an inhibitor arc.

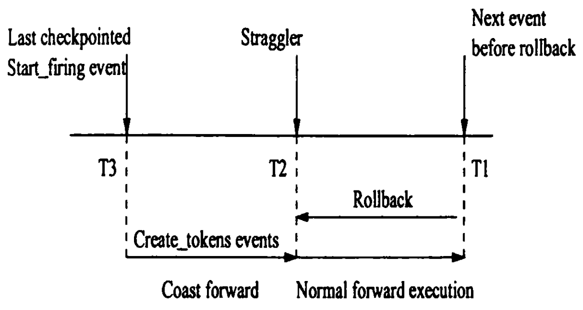


Figure 6.1: Rollback in TW ECATNet Simulation.

order to achieve the best possible execution time, it is important to reduce the overhead associated with saving and restoration states. It is important to realise that, in general, the rollback overhead is comprised of two components: state saving (i.e., checkpointing) and state restoration (i.e., the costs associated with recovering an earlier state after a rollback).

In Time Warp ECATNet simulation, we first made a LP save its state each time an event of type **Destroy_Tokens** or **Create_tokens** is executed (frequent checkpointing). We realised that the set of LPs tend to consume all the allocated memory because of the large size of the state to save per executed event. We later had to turn to infrequent state saving.

State checkpointing is done each time a transition fires (Figure 6.1) and the form of the entries in ES is (t_i,LVT,M) where t_i is the transition that has fired at time LVT yielding a new marking M. In case a LP receives a straggler message m, LVT is set to `TokentTime(m)` and a rollback is performed to time T_3, the timestamp of the most recent checkpointed event in ES but not exceeding LVT, and resumes execution from that point. The coasting-forward phase consists mainly in the re-execution of **Create_tokens** messages in IQ which execution did not lead to a transition firing.

6.3.4 Message Cancellation

When a LP rolls back, it first inserts the straggler message into IQ and updates LVT. The state at (new) time LVT is restored. All incorrect computation is undone by popping out all the records prematurely pushed in ES. If rollback is applied with *aggressive cancellation*, all messages in OQN with token time > LVT are annihilated by removing them from OQN and sending them. The simulator can also apply *lazy cancellation*. In the case reevaluation yields exactly the same positive messages as already sent before, the new positive message is not resent. This will prevent unnecessary message transfers as well as possibly new rollbacks in other LPs.

6.3.5 GVT Computation

The *Global Virtual Time* (GVT) is considered as the virtual clock for the system as a whole. The knowledge of GVT reduces past state savings in IQ, ES and OQN.

In the GVT computation method we implemented, the computation is performed in a fully distributed fashion using a token-passing scheme. LPs are organised into a logical, unidirectional ring. A special message, a *GVT_PACKET* circulates on a complete closed predefined path among the processors. A LP called *Coordinator* (eg. LP_0) is responsible for generating GVT_PACKET which it forwards to its successor after calculating the minimum of its present LVT and the timestamps of all unconfirmed messages (**minLVT**), and posting this in the GVT_PACKET. Thus, in a network of 4 processors, GVT_PACKET will circulate on the path 1-2-3-0. A GVT estimate is calculated by taking the minimum of all **minLVTs** in GVT_PACKET. When GVT_PACKET returns to its owner (the coordinator) after traversing the ring, the computed GVT value is known and is then transmitted via a *GVT_MESSAGE* to the next LP in the ring.

We faced the situation where the LP responsible for generating GVT_PACKET message was ready to send a new one but did not receive the acknowledgement of the previous one, thus was not able to send a GVT_MESSAGE.

To solve this problem, we added a certain degree of "conservatism" into this LP. With a purely optimistic scheme, the LPs advance unboundedly (in fact, LP_0 advances too fast). The conservatism is imposed by limiting the ability to go into the future: LP_0 is allowed to advance to a certain degree, calculated as the value of the GVT plus a *time window size.* If it tries to go beyond, it is temporarily blocked. The window size is not dynamically changed and is the time interval between two GVT times computations.

6.3.6 Algorithms

A LP processes messages in IQ by checking the sign (positive or negative) and the timestamp of each one (these messages are ordered by their timestamp, the head of the queue corresponds to the smallest one). Messages with timestamp $\geq$ LVT are inserted in IQ. In case of a positive message, it is inserted in timestamp order, otherwise (the sign is negative) it annihilates the positive message in IQ previously sent. If the message is a *straggler* (timestamp of the message $<$ LVT), the LP must roll back and restore a valid state. As for the conservative simulator, the processing of the first event in EVL or the first message in IQ generates either new (internal) events in EVL or (external) output messages.

```
while (GVT <= end_of_simulation) {
    ReceiveMessage();
    /* test type of message m just received */
    if TypeMessage(m) == GVT_PACKET
       ManageGVT();
    else InsertInIQ(m);
}
```

InsertInIQ() is invoked each time a message is received, to store it in IQ. The received message may be a **Create_tokens** scheduled for the future, which is simply inserted in the right position. It may also be a straggler, which causes a rollback before its insertion in the queue. Finally, it can be a negative message, which requires a positive message to be annihilated, either without triggering a rollback (if its corresponding positive message has not been executed yet), or after a rollback (if its corresponding positive message has already been executed yet).

```
InsertInIQ(m){
    if(TokenTime(m) > LVT) {
       if(TypeMessage(m) == '+') InsertFuture(m);
       else AnnihilateFuture(m);
    }
    else if(TokenTime(m) < LVT) {
       if(TypeMessage(m) == '+') InsertPast(m);
       else AnnihilatePast(m);
    }
    else /* TokenTime(m) == LVT */ {
       if(TypeMessage(m) == '+') InsertFuture(m);
       else {
          if(AlreadyConsumed(m)) AnnihilatePast(m);
          else AnnihilateFuture(m);
       }
}
```

`InsertPast()` and `AnnihilatePast()` both include the execution of `Rollback()`. `InsertPast()` is executed whenever a straggler arrives. It performs the following steps:

- search in IQ the location where the straggler has to be inserted;
- update LVT to match the straggler's timestamp. Recover the state of the LP at that time. This can be found in the ES, in the position just before the straggler, or may require an additional search in the past plus a coast-forwarding phase;
- clear ES and OQN, i.e., eliminate all the elements whose timestamp is larger than the straggler's;
- execute the straggler (invoke `ExecuteMessage()`).

`AnnihilatePast()` is executed when a negative version of an already executed message is received. This function is very similar to the previous one:

- search in IQ the location where the positive message is stored;
- recover the state of the LP saved just before the corresponding positive message was consumed. As before, this can be found in ES, in the position just before the annihilated message, or may require an additional search in the past plus a coast-forwarding phase;
- clear ES and OQN, i.e., eliminate all the elements whose timestamp is larger than the straggler's;

Each time a LP receives a GVT message, it performs the fossil collection procedure, retrieving memory from the input (IQ), state (ES) and output queues (OQN).

```
FossilCollection(GVT) {
    /* Discard Old Messages in IQ with TokenTime() < GVT */
    DiscardIQ(GVT);
    /* Discard Old Messages in OQN with TokenTime() < GVT */
    DiscardOQN(GVT);
    /* Discard Old State Entries in ES with TT < GVT */
    DiscardES(GVT); }
```

The LP processes the first event of EVL if there is no token message in IQ with smaller timestamp, or the token message with the minimum token time in IQ. As in CMB-DA, the execution of `ExecuteEvent()` or `ExecuteMessage()` may invoke `SendMessage()` to send output messages:

```
 if(TokenTime(First(EVL)) < First(IQ))
    ExecuteEvent(First(EVL));
else ExecuteMessage(First(IQ));
```

The execution of a `Start_firing` event yields to a new state which is saved in ES by invoking `SaveState()`.

6.3.7 Places with Limited capacity

If transition t and its output place p are not assigned to the same LP ($t \in OT_i$ and $p \in P_{t_{out}}$), synchronisation and communication are not necessary because of the "optimism" of the protocol. However, when a **Create_tokens**(t,p,T1,CT,'+') message is generated by LP$_i$ after transition t fires and cannot be accepted by LP$_j$ because the capacity of the destination place p is limited and its marking is full, the message is rejected (it is lost). In this case LP$_j$ sends a **Create_tokens**(t,p,T1,T2,CT,'-') message:

- to inform LP$_i$ that an incorrect computation has been done; This will force LP$_i$ to rollback;
- to inform LP$_i$ about the time transition t could fire (T2 in this case).

6.4 Results of the Experiments

6.4.1 Output Data

Like in the CMB-DA simulator, the description of the ECATNet models is detailed enough and the simulator can measure and give information about:

- the maximum and average number of algebraic terms in places;
- the number of generated (positive and negative messages) and consumed messages;
- the number of generated and consumed events;

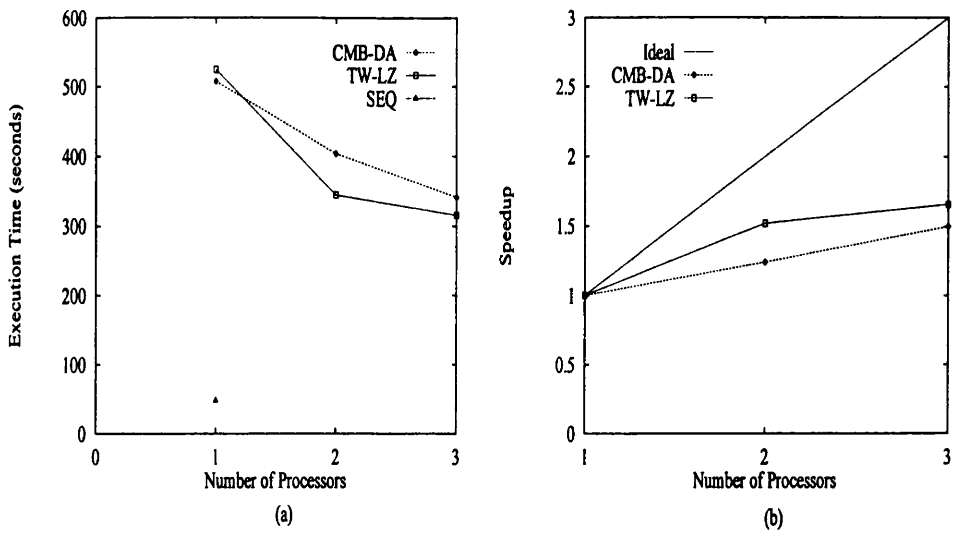


Figure 6.2: (a) Execution Time of TW-LZ and CMB-DA; (b) Speedup over their One Processor Execution.

- time statistics: time spent processing event (*Event Proc.*), sending positive and negative messages (*Comm. Send*), receiving positive and negative messages (*Comm. Recv*), rolling back (*Rollback*), awaiting for messages to be received (*Blocking*), saving states (*Checkpoint*), managing GVT (*GVT*)), awaiting at the end barrier to terminate the protocol (*Term. Protoc*).

When GVT reaches the *end_of_simulation* value, each LP collects statistics, summarises them and sends the final results to be shown on the screen or saved in a file.

6.4.2 Performance Results

In section 5.7 an evaluation of the CMB-DA algorithm and results of the distributed simulation of the ECATNet model of the Ethernet transmitting station have been presented. One we started experimenting with the TW simulator, we found that it was terribly greedy in memory demands: a big deal of memory space is required to store antimessages and copies of the state. In the other hand, the density of events in LP_2 is very high and, for this reason, the probability of a straggler to appear is very high too. Whenever a causal error was detected, we faced the situation where a straggler positive message changes the marking in LP_2 but does not cause the enabling of any new event in the past. In such case, LP_2 does not have to rollback. The simplified mechanism which has been used to recover was an appropriate insertion of firings made on ES, and the top of ES was copied considering a potential change

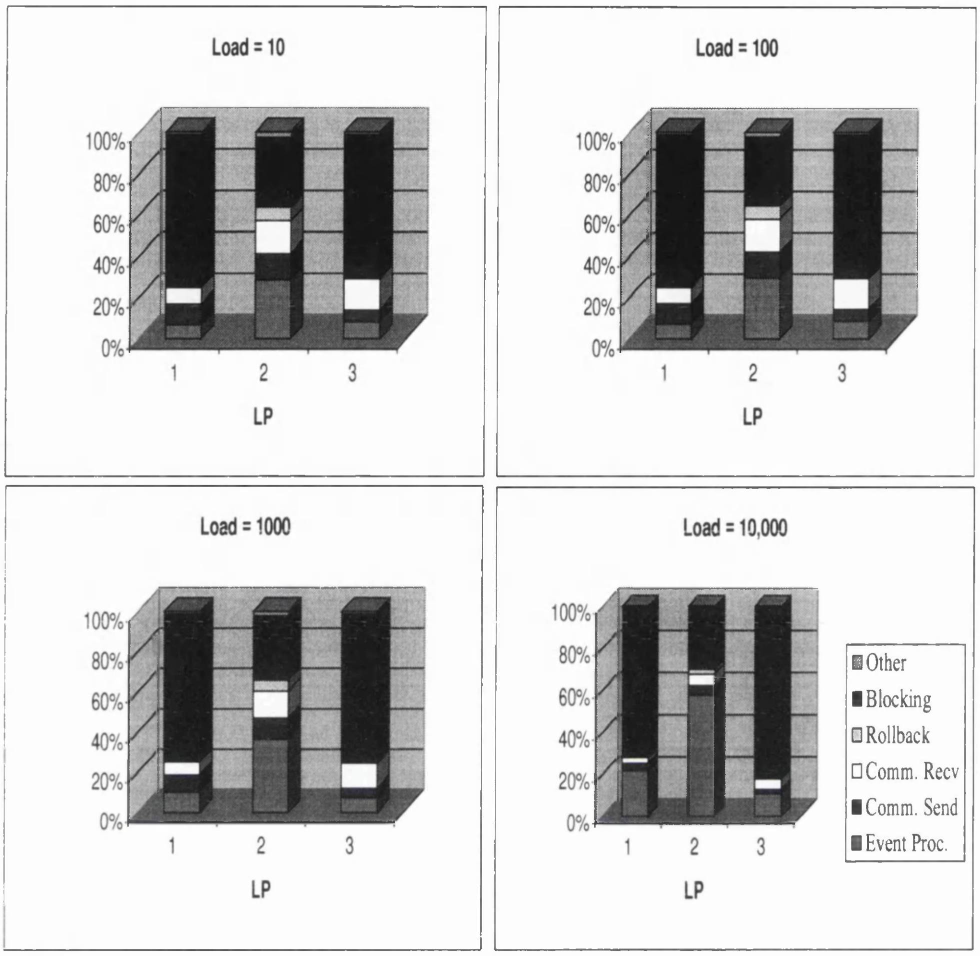


Figure 6.3: TW-LZ: Impact of the Workload onto the Execution Profile (3 PEs).

in the marking. This prevented sending antimessages, which could have given rise to an overflow in the message passing system.

Figure 6.2 shows the execution times of CMB-DA and TW-LZ simulators implementations with the same scenario as in section 5.7. Speedup (by comparison with the distributed simulation code running on a single processor) of 1.6 using 3 processors was observed using the TW-LZ engine. It took the simulation 5 minutes 15 seconds to execute on 3 processors.

We report that in the case of the Ethernet transmitting station the TW's performance is better than CMB-DA's. Although antimessages are not always needed, null messages are, and their number is not negligible in the CMB-DA simulator. Although the global time computation mechanism needs a continuous interchange

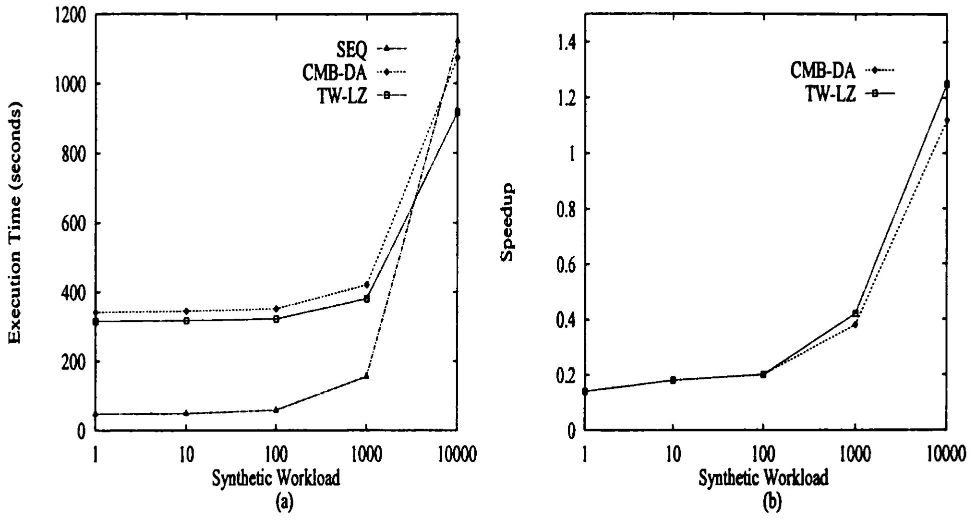


Figure 6.4: Influence of the Synthetic Workload in the Simulation. (a) Execution Times for SEQ, TW-LZ and CMB-DA (3 PEs); (b) Achieved Speedup over SEQ.

of messages among LPs to avoid an exhaustion of the memory space, it did not add any burden to the simulator.

As for CMB-DA, we investigated the influence of the synthetic workload on the TW-LZ simulator's performance by varying the value of *W* to correspond to 1, 10, 100, 1000 and 10,000. In Figure 6.3, we see that experiments with significant synthetic workload exhibit a more balanced communication/computation ratio. Figure 6.4 shows: (a) the execution times for various values of *W*; (b) the speedup obtained by comparison with the sequential simulation code running on a single processor. It can be seen that a moderate amount of speedup of 1.25 is obtained with the larger value of *W* on 3 processors (Figure 6.5).

6.5 Conclusion

To summarise our experience with the TW simulator, we say that it required much more programming effort than the CMB-DA, because the management of the data structures of TW (i.e., the memory management) is anything but trivial. Additionally, it was much harder to debug. Due to the experience carried out on the selected model, we are not able to state a definite set of characteristics of the simulated models that can help in obtaining a good performance from TW over CMB-DA. Nevertheless, the TW simulator is more general and offers a higher level interface to the user.

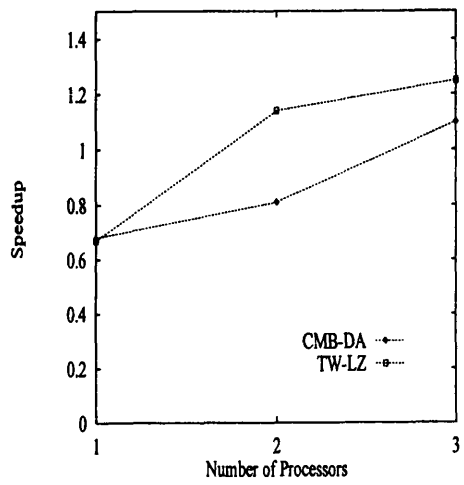


Figure 6.5: Speedup of TW-LZ and CMB-DA over SEQ (W=10,000).

Chapter 7

Synchronous Simulation of ECATNets

In this chapter we describe the design of a synchronous event-driven ECATNet distributed simulator, assessing its correctness and its performance potential. As for CMB-DA and TW-LZ, two partitioning techniques (SCS and EP) are considered.

7.1 Introduction

The experience by Kona and Yew [KY91] with a synchronous parallel event-driven (SYNC) simulator encouraged us to implement and test an ECATNet synchronous simulation engine on the network of workstations. It was easy to redesign the LPs to work synchronously by sharing the same *global simulation clock* instead of working asynchronously as it was the case for CMB and TW. Another motivation of the design is the rewriting logic which is able to find within a LP the set of transitions to be fired in parallel.

The basic design of an ECATNet LP in the implementation of the SYNC simulator follows the description given for CMB-DA, with some modifications that take advantage of the set of communication operations offered by the MPI library. MPI offers an excellent support for global operations such as broadcast and reduction operations.

7.2 The Synchronous Simulator

7.2.1 Logical Processes

Like conservative algorithms, ECATNet synchronous algorithms do not permit any causality errors. The set of LPs in the simulation process incoming messages only when the underlying synchronisation algorithm can guarantee that they will not subsequently receive a message with a smaller timestamp. The data structures according to the synchronous approach are: (1) a Local Virtual Time (LVT) representing an accumulated value of firing times in a LP and whose value is equal to the *global clock*; (2) an event list (EVL) ordered by time of occurrence; (3) input queues (IQ) (one queue per each input channel), which collect recently arrived messages ordered by time; (4) output queues (OQ) (one queue per output channel) which keep messages to send, ordered by time.

7.3 Simulation Engine

7.3.1 Types of Messages

The causality of events is preserved over all LPs by sending timestamped token messages of two types: **Create_tokens** (t,p,TT,*number*) is a synchronisation timestamped message sent by LP_i to LP_j to indicate the *number* of the actual **Create_tokens** messages it is ready to send; **Create_tokens**(t,p,TT,CT) is a message carrying CT when t in LP_i fires leading to a deposit of tokens in place p in LP_j.

7.3.2 Algorithm

The synchronous simulator is very much like the conservative one: SYNC has the same data structures and behaves as CMB-DA by not violating the causality constraint. An ECATNet LP takes into account the following considerations:

- when partitioning with SCS, the distributed conflict resolution algorithm described in §5.6.4 is needed. After the LPs synchronise globally and the new *global clock* is known, a transition $t \in \text{LP}_i$ has to wait for any **Tokens_available** messages sent from $p \in P_{t_{in}}$. As soon as the marking of place p is updated because of the process of an event **Create_tokens**, p has to inform $t \in T_{p_{out}}$ so that t can compete for the available tokens. The LPs synchronise and communicate via the four messages: **Tokens_available**, **Tokens_requested**, **Tokens_allowed** and **Tokens_consumed**;

- for both partitioning techniques (SCS and EP), the algorithm described in §5.6.6 regarding the places with limited capacity is needed when the next transition to fire $t \in$ LP$_i$ has its enabling conditions IC and TC true and has to request a deposit of tokens to $p \in P_{t_{out}}$ via messages **Deposit_request** and **Deposit_request_ACK**.

Each ECATNet LP$_i$ executes a loop of four basic operations :

1. LP$_i$ computes the minimum timestamp T_i among the events stored in the event list (**Start_Fire, End_Fire, Create_tokens**) and the messages stored in IQ (**Create_tokens**). Collectively, the LPs compute the minimum among all those values, $T_{min} = \min(T_i)$ using a reduction operation. This global operation also performs a barrier synchronisation.

2. Event consumption. All the events in the with timestamp $\leq T_{min}$ can be executed safely, because there are no relationships among them. During this step, internal events (**Start_Fire, End_Fire, Create_tokens**) are stored in the event list, while external events are stored as messages in OQ. LP$_i$ advances its clock to reach T_{min}.

3. Message distribution. LP$_i$ sends the messages generated in the previous step. In order to avoid deadlock situations, this is done in two phases:
   - every neighbour is informed about how many messages will be sent to it via **Create_tokens**(t,p,TT,*number*) messages, and
   - messages are actually sent (**Create_tokens**(t,p,TT,CT)).

4. Message Reception. LP$_i$ receives all the external events sent to it by other LPs. This is also done in two phases:
   - gathering from the neighbours the number of messages to receive, and
   - messages are actually received.

Messages distribution and gathering phases are designed in such a way that all the messages generated in one iteration are safely received and stored in the same iteration, without interfering with the next one. The resulting algorithm for a LP in the SYNC simulator is as follows:

```
clock = 0; LVT = 0;
while (clock ≤ end_of_simulation) {
    T = MinimumTimestamp(); /* Minimum among Events/Messages in EVL/IQ */
    clock = GlobalMinimum(T); /* Minimum among all the LPs */
    LVT = clock;
    while (NextEventTime() == LVT){
        if(TokenTime(First(EVL)) < TokenTime(First(IQ)))
            ExecuteEvent(First(EVL));
        else ExecuteMessage(First(IQ));
    }
    SendMessage();
    ReceiveMessage();
}
```

7.4 Results of the Experiments

7.4.1 Output Data

The output of SYNC also consists of a set of statistics about the simulated model such as the maximum and average number of algebraic terms in places, the number of transitions fired, ... plus a set of measurements about the behaviour of the simulator itself. Some interesting measurements are also obtained, for example:

- number of positive and synchronisation messages;
- the number of barrier invocations in the simulator.
- time statistics: time spent processing event (*Event Proc.*), sending positive and synchronisation messages (*Comm. Send*), receiving positive and synchronisation messages (*Comm. Recv*), awaiting for messages to be received (*Blocking*), barrier synchronising (*Sync.*), awaiting at the end barrier to terminate the protocol (*Term. Protoc*).

7.4.2 Performance Measures

The experiment on the distributed simulation of the ECATNet model of the Ethernet transmitting station presented in section 5.7 was performed with SYNC. It took the simulation 15 minutes 47 seconds to execute on 3 processors.

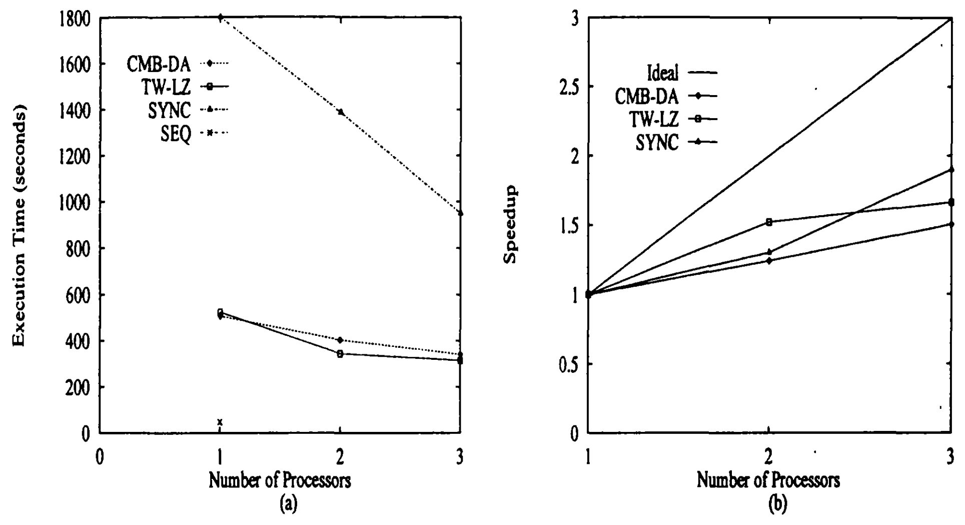


Figure 7.1: (a) Execution Time of SYNC, TW-LZ and CMB-DA; (b) Speedup over their One Processor Execution.

We found that synchronisation overhead considerably outweighs the time spent doing simulation work and that the simulator was terribly greedy at synchronisation barriers where a global reduction operation is performed first to compute the new value of the *global clock* (by invoking `GlobalMinimum`), followed by a broadcast operation to make this value known to each LP. Software support for global synchronisation in MPI makes each global barrier relatively easy to use. We were not surprised to find that synchronisation overhead can account for up to 65% of total simulation runtime. Figure 7.1 shows the execution times of SYNC, CMB-DA and TW-LZ simulators implementations with the same scenario as in section 5.7. Speedup (by comparison with the distributed simulation code running on a single processor) of 1.8 using 3 processors was observed using the SYNC engine.

We investigated the influence of the synthetic workload on the SYNC simulator's performance by varying the value of *W* to correspond to 1, 10, 100, 1000 and 10,000. The experiments with significant synthetic workload in Figure 7.2 exhibit a more balanced communication/computation ratio. Figure 7.3 shows: (a) the execution times for various values of *W*; (b) the speedup obtained by comparison with the sequential simulation code running on a single processor. It can be seen that a poor amount of speedup of 0.7 is obtained with the larger value of *W* on 3 processors (Figure 7.4).

In SYNC, the parallelism is exploited efficiently when multiple events occur at exactly the same time. However, the amount of attainable parallelism and the granularity could be too small to be useful. Also, we realise that the simulator per-

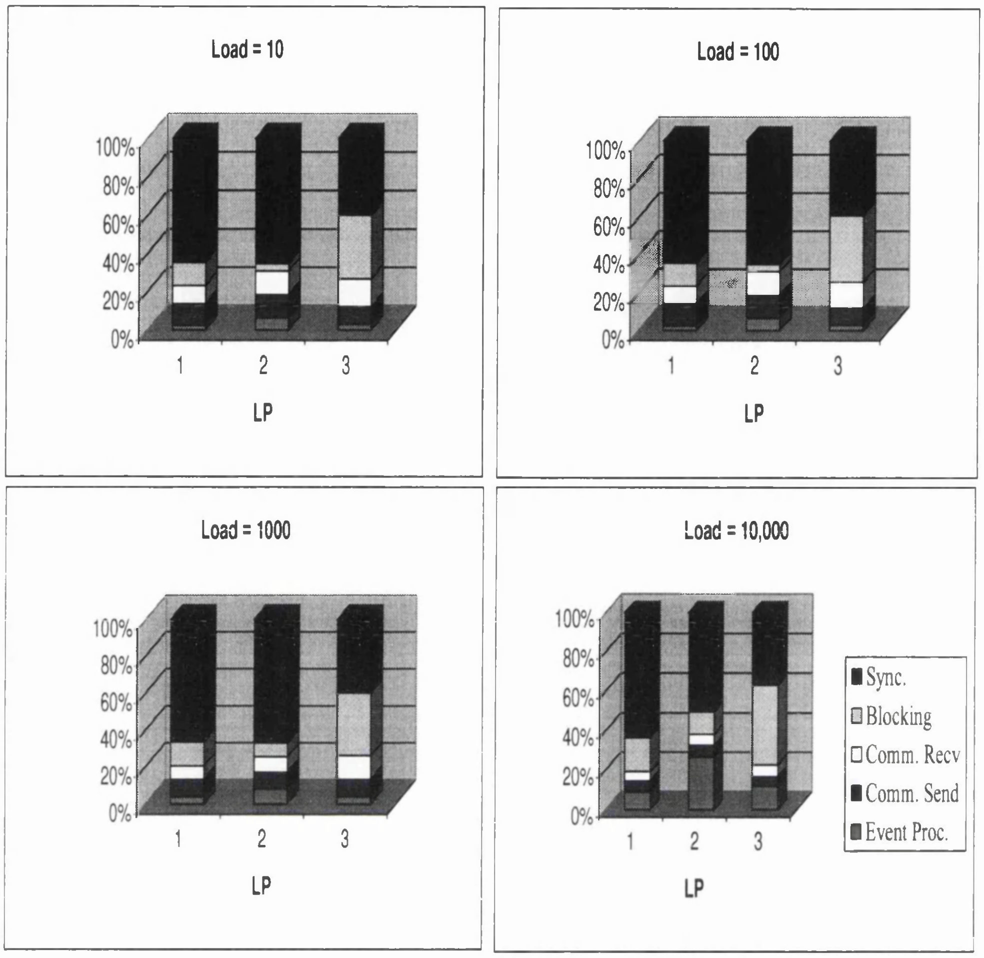


Figure 7.2: SYNC: Impact of the Workload onto the Execution Profile (3 PEs).

formance suffers if the barrier synchronisations must be performed very frequently. This situation occurs when few events occur at exactly the same time, or if load imbalance causes long waiting times at synchronisation points. From the experiment, we see that synchronisation overhead depends on four factors:

1. the frequency of sysnchronisations is controlled by the synchronisation time which depends on Min(LVT$_i$) of each ECATNet partition;

2. the duration of a synchronisation depends on the time it takes to execute the synchronisation operation and on the time spent waiting at the synchronisation point (which is enormous on a NOW). In MPI, all collective operations are blocking. Therefore, this could severely limits speedup;

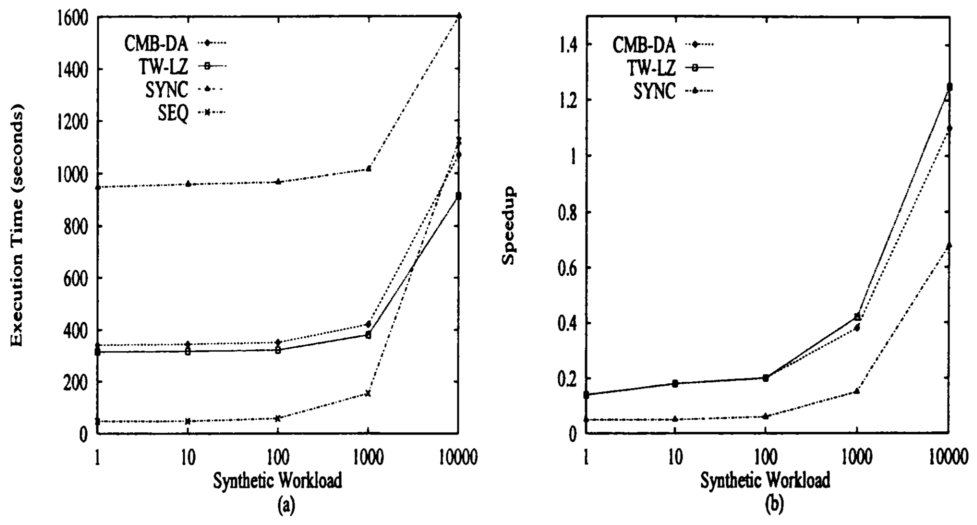


Figure 7.3: Influence of the Synthetic Workload in the Simulation. (a) Execution Times for SEQ, SYNC, TW-LZ and CMB-DA (3 PEs); (b) Achieved Speedup over SEQ.

3. the level of detail: the lengthier and/or more frequent the synchronisations, the larger the synchronisation overhead. Exploiting the parallelism where multiple events occur at exactly the same time to attain a large and useful parallelism depends merely on transition firing times: if there is a set of transitions in the system that have the same (*deterministic* or *stochastic*) time value, these transitions might **Start_firing / End_firing** in parallel. On the other hand, if very few events occur at the same time, the performance of the synchronous simulator is expected to be poor because of frequent synchronisations leading to a very small number of events to be executed;

4. the number of simulated ECATNet LPs: it takes m LPs less time to execute the synchronisation operation than n LPs, when $m < n$.

7.5 Conclusion

In SYNC, events are consumed in timestamp order preventing causality errors to occur. Only those events with the same timestamp are executed concurrently and they are causally independent. Although it was easy to design (it actually re-uses most of the code of CMB-DA), the cost at barrier synchronisation in a NOW environment could be enormous. However, it is possible for some models that simulation work can outweigh synchronisation overhead (even if the synchronisations are frequent): more

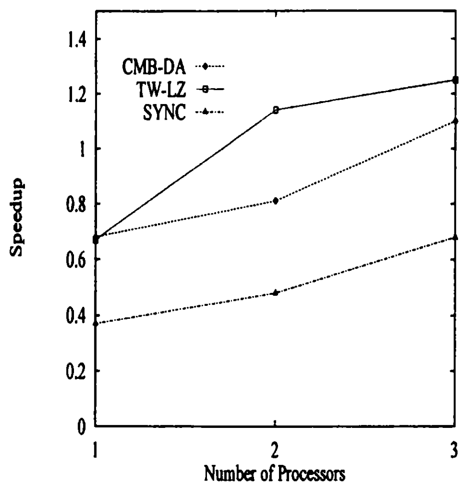


Figure 7.4: Speedup of SYNC, TW-LZ and CMB-DA over SEQ (W=10,000).

(useful) work can be done between synchronisations, and synchronisation overhead is amortised over many target processors.

In terms of difficulty of implementation, we can say that the synchronous simulator has been the simplest to program and debug. The conservative one needed considerably more development time, but it was done in first place, so it served to acquire most of the experience used with the others. The optimistic has been the most difficult to program, and the hardest to debug.

Chapter 8

Case Studies and Performance Results

In the previous chapters we have presented three distributed simulators able to simulate ECATNet models. This chapter presents experiments that have been executed with those simulators, with the aim of characterising how the parameters of the ECATNet model, the synchronisation strategy and the ways of organising the simulator influence the achieved performance. An analysis of the results allows us to suggest suitable combinations of algorithms to efficiently carry out simulations of ECATNet models.

8.1 Introduction

In this chapter we present the results obtained after performing a set of experiments with the distributed simulators and the sequential one, whose purpose was to get an insight into their behaviour under different conditions: parameters of the ECATNet models and ways of organising the LPs. We are more interested in the behaviour of the simulators than in getting useful informations about the model. What we want is to show how some approaches to DDES are effectively useful for the evaluation of systems modelled by ECATNets.

The presentation of the experiments is organised according to the synchronisation mechanism: first CMB-DA, then TW-LZ and finally SYNC. An analysis of the performance of each simulator is done after showing the obtained results and a series of overall conclusions is given.

This chapter is structured as follows: the models under study and the rules we follow during the experiments are presented in §8.2. The results of the experiments with the different models are discussed respectively in §8.3 (producer consumer model), §8.4 (manufacturing system model) and §8.5 (pipeline model). Conclusions are finally summarised in §8.6.

8.2 Models under study

We carry out a set of experiments using the following ECATNet models:

1. Producer Consumer Model (4 LPs model);
2. Manufacturing System Model (8 LPs model);
3. Pipeline Model (16 LPs model).

The following rules are applied to simulate these models:

1. the partitioning has to be related to the firing rule: a LP should be a set of transitions along with their input places such that local information is sufficient to decide upon the enabling and firing of any transition. This is in order to reduce the number of messages exchanged between LPs and minimises overhead (Efficient Partitioning);
2. when a transition is enabled, it fires in a three phase firing;
3. the execution time of the sequential event-driven simulator running with the same set of model parameters will serve as the reference point for the computation of speedup values;
4. synthetic workload: the efforts for real event processing work in the distributed simulators must exceed the communication effort in order to achieve speedup. To do so, the load is (hypothetically) increased on the processors ($W = 1$, 10, ..., 10000);
5. the obtained performance results are represented in the form of collections of execution time and speedup curves. LPs profiles according to the simulation engine are also presented. The execution times represented in Tables and the LPs profiles (distribution of total execution time among simulation and synchronisation) are with no synthetic workload whereas the speedup figures are represented with a large value of W (10,000) if not stated otherwise;
6. mapping: each LP is assigned to a processor and resides there until the end of simulation;

7. the simulation is run for long duration cycles to prevent the performance results to be dependent on initial conditions and to make them representative of the real system.

After showing the experimental results, we proceed to analyse the effect that each parameter of the model or of the simulator has in the execution time. When convenient, several parameters are grouped and studied together. It is clear that the following parameters of the ECATNet model have a significant impact on the execution time of the simulation: (1) the size of the model; (2) the structure of the model; and (3) the scenario of the simulation.

8.3 Producer Consumer Model

Many distributed systems are constructed as producer consumer systems, where producer processes generate some data and consumer processes use the generated data for further computation. As the producer and consumer processes may proceed at different rates, applications use either an unlimited-size or fixed-size buffer for temporary storage of the data. Instead of sending data directly to the consumer, the producer deposits it in the buffer; similarly the consumer process requests data from the buffer rather than directly from the producer process. The buffer must ensure that the data is sent to the consumer in the same order that it is received from the producer. Also, a producer process is blocked if the buffer is full; a consumer is blocked if the buffer is empty.

Figure 8.1 describes an ECATNet model of a producer consumer system. The model parameters include produce-rate, consume-rate, and buffer-size, where the first two parameters represent the mean rates at which the producer and consumer processes generate data and the third parameter is the size of the buffer. Produce-rate and consume-rate are represented by timed transitions **Produce** in LP_1 and **Consume** in LP_4 respectively. We suppose the remaining transitions **Send** in LP_2 and **Receive** in LP3 are immediate. The buffer is modelled by place **Buffer** in LP_3 (with possible limited capacity). In the following we study two applications for temporary storage of data: in the first one the buffer's size is unlimited whereas in the second one we make it fixed.

8.3.1 Unbounding the Buffer's Capacity

The ECATNet model described in Figure 8.1 is a set of four subnets. The buffer, which is unbounded, is represented with place **buffer** in LP_3 with an unlimited

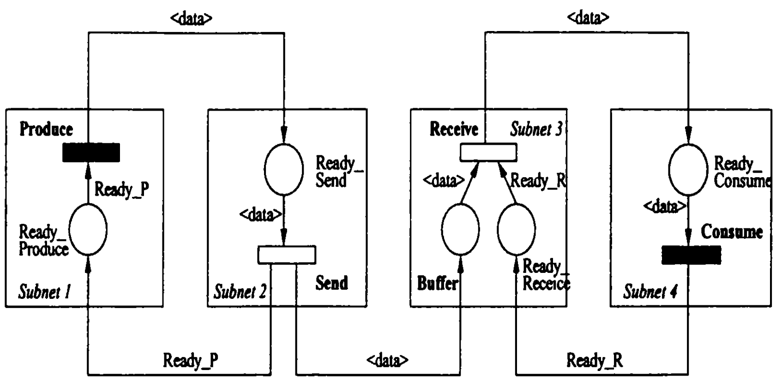


Figure 8.1: ECATNet Producer Consumer Model.

PEs	SEQ	CMB-DA	TW-LZ	SYNC
1	8 secs	5 mins 47 secs	4 mins 27 secs	40 mins 21 secs
2	-	3 mins 08 secs	2 mins 16 secs	22 mins 07 secs
4	-	2 mins 33 secs	1 min 48 secs	16 mins 03 secs

Table 8.1: Producer Consumer Model (Unbounded Buffer's Capacity). Execution Times of the Different Simulators.

capacity. The rewrite rule associated with the transitions are (in the model, IC = DT):

Produce$_{LP_1}$: (Ready_Produce$_{LP_1}$,Ready_P) $\rightarrow$ (Ready_Send$_{LP_2}$,<data>)

Send$_{LP_2}$: (Ready_Send$_{LP_2}$,<data>) $\rightarrow$ (Ready_Produce$_{LP_1}$,Ready_P) $\otimes$ (Buffer$_{LP_3}$,-<data>)

Receive$_{LP_3}$: (Buffer$_{LP_3}$,<data>) $\otimes$ (Ready_Receive$_{LP_3}$,Ready_R) $\rightarrow$ (Ready_Consume$_{LP_4}$,-<data>)

Consume$_{LP_4}$: (Ready_Consume$_{LP_4}$,<data>) $\rightarrow$ (Ready_Receive$_{LP_3}$,Ready_R)

The scenario of the simulation is as follows: duration = 16,000 cycles; the transition firing times are exponentially distributed with mean 2.0 (produce-rate) and 1.0 (consume-rate). The ECATNet model is initially marked M(Ready_Produce) = Ready_P in LP$_1$, and M(Ready_Receive) = Ready_R in LP$_3$.

In the simulation, a total of 32352 transitions have been fired. The execution times of the different simulators are shown in Table 8.1 and Figure 8.2. It is worth mentioning that: (1) each of LP$_1$, LP$_2$ and LP$_4$ has a single IQ; (2) No LP generates **End_firing** internal events in EVL.

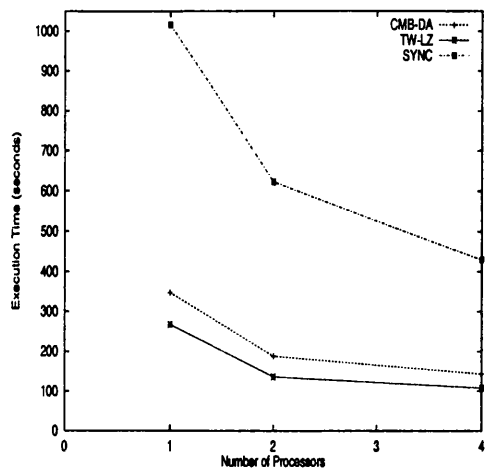


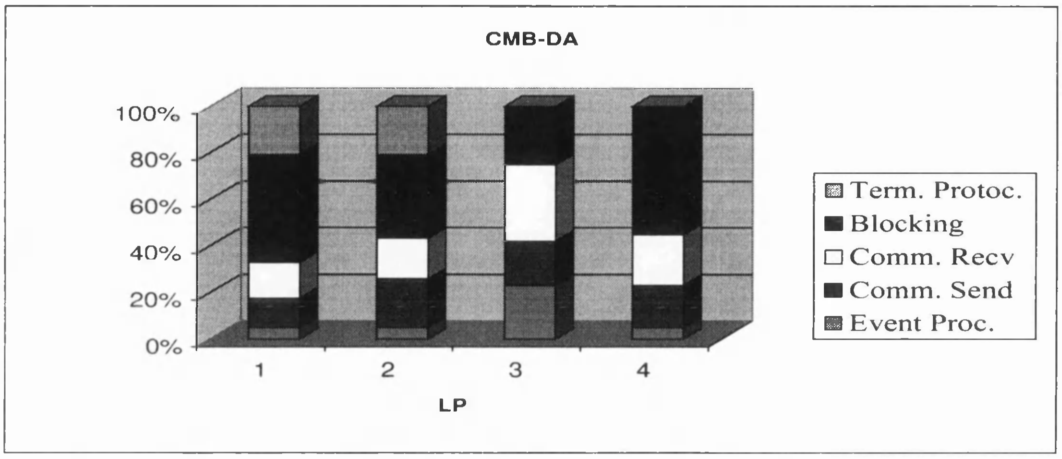
Figure 8.2: Producer Consumer Model (Unbounded Buffer's Capacity). Execution Times of the Different Simulators.

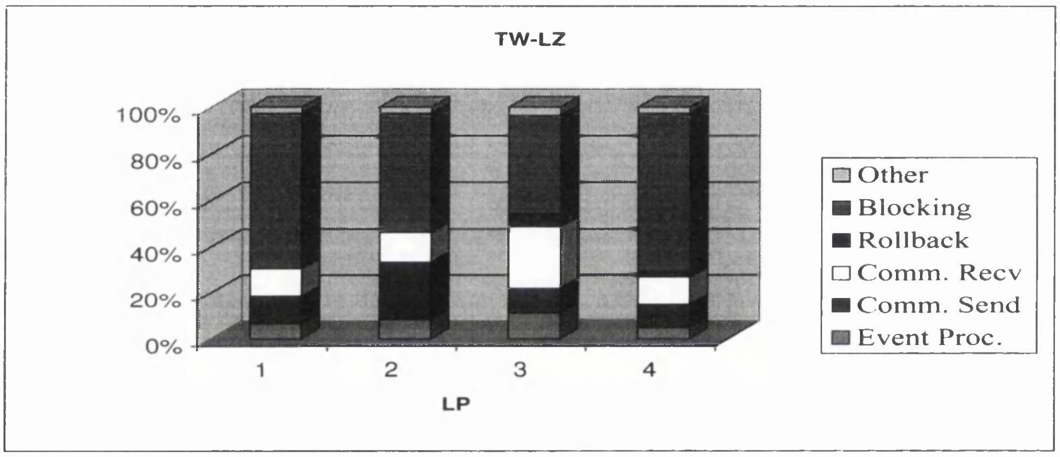
Results of CMB-DA

From the structure of the model, we see that LP_1 sends a **Create_tokens**(<data>) message to LP_2 after transition **Produce** fires (say with timestamp TT). In turn LP_2 sends a **Create_tokens**(Ready_P) message to LP_1 and a **Create_tokens**(<data>) message to LP_3 after transition **Send** fires with the same timestamp (TT) because this transition is immediate.

According to the algorithm described in §5.6.3, a transition $t \notin OT_i$ generates a null message after computing lookahead. As null messages cannot be generated because no transitions of this type exist in the model, we faced a deadlock situation mainly because of LP_3 which has to check the channel clocks CC_2 and CC_4. We modified the algorithm so that LP_3 sends a null message to LP_4 (say with timestamp CC_2), which when received by LP_4 will advance its timestamp by $\lambda_{Consume}$. This advance increases the message-acceptance horizon and thus may allow any awaiting **Create_tokens** message in LP_3 to be consumed. In this case null messages are exchanged between LP_3 and LP_4 only.

From the LPs profiles in Figure 8.3 we can see that LP_1 and LP_2 terminate the simulation earlier than LP_3 and LP_4: both LPs wait 34% of the total execution time at the synchronisation barrier before terminating the simulation (using 4 PEs). This is perfectly understandable because as long as the size of the buffer is not fixed, LP_1 and LP_2 do not "block" (especially LP_2 before depositing the data in the buffer), thus work "faster" than LP_3 (and consequently than LP_4 as well) which has to





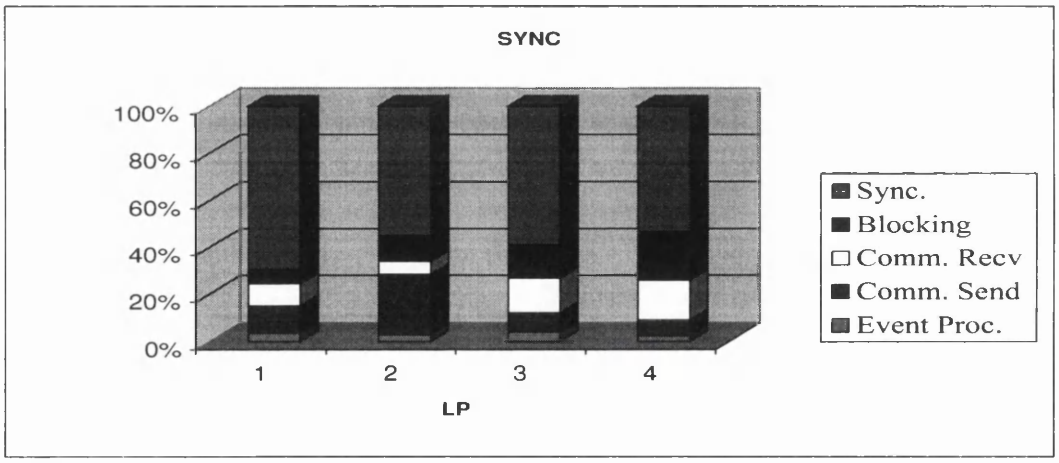


Figure 8.3: Producer Consumer Model (Unbounded Buffer's Capacity). Execution Profiles (4 PEs).

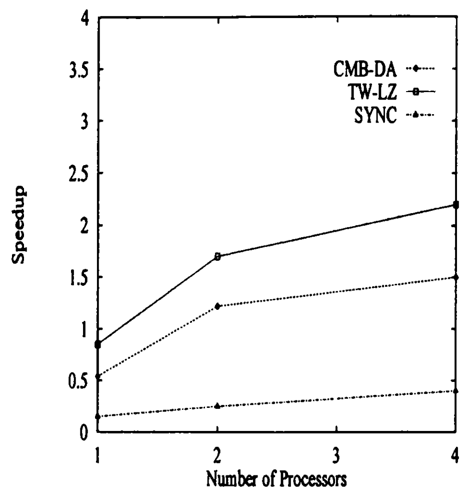


Figure 8.4: Producer Consumer Model (Unbounded Buffer's Capacity). Speedup of the Different Simulators over SEQ.

manage the reception of messages in two input queues (IQ_2 and IQ_4).

The behaviour of CMB-DA depends mainly on the way LPs synchronise in the simulator. The number of "useful" messages (i.e., positive messages) managed by the simulator is affected by the size of the model, its structure and the scenario of the simulation. An increment in the number of these messages means that the LPs have more opportunities to synchronise, while doing useful computation. Null messages are needed less often, as LPs do not block frequently. We can say that there is a high degree of "natural" synchronisation. However, when there are only a few useful messages to process, LPs block often, and null messages are needed to maintain the LPs' clocks updated. Then LPs spend most of their (real) time blocked or processing null messages, i.e., synchronising, instead of making progress.

Results of TW-LZ

LP_1 and LP_2 never rollback: the timestamp of each **Create_tokens**(Ready_P) sent from LP_2 to LP_1 is equal to the timestamp of the **Create_tokens**(<data>) previously sent from LP_1 to LP_2 as transition **Send** is immediate. LP_3 is the only LP to receive staggler messages from LP_2 and LP_4, which could lead to a rollback in LP_4 as well (Figure 8.3).

Regarding the effect of the synthetic workload on the performance of TW, a large value of *W* may have an important effect on the speedup. In the CMB-DA simulator, the busy-wait loop which emulates a high workload is done just whenever it is needed, and never has to be undone. On the other hand, with TW many jobs are processed in a speculative way, and their effect might be undone in the future. This means that, when the workload is high, the effect of erroneous computations is also a serious drawback: it could be better to wait and consume a message just when it has to be consumed, than to consume it and later undo its effects.

Results of SYNC

SYNC showed very poor performance for the following reasons. Firstly, each LP can expect executing a **Start_firing** internal event or a **Create_tokens** message making the density of events low. Secondly, if several transitions fire *at the same time*, this does not necessarily mean that this occurs *at the same synchronisation step*. To highlight this, suppose that LP2 receives a **Create_tokens**(<data>) message with timestamp TT from LP_1 after transition **Produce** fires. If *global clock* is now equal to TT at synchronisation $step_i$ and M(Ready_Receive) = Ready_R, transition **Send** in LP_2 fires first at TT. At $step_{i+1}$, transitions **Produce** in LP_1 and **Receive** in LP_3 fire at TT in parallel. Then at $step_{i+2}$, transition **Consume** fires at TT as well. Thirdly, in addition to the excessive amount of time the LPs spend synchronising to calculate and broadcast the new value of the *global clock*, additional overhead is introduced by **Create_tokens**(t,p,TT,*number*) messages. In order to avoid deadlock situations, at each step of the simulation each LP informs its neighbours about how many positive messages will be sent to them via **Create_tokens**(t,p,TT,*number*) messages, and actually sends them via **Create_tokens** (t,p,TT,CT). As the protocol is synchronous, at least one **Create_tokens**(t,p,TT,*number*) message is sent by a LP to its neighbours after each synchronisation step even when there are no "positive" messages to follow at all.

TW-LZ performed better than CMB-DA: a speedup of 2.2 and 1.5 using 4 PEs was observed respectively for these simulators (Figure 8.4). SYNC showed very poor performance.

8.3.2 Bounding the Buffer's Capacity

In case the buffer is bounded, it is represented with place **buffer** in LP_3 with a limited capacity. The conditional rewrite rule associated with transition **Send** in LP_2 is now:

Send_{LP_2}: (Ready_Send_{LP_2},<data>) $\rightarrow$ ($\text{Ready\_Produce}_{LP_1}$,Ready_P) $\otimes$ (Buffer_{LP_3},-<data>) **if** M(Buffer_{LP_3}) $\oplus$ <data> $\leq$ C(Buffer_{LP_3})

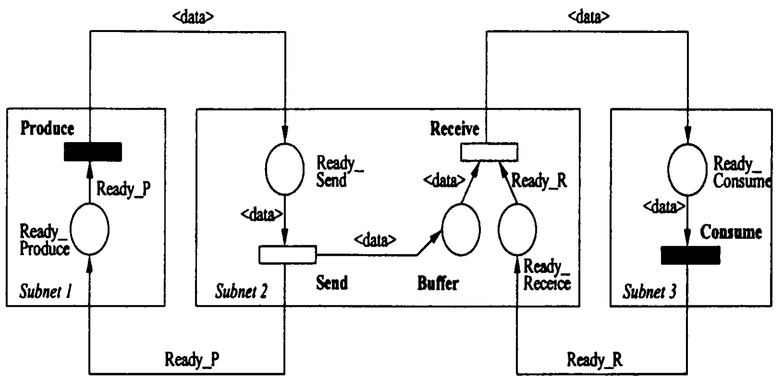


Figure 8.5: ECATNet Producer Consumer Model (P3 Partitoning).

Partitioning Performance Impact

In addition to the experiments carried out with the partitioning into four subnets (P4) described in the previous section (§8.3), we wanted to test the impact that a different alternative of partitioning (and therefore of grain size) has on the performance of the simulators, simply by assigning all conflicting transitions to the same subnet, not only with their input places, but with their limited capacity ouput places as well. The partitioning in Figure 8.5 is now a set of three subnets (P3) where LP_2 generates an **End_firing** internal event in EVL after firing transition **Send.**

A transition t_i in subnet$_i$ may have one or more output places with limited capacity in different LPs. Whether this transition is enabled will depend not only on the marking in all its input places but in all its output places (with limited capacity) as well. Assigning a transition and its output places with limited capacity to the same subnet prevents LPs from exchanging synchronisation additional messages because the enabling conditions can be tested locally: local information is sufficient to decide upon the enabling and firing of any transition.

Four experiments were performed with different simulation engines and the parameters used in the experiments are summarised in table 8.2. The simulation has been run with two different scenarios. In scenario 1 (**S1**), we make the produce-rate < consume-rate; produce-rate = 1.0, consume-rate = 3.0. In scenario 2 (**S2**), we make the produce-rate > consume-rate; produce-rate = 3.0, consume-rate = 1.0. For both scenarios, the duration of the simulation = 16,000 cycles, and the capacity of buffer is <data ⊕ data ⊕ data>. S1 makes the producer a *fast* sender which cannot continuously tansmit data faster than the consumer can absorb it, whereas S2 makes the producer a *slow* sender which continuously tansmits data slower than the consumer can absorb it.

Scenario	S1 - P3	S1 - P4	S2 - P3	S2 - P4
Number of subnets	3	4	3	4
Number of PES	1..3	1..4	1..3	1..4
Results	Fig. 8.6	Fig. 8.6	Fig. 8.9	Fig. 8.9
	Fig. 8.8	Fig. 8.8	Fig. 8.11	Fig. 8.11

Table 8.2: Producer Consumer Model (Bounded Buffer's Capacity). Experiments Performed with Scenarios **S1** and **S2**.

Results of Experiment S1

A total of 21343 transitions have been fired, and the producer had 66.6% idle time whereas the consumer 0%. The execution times of the simulators running scenario S1 are shown in Table 8.3: the three distributed simulators perform better with partition P3 than with partition P4.

Partition		P3			P4		
PEs	SEQ	CMB-DA	TW-LZ	SYNC	CMB-DA	TW-LZ	SYNC
1	5 secs	2 mins 20 secs	2 mins 51 secs	16 mins 01 secs	3 mins 51 secs	7 mins 30 secs	44 mins 48 secs
2	-	2 mins 08 secs	2 mins 14 secs	12 mins 29 secs	3 mins 21 secs	5 mins 40 secs	22 mins 42 secs
3	-	1 min 57 secs	1 min 38 secs	8 mins 04 secs	-	-	-
4	-	-	-	-	2 mins 54 secs	3 mins 18 secs	19 mins 54 secs

Table 8.3: Producer Consumer Model (Bounded Buffer's Capacity). Execution Times for Partitions P3 and P4 - Experiment S1.

CMB-DA and **SYNC**: with P4, *each time* transition **Send** in LP$_2$ has its input conditions enabled (presence of a token <data> in place Ready_Send), it sends a message to LP$_3$ to request a deposit of tokens <data> in place **Buffer** via a **Deposit_request** message (say with timestamp TT). After checking the marking of place **Buffer**, LP$_3$ replies by sending a **Deposit_request_ACK** message specifying in its timestamp the time the deposit is possible. This time depends mainly on the enabling conditions of transition **Receive** which, when it fires, destroys <data> tokens in place **Buffer**, and consequently makes a new deposit of <data> possible in this place. It is worth to mention that the communication time of LP$_2$ and LP$_3$, and the blocking time of LP$_1$ and LP$_4$ are quite important (Figure 8.7) especially for CMB-DA. With P3, the simulators do not need to use **Deposit_request** and **De-**

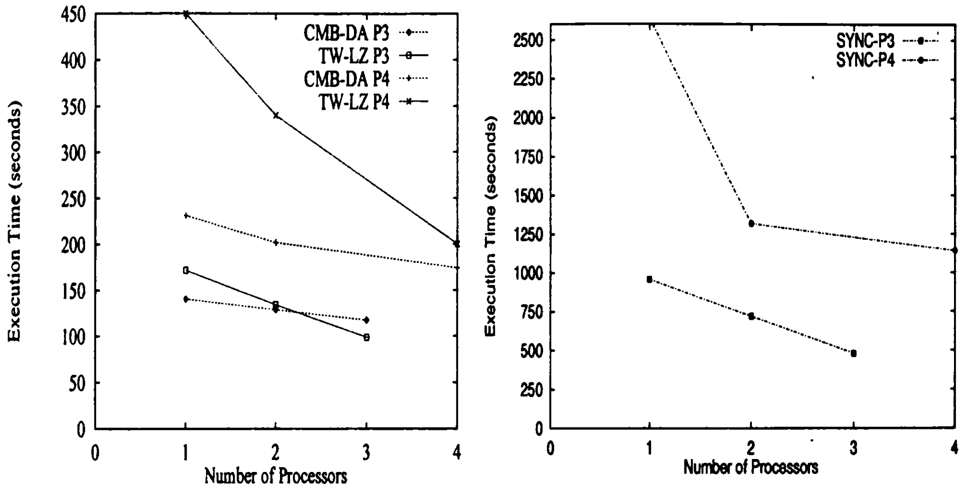


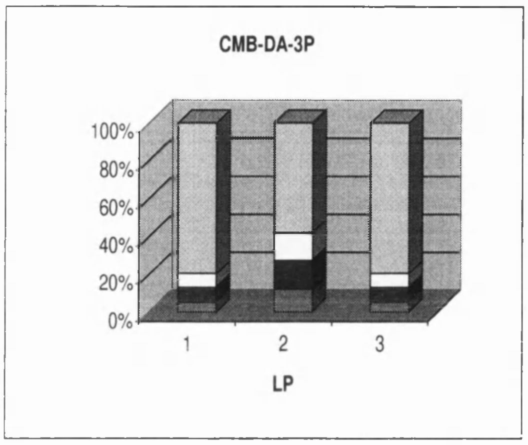
Figure 8.6: Producer Consumer Model (Bounded Buffer's Capacity). Execution Times for Partitions P3 and P4 - Experiment S1.

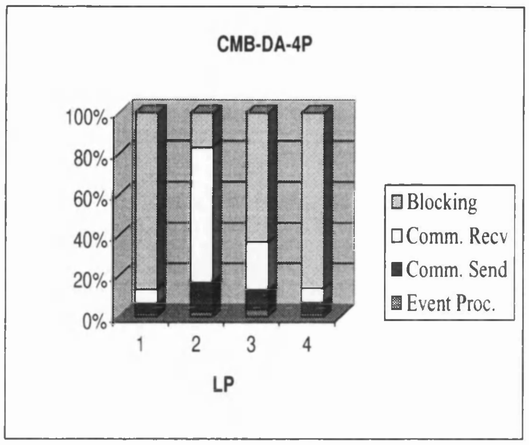
posit_request_ACK messages because the information concerning place **Buffer**'s marking is available locally in LP_2. Thus the overhead is substantially reduced. This also results in an important event processing time in this LP.

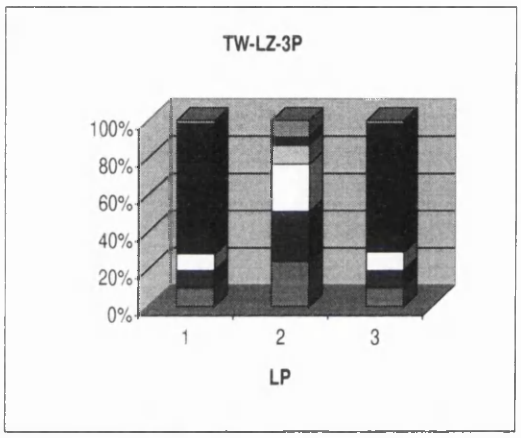
TW-LZ: with P4, as the protocol is optimistic, transition **Send** fires as soon as its input conditions are enabled: LP_2 does not really care about the actual marking of place **Buffer** in LP_3 and sends a **Create_tokens** message leading to a deposit of <data> token in this place. When receiving this message, LP_3 checks the marking of place **Buffer**. If the deposit does not result in a capacity overflow, the message is accepted and the deposit is performed. But in case a deposit of <data> exceeds what **Buffer** can handle, LP_3 will send a negative message asking LP_2 to rollback because an incorrect computation has been done. In this message, LP_3 will specify the time firing of place **Send** can take place (say TT)[1]. A rollback in LP_2 may result in a rollback in LP_1 as well (Figure 8.7). LP_2 resumes its execution by sending a new **Create_tokens** message (at TT). With P3, LP_2 is prevented from receiving a **Create_tokens** message which could result in an overflow of place **Buffer**'s capacity leading to a rollback in LP_2 because the information concerning place **Buffer**'s marking is available locally. This again results in a reduction of overhead.

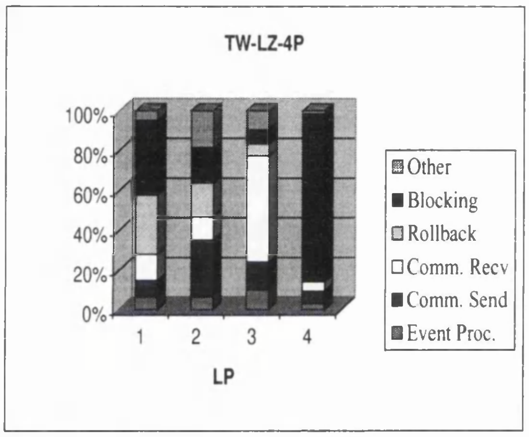
Although the frequent sending of **Deposit_request** and **Deposit_request_ACK**

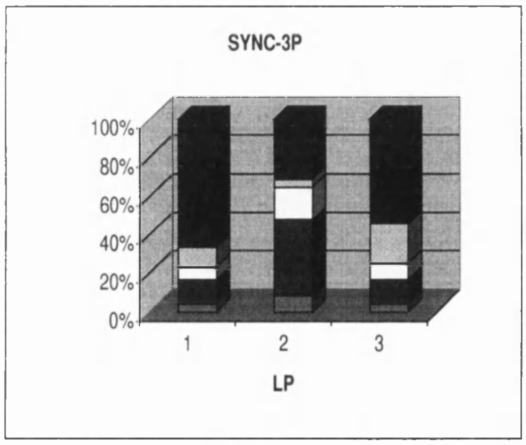
[1]This situation is similar to a flow control needed to force a sender to stop frequently to give the receiver a chance "to breathe".











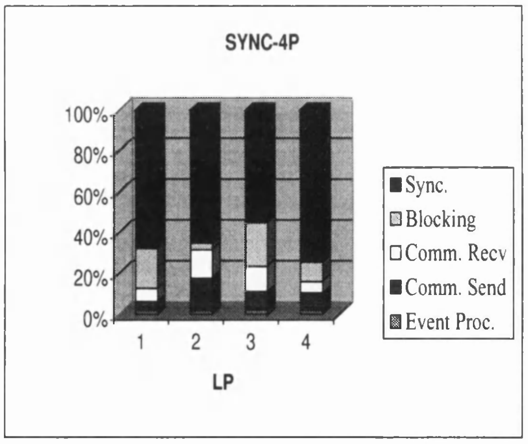


Figure 8.7: Producer Consumer Model (Bounded Buffer's Capacity). Execution Profiles for Partitions P3 and P4 - Experiment S1.

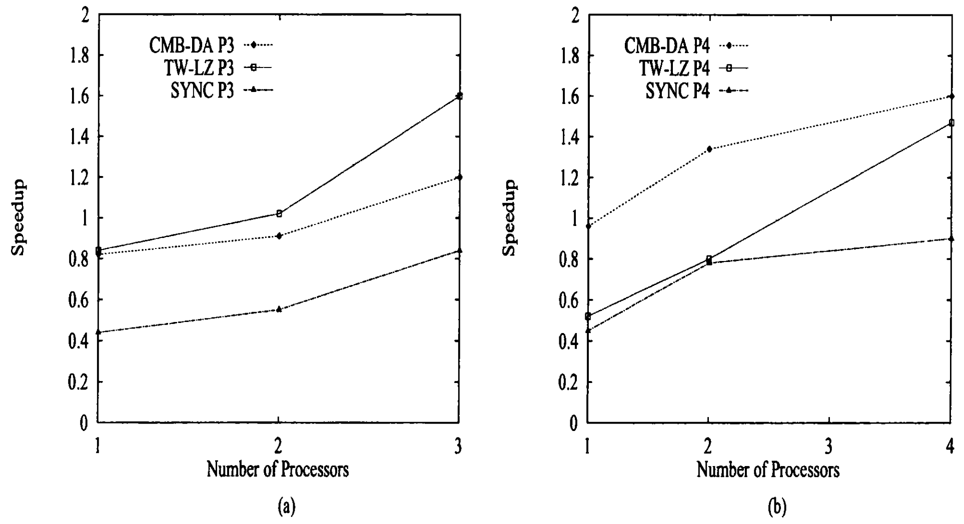


Figure 8.8: Producer Consumer Model (Bounded Buffer's Capacity). Speedup (a) Partition P3; (b) Partition P4 - Experiment S1 (W=10,000).

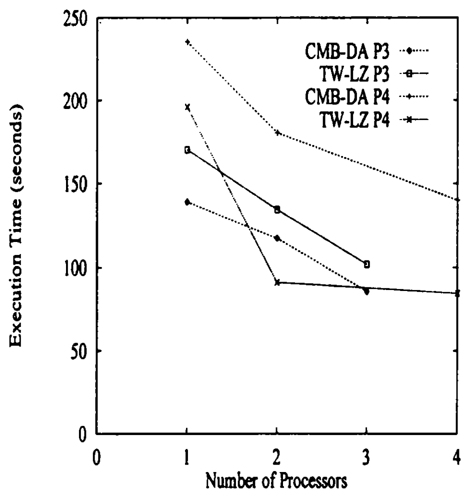
messages using partition P4, CMB-DA performed better than TW-LZ which had to manage the cascaded rollbacks in LP_2 and LP_1. A speedup of 1.6 and 1.5 using 4 PEs was observed respectively for these simulators (Figure 8.8(a)). However, TW-LZ performed better using partition P3 (Figure 8.8(b)) where a speedup of 1.6 and 1.2 using 3 PEs was observed respectively for CMB-DA and TW-LZ. SYNC showed poor performance.

Results of Experiment S2

A total of 21333 transitions have been fired, and the consumer had 66.6% idle time whereas the producer 0%. The execution times of the simulators running scenario S2 are shown in Table 8.4 and Figure 8.9: the simulators still perform better with partition P3 than with partition P4, except for TW-LZ.

Partition		**P3**			**P4**		
PEs	SEQ	CMB-DA	TW-LZ	SYNC	CMB-DA	TW-LZ	SYNC
1	5 secs	2 mins 19 secs	2 mins 50 secs	13 mins 55 secs	3 mins 55 secs	3 mins 16 secs	49 mins 40 secs
2	-	1 min 57 secs	2 mins 14 secs	11 mins 09 secs	3 mins 01 secs	1 min 30 secs	28 mins 21 secs
3	-	1 min 25 secs	1 min 41 secs	7 mins 43 secs	-	-	-
4	-	-	-	-	2 mins 20 secs	1 min 24 secs	26 mins 17 secs

Table 8.4: Producer Consumer Model (Bounded Buffer's Capacity). Execution Times for Partitions P3 and P4 - Experiment S2.



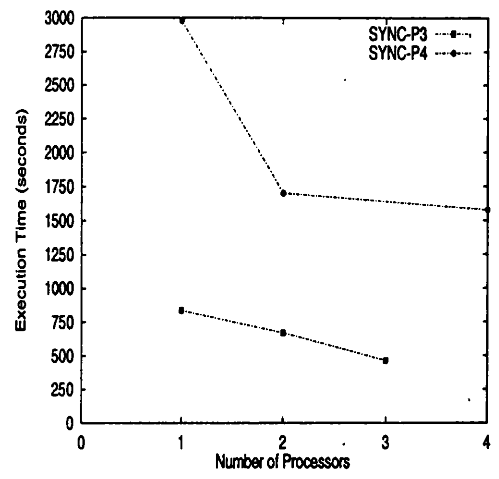


Figure 8.9: Producer Consumer Model (Bounded Buffer's Capacity). Execution Times for Partitions P3 and P4 - Experiment S2.

CMB-DA and **SYNC**: with P4 and as for S1, the same observations are made. *Each time* transition **Send** in LP_2 has its input conditions enabled (presence of a token <data> in place Ready_Send), it sends a **Deposit_request** message to LP_3 (say with timestamp TT). However, with scenario S2 LP_3 always replies via a **Deposit_request_ACK** message specifying that the deposit is possible at TT because *no tokens* are available in place **Buffer**. It is worth to mention that the communication time of LP_2 and LP_3, and the blocking time of LP_1 and LP_4 are quite important (Figure 8.10) especially for CMB-DA.

TW-LZ: with P4 we did not expect any rollbacks in LP_3 with scenario S2, but were not surprised to see that they do occur. The reason is that LP_3 keeps accepting **Create_tokens**(<data>) messages from LP_2 which cause an overflow in place **Buffer** because of a "late" reception of **Create_tokens**(Ready_R) messages from LP_4 (deposit in place **Ready_Receive**). The arrival of these straggler messages not only prevents transition **Receive** to fire "on time", but causes rollbacks. LP_3 sends a negative message asking LP_2 to rollback as well because of incorrect computation and specifies the time firing of transition **Send** can take place (say TT). A rollback in LP_2 may result in a rollback in LP_1 as well (Figure 8.10). LP_2 resumes its execution by sending a new **Create_tokens**(<data>) message (at TT).

With partition P4, TW-LZ performed better than CMB-DA which had to man-

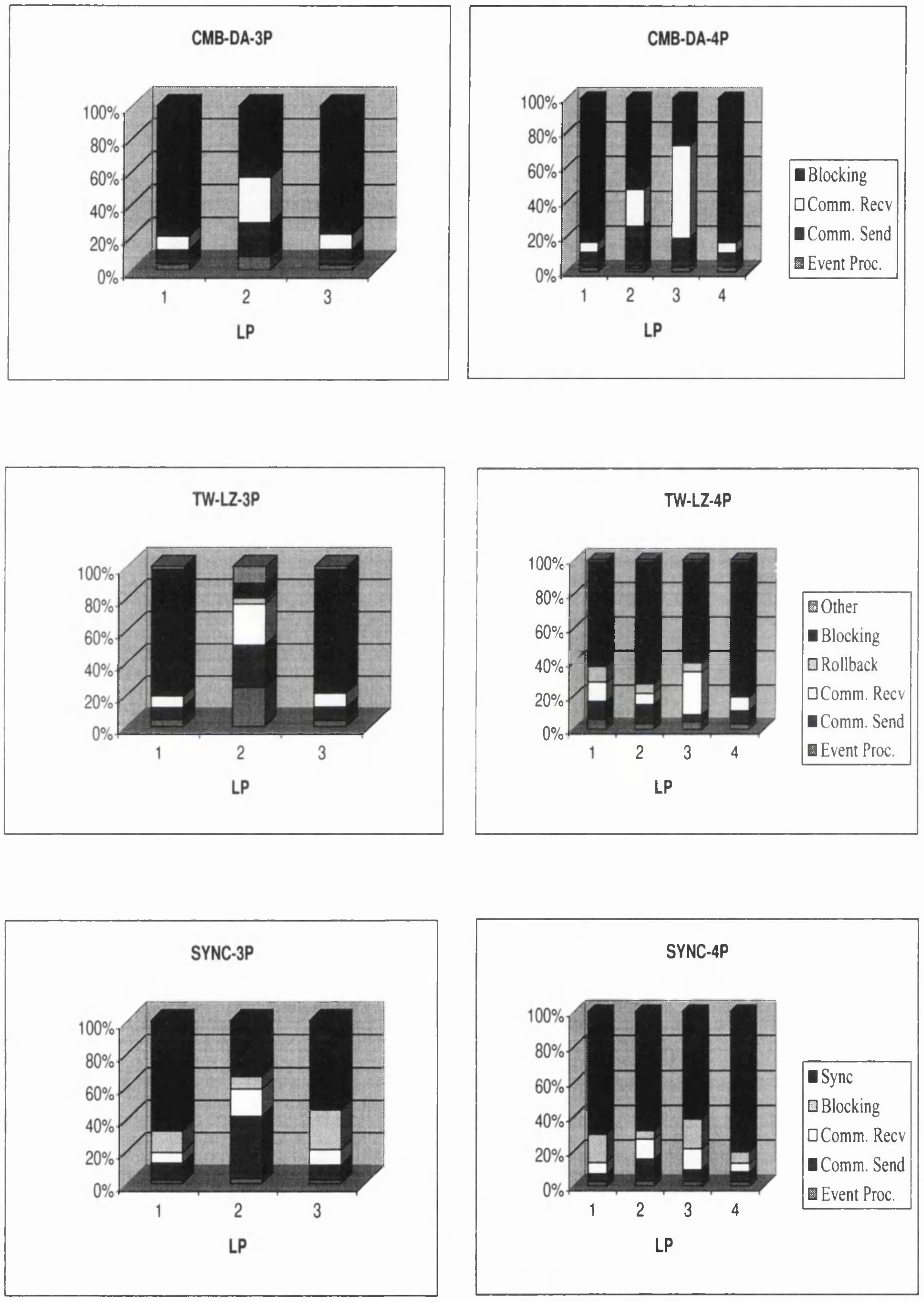


Figure 8.10: Producer Consumer Model (Bounded Buffer's Capacity). Execution Profiles for Partitions P3 and P4 - Experiment S2.

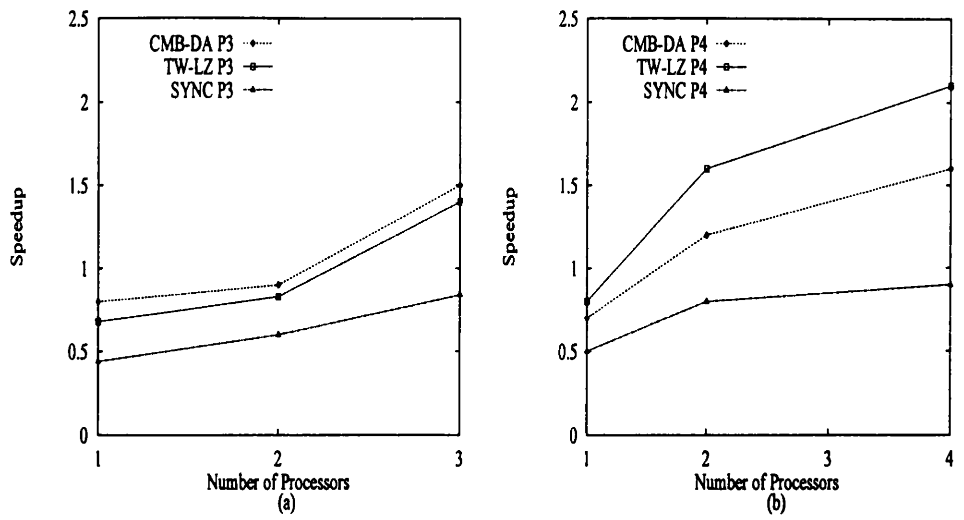


Figure 8.11: Producer Consumer Model (Bounded Buffer's Capacity). Speedup (a) Partition P3; (b) Partition P4 - Experiment S2 (W=10,000).

age the frequent sending of **Deposit_request** messages by LP_2 and the reply to these messages of LP_3 via **Deposit_request_ACK** messages, even when the responses were always positive, i.e., the firing of transition **Send** always took place at the time specified by LP_2 in the request. A speedup of 2 and 2.1 using 4 PEs was observed respectively for TW-LZ and CMB-DA respectively (Figure 8.11(a)). However, CMB-DA performed better than TW-LZ using partition P3 (Figure 8.11(b)) where a speedup of 1.5 and 1.4 using 3 PEs was observed. SYNC showed poor performance.

8.4 Manufacturing System Model

There has been a dramatic increase in the use of simulation to design and optimise manufacturing systems. One of the reasons is that the increased competition in many industries has resulted in a greater emphasis on automation to improve productivity and quality and also to reduce costs. Since automated systems are more complex, they can typically be analysed only by simulation.

The ability of ECATNet models for prototyping and analysing concurrent systems to exploit supervisor evaluation of discrete event systems is shown in [BCD98]. The recent work by Turner et al. shows how to bridge the gap between the users of industrial simulation packages and the parallel simulation community [TLL+98]. This work consists in developing a methodology for automating the parallelisation of manufacturing simulations by constructing a mapping from the sequential simulation model to an efficient parallel implementation.

We assume a *physical system* of four machines participating in a manufacturing process. In a processing step, $machine_1$ produces subpart A_1 of a product A. Subpart A_2 is produced by machines 2 and 3 (both of which can produce concurrently). Once one subpart A_1 and one subpart A_2 are assembled, $machine_4$ produces subpart A_3. One piece of A is an assembly of A_1-A_2-A_3, and all subparts A_1, A_2 and A_3 require a single amount of assembly steps.

The system is modelled in terms of ECATNets (Figure 8.12):

LP_1: *source* process, generates the production orders;
LP_2: $machine_1$ produces subpart A_1
LP_3: *fork* process
LP_4: $machine_2$ produces subpart A_2
LP_5: $machine_3$ produces subpart A_2
LP_6: *join* process
LP_7: $machine_4$ produces subpart A_3
LP_8: *sink* process.

Transition $SendM_1$ in LP_1 models the generation of the orders for processing. Transition $ProdA_1$ in LP_2, $ProdA_2$ in LP_4, $ProdA_2$ in LP_5 and $ProdA_3$ in LP_7 model the processing steps of parts A_1, A_2, A_2 and A_3 by machines 1, 2, 3 and 4 respectively. Machines in the processing phase are represented by algebraic terms m_1, m_2, m_3 and m_4 in place p_2 in LP_2, LP_4, LP_5 and LP_7 respectively, finished parts A_1 by CT to be deposited in place ReadyF in LP_3, finished parts A_1A_2 by CT to be deposited in place ReadyJ in LP_6, and the final product $A_1A_2A_3$ by CT to be deposited in place Collect in LP_8 (assuming this latter has an infinite capacity).

The rewrite rules associated with the model are:

SendM1_{LP_1}: $(\text{ReadyS}_{LP_1}, \text{s}) \rightarrow (\text{ReadyS}_{LP_1}, \text{s}) \otimes (\text{Q1}_{LP_2}, \text{s})$
ProdA1_{LP_2}: $(\text{Q1}_{LP_2}, \text{s}) \otimes (\text{M1}_{LP_2}, \text{m1}) \rightarrow (\text{M1}_{LP_2}, \text{m1}) \otimes (\text{ReadyF}_{LP_3}, \text{A1})$
SendM2_{LP_3}: $(\text{ReadyF}_{LP_3}, \text{A1}) \rightarrow (\text{Q2}_{LP_4}, \text{A1})$
SendM3_{LP_3}: $(\text{ReadyF}_{LP_3}, \text{A1}) \rightarrow (\text{Q3}_{LP_5}, \text{A1})$
ProdA2_{LP_4}: $(\text{Q2}_{LP_4}, \text{A1}) \otimes (\text{M2}_{LP_4}, \text{m2}) \rightarrow (\text{M2}_{LP_4}, \text{m2}) \otimes (\text{ReadyJ}_{LP_6}, \text{A1A2})$
ProdA2_{LP_5}: $(\text{Q3}_{LP_5}, \text{A1}) \otimes (\text{M3}_{LP_5}, \text{m3}) \rightarrow (\text{M3}_{LP_5}, \text{m3}) \otimes (\text{ReadyJ}_{LP_6}, \text{A1A2})$

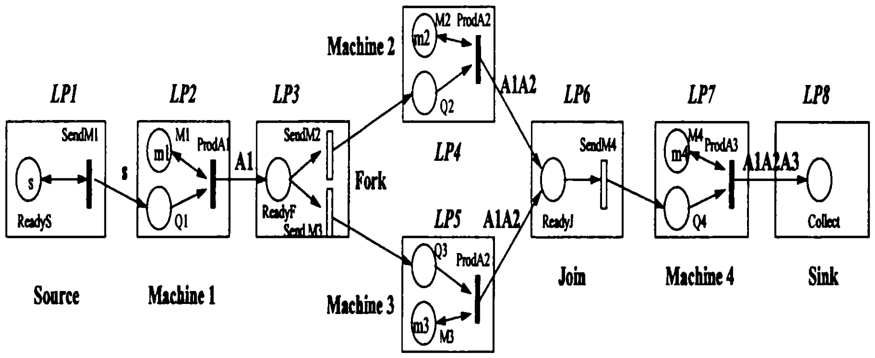


Figure 8.12: ECATNet Manufacturing System Model.

SendM4$_{LP_6}$: (ReadyJ$_{LP_6}$, A1A2) $\rightarrow$ (Q4$_{LP_7}$, A1A2)
ProdA3$_{LP_7}$: (Q4$_{LP_7}$, A1) $\otimes$ (M4$_{LP_7}$, m4) $\rightarrow$ (M4$_{LP_7}$, m4) $\otimes$ (Collect$_{LP_8}$, A1A2A3)

The time behaviour of the system is modelled by associating timing information to transitions:

λ_{SendM1} = 1.0 (in LP$_1$); λ_{ProdA1} = 1.0 (in LP$_2$); λ_{ProdA2} = 3.0 (in LP$_4$); λ_{ProdA2} = 0.5 (in LP$_5$); λ_{ProdA3} = 0.5 (in LP$_7$). The remaining transitions are immediate. In LP$_3$, if transitions SendM2 and SendM3 cannot fire in parallel, the decision place ReadyF in LP$_3$ selects randomly a transition to fire with equal probability 0.5.

The initial state of the system is represented by the marking of the ECATNet where there is a "signal" (order) term in ReadyS (in LP$_1$), and machines M1, M2, M3 and M4 are in state "idle" (terms m1, m2, m3 and m4 in places M1, M2, M3 and M4 in LP$_2$, LP$_4$, LP$_5$ and LP$_7$ respectively).

8.4.1 Results of the Experiments

A total of 95894 transitions have been fired, and 15963 pieces of product A have been produced. The execution times of the simulators are shown in Table 8.5 and Figure 8.13.

Results of CMB-DA

LP$_3$, LP$_4$, LP$_5$, LP$_6$ are the only LPs to use null messages. From the structure of the model, we see that each LP has a single IQ, except LP$_6$ which has to manage its two

PEs	SEQ	CMB-DA	TW-LZ	SYNC
1	27 secs	13 mins 27 secs	13 mins 19 secs	2 hrs 14 mins 04 secs
2	-	8 mins 01 secs	7 mins 34 secs	1 hr 16 mins 18 secs
4	-	4 mins 51 secs	4 mins 17 secs	1 hr 4 mins 17 secs
8	-	3 mins 30 secs	2 mins 19 secs	32 mins 54 secs

Table 8.5: Manufacturing System Model. Execution Times of the Different Simulators.

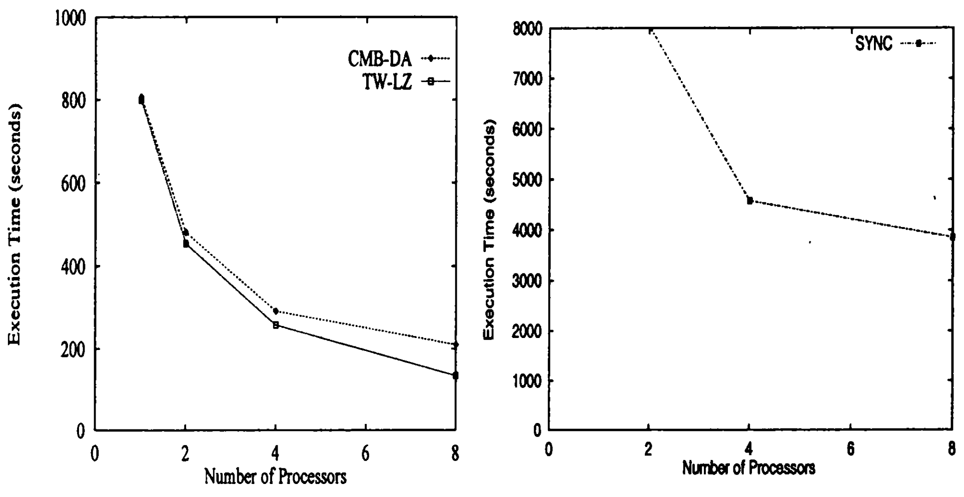
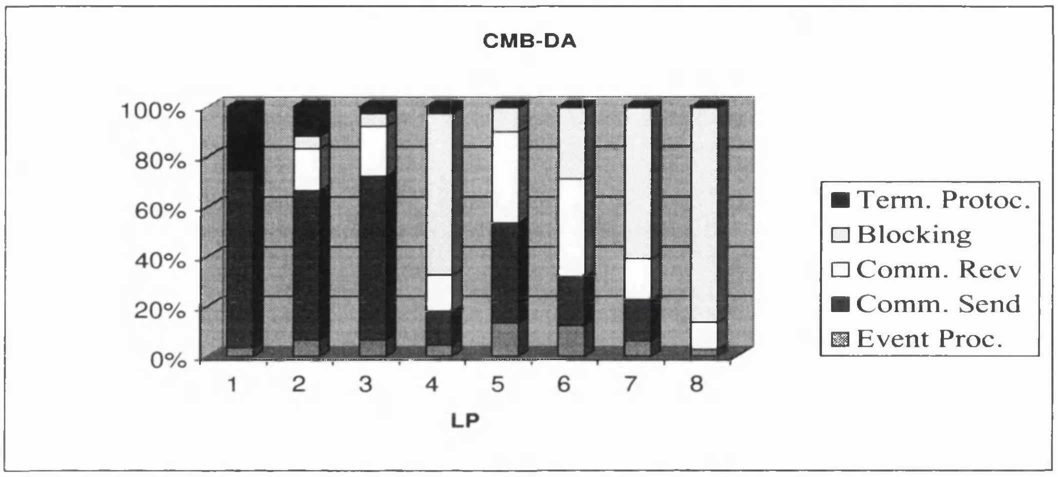
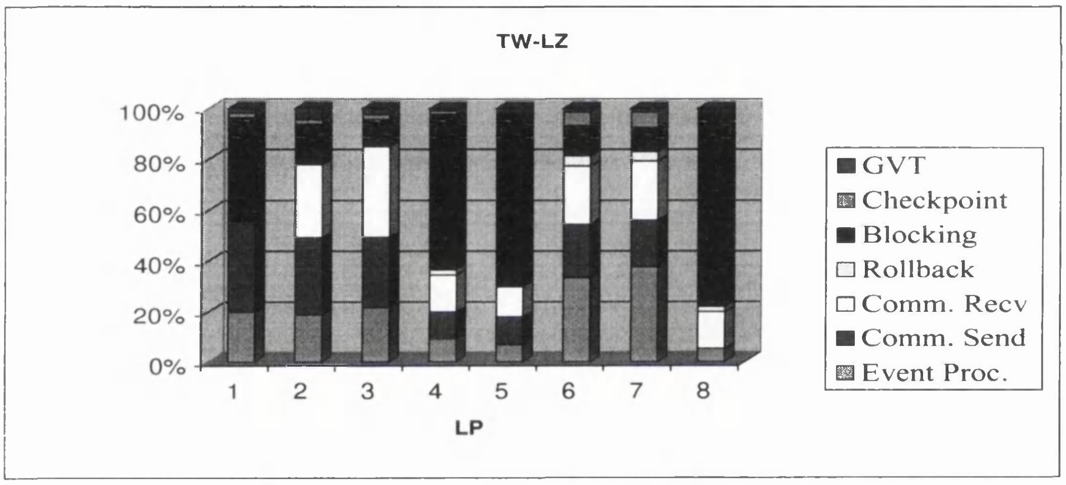


Figure 8.13: Manufacturing System Model. Execution Times of the Different Simulators.

channel clocks CC_4 and CC_5. As for the producer consumer model, in the absence of a transition in LP_4 and LP_5 to generate a null message and increase lookahead, LP_6 often blocks, resulting in the blocking of LP_7 and LP_8 as well. To remedy this, LP_3 has to send a null message to LP_5 every time transition SendM2 fires, and a null message to LP_4 every time transition SendM3 fires with timestamp equal to LVT_3 (transitions SendM2 and SendM3 are actually immediate). These null messages, when received by LP_4 and LP_5, will be forwarded to LP_6 after increasing lookahead by λ(PrA2) respectively in LP_4 and LP_5. When received by LP_6, they break the deadlock and allow any awaiting **Create_tokens** messages (either in IQ_4 or IQ_5) to be consumed. It is worth to mention that from the structure of the model, LP_1 does not receive any incoming messages. Consequently it is the fastest LP and waits 25% of the total execution time at the synchronisation barrier before terminating





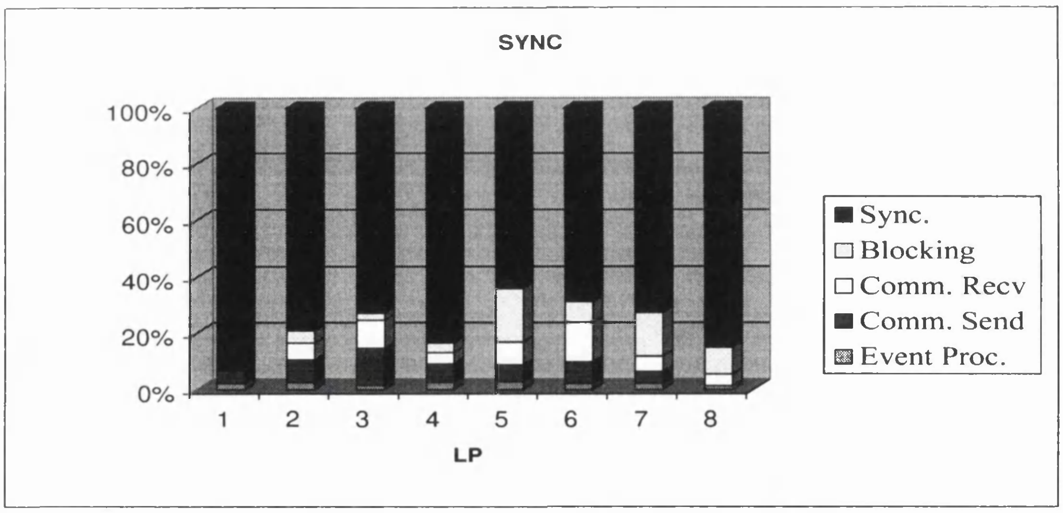


Figure 8.14: Manufacturing System Model. Execution Profiles (8 PEs).

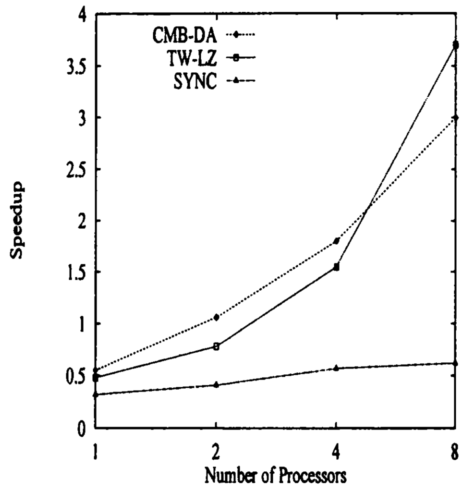


Figure 8.15: Manufacturing System Model. Speedup of CMB-DA, TW-LZ and SYNC (W=10,000).

the simulation (Figure 8.14).

Results of TW-LZ

LP_4 rolls back frequently after reception of **Create_tokens**(A1) positive straggler messages from LP_3 whose effect is a deposit of tokens in place Q2. After rolling back and restoring a correct state, no antimessages are transmitted because the previously sent messages to LP_6 are re-generated. However, this is not the case for LP_6 which receives **Create_tokens** positive straggler messages from LP_4 and LP_5. This may result in sending antimessages to LP_7, and consequently in cascaded rollbacks in LP_7 and LP_8 (Figure 8.14).

Also, LP_1 works faster than the other LPs (2..8). It is also responsible for GVT calculation (coordinator). We faced the situation where LP_1 was ready to send a new GVT_PACKET message but did not receive the acknowledgement of the previous one, thus was not able to send a GVT message (§6.3.5). The addition of a certain degree of "conservatism" into LP_1 which advanced unboundedly imposed a limitation to its ability to go into the future. LP_1 was allowed to advance to a certain degree, calculated as the value of GVT plus a *time window size* (the time interval between two GVT times computations). LP_1 would temporarily block if it tried to go beyond. LP_1 resumed execution after receiving a GVT_PACKET message acknowledgement.

Results of SYNC

SYNC exhibited poor performance mainly due to the time LPs spend synchronising (Figure 8.14). We observed a maximum of five events processed in parallel by the LPs.

TW-LZ performed better than CMB-DA and SYNC as shown in Figure 8.15 where a speedup of 3.7 using 8 PEs was observed.

8.5 Pipeline Model

In the ECATNet model of Figure 8.16, each LP_i models a processing element in a pipeline system. The element waits for one input from its *previous* neighbour LP_{i-1}, performs a computation, then sends the result to its *next* neighbour LP_{i+1}. The model comprises 16 LPs, and each LP has 2 places and 2 transitions. The rewrite rules associated with the model are:

$t1_{LP_i}$: $(p1_{LP_i}, x) \rightarrow (p2_{LP_i}, y)$
$t2_{LP_i}$: $(p1_{LP_i}, \emptyset) \otimes (p2_{LP_i}, y) \rightarrow (p1_{LP_{i+1}}, x)$ /* $(p1_{LP_i}, \emptyset)$ is equivalent to the inhibitor arc concept */

We ran the simulation with the following scenario: simulation duration = 16,000 cycles; $\lambda(t1) = 0.0$, and $\lambda(t2) = 1.0$, and $M(p_1) = x$. A total of 512032 transitions have been fired.

8.5.1 Experiment With Different Grain Sizes

Mapping the ECATNet model onto the NOW requires two steps: mapping subnets onto LPs, and mapping LPs onto PEs. There are two trivial possibilities:

1. map each subnet onto a single LP, and then map each LP onto a single PE as we did for the previous models. We say that the grain size of the LP is *minimum*. A good deal of interprocess communication is needed, because **Create_tokens** external events need to be sent as messages;

2. map several subnets onto one LP, and then each LP onto a different PE. In this case, many of the **Create_tokens** events are internal and the interprocess communication is significantly reduced.

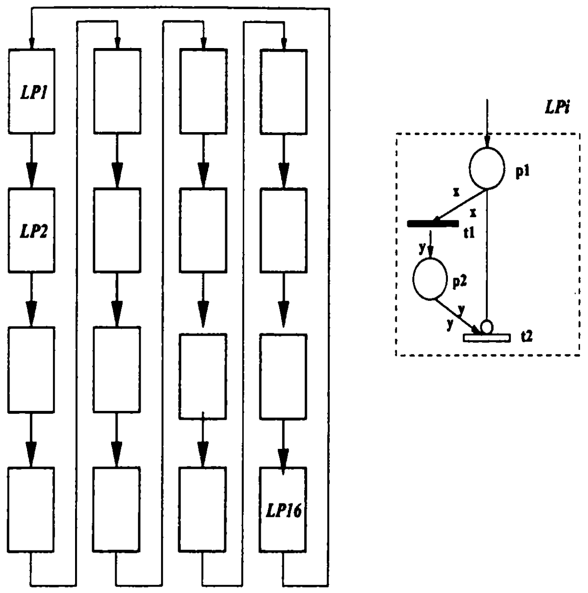


Figure 8.16: ECATNet Pipeline Model (16 Subregions).

We redesigned the LPs to allow the simulation of square of subnets while the version previously used (Figure 8.16) was only designed for minimum grain size. It is worth mentioning that in the case of a *maximum* grain size, any of the distributed simulators (CMB-DA, TW and SYNC) running onto one PE behaves exactly like the sequential simulator, except that the LP (simulating a single partition P1) sends **Create_tokens** messages (from LP_16 to LP_1 in Figure 8.16) to itself rather than inserting them in EVL.

Additionally, we wanted to test the impact of the different alternatives of grain size have on the performance of the simulator. Various alternatives of grain size are possible. Figure 8.17a represents the mapping for minimum grain size (partition P16). Figures 8.17b, c and d represent a mapping for *intermediate* grain size leading to 8, 4 and 2 LPs respectively (partitions P8, P4 and P2). Note that in partition P2 a subnet simulated by a LP is 8 times larger than that used in partition P16. The execution times of the simulators for various partitionings are shown in Table 8.6 and Figure 8.18.

Experiments with partitions P16 and P8 are cases with workload W = 10,000 whereas P4 and P2 are cases without synthetic workload. We do use synthetic workload when experimenting with partitions P16 and P8, but not with P4 and P2: the way of giving more work to a LP is mapping onto it a larger number of subnets. In

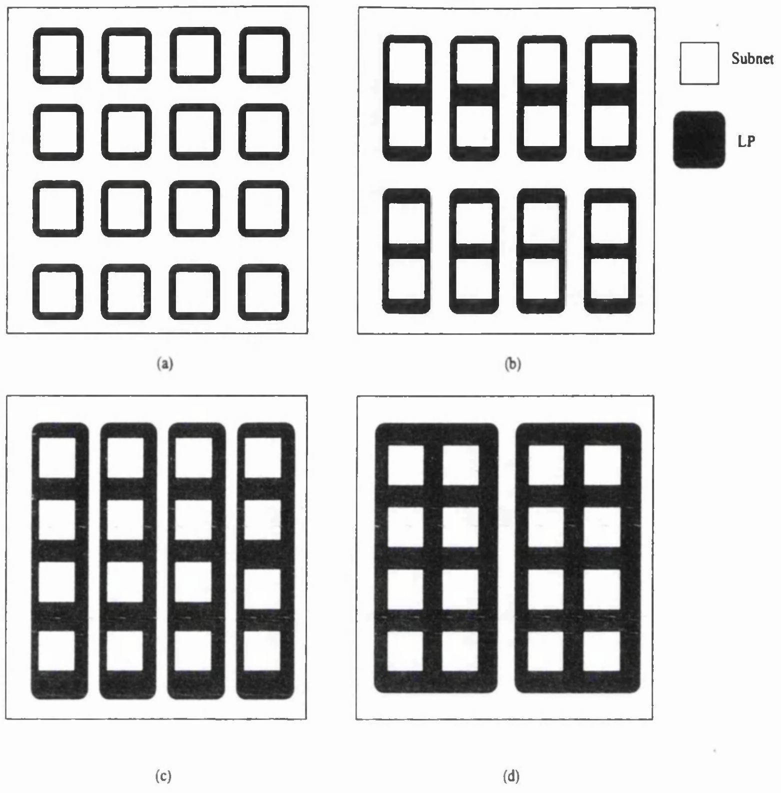


Figure 8.17: Mapping the Pipeline Model onto LPs. (a) Minimum Grain Size (16 LPs). (b) (c) (d) Intermediate Grain Size, respectively 8, 4, 2 LPs.

models with larger grain size (P4 and P2), the computation/communication ratio is more balanced this way, and the local simulation work in terms of physical processor cycles exceeds the computation/communication threshold.

8.5.2 Discussion

Results of CMB-DA

The possible imposition of extra overhead by the use of null messages in distributed simulation has caused much criticism on the useful of such approaches. When analysing LP_i in partition P16, we see that (1) it has a single input queue where messages received from LP_{i-1} are stored; (2) when the immediate transition t_1 fires, a null message with timestamp $Tsim + \lambda(t_1) + \lambda(t_2)$ is generated. As transition t_1 is immediate, this timestamp is equal to the timestamp of the positive **Create_tokens** message generated when transition t_2 fires. Thus there is no need to generate null

Partition	SEQ	CMB-DA	TW-LZ	SYNC
P16 (Min, 16 PEs)	-	2 mins 32 secs	3 mins 02 secs	20 mins 16 secs
P8 (Int, 8 PEs)	-	2 mins 21 secs	2 mins 54 secs	19 mins 42 secs
P4 (Int, 4 PEs)	-	2 mins 32 secs	3 mins 53 secs	11 mins 30 secs
P2 (Int, 2 PEs)	-	3 mins 39 secs	4 mins 58 secs	8 mins 23 secs
P1 (Max, 1LP)	6 mins 02 secs	7 mns 40 secs	10 mns 28 secs	8 mns 41 secs

Table 8.6: Pipeline Model. Execution Times for Various Partitionings.

messages and consequently the execution time for the CMB-DA simulator by reducing them improved by 40%.

From the experiments performed with different partitionings, the performance of CMB-DA increases with the grain size of the LPs. This way, the computation/ communication ratio is more balanced. If, looking at the results of the experiments, we compare maximum vs. minimum grain size, we can see that corse grain simulation is more effective that fine grain simulation because a small number of LPs synchronising results in lower overhead. If now we compare with the intermediate grain size alternatives, it is clear that these have the best perfomance (P8 and P4). From these experiments, we learned that it is good to have models with large grain size, to balance the computation/communication ratio of the PEs.

Results of TW-LZ

Preliminary tests done with TW-LZ showed good performance figures when partitioning with P16 and P8, and no rollbacks were observed. This is understandable because as long as the accumulated firing times of transitions within each LP is (λ(t1) + λ(t2) = 1.0) and ((λ(t1) + λ(t2))*2 = 2.0) respectively, there is no possibility of receiving a straggler message. But the reason CMB-DA performed better was because of state saving, antimessages management and GVT computation in TW-LZ.

However, rollbacks were observed when partitioning with P4 and P2. We verified that the big size of the state to save after firing a transition was also partially responsible for the poor results of TW-LZ: the state of a LP is the combination of all the state variables which represent the ECATNet subnet assigned to that LP, plus a number of variables for statistics gathering. Also, with P4 and P2, the density of events is, in general, very high and, for this reason, the probability of a straggler to appear is very high too. This leads to continuous rollbacks in the LPs. It has

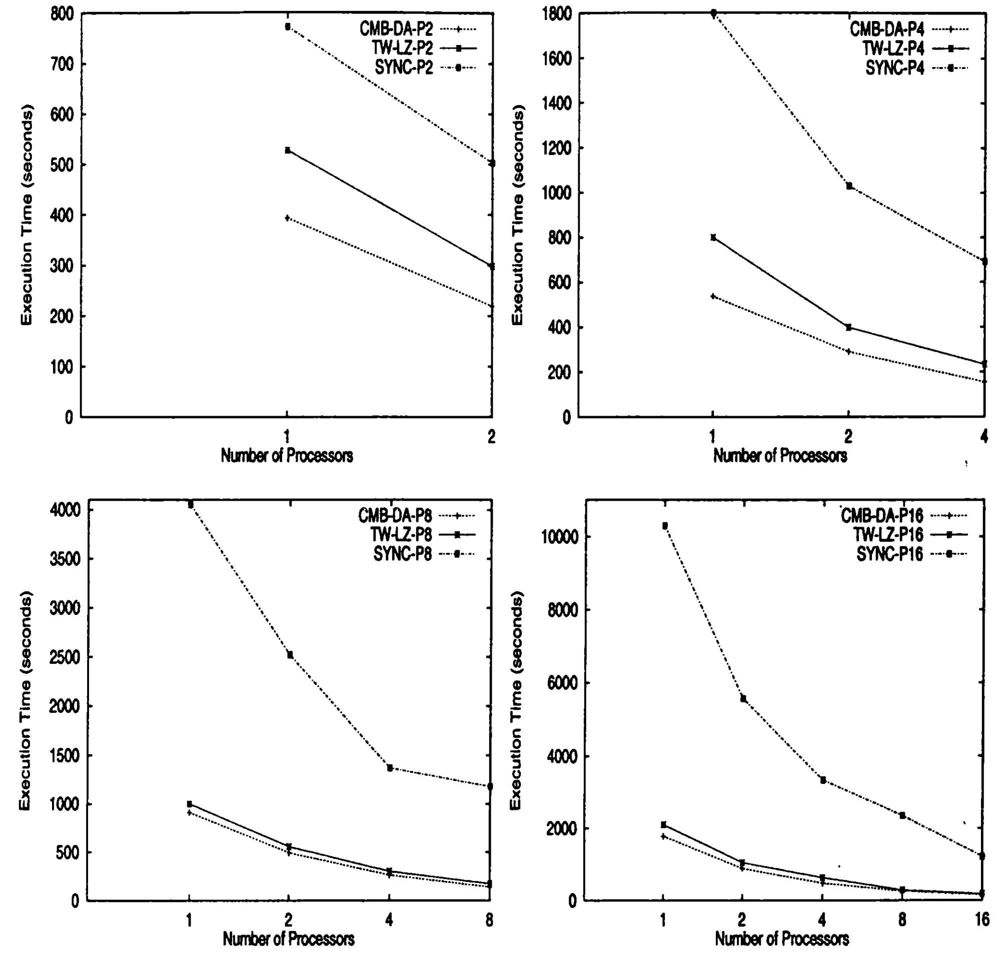
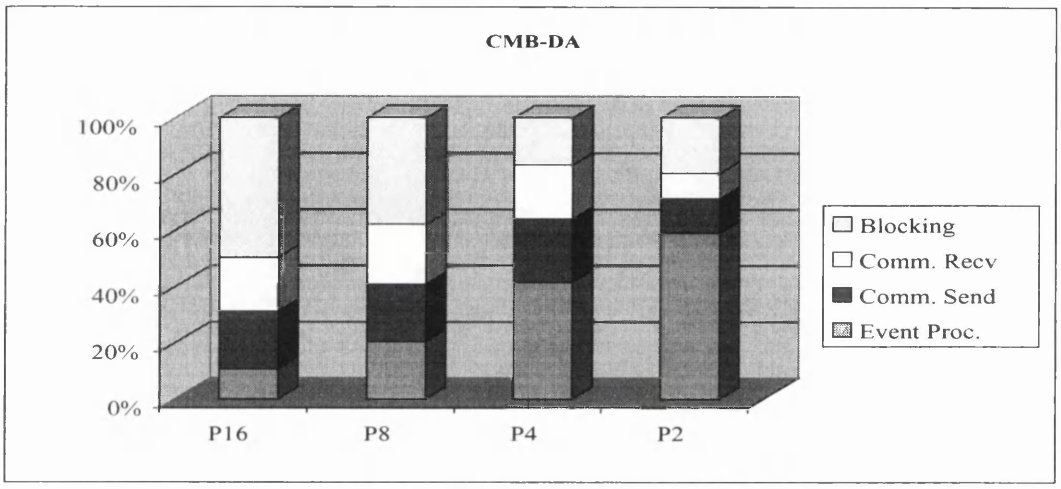
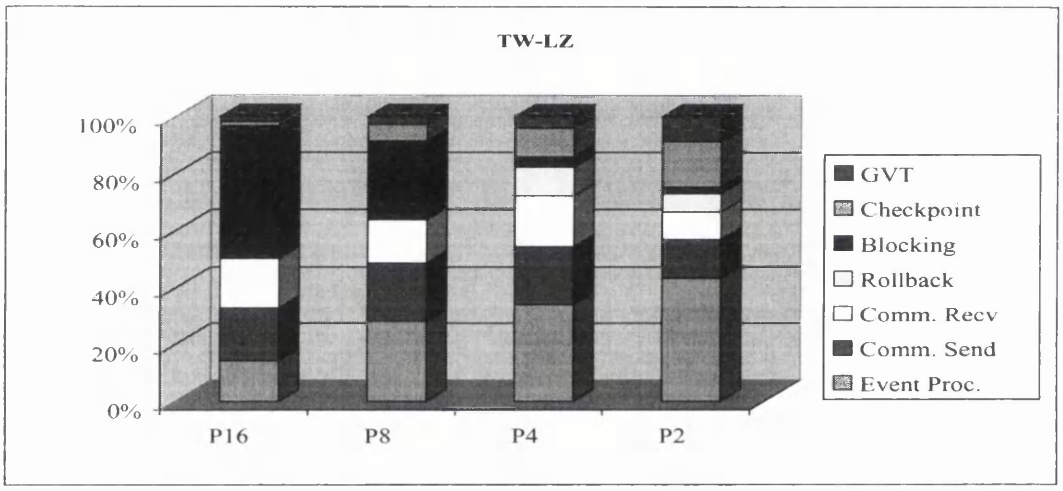


Figure 8.18: Pipeline Model. Experiments with Various Partitions. Execution Times.

been monitored that almost all the events executed in a speculative way have to be undone. The result is that the simulator looses most of its time doing the following "management" activities:

- storing state copies and antimessages;
- traversing the input queue to locate events to annihilate, or to find the right place to insert a straggler;
- coast-forwarding to construct appropriate state versions;
- sending and managing antimessages, and





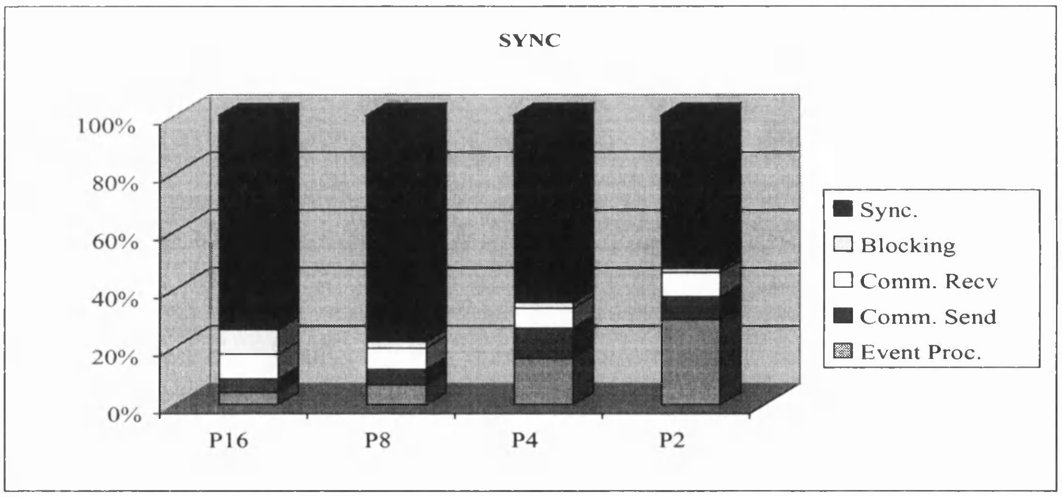


Figure 8.19: Pipeline Model: Execution Profile of LP_1 for Various Partitionings.

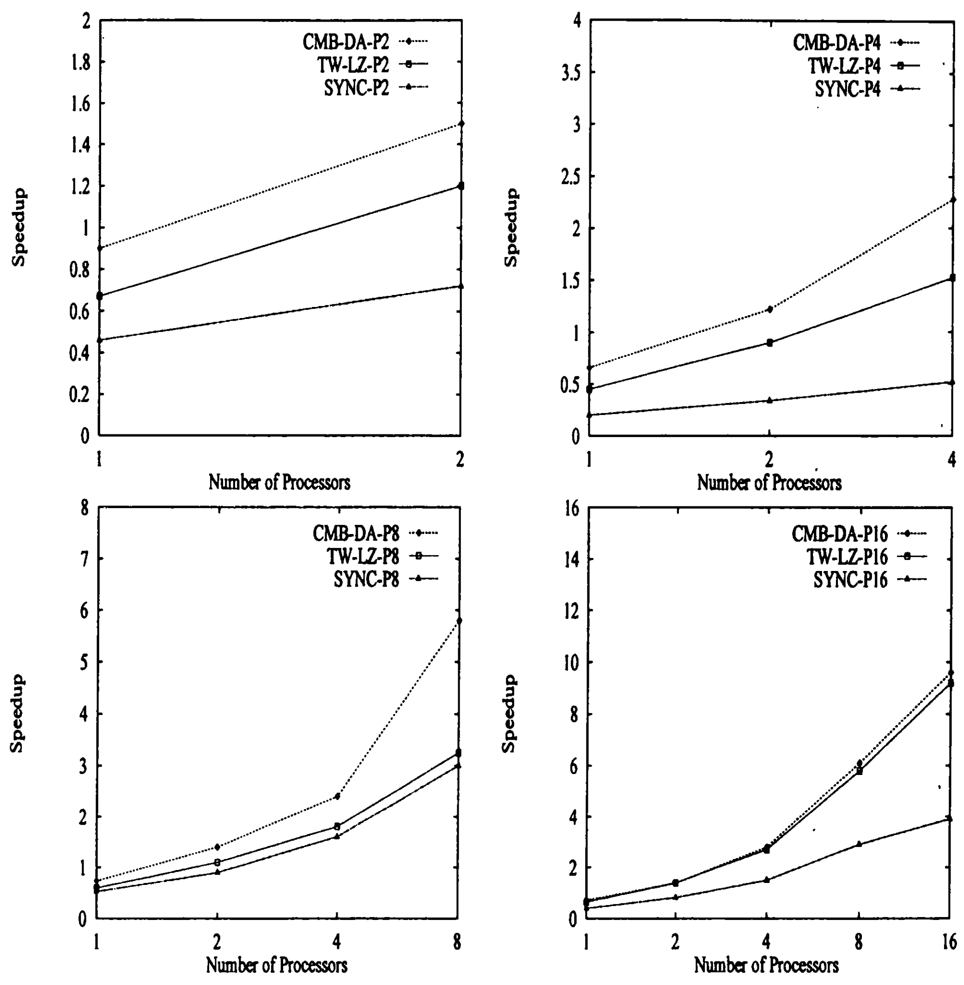


Figure 8.20: Pipeline Model. Experiment with Various Partitions. Speedup (W=10,000 for P16 and P8, No Synthetic Workload for P4 and P2).

- computing GVT.

For ECATNet applications, the cost of executing an event is very low, while the state size could be huge. Even when incremental state saving is used the amount of data moved for state saving is several times larger than the data moved to execute an event.

Results of SYNC

According to the ECATNet structure and to the scenario of the simulation, the set of LPs do the same work at each step of the simulation. Also, SYNC simply avoids sending/receiving **Create_tokens** (t,p,TT,*number*) messages when no external events are generated. This minimises the number of messages exchanged between LPs which is a critical factor for the simulation of complex systems and exploits the net structure to obtain efficiency.

As mentioned previously, decreasing the number of LPs imposes a higher workload to the LPs. With partition P2, the ECATNet submodels have the largest size. This results in a workload large enough to keep all the LPs busy, performing useful computation most of the time and minimising the relative effort devoted to synchronisation. If compared with performances of P16, P8 and P4, the performance of the simulation with P2 is improved by reducing communication time and waiting time at synchronisation points.

The number of barriers executed by the LPs was observed to be 16,000, i.e., as many barriers as simulation cycles. The reason is simple: the density of events **Start_firing**, **End_firing** and **Create_tokens** is big enough to always have at least one event per cycle. Thus, with this scenario SYNC behaves as a time-driven algorithm.

SYNC behaves very much like CMB-DA, with only a significant difference: it still takes the LPs a huge amount of time to perform reduction and broadcast operations in a NOW, sooner degrading its performance. About the effective simulation time of SYNC compared with CMB-DA, we can say that the difference is only due to the way both programs are instrumented. The synchronous nature of SYNC allows for a more precise measure of the time spent in the different activities a LP performs at each iteration. In CMB-DA the measurements are taken in an event-by-event basis, which means a significantly higher number of calls to the function that gives the real time clock value, and this affects the achieved accuracy.

In all the experiments with various partitionings (P16, P8, P4 and P2), CMB-DA performed better than TW-LZ and SYNC as shown in Figure 8.20 where a speedup of 1.5, 2.3, 5.8 and 9.1 using 2, 4, 8 and 16 PEs respectively was observed. For SYNC, it is worth to mention the speedup observed with partitions P16 and P8 (3.9 and 3 using 16 and 8 PEs respectively).

8.6 Conclusion

In this chapter we have presented our experiences using three distributed discrete event simulation strategies to study three systems modelled by ECATNet models:

a producer consumer system, a manufacturing system and a pipeline system. The knowledge obtained from previous experiments with the Ethernet transmitting station model has been applied, when possible, to the simulation of these models.

In general, we have confirmed our preliminary ideas about which characteristics of the simulated ECATNet model help to improve the performance of the distributed simulators: partitioning, structure of the net and scenario of the simulation.

1 Partitioning has the strongest influence on the performance of the simulators. The initial ECATNet model has to be partitioned efficiently into subnets for two reasons:

- to avoid the overhead due to the distributed conflict resolution algorithm in CMB-DA, TW-LZ and SYNC by assigning the set of conflicting transitions together with their input places to the same subnet;
- for models with places with limited capacity, to avoid (a) the overhead due the synchronisation via **Deposit_request** and **Deposit_request_ACK** messages in CMB-DA and SYNC; and (b) possible rollbacks in TW-LZ.

2 In order to take advantage of using CMB-DA, we need ECATNet models with a high degree of internal communication, which allows the processes to remain synchronised without sending null messages.

3 The knowledge of the behaviour of the ECATNet model may allow CMB-DA to exploit some lookahead information, which helps maintaining a good performance when the simulator has not useful work to do. The experiments performed with the three case studies show that the structure of the ECATNet model must be exploited so that a LP sends null messages *only when they are needed.*

4 It is important to use intermediate and corse grain processes, that is, to assign a significant amount of work to each process. This way, less processes are used to run the model, and the synchronisation effort is reduced. We have seen how using intermediate grain sizes in the pipeline model always led to a good performance.

5 We can say that the experience with CMB-DA and TW-LZ has been positive, but that the experience with SYNC has been quite discouraging. In all the experiments, CMB-DA and TW-LZ performed better than SYNC. The performance of the tested SYNC implementation is very poor, even after introducing a series of optimisations. The conclusion for SYNC is that, although it exploits efficiently the parallelism inherent in ECATNet models, it is not the right tool for the kind of parallel environment used in this study. However, SYNC has the advantage of a simple implementation, and can be used in environments where CMB-DA is not applicable because of the presence of loops with zero timestamp increment. Another good characteristic of SYNC is that it is based on a very simple algorithm, without the complicated memory management strategies of TW.

6 It has been noticed that the communication demands of CMB-DA, TW-LZ and SYNC are very strong in a network of workstations environment, where comunication costs are very high compared to computation costs: a collection of workstations in an Ethernet network does not perform so well.

7 Concerning the parallel programming environment, the outstanding point is that some results are really poor, mainly due to the characteristics of the computing system used in the experiments (NOW), and the way interprocess communication is achieved. In fact, for all the experiments except the one with the pipeline model, the sequential simulator was *much faster* than the distributed simulators *without* synthetic workload. A message passing mechanism is used for synchronisation and communication, and a general purpose Ethernet network with the TCP/IP protocols over it provides the necessary connectivity. This means that communication in this environment is relatively slow, because of:

- the peak data rate of Ethernet: 10 Mb/s;
- the shared nature of Ethernet: the available data rate must be shared among all the devices connected to the network, being or not part of the simulation environment, and
- the software overhead imposed by the use of several layers of protocols (Ethernet, IP, TCP, MPI). The communication protocols used inside a parallel computer are much simpler; in particular, there are not as many layers. As layering means encapsulation, i.e., addition of control informations, its effects are worse for short messages than for long messages [2] [3].

[2]We are speaking in this context about real messages interchanged between processing elements, not about simulated messages.

[3]The actual messages managed by the distributed simulators are short: about 48 bytes.

Chapter 9

Conclusion and Further Research

9.1 Summary

Throughout this thesis we have made a study of techniques for distributed simulation of discrete event systems, with special attention to a particular kind of high-level algebraic Petri nets models: Extended Concurrent Algebraic Term Nets. We started reasoning the interest of this study:

1. the need to introduce the concept of time in ECATNets to specify practical applications;
2. the need to exploit efficiently the parallelism inherent in the models;
3. the need to turn to simulation because the formal specification of real systems modelled by ECATNets is based on implementation concepts rather than theoretical ones;
4. to accelerate simulations, and
5. the consideration of simulators as interesting applications for their implementations in network of workstations and parallel computers.

The identification of causal dependencies among events allowed the relaxation of some constraints the sequential simulators impose on the order events are simulated, in such a way that several events can be processed in parallel, after splitting the simulator into a collection of collaborating logical processes. However, a synchronisation mechanism must be added to guarantee that the collection of logical processes progresses, as a whole, in a consistent way.

We have presented three different synchronisation alternatives: *conservative*, *optimistic* and *synchronous*. CMB-DA, TW-LZ and SYNC are particular realisations of those alternatives, which have been implemented and tested, first to study a model from the domain of communication networks and to analyse the general characteristics of each simulator, and then using three models to test how well these simulators work with real world applications.

Implementations of the three simulators have been performed in a network of workstations environment. A study of the characteristics of this environment has been done, allowing us to understand how the execution times of simulations depend on different characteristics of the synchronisation algorithm, the model under study and the target multicomputer.

9.2 Contributions

We have shown how DDES has been successfully used to study a variety of real-world systems, including the study of different aspects of parallel computing. Our contribution in this field is the use of DDES to analyse the behaviour of Extended Concurrent Algebraic Term Nets when used to model any existing or hypothetical system. The contributions of this work can be summarised in the following points:

1. The introduction of time to ECATNets. These nets enriched with temporal specification are suitable to discrete simulation, thus making an important step in their quantitative performance evaluation. Timed transition ECATNet formalism provides a substantial contribution to the implementation of efficient, general purpose discrete event simulation techniques.

2. A description of a collection of alternatives for distributed simulation of discrete event systems, with an evaluation of them using ECATNet models designed as a test. Similar evaluations can be found in the literature, most of them using shared memory multiprocessors, which allowed the implementation of a variety of optimisations. However, our work has been developed in a distributed memory environment where message passing is the only means of communication and synchronisation.

3. A description of an environment for parallel programming, identifying the characteristics that have an impact on the implementation of parallel algorithms. In this context, distributed simulation algorithms can be considered as a case study on parallel programming.

4. A detailed description of ECATNet models, along with the way they can be simulated using an event-driven approach. A description of these models using a C-like language has been done, able to be used by any of the simulation engines as part of this work.

5. The implementation and analysis of three distributed simulators, using three different synchronisation mechanisms:

- CMB-DA, a conservative simulator;
- TW-LZ, an optimistic simulator, and
- SYNC, a synchronous simulator.

Additionally, a sequential simulator (SEQ) has been implemented and tested, to use the obtained results as a reference point to compute speedups.

6. The design of an optimisation on CMB-DA that allows a reduction on the number of null messages used for synchronisation. Null messages are sent only when they have a positive impact on the receiving LP. Additionally, null messages are not stored in the receiving LP: the only effect of the reception of a null message is an increment in the receiving channel's clock. With this optimisation, the synchronisation effort of the simulator is notably reduced.

7. The introduction of the concept of grain size of ECATNet LPs in the distributed simulators. For example, the synchronisation mechanism for CMB-DA requires LPs to block frequently, while they await until it is sure that advance is possible without causal risks. If only one LP is assigned to each processor, CPU power is wasted while the LP is blocked. Assigning several LPs of smaller size to each processor (which means that each LP simulates a smaller part of the system under study), the CPU can be assigned to a non blocked LP. However, LP's grain size should not be too small, because in that case the overall number of LPs would be increased, and more null messages would be needed to keep the simulator synchronised. After a

set of evaluations, it was shown that it is more advantageous to use intermediate or maximum grain size LPs instead of minimum grain size LPs.

8. The characterisation of the event density of the simulated systems as an important parameter to achieve good performance in CMB-DA simulator. ECATNet models with high event density achieve good performance, because the message interchange for event scheduling provides the LPs with the necessary sychronisation, and a few number of messages are required.

9. The characterisation that TW-LZ is a viable approach to the distributed simulation of ECATNet models. We confirm that, in different scenarios, TW can be very effective, although the nature of the ECATNet models and the large size of the data structures that represent the state of a LP. However, receiving stragglers due to a violation of the causality constraint and/or an overflow in a place with limited capacity make it the worst possible scenario for TW.

10. The conclusion that in the same conditions as in (8), SYNC performs well, because all the LPs have a similar work to do between barrier synchronisations, exploiting parallelism efficiently. If the message density reduces, the performance also reduces. However, it was shown after a set of evaluations that SYNC is definitely not appropriate for ECATNet distributed simulation in a network of workstations.

11. The conclusion that a collection of workstations in an Ethernet network does not perform so well. The cost of synchronisation is very high due to the slow communication infrastructure based on TCP/IP over Ethernet. The sources of overhead at a sender CPU, the network of routers and the receiver CPU lead to poor performance of the distributed simulators compared with the sequential one.

12. Through the experiments that have been executed with the distributed simulators, we got an insight into their behaviour under different conditions. We characterised how the structure and parameters of the ECATNet model, the synchronisation strategy, the ways of organising the simulator and the scenario of the simulation influence the achieved performance. We conclude that CMB-DA and TW-LZ can be considered as suitable approaches for the distributed simulation of ECATNets. The main requirement remains a fast message passing mechanism for better performance.

9.3 Further Work

There are many ways to further extend the work presented here. The most appealing ones are given in the following points:

1. To build a complete analysis tool for ECATNets, based on conservative, Time Warp and synchronous distributed simulation engines. The tool should be able to allow a researcher to describe the system under study, given its ECATNet model and a collection of parameters such as timings. Our system for modelling and simulation with ECATNets currently consists of a sequential and three distributed simulators. It is planned to extend the simulation system with another component: a net editor which will allow the entering of a graphical representation of ECATNet models. A graphical editor of an earlier style of algebraic Petri nets has been developed [BB92] and will be modified to accomodate ECATNets and to offer a way of providing the information to describe in a user friendly way.

2. Implementation on Massively Parallel Processing architectures:

   - Distributed memory environment: the communication in the distributed simulators is modelled via message passing between different processors. This yields a very portable implementation. Since the MPI library has been adopted, the three distributed simulators (CMB-DA, TW-LZ and SYNC) can be successfully ported on Massively Parallel Processing architectures, such as the the Cray T3E, the Thinking Machine CM-5 or the Intel Paragon.

   - Shared memory environment: The simulation algorithms developed for a message passing environment trivially adapt without performance loss to shared memory by emulating *message exchange* via *shared variables.* The protocols we proposed run in a distributed memory environment. Since distributed memory is not the optimal architecture for parallel simulation algorithms because of the *tight* global synchronisation these algorithms impose, it will be interesting to investigate the performance of these algorithms on shared memory architectures. In this case, messages are exchanged through a global memory.

   - In relation to the SYNC simulator, the current implementations follows a SPMD programming model. It would be very interesting to implement a version for SIMD machines in order to test its behaviour using several thousands of processing elements.

3. Implementation of an ECATNet *concurrent* simulator. The distribution at the event level with a centralised event list is particularly appropriate for shared memory multiprocessors [JCRB89]. The model parallelism exploitation aims at a distribution of single events among processors for their *concurrent* execution. If the event list is a centralised data structure maintained by a *master* processor, concurrent events are distributed to a pool of *slave* processors dedicated to execute them. In ECATNets, the master processor in this case takes care that consistency in the event structure is preserved, i.e., prohibits the execution of events potentially yielding causality errors due to overlapping effects of events being concurrently processed. In addition to this, the application of rewrite rules will find the set of transitions to be fired in parallel, and the events processed in parallel are typically the ones located at the same time.

4. Implementation in distributed shared memory environment. Most of the parallel simulations applications found in the literature were run on shared memory architectures, very few on a network of workstations, a widely used hardware platform which cannot be ignored by parallel simulation researchers. However, the implementation of an ECATNet concurrent simulator prototype using *Phosphorus* [CDMB95] has been quite disappointing. Phosphorus is a distributed shared memory system developed on top of the PVM, which makes the performance of the various simulators (distributed and concurrent) built on top of two messages passing libraries (MPI and PVM) difficult to compare.

A future ECATNet concurrent simulator in DSM can be implemented in MPI2, which permits the use of one-sided communications. This is important to the application programmer, because even loose synchronisation of the send-receive pair in the algorithms imposes constraints and is no longer necessary. It is difficult to gain acceptable speedup for parallel simulation in a network of workstations due to long communication delays. One possible solution is to connect the workstations using a high speed network, such as a fast Ethernet or an ATM network.

5. This study has, fundamentally, an empirical basis. An analytical study of the way the different characteristics of ECATNet models, simulators and parallel programming environment affect the execution time of simulators should be of great interest. The difficult part is that the spectrum of parameters to consider is too large. Several studies have been done in this direction, but most of them make a series of simplifying assumptions which limit its applicability.

6. The use of parallel simulation of ECATNets in rewriting logic. Rewriting has been recognised as an efficient concurrent computational paradigm. The notion of rewriting has been generalised so that functional computations, as well as other parallel computations that are highly non-functional in nature, can be expressed using a declarative and machine independent parallel programming [LMOM94]. A new type of rewriting, Object-Oriented Rewriting, corresponds to *actor-like objects* [Agh90, Agh96] that interact with each other by asynchronous message-passing. Many discrete event simulation applications can be naturally expressed and parallelised in this way.

Rewriting logic has been used as a semantic framework suitable for object-oriented specification [LLW95] and implementation of the style of discrete event simulation within rewriting logic [Lan96]. Further work will include the implementation of an ECATNet parallel simulator in Maude programming language [CDE$^+$98] which is an extension of OBJ [GKK$^+$88], thus extending the domain of parallel simulation and rewriting logic. Maude is a designed language of rewriting logic proposed as a machine-independent parallel programming language that can be efficiently implemented in parallel on many different machines, ranging from sequential, SIMD, MIMD, and MIMD/SIMD machines [LMOMR94]. The description of the use of object-oriented rewriting logic in the field of distributed simulation of ECATNets has been given in [DB98]. This modest approach is driven from practical considerations from the work on developing the distributed simulation framework for ECATNets, and must be considered as a first step towards a solution of the problem. The interesting thing about rewriting logic and parallel simulation is to see that concurrency occurs *explicitly* in the simulation model and *implicitly* in the logic.

7. The work presented in the thesis should be considered as a step in the direction of efficiently implemented distributed simulation techniques for high-level nets such as CPNets. An approach is proposed in [BP97] to relate CPNets to ECATNets. Results concerning analysis techniques of CPNets can be used for ECATNets analysis.

8. The simulators developed should be used in the development of real applications: distributed real time systems, distributed measurements systems and multimedia applications.

9. It will be interesting to implement the synchronisation mechanisms in Java so as to improve portability and move towards distributed simulation at the Web level.

Appendix A

Parallel Programming Environment

A.1 Introduction

In this appendix we present the parallel programming environment used for our research in distributed simulation. All our experiments with the simulators: CMB-DA, TW, and SYNC were made using an implementation of MPI (Message Passing Interface) over a network of Sun Sparc workstations as a platform for parallel programming.

A.2 Message Passing Interface

The goal of MPI, simply stated, is to develop a widely used standard for message passing programs. The interface should establish a practical, portable, efficient and standard for message passing [Mes95]. Over the last ten years, substantial progress has been made in casting significant applications in this paradigm. Each parallel computer vendor has implemented its own variant. More recently, several systems have demonstrated that a message passing system which is both efficient and portable can be implemented. The purpose of MPI is to define both the syntax and semantics of a core of library routines that will be useful to a wide range of users and efficiently implementable on a wide range of computers.

MPI has emerged as the future standard for message passing in both distributed and parallel computing environments. This standard is the result of contributions from more than 40 organisations (hardware/software vendors, federal laboratories, and universities). In designing MPI, the MPI forum sought to make use of the most attractive features of a number of existing message passing systems, rather than selecting one of them and adopting it as a standard. Thus, MPI has been strongly influenced by many works at the IBM T. J. Watson research Center, Intel's NX/2, Express, nCUBE's Vertex, p4, and PARMACS. Other important contributions have come from Zipcode, Chimp, PVM, Chameleon, and PICL [Mes95]. MPI offers, for the first time, an accepted standard for message passing. It is based on point-to-point communication between pairs of processes and collective communications within groups of processes. Additionally, it includes the specification of a much richer set of features than previous message-passing models such as p4 and PVM. These allow the programmer to manipulate entire process groups, define topological structure for process groups, and explicit facilities to aid in the development and use of parallel libraries.

Several implementations of MPI are freely available. One of those is MPICH developed by Argonne National Laboratory and Mississipi State University. It is supported on a variety of parallel computers and networks of workstations. Parallel computers that are supported include: IBM SP1, SP2 (using various communication options), Thinking Machine CM-5, Intel Paragon, IPSC860, Touchstone Delta, Ncube2, Meiko CS-2, Kendall Square KSR-1 and KSR-2, SGI and Sun microprocessors. Supported workstations include: Sun4 family running SunOS or Solaris, HP, DEC 3000 and Alpha, IBM RS/6000 family, SGI, Intel based PC clones running Linux. Other available MPI implementations are:

- Edinburgh Parallel Computer Centre CHIMP implementation
- Mississipi State University UNIFY implementation
- Ohio Supercomputer Center LAM implementation
- University of Nebraska at Omaha WinMPI implementation.

MPI-2, recently developed [Mes97], provides extensions to the first release of MPI. The areas of expansion are:

- Input/Output
- Active messages
- Process startup
- Dynamic process control

- Remote store/access
- Language bindings for Fortran 90 and C++
- Graphics
- Real-time support.

A.2.1 MPI Programs

The current MPI specification [Mes95] includes bindings for C and Fortran 77 programming languages. MPI-2 [Mes97] include bindings for C++ and Fortran 90. In the remaining, we will only discuss the C binding.

Any MPI program must include the header file "mpi.h", where the definitions, macros and function prototypes necessary for compiling the programs can be found. Before using any other MPI function, a program must invoke `MPI_Init()`, to do all the necessary set-up operations. After finishing (but before exiting) the function `MPI_Finalize()` must be called for cleaning up.

```
int MPI_Init(int argc, char **argv);

int MPI_Finalize();
```

After the initialisation, functions are available for point to point communication, collective communication, and several environment management functions. In the following sections we give a description of the functions which are relevant for our work.

A.2.2 Communicators

MPI provides the function `MPI_Comm_rank()` which returns the rank of a process in its second argument. Its syntax is:

```
int MPI_Comm_rank (MPI_Comm comm, int rank);
```

The first argument is a *communicator*. Essentialy a communicator is a collection of processes that can send messages to each other. For basic programs, the only communicator needed is `MPI_COMM_WORLD`. It is predefined in MPI and consists of all the processes running when the program execution begins.

MPI provides the function `MPI_Comm_size` for determining the number of processes executing the program. Its first argument is a communicator. It returns the number of processes in a communicator in its second argument. Its syntax is:

```
int MPI_Comm_size (MPI_Comm comm, int size);
```

A.2.3 Point to Point Communication

MPI offers two communication models, *blocking* and *nonblocking*, and four communication modes: *basic*, *synchronous*, *buffered* and *ready*.

Basic: MPI_Send completes when the message data and envelope have been safely stored away so that the sender is free to access and overwrite the send buffer. The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer.
Buffered: MPI_Bsend completes immediately after storing the message in a local buffer, managed by the user. Its completion never depends on the occurrence of a matching receive.
Synchronous: MPI_Ssend only completes when a matching receive has been posted and the message interchange has been completed.
Ready: MPI_Rsend can be started only if a matching receive has been already posted. Otherwise the operation is erroneous and its outcome is indefined.

All four blocking send operations take the same arguments. The prototype of MPI_Send is as follows:

```
int MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int
tag, MPI_Comm comm);
```

The blocking receive operation can match any of the send modes, and returns only after the receive buffer contains the newly received message.

```
int MPI_Recv (void *buf, int count, MPI_Datatype datatype, int src, int
tag, MPI_Comm comm);
```

For the nonblocking model of communication, the functions: `MPI_Isend()`, `MPI_Ibsend()`, `MPI_Issend()`, `MPI_Irsend()`, `MPI_Irecv()` are available to start an operation. Several forms of `MPI_Wait()` allow a process to block until a previously started operation (or set of operations) has been completed. Alternatively, a non-blocking test of completion can be done using `MPI_Test()`. A non-committed posted operation can be canceled with `MPI_Cancel()`.

It is also possible to check if messages are pending to be received. `MPI_Probe()` is blocking, i.e., blocks the caller if no message is ready, until one is received. `MPI_Iprobe()` is just a nonblocking test. In either case, the message is not actually received until a receive operation is done.

A.2.4 Collective Communication

A collective communication is a communication that involves a group of processes. All collective operations are *global* and *blocking*, that is, in order to perform an operation all the members of a group must call it, and control returns when it has been completed. In many cases a *root* process is mentioned. Any process in a group can be the root of an operation, but it is necessary that all members agree in defining which process is the root. The collective communication functions provided by MPI are:

`MPI_Barrier()`: barrier synchronisation across all group members.
`MPI_Bcast()`: broadcast from one member to all members of a group.
`MPI_Gather()`: gather data from all group members to one member.
`MPI_scatter()`: scatter data from one member to all members of a group.
`MPI_Allgather()`: a variation of MPI_Gather where all members of the group receive the result of the gather operation.
`MPI_Reduce()`: global reduction operation such as sum, max, min, or user-defined functions, where the result is returned to only one member.
`MPI_Allreduce()`: a variation of `MPI_Reduce()` where the result is returned to all group members.
`MPI_Reduce_scatter()`: a combined reduction and scatter operation. First a reduction is done, and then the result (a vector) is scattered along the processes in the group.
`MPI_Scan()`: scan across all members of a group.

A.2.5 Data Types

One of the parameters of send and receive operations is the type of the data units being transmitted (other being the number of units transmitted). Both sender and

MPI data type	C data type
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Table A.1: MPI data types (C binding)

receiver must agree in the specified data type. A series of constants are defined to choose the appropriate data type. Table A.1 lists the data types available for the C binding.

The last two types do not correspond to any C data type. A value of type MPI_BYTE consists of an octet, which is uninterpreted and different from a character. MPI_PACKED is used to interchange blocks of non-contiguous data items which have been packed into a contiguous buffer. As MPI supports parallel computations across heterogeneous environments, the MPI_BYTE data type can be used to transfer data without any conversion of the different data representation formats of the communicating machines.

A.3 Running an MPI Application on a NOW

The implementation of MPI includes a configuration program which sets up CHIMP MPI (Edinburgh Parallel Computer Centre) for running in a network of Sun workstations running Solaris. The actual CHIMP version is 2.0 which is a distributed Unix version SPARC running Solaris 2.1 (or later). Once the environment has been configured, a Makefile is provided to build:

- a library with all MPI functions;
- scripts for compilation of MPI application code (*mpicc* for the C programming language);
- *mpirun*, a tool for starting parallel programs in a way independent of the target machine;
- a collection of manuals and other documentation.

Once the set of executables which make up the MPI application has been generated, we must write a *configuration file* for the application. Items enclosed in brackets () and separated by commas represent MPI processes. Processes are specified by the command that would execute that process; usually this will be the executable name followed by any command arguments. These processes are normally placed arbitrarily on the available processors: the only relationship automatically preserved from configuration file to process placements is the co-locality of processes on processors, i.e., if two processes appear in the same set of brackets in the configuration file, it is guaranteed that they will be run on the same processor. The *loader tool* interprets a configuration file and places processes on computing resources as specified. All the machines of the configuration file have the same view of the file system, which usually means that NFS (Network File System) is in use.

Appendix B

Glossary

B.1 Petri Nets

Inhibitor arc: arc that allows zero testing. An inhibitor arc from a place p_i to a tansition t_j has a small circle rather than an arrowhead at the transition. The firing rule is changed as follows: a transition is enabled when tokens are all of its (normal) inputs and zero tokens are in all of its inhibitor inputs. The transition fires by removing tokens from all of its (normal) inputs.

Random switch: probability distribution associated with the choice of the transition to be fired among a set of more immediate transitions in conflict at a certain reachable marking. The distribution is in general marking-dependent.

Let $N = (P,T,F)$ be a net and $x \in X$. Then:

Pre-set: noted $\bullet x = \{y \mid (y,x) \in F\}$ (Pre-set of $x \in X$).

Post-set: noted $x\bullet = \{y \mid (x,y) \in F\}$ (Post-set of $x \in X$).

Siphon: let $P_0 \subseteq P$ a set of places. P_0 is called a trap iff $P_0\bullet \subseteq \bullet P_0$.

Deadlock: P_0 is called a deadlock iff $\bullet P_0 \subseteq P_0\bullet$.

B.2 Parallel Processing

Bandwidth: amount of data that can be sent through a given communication circuit per second (an important communication channel's parameter).

Barrier: a point in a program at which *barrier synchronisation* occurs.

Barrier synchronisation: an event in which two or more processes belonging to some implicit or explicit group *block* until all members of the group have blocked.

Broadcast: to send a message to all possible recipients.

Deadlock: a situation in which each possible activity is *blocked*, waiting on some other activity that is also blocked.

Granularity: the size of the operations done by a processor between communication events. A process may be *fine-grained* or *coarse-grained.*

Host: a computer connected to a network.

Latency: time it takes for a packet to cross a network connection, from sender to receiver (an important communication channel's parameter).

Load balancing: techniques which aim to spread tasks among the processors in a parallel processor to avoid some processors being idle when other have tasks queueing for execution.

Mapping: an allocation of processes to processors.

Message passing: a style of interprocess communication in which processes send discrete messages to one another.

Overhead: information, such as control, routing, and error checking characters, that is transmitted along with the user data. It also includes information such as network status or operational instructions, network routing informations and retransmissions of user data received in error.

Parallel speedup: ratio of the serial execution time of the best known serial algorithm (TS) to the parallel execution time of the chosen algorithm (TP).

Reduction operation: an operation applying an associative or commutative binary operator $\oplus$ to a list of values $\{v_0\ v_1\ \ldots\ v_{n-1}\}$ to produce $(v_0 \oplus v_1 \oplus \ldots \oplus v_{n-1})$.

Shared memory: memory that appears to the user to be contained in a single *address space* and that can be accessed by any process.

Throughput: (1) the rate at which a processor can work expressed in instructions per second or jobs per hour or some other unit of performance; (2) the amount of data a communication's channel can carry, usually in bytes per second.

References

[ABC$^+$91] M. Ajmone Marsan, G. Balbo, G. Chiola, G. Conte, S. Donatelli, and G. Franceschinis. An Introduction to Generalized Stochastic Petri Nets. *Microelectronics and Reliability*, 31(4):699–725, 1991.

[ABC$^+$95] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets.* John Wiley and Sons, 1995.

[ABL94] D.K Arvind, R. Bagrodia, and J.Y. Lin, editors. *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, Edinburgh, Scotland, UK, Jun 1994. ACM/IEEE/SCS. SIGSIM Newletter 24(1), July 1994.

[AC87] M. Ajmone Marsan and G. Chiola. On Petri Nets with Deterministic and Exponentially Distributed Firing Times. In Rozenberg [Roz87], pages 132–145.

[ACB84] M. Ajmone 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(2):93–122, May 1984.

[ACLS94] D. Agrawal, M. Choy, H.V. Leong, and A.K. Singh. Maya: a Simulation Platform for Distributed Shared Memories. In Arvind et al. [ABL94], pages 151–155.

[AD91] H. Ammar and S. Deng. Parallel Simulation of Petri Nets Using Spatial Decomposition. In *Proceedings of the 1991 IEEE International Symposium on Circuits and Systems*, pages 826–829, Singapore, Jun 1991.

[Agh90] G. Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, Sep 1990.

[Agh96] G. Agha. Modeling Concurrent Systems: Actors, Nets, and the Problem of Abstraction and Composition. In J. Billington and W. Reisig, editors, *Proceedings of the 17th International Conference on Application and Theory of Petri Nets*, pages 1–10, Osaka, Japan, Jun 1996. Lecture Notes in Computer Science 1091, Springer.

[Ajm89] M. Ajmone Marsan. Stochastic Petri Nets : an Elementary Introduction. In G. Rozenberg, editor, *Advances in Petri Nets 1989*, pages 1–29, 1989. Lecture Notes in Computer Science 424, Springer-Verlag.

[Ajm93] M. Ajmone Marsan, editor. *Proceedings of the 14th International Conference on Application and Theory of Petri Nets*, Chicago, Illinois, Jun 1993. Lecture Notes in Computer Science 691, Springer-Verlag.

[AL97] T. Aura and J. Lilius. Time Processes for Time Petri Nets. In P. Azéma and G. Balbo, editors, *Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, pages 136–155, Toulouse, France, Jun 1997. Lecture Notes in Computer Science 1248, Springer.

[Aya93] R. Ayani. Parallel Simulation. In L. Donatiello and R. Nelson, editors, *Performance Evaluation of Computer and Communication Systems*, pages 1–20, 1993. Lecture Notes in Computer Science 729, Springer-Verlag.

[Bal92] G. Balbo. Performance Issues in Parallel Programming. In K. Jensen, editor, *Proceedings of the 13th International Conference on Application and Theory of Petri Nets*, pages 1–23, Sheffield, UK, Jun 1992. Lecture Notes in Computer Science 616, Springer-Verlag.

[BB92] O. Bounouioua and M. Bettaz. A Graphical Editor-Simulator for Algebraic Term Nets. In *Proceedings of the Second Maghrebin Conference on Software Engineering and Artificial Intelligence*, pages 177–190, Tunis, Tunisia, Apr 1992.

[BBB$^+$94] F. Bacceli, G. Balbo, R.J. Boucherie, J. Campos, and G. Chiola. Annotated Bibliography on Stochastic Petri Nets. In O.J. Boxma and G.M. Kode, editors, *Proceedings of the 3rd QMIPS Workshop (Part I)*, pages 25–44, 1994. Performance Evaluation of Parallel and Distributed Systems, Vol.105, CWI Tract.

[BBCC92] G. Balbo, S.C. Bruell, P. Chen, and G. Chiola. An Example of Modeling and Evaluation of Concurrent Program using Colored Stochastic Petri Nets. Lamport's Fast Mutual Exclusion Algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):221–240, Mar 1992.

[BBCC95] E. Battiston, O. Botti, E. Crivelli, and D. De Cindio. An Incremental Specification of a Hydroelectric Power Plant Control Systems using a Class of Modular Algebraic Nets. In G. De Michelis and M. Diaz, editors, *Proceedings of the 16th International Conference on Application and Theory of Petri Nets*, pages 84–102, Torino, Italy, Jun 1995. Lecture Notes in Computer Science 935, Springer.

[BC92] M. Bettaz and A. Choutri. Algebraic Term Nets: a Formalism for Specifying Communication Software in the OSI Framework. In *Proceedings of the Unified Computation Laboratory*, pages 293–305. Oxford University Press, 1992.

[BC93] F. Bacceli and M. Canales. Parallel Simulation of Stochastic Petri Nets using Recurrence Equations. *ACM Transactions on Modeling and Computer Simulation*, 3(1):20–41, Jan 1993.

[BCD98] K. Barkaoui, A. Chaoui, and K. Djemame. On the Use of Algebraic Petri Nets for Supervisor Evaluation of Discrete Event Systems. In P. Borne and M. Ksouri, editors, *Proceedings of CESA'98 (Computational Engineering in Systems Applications)*, pages 465–469, Nabeul-Hammamet, Tunisia, April 1998. IEEE.

[BCM88] E. Battiston, F. De Cindio, and G. Mauri. OBJSA Nets: a Class of High-Level Nets Having Objects as Domains. In G. Rozenberg, editor, *Advances in Petri Nets 1988*, pages 20–43, 1988. Lecture Notes in Computer Science 340, Springer Verlag.

[BDF92] G. Balbo, S. Donatelli, and G. Franceschinis. Understanding Parallel Program Behavior Through Petri Net Models. *Journal of Parallel and Distributed Computing*, 15(3):171–187, Jul 1992.

[BEM90] B. Butler, R. Esser, and R. Mattmann. A Distributed Simulator for High Order Petri Nets. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, pages 47–63, 1990. Lecture Notes in Computer Science 483, Springer-Verlag.

[Bet91] M. Bettaz. An Association of Algebraic Term Nets and Abstract Data Types for Specifying Real Communication Protocols. In H. Ehrig, K.P. Jantke, F. Orejas, and H. Reichel, editors, *Proceedings of the 7th Workshop on Specification of Abstract Data Types*, pages 11–30, Wusterhausen/Dosse, Germany, 1991. Recent Trends in Data Type Specification, Lecture Notes in Computer Science 534, Springer-Verlag.

[Bet93] M. Bettaz. Specification Hautement Compacte et Modulaire de l'Ethernet: la Station Emettrice. In *Proceedings of CFIP'93*, Montreal, Canada, 1993. Hermes, Paris. (In french).

[Bil89] J. Billington. Many Sorted High-Level Nets. In *Proceedings of the Third International Workshop on Petri Nets and Performance Models*, Kyoto, Japan, Dec 1989.

[BL94] R.L. Bagrodia and W. Liao. Maisie: A Language for the Design of Efficient Discrete-Event Simulations. *IEEE Transactions on Software Engineering*, 20(4):225–238, Apr 1994.

[BM93a] M. Bettaz and M. Maouche. How to Specify Non Determinism and True Concurrency with Algebraic Term Nets. In M. Bidoit and C. Choppy, editors, *Proceedings of the 8th Workshop on Specification of Abstract Data Types*, pages 164–180, Dourdan, France, 1993. Recent Trends in Data Specification, Lecture Notes in Computer Science 655, Springer-Verlag.

[BM93b] M. Bettaz and A. Mehemmel. Modeling and Proving of Truly Concurrent Systems with CATNets. In *Proceedings of 1st Euromicro Workshop on Parallel and Distributed Processing*, pages 265–272. IEEE/CS, 1993.

[BM95] M. Bettaz and M.Maouche. Modeling of Object Based Systems with Hidden Sorted ECATNets. In *Proceedings of MASCOTS'95*, pages 307–311, Durham, North Carolina, Jan 1995. IEEE.

[BMSB92] M. Bettaz, M. Maouche, M. Soualmi, and M. Boukebeche. Using ECATNets for Specifying Communication Software in the OSI Framework. In W.W. Koczkodaj, P.E. Lauer, and A.A. Toptsis, editors, *Proceedings of the 1992 International Conference on Computing and Information*, pages 410–413, Toronto, Canada, May 1992. IEEE.

[BMSB93a] M. Bettaz, M. Maouche, M. Soualmi, and M. Boukebeche. Compact Modeling and Rapid Prototyping of Communication Software with ECATNets: a Case Study. *Simulation Series*, 25(1):149–154, 1993. SCS and IEEE.

[BMSB93b] M. Bettaz, M. Maouche, M. Soualmi, and M. Boukebeche. Protocol Specification using ECATNets. *Networking and Distributed Computing*, 3(1):7–35, 1993. Hermes, Paris.

[BMSB94] M. Bettaz, M. Maouche, M. Soualmi, and M. Boukebeche. On Reusing ATNet Modules in Protocol Specification. *Journal of Systems and Software*, 27(2):119–128, Nov 1994.

[BP97] F. Belala and L. Petrucci. Sémantique des ECATNets en Termes de CPNets : Application a un Exemple de Production. In *Proceedings of MOSIM'97*, pages 367–375, Rouen, France, Jun 1997. Hermes. (In french).

[Bräunl93] T. Bräunl. *Parallel Programming*. Prentice Hall, 1993.

[BRR86] W. Brauer, W. Reisig, and G. Rozenberg, editors. *Advances in Petri Nets 1986, Part I.* Lecture Notes in Computer Science 255, Springer-Verlag, 1986.

[Bry77] R.E. Bryant. Simulation of Packet Communications Architecture Computer Systems. Technical Report MIT-LCS-TR-188, MIT, 1977.

[BT94] A. Boukerche and C. Tropper. A Static Partitioning and Mapping Algorithm for Conservative Parallel Simulations. In Arvind et al. [ABL94], pages 164–172.

[Car89] J.A. Carrasco. Automated Construction of Compound Markov Chains from Generalized Stochastic High Level Petri Nets. In *Proceedings of the 3rd International Workshop on Petri nets and Performance Models*, pages 93–102, Kyoto, 1989. IEEE.

[CD97] A. Chaoui and K. Djemame. Static Deadlock Detection in Ada Tasking using a Logic of Concurrency. Technical report, Computing Science Institute, University of Constantine, Algeria, Jul 1997.

[CDE+98] M. Clavel, F. Durán, S. Eker, P. Lincoln, and J. Meseguer. An Introduction to Maude (Beta Version). Technical report, SRI International, Menlo Park, CA., Mar 1998. http://maude.csl.sri.com/manual/.

[CDFH93] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic Well-Formed Coloured Nets and Symmetric Modelling Applications. *IEEE Transactions on Computers*, 42(11):1343–1360, Nov 1993.

[CDMB95] R. Cabrera, I. Demeure, P. Meunier, and V. Bartro. Phosphorus: a Tool for Shared Memory Management in a Distributed Environment. Technical Report 95D003, Ecole Nationale Superieure des Telecommunications, Departement d'Informatique, Paris, France, Nov 1995. http://www-inf.enst.fr/~research/publications_ec/demeure/demeure_95c.ps.

[CF93a] G. Chiola and A. Ferscha. Distributed Simulation of Petri Nets. *IEEE Parallel and Distributed Technology*, pages 33–50, Aug 1993. http://sokrates.ani.univie.ac.at/~ferscha/E-PAPERS/ieeepdt93.ps.-gz.

[CF93b] G. Chiola and A. Ferscha. Distributed Simulation of Timed Petri Nets: Exploiting the Net Structure to Obtain Efficiency. In Ajmone Marsan [Ajm93], pages 146–165. http://sokrates.ani.univie.ac.at/~ferscha/E-PAPERS/petrinets93.ps.-gz.

[CF95] G. Chiola and A. Ferscha. Performance Comparable Design of Efficient Synchronization Protocols for Distributed Simulation. In *Proceedings*

of MASCOTS'95, pages 59–65, Durham, North Carolina, Jan 1995. IEEE.
http://sokrates.ani.univie.ac.at/~ferscha/E-PAPERS/mascots.ps.gz.

[CFGR95] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN 1.7: GRaphical Editor and Analyzer for Timed and Stochastic Petri Nets. *Performance Evaluation*, 24(1,2):47–68, Nov 1995.

[CGU$^+$94] J. Cleary, F. Gomes, B. Unger, X. Zhonge, and R. Thudt. Cost of Sate Saving and Rollback. In Arvind et al. [ABL94], pages 94–101.

[CH94] R.D. Chamberlain and C.D. Henderson. Evaluating the Use of Pre-Simulation in VLSI Circuit Partitioning. In Arvind et al. [ABL94], pages 139–146.

[CM79] K.M. Chandy and J. Misra. Distributed Simulation: a Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–453, Sep 1979.

[CM81] K.M. Chandy and J. Misra. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM*, 24(11):198–206, Apr 1981.

[CN93] H. Choi and B. Narahari. Efficient Algorithms for Mapping and Partitioning a Class of Parallel Computations. *Journal of Parallel and Distributed Computing*, 19(1):349–363, Dec 1993.

[CR83] J.E. Coolahan and N. Roussopoulos. Time Requirements for Time-Driven Systems Using Augmented Petri Nets. *IEEE Transactions on Software Engineering*, SE-9(5):603–616, Sep 1983.

[CSR93] K. Chung, J. Sang, and V. Rego. A Performance Comparison of Event Calendar Algorithms: an Empirical Approach. *Software - Practice and Experience*, 23(10):1107–1138, Oct 1993.

[CT95] W. Cai and S.J. Turner. An Algorithm for Reducing Null-Messages of CMB Approach in Parallel Discrete Event Simulation. Technical Report TR-333, Department of Computer Science, University of Exeter, 1995. ftp://ftp.dcs.exeter.ac.uk/pub/parallel/simul/rednull.ps.Z.

[CT96a] Cleary and J.J. Tsai. Conservative Parallel Simulation of ATM Networks. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pages 30–38, Philadelphia, May 1996. ACM/IEEE/SCS.

[CT96b] Q.M. Cui and S.J. Turner. A New Approach to the Distributed Simulation of Timed Petri Nets. In *Proceedings of SCS European Simulation Multiconference*, pages 90–94, Budapest, Hungary, Jun 1996. Society for Computer Simulation.

[DB94] K. Djemame and M. Bettaz. On the Parallel Simulation of ECAT-Nets. Technical report, Computing Science Institute, University of Constantine, Algeria, Jun 1994.

[DB98] K. Djemame and M. Bettaz. Parallel Simulation in Rewriting Logic: Some Observations. In *Proceedings of 6th Euromicro Workshop on Parallel and Distributed Processing*, pages 197–203, Madrid, Spain, Jan 1998. IEEE/CS.

[DBGM95] K. Djemame, M. Bettaz, D.C. Gilles, and L.M. Mackenzie. Time Warp Simulation of ECATNets. In *Proceedings of 7th European Simulation Symposium*, pages 171–175, Erlangen, Nuremberg, Germany, Oct 1995. SCS.

[DBGM96a] K. Djemame, M. Bettaz, D.C. Gilles, and L.M. Mackenzie. Distributed Simulation of ECATNets: A Conservative Approach. In *Proceedings of 4th Euromicro Workshop on Parallel and Distributed Processing*, pages 518–525, Braga, Portugal, Jan 1996. IEEE/CS.

[DBGM96b] K. Djemame, M. Bettaz, D.C. Gilles, and L.M. Mackenzie. Performance Comparison of High Level Algebraic Nets Distributed Simulation Protocols. In J.M. Charnes, D.M. Morrice, D.T. Brunner, and J.J. Swain, editors, *Proceedings of 1996 Winter Simulation Conference*, pages 621–628, Coronado, CA., Dec 1996. ACM/IEEE/SCS.

[DBGM98] K. Djemame, M. Bettaz, D.C. Gilles, and L.M. Mackenzie. Performance Comparison of High Level Algebraic Nets Distributed Simulation Protocols (*full paper*). *Journal of Systems Architecture*, 44:457–472, 1998.

[DF97] S.R. Das and R.M. Fujimoto. Adaptive Memory Management and Optimism Control in Time Warp. *ACM Transactions on Modeling and Computer Simulation*, 7(2):239–271, Apr 1997.

[DH90] C. Dutheillet and S. Haddad. Regular Stochastic Petri Nets. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, pages 186–210, 1990. Lecture Notes in Computer Science 483, Springer-Verlag.

[Dje98] K. Djemame. Queueing Networks Versus Petri Nets: Two Performance Case Studies in Distributed Simulation. In *Proceedings of CNP'98 (Colloque National sur la Productique)*, pages 136–141, Tizi-Ouzou, Algeria, May 1998.

[DTGN84] J.B. Dugan, K.S. Trevedi, R.M. Geist, and V.F. Nicola. Extended Stochastic Petri Nets: Applications and Analysis. In E. Gelenbe, editor, *Proceedings of Performance'84*, pages 507–519, Paris, France, Dec 1984. Elsevier Science Publishers, North Holland.

[Dun90] R. Duncan. A Survey of Parallel Computer Architectures. *IEEE Computer*, 23(2):5–16, Feb 1990.

[EM85] E. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 1*. EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1985.

[Ert94] W. Ertel. On the Definition of Speedup. In Halatsis et al. [HMPT94], pages 289–300. http://www.fh-weingarten.de/~ertel/parle.ps.gz.

[FC95] A. Ferscha and G. Chiola. Performance Comparison of Distributed Petri Net Simulations. In *Proceedings of the 1995 Summer Simulation Conference*, Ottawa, Canada, Jul 1995. SCS.
http://sokrates.ani.univie.ac.at/~ferscha/E-PAPERS/scs95ch.ps.gz.

[Fel93] F. Feldbrugge. Petri Net Tool Overview. In G. Rozenberg, editor, *Advances in Petri Nets 1993*, pages 169–209, 1993. Lecture Notes in Computer Science 674, Springer Verlag.

[Fer92] A. Ferscha. A Petri Net Approach for Performance Oriented Parallel Program Design. *Journal of Parallel and Distributed Computing*, 15(3):188–206, Jul 1992.
http://sokrates.ani.univie.ac.at/~ferscha/E-PAPERS/jpdc92.ps.gz.

[Fer96] A. Ferscha. *Parallel and Distributed Simulation of Discrete Event Systems*, chapter 35, pages 1003–1039. In Zomaya [Zom96], 1996.
http://sokrates.ani.univie.ac.at/~ferscha/E-PAPERS/handbook.ps.-gz.

[Fos95] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995.
http://www.hensa.ac.uk/parallel/books/addison-wesley/dbpp/.

[Fuj88] R. Fujimoto. Lookahead in Parallel Discrete Event Simulation. In *Proceedings of the 1988 International Conference on Parallel Processing*, volume 3, pages 34–41, Aug 1988.

[Fuj89] R. Fujimoto. Time Warp on a Shared Memory Multiprocessor. *Transactions of the Society for Computer Simulation*, 6(3):211–239, Jul 1989.

[Fuj90] R. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):31–53, Oct 1990.

[Fuj93] R. Fujimoto. Future Directions in Parallel Simulation Research. *ORSA Journal on Computing*, 5(3):245–248, Summer 1993.

[FW94] B. Falsafi and D.A. Wood. Cost/Performance of a Parallel Computer Simulator. In Arvind et al. [ABL94], pages 173–182.

[Gar90] M. Garzia. Discrete Event Simulation Methodologies and Formalisms. *Simulation Digest*, 21(1):3–13, 1990.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge Tennessee 37831, Sep 1994. http://www.netlib.org/pvm3/ug.ps.

[Gen86] H.J. Genrich. Predicate/Transition Nets. In Brauer et al. [BRR86], pages 207–247.

[GKK+88] J. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An Introduction to OBJ3. In S. Kaplan and J.P. Jouannaud, editors, *Proceedings of the 1st International Workshop on Conditional Term Rewriting Systems*, pages 258–263, Orsay, France, 1988. Lecture Notes in Computer Science 308, Springer-Verlag.

[GKP96] A. Geist, J.A. Kohl, and P.M. Papadopoulos. PVM and MPI: A Comparison of Features. *Calculateurs Parallèles*, 8(2):137–150, Jun 1996. http://www.epm.ornl.gov/pvm/PVMvsMPI.ps.

[GMMP91] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzè. A Unified High-Level Petri Net Formalism for Time-Critical Systems. *IEEE Transactions on Software Engineering*, 17(2):160–172, Feb 1991.

[GT93] D.W. Glazer and C. Tropper. On Process Migration and Load Balancing in Time Warp. *IEEE Transactions on Parallel and Distributed Systems*, 4(3):318–354, March 1993.

[Hei88] P. Heidelberger. Discrete Event Simulations and Parallel Processing: Statistical Properties. *SIAM Journal on Scientific and Statistical Computing*, 9(6):1114–1132, Nov 1988.

[HMPT94] C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors. *Proceedings of the 6th International PARLE Conference*, Athens, Greece, Jul 1994. Lecture Notes in Computer Science 817, Springer-Verlag.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Englewood Cliffs, N.J., Prentice-Hall, 1985.

[Inm89] Inmos Limited. *OCCAM2 Reference Manual.* Inmos Ltd., Bristol, UK, 1989.

[Inm90] Inmos Limited. *ANSI C Toolset.* Inmos Ltd., Bristol, UK, 1990.

[JBW+87] D.R. Jefferson, B. Beckman, F. Wieland, L. Blume, M. Diloreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren,

J. Wedel, H. Younger, and S. Bellenot. The Time Warp Operating System. In *Proceedings 11th Symposium on Operating Systems Principles*, volume 21, pages 77–83, 1987.

[JCRB89] D.W. Jones, C.C. Chou, D. Renk, and S.C. Bruell. Experience with Concurrent Simulation. In *Proceedings of the 1989 Winter Simulation Conference*, pages 756–764. E.A. MacNAIR, K.J. Musselman, P. Heidelberger (Ed.), 1989.

[JE91] K. Jensen and G. Rozenberg (Ed.). *High-Level Petri Nets.* Springer-Verlag, Berlin, 1991.

[Jef85] D.R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, Jul 1985.

[Jen92] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts.* EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1992.

[Kau87] F.J. Kaudel. A Literature Survey on Distributed Discrete Event Simulation. *Simuletter*, 18(2):11–21, Jun 1987.

[Kel95] C. Kelling. TimeNET-Sim - a Parallel Simulator for Stochastic Petri Nets. In *Proceedings of the 28th Annual Simulation Symposium*, pages 250–258, Phoenix, AZ, 1995. IEEE Computer Society Press.

[KGS93] P.G. Kropf, K. Guggisberg, and M. Studer. Parallel Simulations with High-Level Petri Nets on Transputers. *Speedup Journal*, 7(2):24–28, Sep 1993.

[KY91] P. Konas and P.C. Yew. Parallel Discrete Event Simulation and Shared-Memory Multiprocessors. In A.H. Rutan, editor, *Proceedings of the 24th Annual Simulation Symposium*, pages 134–148, New-Orleans, Louisiana, 1991. IEEE Computer Society Press.

[Lak95] C. Lakos. From Coloured Petri Nets to Object Petri Nets. In G. De Michelis and M. Diaz, editors, *Proceedings of the 16th International Conference on Application and Theory of Petri Nets*, pages 278–297, Torino, Italy, Jun 1995. Lecture Notes in Computer Science 935, Springer. ftp://opn.cs.utas.edu.au/pub/postscript/pn95-paper.ps.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, Jul 1978.

[Lan96] C. Landauer. Discrete-Event Systems in Rewriting Logic. In J. Meseguer, editor, *Proceedings of 1st International Workshop on Rewriting Logic and its Applications*, Asilomar, Pacific Grove, CA., Sep 1996. Electronic Notes in Theoretical Computer Science, Vol.4.

http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume4/-landauer.ps.

[Lin90] Y.B. Lin. *Understanding the Limits of Optimistic and Conservative Parallel Simulation.* PhD thesis, Department of Computer Science and Engineering, University of Washington, Seattle 98195, Aug 1990. (Available as Technical Report No. 90-08-02).

[Lin93] Y.B. Lin. Will Parallel Simulation Research Survive? *ORSA Journal on Computing*, 5(3):236–238, Summer 1993.

[LK91] A.M. Law and W.D. Kelton. *Simulation Modeling and Analysis.* McGraw-Hill, 2nd edition, 1991.

[LKP92] C.A. Lakos, C.D. Keen, and E.J. Palmer. A Flexible Distributed Simulator for Object-Oriented Petri Nets. In *Proceedings of Transputer and Parallel Applications Conference*, Melbourne, Australia, Nov 1992. ftp://opn.cs.utas.edu.au/pub/postscript/tapa92-paper.ps.

[LL90] Y.B. Lin and E.D. Lazowska. Exploiting Lookahead in Parallel Simulation. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):457–469, Oct 1990.

[LL91a] Y.B. Lin and E.D. Lazowska. A Study of the Time Warp Rollback Mechanism. *ACM Transactions on Modeling and Computer Simulation*, 1(1):51–72, Jan 1991.

[LL91b] Y.B. Lin and E.D. Lazowska. A Time-Division Algorithm for Parallel Simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(1):73–83, Jan 1991.

[LLW95] U. Lechner, C. Lengauer, and M. Wirsing. An Object-Oriented Airport: Specification and Refinement in Maude. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Proceedings of the 10th Workshop on Specification of Abstract Data Types*, pages 351–367, S.Margherita, Italy, 1995. Recent Trends in Data Type Specification, Lecture Notes in Computer Science 906, Springer. http://www.mcm.unisg.ch/~ulechner/compass.ps.

[LM88] C. Lin and D.C. Marinescu. Stochastic High-Level Petri Nets and Applications. *IEEE Transactions on Computers*, 37(7):815–825, Jul 1988.

[LMOM94] P. Lincoln, N. Marti-Oliet, and J. Meseguer. Specification, Transformation, and Programming of Concurrent Systems in Rewriting Logic. In S. Jagannathan G. Blelloch, K.M. Chandy, editor, *Proc. DIMACS Workshop on Specification of Parallel Algorithms*, DIMACS Series in

Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, May 1994.

[LMOMR94] P. Lincoln, N. Marti-Oliet, J. Meseguer, and L. Ricciulli. Compiling Rewriting onto SIMD and MIMD/SIMD Machines. In Halatsis et al. [HMPT94], pages 37–48.

[LP91] Y.B. Lin and B.R. Preiss. Optimal Memory Management for Time Warp Parallel Simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(4):283-307, Oct 1991.

[LPLL93] Y.B. Lin, B. Preiss, W.M. Loucks, and E.D. Lazowska. Selecting the Checkpoint Interval in Time Warp Simulation. In R. Bagrodia and D. Jefferson, editors, *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pages 3–10, San Diego, CA., May 1993.

[LWS91] B. Lubachevsky, A. Weiss, and A. Shwartz. An Analysis of Rollback Based Simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(2):154–193, Apr 1991.

[MBBP97] M. Maouche, M. Bettaz, G. Berthelot, and L. Petrucci. Du vrai parallelisme dans les réseaux algébriques et de son application dans les systèmes de production. In *Proceedings of MOSIM'97*, pages 417–424, Rouen, France, Jun 1997. Hermes. (In french).

[Mes92a] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[Mes92b] J. Meseguer. On the Semantics of Petri Nets. In W.R. Cleaveland, editor, *Proceedings of the 3rd International Conference on Concurrency Theory*, pages 286–301, Stony Brook, NY, USA, Aug 1992. Lecture Notes in Computer Science 630, Springer-Verlag.

[Mes95] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report CS-93-214, University of Tennessee, Jun 1995. http://www.mcs.anl.gov/mpi/mpi-report/mpi-report.html.

[Mes96] J. Meseguer. Rewriting Logic as a Semantic Framework of Concurrency: a Progress Report. In U. Montanari and V. Sassone, editors, *Proceedings of the 7th International Conference on Concurrency Theory*, pages 331–372, Pisa, Italy, Aug 1996. Lecture Notes in Computer Science 1119, Springer.

[Mes97] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface Standard. Technical report, University of Tennessee, Jul 1997.
http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[MF76] P.M. Merlin and D.J. Farber. Recoverability of Communication Protocols - Implications of a Theoretical Study. *IEEE Transactions on Communications*, COM-24(9):1036–1043, Sep 1976.

[Mis86] J. Misra. Distributed Discrete Event Simulation. *ACM Computing Surveys*, 18(1):39–65, Mar 1986.

[Mol85] M.K. Molloy. Discrete Time Stochastic Petri Nets. *IEEE Transactions on Software Engineering*, SE-11:417–423, Apr 1985.

[MPT91] S. Morasca, M. Pezzè, and M. Trubian. Timed High-Level Petri Nets. *Journal of Real-Time Systems*, 3:165–189, 1991.

[MR94] P. Mussi and H. Rakotoarisoa. PARSEVAL: a Workbench for Queueing Networks Parallel Simulation. Technical Report 2234, INRIA-Sophia Antipolis, France, Apr 1994.
http://www.inria.fr/RRRT/RR-2234.html.

[Mur89] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, Apr 1989.

[NH93] D. Nicol and P. Heidelberger. Parallel Simulation of Markovian Queueing Networks Using Adaptive Uniformization. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 135–145, Santa Clara, CA., Jun 1993. Performance Evaluation Review 21(1).

[Nic88] D. Nicol. Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks. *ACM SIGPLAN*, 23(9):124–137, Jul 1988.

[Nic92] D. Nicol. Conservative Parallel Simulation of Priority Class Queueing Networks. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):294–303, May 1992.

[Nic93] D. Nicol. The Cost of Conservative Synchronization in Parallel Discrete Event Simulation. *Journal of the ACM*, 40(2):304–333, Apr 1993.

[NL93] B. Nandy and W.M. Loucks. On a Parallel Partitioning Technique for Use with Conservative Parallel Simulation. In R. Bagrodia and D. Jefferson, editors, *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pages 43–51, San Diego, CA., May 1993.

[NM95] D. Nicol and W. Mao. Automated Parallelization of Timed Petri-Net Simulations. *Journal of Parallel and Distributed Computing*, 29(2):60–74, 1995.

[NR91] D. Nicol and S. Roy. Parallel Simulation of Timed Petri Nets. In *Proceedings of the 1991 Winter Simulation Conference*, pages 574–583. B.Nelson, D.Kelton, G.Clark (Ed.), 1991.

[Pag86] A. Pagnoni. Stochastic Nets and Performance Evaluation. In Brauer et al. [BRR86], pages 436–459.

[Pet62] C.A. Petri. Kommunikation mit Automaten. Technical report, Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr.3, 1962. Also english translation, Communication with Automata, New York: Griffiss Air Force Base, Technical Report RADC TR-65-377, Vol.1, Suppl.1, 1966.

[Pet81] J.L. Peterson. *Petri Net Theory and the Modeling of Systems.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[Pet86] C.A. Petri. Forgotten topics of net theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Advances in Petri Nets 1986, Part II*, pages 500–514, 1986. Lecture Notes in Computer Science 255, Springer-Verlag.

[PS91] A. Prakash and R. Subramanian. FILTER: an Algorithm for Reducing Cascaded Rollbacks in Optimistic Distributed Simulations. In *Proceedings of the 24th annual simulation symposium*, pages 123–132, New-Orleans, Louisiana, 1991.

[PSHH97] F. De La Puente, J.D. Sandoval, P. Hernandez, and F. Herrera. Parallel Simulation for Queueing Networks on Multiprocessor Systems. In *Proceedings of 5th Euromicro Workshop on Parallel and Distributed Processing*, pages 240–245, London, UK, Jan 1997. IEEE/CS.

[Ram74] C. Ramchandani. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets.* PhD thesis, MIT, Cambridge, MA, Feb 1974. (Available as Scientific Report MAC TR-120).

[Rei85] W. Reisig. *Petri Nets.* EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1985.

[Rei86] W. Reisig. Place/Transition Systems. In Brauer et al. [BRR86], pages 117–141.

[Rei91] W. Reisig. Petri Nets and Algebraic Specifications. *Theoretical Computer Science*, 80:1–34, 1991.

[RH80] C.V. Ramamoorthy and G.S. Ho. Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets. *IEEE Transactions on Software Engineering*, SE-6(5):440–449, Sep 1980.

[RM91] H. Rakotoarisoa and P. Mussi. PARSEVAL: PARallelisation sur Reseaux de Transputers de Simulation pour l'EVALuation de Performances. Technical Report RT-0131, INRIA-Sophia Antipolis, France, Sep 1991. (In french). http://www.inria.fr/RRRT/RT-0131.html.

[RMM88] D.A. Reed, A.D. Malony, and B.D. McCredie. Parallel Discrete Event simulation Using Shared Memory. *IEEE Transactions on Software Engineering*, 14(4):541–553, Apr 1988.

[Roz87] G. Rozenberg, editor. *Advances in Petri Nets 1987*. Lecture Notes in Computer Science 266, Springer-Verlag, 1987.

[RW89] R. Righter and J.C. Walrand. Distributed Simulation of Discrete Event Systems. *Proceedings of the IEEE*, 77(1):99–113, Jan 1989.

[Sch97] K. Schmidt. Verification of Siphons and Traps for Algebraic Petri Nets. In P. Azéma and G. Balbo, editors, *Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, pages 427–446, Toulouse, France, Jun 1997. Lecture Notes in Computer Science 1248, Springer.

[SdSSW95] P. Sénac, P. de Saqui-Sannes, and R. Willrich. Hierarchical Time Stream Petri Net: a Model for Hypermedia Systems. In G. De Michelis and M. Diaz, editors, *Proceedings of the 16th International Conference on Application and Theory of Petri Nets*, pages 451–470, Torino, Italy, Jun 1995. Lecture Notes in Computer Science 935, Springer.

[SS93] T.K. Som and R.G. Sargent. A New Process to Processor Assignment Criterion for Reducing Rollbacks in Optimistic Simulation. *Journal of Parallel and Distributed Computing*, 18(4):509–515, Aug 1993.

[SSW95] S. Schöf, M. Sonnenschein, and R. Wieting. Efficient Simulation of THOR Nets. In G. De Michelis and M. Diaz, editors, *Proceedings of the 16th International Conference on Application and Theory of Petri Nets*, pages 412–431, Torino, Italy, Jun 1995. Lecture Notes in Computer Science 935, Springer.

[Ste92] J.S. Steinman. SPEEDES: a Multiple-Synchronization Environment for Parallel Discrete-Event Simulation. *International Journal of Computer Simulation*, 2:251–286, 1992.

[Tan96] A. Tanenbaum. *Computer Networks.* Prentice-Hall, 3rd Edition, 1996.

[Tau88] D. Taubner. On the Implementation of Petri Nets. In G. Rozenberg, editor, *Advances in Petri Nets 1988*, pages 418–439, 1988. Lecture Notes in Computer Science 340, Springer-Verlag.

[TB93] C. Tropper and A. Boukerche. Parallel Simulation of Communicating Finite State Machines. In R. Bagrodia and D. Jefferson, editors, *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pages 143–150, San Diego, CA., May 1993.

[TLL$^+$98] S.J. Turner, C.C. Lim, Y.H. Low, W. Cai, W.J. Hsu, and S.Y. Huang. A Methodology for Automating the Parallelization of Manufacturing Simulations. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*, pages 126–133, Banff, Alberta, Canada, May 1998. IEEE. (available as Research Report No. R369, Department of Computer Science, University of Exeter). ftp://ftp.dcs.exeter.ac.uk/pub/parallel/simul/method.ps.Z.

[Tur96] L. Turcotte. *Cluster Computing*, chapter 26, pages 762–779. In Zomaya [Zom96], 1996.

[TZ91] G.S. Thomas and J. Zahorjan. Parallel Simulation of Performance Petri Nets: Extending the Domain of Parallel Simulation. In *Proceedings of the 1991 Winter Simulation Conference*, pages 564–573. B.Nelson, D.Kelton, G.Clark (Ed.), 1991.

[Van93] W.M.P. Van Der Aalst. Interval Timed Coloured Petri Nets and their Analysis. In Ajmone Marsan [Ajm93], pages 453–472.

[Vau87] J. Vautherin. Parallel System Specifications with Coloured Petri Nets and Algebraic Specifications. In Rozenberg [Roz87], pages 293–308.

[VdFC95] V. Valero, D. de Frutos, and F. Cuartero. Timed Processes of Timed Petri Nets. In G. De Michelis and M. Diaz, editors, *Proceedings of the 16th International Conference on Application and Theory of Petri Nets*, pages 490–509, Torino, Italy, Jun 1995. Lecture Notes in Computer Science 935, Springer.

[Vri90] R.C. De Vries. Reducing Null Messages in Misra's Distributed Discrete Event Simulation Method. *IEEE Transactions on Software Engineering*, 16(1):82–91, Jan 1990.

[WH87] M. Wilbur-Ham. Numerical Petri Nets, a Guide. Technical Report 111, Telecom Australia, Research Laboratory, 1987.

[WH94] H. Wabnig and G. Haring. Petri Net Performance Models of Parallel Systems - Methodology and Case Study. In Halatsis et al. [HMPT94], pages 301–312.

[WL89] D.B. Wagner and E.D. Lazowska. Parallel Simulation of Queueing Networks: Limitations and Potentials. In *Proceedings of the 1989 SIGMETRICS Conference*, pages 146–155, Berkeley, CA., 1989. ACM, New York.

[WM93] C.M. Woodside and G.G. Monforton. Fast Allocation of Processes in Distributed and Parallel Systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):164–174, Feb 1993.

[Zei91] B.P. Zeigler. Object-Oriented Modeling and Discrete-Event Simulation. In M.C. Yovits, editor, *Advances in Computers*, volume 33, pages 67–114. Academic Press, N.Y., 1991.

[Zom96] A.Y.H. Zomaya, editor. *Parallel and Distributed Computing Handbook.* McGraw-Hill, 1996.