Parallel Simulation of Petri Nets

by

Gregory Scott Thomas

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science

University of Washington

1991

Approved by _____
(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree Department of Computer Science and Engineering

Date _____

University of Washington

Abstract

Parallel Simulation of Petri Nets

by Gregory Scott Thomas

Chairperson of the Supervisory Committee: Professor John Zahorjan
Department of Computer Science
and Engineering

We present a parallel simulation protocol for performance Petri nets, Petri nets in which transition firings take randomly selected amounts of time. This protocol is interesting for two reasons. First, application of standard conservative or optimistic parallel simulation to Petri nets results in either unnecessarily low (possibly no) parallelism or simply fails to produce correct results. Thus, this new protocol may be thought of as addressing a class of models not amenable to standard parallel simulation, with Petri net models being a particular example. Second, performance Petri nets are currently analyzed using numerical techniques that have time and space requirements exponential in the size of the net. Simulation, and particularly parallel simulation, is thus a practical alternate analysis method for these models, as we show by measurement of execution times.

Our new protocol is derived from the rules of conservative parallel simulation. While this approach normally relies on use of the actual or "future" timestamps on the message paths into a component of the model to determine the latest simulation time to which that component can safely progress, we have introduced a new technique that loosens this restriction. Our technique, called *Selective Receive*, allows model components to sometimes ignore certain of their input channels and thus to determine their local clock times based on only a subset of their potential inputs. This technique is helpful in simulating Petri net models. We believe that it may be of use in speeding up the parallel simulation of other systems as well.

Finally, the development of a parallel simulation protocol for Petri nets suggested a small modification to their definition that, while not affecting the expressive power of the nets, allows for much more efficient simulations. We show that parallel simulations using this modification can achieve unboundedly high speedups relative to a sequential simulation that does not employ this approach.

Table of Contents

List of Figures v

List of Tables vii

Chapter 1: Introduction 1

1.1 Motivation 1

1.2 Goals 2

1.3 Related Work 4

1.4 Organization 4

Chapter 2: Petri Nets 6

2.1 Background 6

2.2 Basic Nets 7

2.3 Extensions 9

2.3.1 Standard 9

2.3.2 Nonstandard 13

2.4 Performance Petri Nets 15

2.5 Analysis 18

2.5.1 Problems 18

2.5.2 Complexity 19

2.5.3 Techniques 20
2.6 Tools 22
2.7 Applications 22
2.7.1 Examples from the Literature 23
2.7.2 Modeling Examples 23

Chapter 3: Parallel Simulation 33
3.1 Simulation 33
3.2 Parallel Simulation 34
3.3 Parallel Simulation and PPNs 35

Chapter 4: The Transition Firing Protocol 39

Chapter 5: Selective Receive and TFP 43
5.1 Place LP 45
5.2 Transition LP 46

Chapter 6: Conflict Resolution 48

Chapter 7: Implementation 51
7.1 Environment 51
7.2 Execution 52
7.3 Issues 54
7.3.1 Firing Delay Distributions 54
7.3.2 Lookahead 54
7.3.3 Termination 56

Chapter 8: Performance 57

8.1 Speedup Versus Number of Processors 58
8.2 Execution Time Versus Model Size 61

Chapter 9: Summary 67
9.1 Contributions 67
9.2 Areas for Further Work 68

Glossary 70

Bibliography 74

List of Figures

2.1 A net before firing. 7

2.2 A net after firing. 8

2.3 A net with a decision place. 8

2.4 A net with multiple arcs. 11

2.5 A net with arc multiplicities. 11

2.6 A net with an inhibitor arc. 12

2.7 A net with arc weights for conflict resolution. 13

2.8 A net with arc weights for deposit branching. 14

2.9 A net without deposit branching. 14

2.10 Central server model. 24

2.11 Central server net. 25

2.12 Communication channel model. 26

2.13 Communication channel net. 26

2.14 Multiprocessor model. 27

2.15 Multiprocessor net. 28

2.16 Net for database locking strategy ALL. 30

2.17 Net for database locking strategy ONE. 31

3.1 Taxonomy of parallel discrete-event simulation. 34

4.1 Overview of TFP. 40

5.1 State diagram for place LP. 45

5.2 State diagram for transition LP. 47

7.1 Use of Persephone. 53

8.1 Instruction pipeline net. 58

8.2 Measured speedups for Persephone on the PPN models. 59

8.3 Normalized execution time on CentralServer, varying jobs. 62

8.4 Normalized execution time on CentralServer, varying disks. . . . 63

8.5 Normalized execution time on Multiprocessor, varying processors. 63

8.6 Normalized execution time on Multiprocessor, varying memories. 64

List of Tables

2.1 Net token flow through a place due to an adjacent transition. . . 18

8.1 Number of conflict sets and maximum speedups of PPN models. 60

8.2 Execution times (seconds) of Persephone (1 processor) and Seq on DB models. 66

Acknowledgements

I am deeply indebted to my advisor, John Zahorjan, for his support and guidance. This research has benefited immensely from his effort, insight, and creativity. His indefatigable optimism ("How hard can this be?") provided a refreshing break from reality on numerous occasions, and his sense of humor was always welcome.

Ed Lazowska deserves special thanks for bringing me to the University of Washington, acting as my initial advisor, and serving on my supervisory committee.

Dave Wagner made Synapse available to us and answered a huge number of questions about it and about parallel simulation in general. Mary Vernon made GTPNA available to us. Tom Anderson assisted with the ideas behind the decentralized approach to conflict resolution and offered some analytic insight into its relationship to the standard centralized approach. Andreas Hagerer, Larry Landweber, Mike Molloy, Jerre Noe, and Louis Rosier each provided pointers into the literature. I am grateful to them for this assistance.

Typesetting a thesis is an arduous task, for which I received invaluable help from several people. Kathy Armstrong and John Faust provided LaTeX and BibTeX examples, and Reid Brown furnished his PostScript graphing package.

Raj Vaswani answered a number of questions regarding our Sequent Symmetry, as did Bob Sandstrom. Nancy Johnson Burr has been a continuous source of assistance for all types of systems issues. I appreciate their efforts.

Among the many gifted teachers with whom I have been blessed, I wish to particularly thank Dave Barton, Finbarr O' Sullivan, and Aram Thomasian, for taking a special interest in me while I was at Berkeley; and Linda Hudak, for her encouragement through the years. Additionally, Dev Chen has been an outstanding role model and friend.

Finally, I would not have reached this point without the love, support, encouragement, patience, and understanding of my parents, Gary and Sharon, and

the rest of my family. My biggest thanks goes to them.

This material is based upon work supported by the National Science Foundation (Grants DCR-8352098, CCR-8619663, and CCR-8703049), the Washington Technology Center, and Digital Equipment Corporation (the External Research Program and the Systems Research Center).

In memory of my grandfather,

Thomas F. Pepler,

1912–1982,

who was unable to share some of his grandchildren's brightest moments.

Chapter 1

Introduction

This thesis concerns the application of conservative parallel simulation to the analysis of performance Petri nets. In this chapter we motivate the topic, state the goals of our work, mention work related to ours, and detail the organization of the remainder of the thesis.

1.1 Motivation

Two primary considerations motivated our selection of this topic:

1. The high modeling power of Petri nets is tempered by their low decision power;[1] analytic analysis techniques are either nonexistent, computationally intractable, or of narrow applicability. Decision power can generally be improved by reducing modeling power. Instead, we wish to improve decision power while preserving modeling power by using a nonanalytic method of analysis.

2. To date, applications of parallel simulation have been limited to a few particular areas, e.g., queueing network models; the question of its suitability

[1] *Modeling power* refers to the ability to represent systems, and *decision power* refers to the ability to analyze models to answer questions about the systems they represent [Peterson 77, Peterson 81, Ciardo 87].

for other areas is of interest. We wish to extend the domain of parallel simulation to include Petri nets.

These considerations dovetail: simulation is a popular nonanalytic method of analysis, and Petri nets comprise an application with markedly different semantics than those of queueing network models.

We are interested in Petri nets as a tool for system performance analysis. Although performance Petri nets have seen considerable use in this role, their utility is limited by the shortcomings of the analytic techniques employed in their analysis; these techniques are of exponential complexity or are applicable only to restricted classes of nets. Simulation is an attractive alternative to these analytic methods because it is likely to be of smaller complexity and applicable to unrestricted classes of nets.

Simulation is not, however, itself devoid of shortcomings, one of which is its propensity toward voracious consumption of computational resources. *Parallel* simulation strives to mitigate this shortcoming by engaging multiple processors to decrease execution time. Consequently, it is our method of choice. Furthermore, we focus on *conservative* parallel simulation because we expect its performance characteristics for this application to be more stable (and often better) than those of an optimistic approach.

1.2 Goals

Our intent is to facilitate the analysis of nets which are larger and more complex than those amenable to analytic techniques. The specific goals of this work may be summarized as follows:

1. Devise a conservative parallel simulation protocol, known as the *Transition Firing Protocol* (TFP), which correctly simulates any net belonging to the class of Petri nets defined in Section 2.4, and which provides parallelism that scales with the size of the net.

2. Determine the amenability of performance Petri nets to conservative parallel simulation by implementing TFP and measuring its performance.

3. Compare the computational complexity of conservative parallel simulation of performance Petri nets to that of analytic techniques, again by measurement.

For reasons that shall be expounded in Chapter 3, only certain decompositions of Petri nets map directly to the standard paradigm of conservative parallel simulation. Other decompositions require a new paradigm; the first goal is to create such a paradigm.

The construction of *Persephone*, a Petri net simulator that implements TFP, is central to the second goal. Persephone serves several purposes:

- It is a proof by existence that conservative parallel simulation can be applied to Petri nets.
- It provides a vehicle for experimenting with, testing, and debugging the protocol.
- It enables concrete measurements of performance and complexity to be made.
- It is an analysis tool for systems and Petri nets that is useful in its own right.

Attainment of the first goal and the implementation of Persephone affirmatively answers the question of whether conservative parallel simulation *can* be used to analyze Petri nets; the second goal is to answer the question of whether it can be used *effectively*. For parallel simulation to be considered effective in a problem domain, its use of multiple processors (think of this as an "investment") must provide some benefit (a "return") over the use of a single processor by a sequential simulation. The chief benefit sought is a reduction of execution time in direct proportion to the number of processors applied. For example, ideally

a parallel simulation running on ten processors should execute ten times faster than a sequential simulation of the same model. Therefore, the investigation comprising the second goal is undertaken by measuring the performance of our simulator on a set of benchmark nets.

While the second goal concerns the issue of parallel versus sequential simulation, the thrust of the third goal is simulation versus analytic techniques: how execution time grows as model complexity increases.

1.3 Related Work

TFP resembles in some ways several of the protocols developed by Taubner [Taubner 88]. That work was performed in the context of "Petri net driven execution" of distributed programs—the firing of a transition causes the invocation of a procedure, with the Petri net itself used to determine the flow of control (e.g., [Hartung 88])—rather than in the context of simulation of performance Petri nets.

Our work differs significantly from Taubner's in at least three ways. First, Taubner assumes untimed nets, so there is no notion of simulation time. Second, our conflict resolution strategy is different from each of those proposed by Taubner. Finally, because each transition firing results in a procedure execution, the amount of overhead required to run the protocol is less important for execution of distributed programs than it is in our simulation context, where a transition firing involves very little inherent work. Thus, we have been more sensitive to these overheads in designing our protocol and prototype implementation.

1.4 Organization

Chapter 2 discusses the components and semantics of Petri nets. It defines the particular class of nets we use, surveys analysis methods and tools, and describes how Petri nets are applied. Benchmark nets are introduced in modeling examples. Chapter 3 characterizes different simulation paradigms and discusses

various approaches to applying parallel simulation to Petri nets.

An overview of TFP is presented in Chapter 4. TFP exploits a fundamental modification to the standard algorithm for conservative parallel simulation. This modification, which we call *Selective Receive*, is described in Chapter 5.

In Chapter 6 we discuss one aspect of performance Petri net analysis, conflict resolution, for which a slight redefinition of the model semantics suggested by the parallel approach leads to greatly improved analysis performance.

Chapter 7 discusses the implementation of Persephone. Chapter 8 reports the performance of Persephone on a group of benchmark nets and compares it to that of the analytic techniques. Chapter 9 concludes by summarizing the contributions of this thesis and suggesting areas for further work.

Chapter 2

Petri Nets

This chapter serves as an introduction to Petri nets. After discussing the components and semantics of basic nets, various standard and nonstandard extensions of basic nets are presented. These extensions are incorporated in the class of nets we use, which is defined in Section 2.4. The ensuing sections describe methods of analysis, tools for analysis, and how Petri nets are applied. The chapter concludes with examples in which Petri net models of systems are constructed; these nets later serve as benchmarks for performance measurements.

For more extensive introductions and bibliographies, refer to [Peterson 77, Peterson 81, Reisig 85, Murata 89]. Additionally, [Drees et al. 87] is a bibliography containing over two thousand entries.

2.1 Background

Petri nets are a modeling tool that is both graphical and mathematical [Murata 89]. This thesis is concerned primarily with their graphical aspects, because it is topology that has the dominant impact on simulation.

Petri nets have evolved from the ideas in C. A. Petri's dissertation [Petri 62]. They are the focus of *special net theory*, which is a branch of *general net theory* [Petri 80]. Special net theory deals with flow phenomena in single nets; net morphisms and relations between nets are the gist of general net theory.

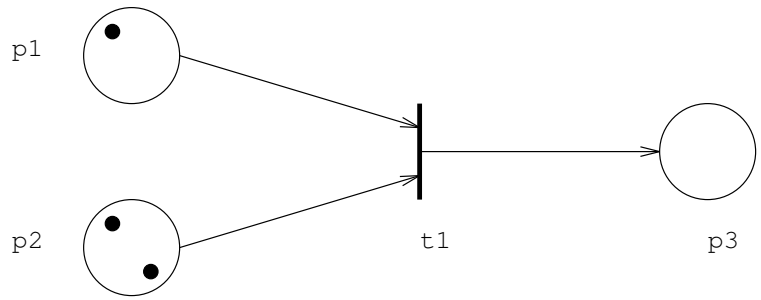

Figure 2.1: A net before firing.

2.2 Basic Nets

There are many kinds of Petri nets, and different ways to represent them. Our nets most closely resemble *place/transition-nets* [Genrich & Stankiewicz-Wiechno 80] without capacities, and we adopt a predominantly graphical representation.

A Petri net is a bipartite[1] directed graph. The two types of nodes are called *places* and *transitions*. Places are drawn as circles, and transitions are drawn as bars or rectangles. When expedient, nodes may be labeled. Arcs are directed from places to transitions or from transitions to places. If there is an arc (p, t) from a place p to a transition t, we say that p is an *input place* of t and that t is an *output transition* of p. Similarly, if there is an arc (t, p) from t to p, we say that p is an *output place* of t and that t is an *input transition* of p.

Zero or more *tokens* reside at each place. Tokens are drawn as small filled circles (dots) inside places; alternatively, the number of tokens at a place may be drawn inside the place.

The number of tokens at a place is called the *marking* of the place. The marking of a net is given by the vector of markings of its places, and may be interpreted as the state of the net.

The net in Figure 2.1 consists of three places (`p1`, `p2`, and `p3`) and one tran-

[1] [Petri 87] contains an interesting discourse on the duality implications of this fact.

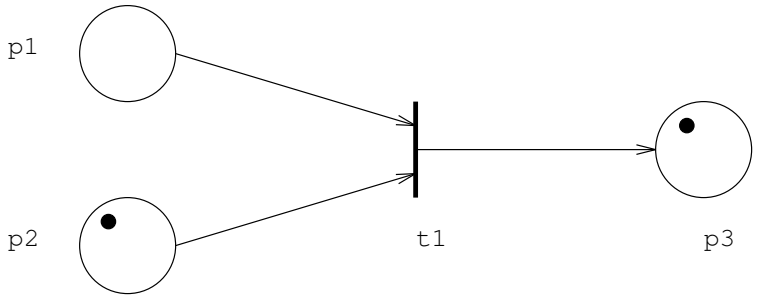

Figure 2.2: A net after firing.

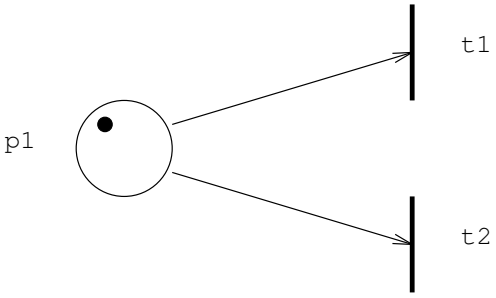

Figure 2.3: A net with a decision place.

sition (`t1`). Transition `t1` has input places `p1` and `p2`, and output place `p3`. `p1` contains one token, `p2` contains two tokens, and `p3` contains zero tokens, so the marking of the net is the vector (1, 2, 0).

A transition is *enabled* when each of its input places contains at least one token. An enabled transition may *fire*, removing a token from each input place and depositing a token at each output place. Each transition firing produces a new marking.

In Figure 2.1, transition `t1` is enabled. Figure 2.2 depicts the net in Figure 2.1 after the completion of the firing of `t1`.

Transition firings are not always so simple. In Figure 2.3, `p1` is a *decision* place—a place which is a source for more than one regular[2] arc. Whenever there is a token at `p1`, `t1` and `t2` are *in conflict* because the firing of one disables the

[2]As explained in Section 2.3.1, a regular arc is an arc that is not an inhibitor arc.

other.

Let P be the set of places of a net, $p_i \in P$, $\mathrm{OT}(p_i)$ the set of output transitions of p_i, and R^* the reflexive transitive closure of the relation $R \subseteq P^2$, defined by $p_i R p_j$ if and only if $\mathrm{OT}(p_i) \cap \mathrm{OT}(p_j) \neq \emptyset$. R^* partitions P into a set of equivalence classes (called *locksets* in [Taubner 88]). The places in such an equivalence class, along with their output transitions, constitute a *static conflict set*. It is possible for multiple transitions in a static conflict set to become enabled at the same time, but for only some subset of them to begin firing at that time (due to the limited number of tokens available in the decision places of the static conflict set).

A *dynamic conflict set* is a set of transitions (and their associated input places) that are actually enabled and competing for tokens (at decision places) at some point in time. A dynamic conflict set is thus a subset of some static conflict set.

When a dynamic conflict set exists, a subset of its transitions must be chosen to fire. This is called *conflict resolution*. A number of different proposals have been made for selecting the subset to fire. We use a technique modeled on that proposed for Generalized Timed Petri Nets [Holliday & Vernon 87], which uses the notion of *maximal sets*. A maximal set is a set of transitions within a dynamic conflict set such that, if those transitions were to fire, no other transitions in the conflict set could also fire. In general, there may be many distinct maximal sets corresponding to a particular dynamic conflict set.

2.3 Extensions

The nets described in the previous section have limited modeling power. To remedy this, a number of extensions of Petri nets have been proposed. These extensions, as well as some novel ones, are the topics of the next sections.

2.3.1 Standard

The extensions presented in this section have been widely adopted, either out of necessity (e.g., time is essential for performance evaluation), or for the sake of convenience.

Time

As formulated originally, Petri nets lack any timing information. The notion of time is most commonly introduced by associating a delay with each transition or with each place.[3] The two approaches are equivalent [Sifakis 80], so the one most natural for the intended application should be chosen. In this thesis, only the approach which associates delays with transitions is considered.

The introduction of delays changes the semantics of firing so that instead of tokens being deposited in a 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. Let δ be such a chosen delay of transition t, which begins firing at time τ. Tokens are removed from the input places of t at τ, and deposited in the output places of t at $\tau + \delta$. More than one firing of t may be in progress concurrently.

Furthermore, the state of a net augmented with time is not completely described by the marking of the net. The remaining firing times of all firings in progress constitute the additional information necessary to fully characterize the state of the net.

Nets can be classified according to whether the delays are deterministic or nondeterministic. Nets having deterministic delays are called *timed* Petri nets; those having nondeterministic delays are called *stochastic* Petri nets. Stochastic nets with exponentially or geometrically distributed delays are isomorphic to homogeneous Markov chains, making them amenable to standard Markovian

[3] [Petri 87] illustrates several semantic difficulties engendered by the introduction of time to nets.

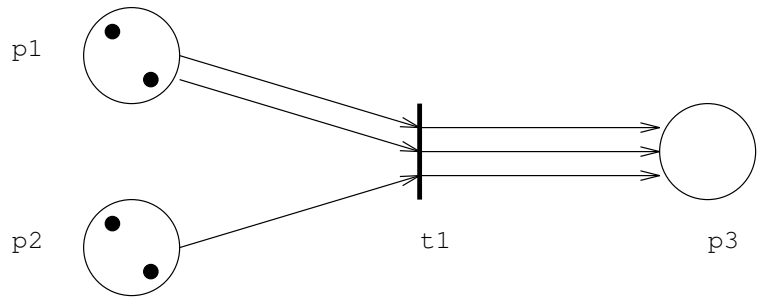

Figure 2.4: A net with multiple arcs.

analysis techniques [Hoel et al. 72]. Although timed nets complicate analysis because they lack the memoryless property possessed by the exponential and geometric distributions, there do exist models which allow both deterministic and nondeterministic delays in a single net [Molloy 85].

From the point of view of simulation, however, the distributions of the delays are of little consequence. One of the advantages that simulation has over analytic techniques is that it allows arbitrary distributions to be used. Section 7.3.1 lists the distributions supported by Persephone.

Arc Multiplicity

It is often convenient to allow multiple tokens to flow between two nodes at each firing. This is represented by multiple arcs between a given source and sink, as shown in Figure 2.4. There must be at least two tokens at `p1` and at least one token at `p2` to enable `t1`, and three tokens are deposited at `p3` when `t1` fires. Figure 2.5 shows the net in Figure 2.4 after the firing of `t1`. It also illustrates how multiple arcs are depicted as a single arc with an attached *multiplicity*, a convention borrowed from circuit diagrams. If not labeled, an arc is assumed to have multiplicity 1.

For place p and transition t, an arc (p, t) of multiplicity m means that m tokens are required at p to enable t. An arc (t, p) of multiplicity m means that each time t finishes firing it deposits m tokens at p.

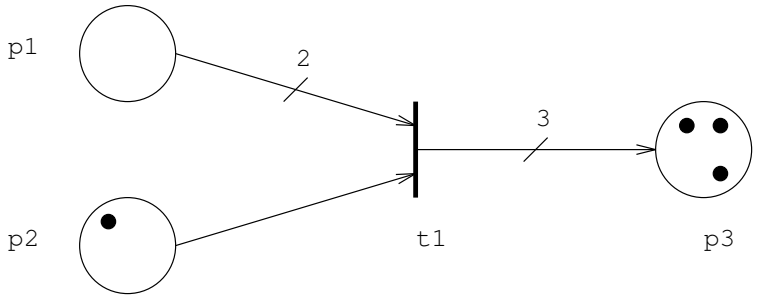

Figure 2.5: A net with arc multiplicities.

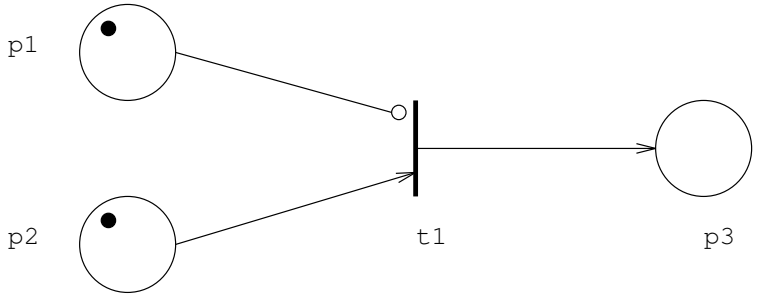

Figure 2.6: A net with an inhibitor arc.

Petri nets in which arcs can have multiplicities greater than one are called *generalized* Petri nets. They are equivalent to nets in which arc multiplicities must equal one [Peterson 77].

Inhibitor Arcs

The ability to determine that a place has zero tokens is useful, and is facilitated by an extension known as *inhibitor* arcs. In Figure 2.6, the inhibitor arc from `p1` to `t1` indicates that `t1` cannot fire unless there are zero tokens at `p1`; since there is a token at `p1`, `t1` is inhibited from firing. The small circle which replaces the arrowhead is intended to suggest inversion, another convention borrowed from circuit diagrams. When it is necessary to distinguish inhibitor arcs from other arcs, the latter shall be referred to as *regular* arcs.

The addition of inhibitor arcs fundamentally changes the modeling power and the decision power of Petri nets. Petri nets with inhibitor arcs are equivalent to

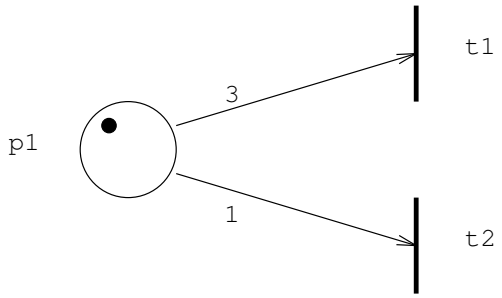

Figure 2.7: A net with arc weights for conflict resolution.

Turing machines [Peterson 77].

2.3.2 Nonstandard

In this section we describe extensions which are novel or which have not been widely adopted.

Arc Weights

Each regular arc may have a *weight* associated with it. The weight is interpreted differently depending on whether the source of the arc is a place or a transition.

Conflict Resolution If the arc's source is a place, then the weight is used to resolve conflicts at that place, as explained in Section 2.2 and Chapter 6. The basic idea is that a higher weight gives a higher priority. In Figure 2.7, for example, `p1` will give a token to `t1` three times as often as it gives a token to `t2`. If not labeled, an arc is assumed to have weight 1; therefore, if all output arcs of a place have equal weights, no weights need be specified.

Deposit Branching If the arc's source is a transition, then the weight is used for *deposit branching* at that transition. Deposit branching allows a transition to deposit tokens in an output place chosen at random each time the transition fires, rather than depositing tokens in each output place. Let $S = S_U \cup S_W$

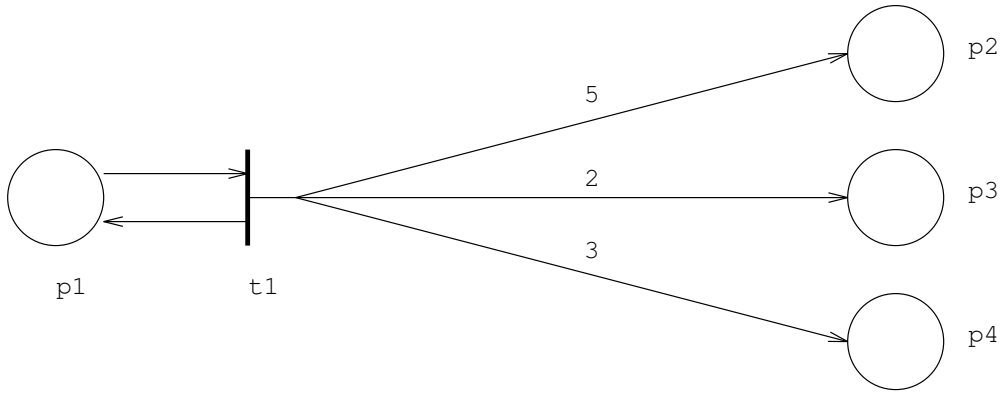

Figure 2.8: A net with arc weights for deposit branching.

denote the set of output arcs for a transition t, with S_U being the unweighted arcs and S_W being the weighted arcs. When t fires, it deposits tokens on each arc in S_U, and randomly chooses one of the arcs in S_W to deposit tokens on. In Figure 2.8, $S_W = \{(\texttt{t1}, \texttt{p2}), (\texttt{t1}, \texttt{p3}), (\texttt{t1}, \texttt{p4})\}$, and $S_U = \{(\texttt{t1}, \texttt{p1})\}$; when `t1` fires, it deposits a token at `p1`, and chooses either `p2`, `p3`, or `p4` (with respective probabilities 0.5, 0.2, and 0.3) to deposit a token at.

Members of S_W are distinguished graphically by having them branch from a single segment emanating from the source. As with conflict resolution weights, deposit branching weights are assumed to be 1 if not specified.

Deposit branching allows some models to be expressed by simpler nets—it does not change the modeling or decision power of Petri nets. This is illustrated by Figure 2.9, a model without deposit branching which is equivalent to the model in Figure 2.8. It uses the decision place `p5` and the three zero-delay transitions `t1`, `t2`, and `t3` to achieve the same effect as that achieved by `t1` with deposit branching in Figure 2.8. In general, an n-way deposit branch replaces one decision place, n zero-delay transitions, and $n + 1$ arcs.

Generalized Inhibitor Arcs

Consider a regular arc r of multiplicity m directed from place p to transition t. The constraint imposed on t by r is that t can fire only if there are m or more

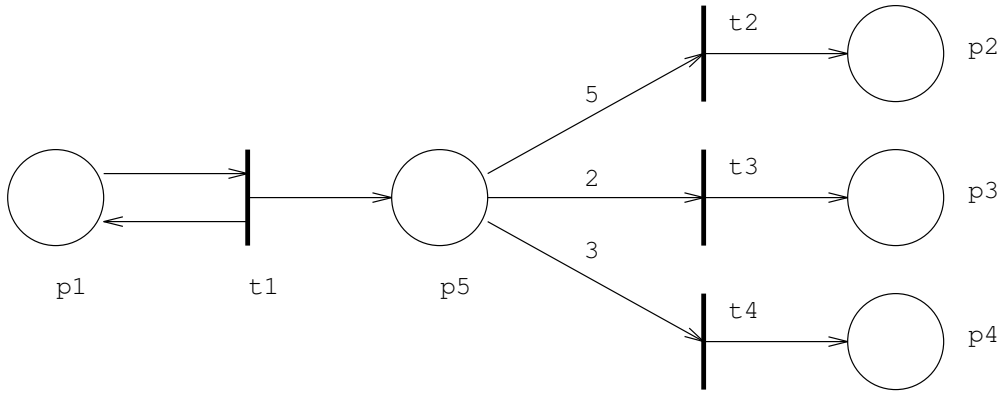

Figure 2.9: A net without deposit branching.

tokens at p. The complementary constraint—that t can fire only if there are *fewer than* m tokens at p—can be expressed by *generalized* inhibitor arcs.

Generalized inhibitor arcs combine inhibitor arcs and arc multiplicities in a natural way. An ordinary inhibitor arc (i.e., an inhibitor arc of multiplicity 1) prevents its sink from firing if there is at least 1 token at its source. A generalized inhibitor arc of multiplicity m prevents its sink from firing if there are at least m tokens at its source. This definition is symmetric to that of generalized regular arcs: a generalized regular arc (from place to transition) of multiplicity m *allows* its sink to fire if there are at least m tokens at its source. As with ordinary inhibitor arcs, the firing of a generalized inhibitor arc's sink consumes no tokens from its source. If not labeled, an inhibitor arc is assumed to have multiplicity 1.

2.4 Performance Petri Nets

Petri nets augmented with delays and branching probabilities to facilitate the derivation of performance measures have been called *performance* Petri nets (PPNs) [Vernon et al. 86]. Definition 2.1 specifies the particular class of PPNs we use. It incorporates the extensions described in Section 2.3, but is otherwise similar to classes of PPNs appearing in the literature. In the sequel, all references to PPNs refer to this class unless noted otherwise. (Petri nets that are not PPNs are sometimes referred to as *classical* or *untimed* Petri nets.)

[Ajmone Marsan & Chiola 87] is recommended as an introduction to PPNs.

Definition 2.1 (Performance Petri Net) A *performance Petri net* N is a 7-tuple:

$$N = (P, T, A, U, W, \Delta, M_0)$$

P	$=$	$\{p_1, p_2, \ldots, p_n\}$	places
T	$=$	$\{t_1, t_2, \ldots, t_m\}$	transitions
A_R	$\subseteq$	$\{P \times T\} \cup \{T \times P\}$	regular arcs
A_I	$\subseteq$	$\{P \times T\}$	inhibitor arcs
A	$=$	$A_R \cup A_I$	arcs
U	$:$	$A \rightarrow \{1, 2, \ldots\}$	arc multiplicities
W	$:$	$A_R \rightarrow \mathcal{R}$	arc weights
Δ	$=$	$(\delta_1, \delta_2, \ldots, \delta_m)$	firing delay distributions
M_0	$=$	$(\mu_1^0, \mu_2^0, \ldots, \mu_n^0)$	initial marking

P and T are finite sets. U is a mapping that assigns to each $a \in A$ its multiplicity $U(a)$. Similarly, W assigns to each $a \in A_R$ its weight $W(a)$. If $a \in \{P \times T\}$, then $W(a) \geq 0$, but if $a \in \{T \times P\}$, $W(a)$ can be negative; $W(a) < 0$ implies that a does not participate in deposit branching—it is "unweighted."

Δ can be viewed as a mapping from a transition t_i to its firing delay distribution δ_i, i.e., $\Delta(t_i) = \delta_i$. Each δ_i is a probability distribution, e.g., uniform(a, b). Unlike many other definitions of PPNs, our PPNs are not restricted to having only a certain kind of distribution.

M_0 assigns to each $p_i \in P$ its initial marking μ_i^0. Similarly, for any marking M, the mapping $M : P \rightarrow \{0, 1, \ldots\}$ is defined as $M(p_i) = \mu_i$. Further mappings are given in Definition 2.2.

Definition 2.2 (Node I/O Set Functions) The following functions each map a node of a performance Petri net N to a subset of its input or output nodes.

$I_{PR}(t)$	=	$\{p \mid (p,t) \in A_R\}$	input places via regular arcs
$I_{PI}(t)$	=	$\{p \mid (p,t) \in A_I\}$	input places via inhibitor arcs
$I_P(t)$	=	$\{p \mid (p,t) \in A\}$	input places
$I_T(p)$	=	$\{t \mid (t,p) \in A\}$	input transitions
$O_{PU}(t)$	=	$\{p \mid (t,p) \in A \wedge W((t,p)) < 0\}$	output places via unweighted arcs
$O_{PW}(t)$	=	$\{p \mid (t,p) \in A \wedge W((t,p)) \geq 0\}$	output places via weighted arcs
$O_P(t)$	=	$\{p \mid (t,p) \in A\}$	output places
$O_{TR}(p)$	=	$\{t \mid (p,t) \in A_R\}$	output transitions via regular arcs
$O_{TI}(p)$	=	$\{t \mid (p,t) \in A_I\}$	output transitions via inhibitor arcs
$O_T(p)$	=	$\{t \mid (p,t) \in A\}$	output transitions

The value of zero or more of these functions may be $\emptyset$ for zero or more nodes.

The rules for transition enabling and firing can now be stated.

Definition 2.3 (Transition Enabling) A transition $t \in T$ is *enabled* if and only if

$$(\forall p \in I_{PR}(t), M(p) \geq U((p,t))) \bigwedge (\forall p \in I_{PI}(t), M(p) < U((p,t))).$$

Definition 2.4 (Transition Firing) Define the indicator function $\beta : T \times P \rightarrow \{0,1\}$ by

$$\beta(t,p) = \begin{cases} 1, & (t,p) \in O_{PW}(t) \text{ and} \\ & (t,p) \text{ is the arc selected by } t \text{ to receive the deposit} \\ 0, & \text{otherwise} \end{cases}$$

Let M_1 be the marking of N at the start of the firing of transition $t \in T$, and M_2 the marking at the end of the firing of t. Assuming no transitions fire concurrently, the effect of the firing of t is that $\forall p \in P$,

$$M_2(p) = M_1(p) + \phi(p,t),$$

where $\phi(p,t)$ is the net token flow through p due to t and is given in Table 2.1. It is necessary to use net flow rather than unidirectional flow because of the possibility that p is both an input and output place of t.

Table 2.1: Net token flow through a place due to an adjacent transition.

$(p,t) \in I_{PR}(t)$	$(t,p) \in O_{PU}(t)$	$(t,p) \in O_{PW}(t)$	$\phi(p,t)$
no	no	no	0
no	no	yes	$U((t,p)) * \beta(t,p)$
no	yes	no	$U((t,p))$
yes	no	no	$-U((p,t))$
yes	no	yes	$-U((p,t)) + U((t,p)) * \beta(t,p)$
yes	yes	no	$-U((p,t)) + U((t,p))$

2.5 Analysis

After a Petri net model of a system has been constructed, the net must be analyzed to determine properties of the model. Analysis problems, their complexities, and techniques for solving them are discussed in the next sections.

2.5.1 Problems

From their inception, Petri nets have seen widespread use as a tool for verifying the correctness of systems. For example, analysis of a Petri net model of a system can show that the system cannot deadlock. To address various correctness issues, a number of analysis problems for classical Petri nets have been formulated [Peterson 81, Murata 89], several of which are paraphrased below.

Reachability A marking M' is *immediately reachable* from a marking M if $\exists t \in T$ such that the firing of t transforms M to M' (written $M \xrightarrow{t} M'$). The "reachable" relation is the reflexive transitive closure of the "immediately reachable" relation: M' is *reachable* from M if $\exists$ a sequence of transition firings $\sigma = t_1 t_2 \cdots t_k$ that transforms M to M' (written $M \xrightarrow{\sigma} M'$). The *reachability set* $R(N, M)$ of net N with marking M is the set of all markings reachable from M. The *reachability problem* is to determine, given a marking M, if $M \in R(N, M_0)$. Many other Petri net problems can be stated in terms of this basic problem.

Liveness A net N is *live* if $\forall t \in T,\ \forall M \in R(N, M_0),\ \exists \sigma\ \exists M'$ such that $M \xrightarrow{\sigma} M'$ and t is enabled in M'. Liveness can be used to verify the absence of deadlock in a system.

Boundedness A net N is *k-bounded* if $\forall p \in P,\ \forall M \in R(N, M_0),\ M(p) \leq k$. Boundedness is important for modeling components having finite capacities, e.g., buffers. A bounded net has a finite reachability set, which is why boundedness is a property crucial for analytic techniques requiring a finite state space.

Conservation A net N is *strictly conservative* if $\forall M \in R(N, M_0)$, $\sum_{p \in P} M(p) = \sum_{p \in P} M_0(p)$. N is conservative with respect to a weighting vector $\omega = (\omega_1, \omega_2, \ldots, \omega_n), n = |P|, \omega_i > 0$, if $\forall M \in R(N, M_0)$, $\sum_i \omega_i \cdot M(p_i) = \sum_i \omega_i \cdot M_0(p_i)$. Conservation is helpful for modeling resources which are neither created nor destroyed.

More recently, PPNs have been used to model systems in which time plays a critical role. Generally we are interested not only in the correctness of such a system, but also in its performance. Thus, in addition to solving the problems stated above, for PPNs we want to determine properties such as the average number of tokens at a place or the rate at which a transition fires [Molloy 85, Murata 89].

2.5.2 Complexity

The extreme difficulty of many Petri net analysis problems has led researchers to restrict the classes of nets they analyze in the hope that relatively efficient solution techniques can be found for certain subclasses of nets. Representative examples of complexity results for classical Petri net analysis problems are:

- The problem of determining whether the reachability set of one net is a subset of the reachability set of another net is undecidable.
- The preceding problem, but for *bounded* reachability sets, is of nonprimitive recursive complexity.

- The reachability problem requires at least exponential space and time.
- The boundedness problem requires at least exponential space and time.

Details appear in [Peterson 81, Jantzen 87].

Although these results were derived for untimed nets, they are applicable to performance nets as well. In particular, the exhaustive enumeration algorithms used to solve reachability-type problems are of exponential complexity regardless of whether or not the net is timed, because the worst-case size of the reachability set is exponential in the size of the net. More importantly, the Markovian analysis techniques applied to PPNs are also of exponential complexity, as demonstrated vividly by the measurements presented in Section 8.2.

2.5.3 Techniques

Nearly all techniques for solving Petri net analysis problems are analytic. A *non*analytic technique which may be applied to certain problems is simulation. These techniques are the subjects of the next sections.

Analytic

Analytic methods are important because they provide exact solutions. Moreover, to solve any of the correctness problems mentioned in Section 2.5.1, an analytic method *must* be used—simulation cannot be used to *prove* properties of models.

There are three categories of analytic techniques for solving classical Petri net analysis problems [Peterson 81, Murata 89]:

- The *coverability tree* algorithm exhaustively constructs a tree of reachable markings. For a bounded net the coverability tree contains all possible reachable markings, so it is called the *reachability tree*. For an unbounded net the reachability set is infinite; the coverability tree employs a representation which keeps it finite but loses information.

Numerous problems, including those in Section 2.5.1, can be solved using the reachability tree. The coverability tree, because of the information lost, cannot solve as many problems; of those in Section 2.5.1, it can solve only boundedness and conservation.

The two major drawbacks to this approach are its exponential complexity and the loss of information.

- In the *matrix equation* approach, nets are represented by matrices. This allows many problems, e.g., conservation, to be formulated as matrix equations.

  This approach possesses two serious deficiences. The first is that solutions give no information about the order of transition firings. The second is that some solutions correspond to transition firing sequences which are not possible.

- The *reduction* method 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 analyzed by one of the standard (expensive) techniques. Thus, this method by itself does not solve problems, but is used in conjunction with other methods to solve problems.

The basic analytic technique for solving PPN analysis problems involves computing the steady-state probability distribution of the underlying Markov chain [Molloy 85, Murata 89]. The states of the Markov chain correspond to the markings in the reachability set, so this technique suffers from the same exponential complexity that plagues the coverability tree approach.

Simulation

Although simulation is ill suited for classical analysis problems, it is especially well suited for the performance analysis domain. Its most attractive feature is its broad applicability: it can handle nets of virtually any size, nets having

arbitrary firing delay distributions, and nets incorporating arbitrary extensions. Furthermore, simulation can easily provide transient solutions, while analytic techniques supply only steady-state solutions.

The major disadvantage of simulation is that it does not, in general, provide exact solutions. Depending on the application, this may or may not be important (e.g., if the additional inaccuracy introduced by simulation is insignificant compared to the inaccuracy inherent in the model, then exact solutions are unlikely to be important).

2.6 Tools

The difficulty of Petri net analysis problems precludes analyzing all but the most minuscule nets by hand; automated tools are necessary. Issues involved in building tools for the creation and analysis of Petri nets are treated in [Jensen 87]. Dozens of such tools exist; summaries of a number of them appear in [Feldbrugge 86, Feldbrugge & Jensen 87].

A prominent example of a performance-oriented tool is the Generalized Timed Petri Net Analyzer (GTPNA) [Holliday & Vernon 86]. Among the extensions to classical Petri nets made by Generalized Timed Petri Nets (GTPNs) are arc multiplicities and state-dependent transition firing durations and frequencies [Holliday & Vernon 87]. GTPNA builds the reachability graph of a bounded net and then computes performance measures from the Markov chain associated with the graph. Optimization is performed to reduce the state space, but the complexity is still exponential in the worst case.

2.7 Applications

Uses of Petri nets include analysis, design, modeling, performability, performance evaluation, representation, specification, synthesis, validation, and verification. These uses are not mutually exclusive; for example, a system might be designed using Petri nets, which can be analyzed to verify that the system possesses certain

properties. This versatility is one of the strengths of Petri nets.

In the following sections we mention areas in which Petri nets have been applied, and present a series of modeling examples.

2.7.1 Examples from the Literature

Petri nets have been applied in a multitude of domains. Areas of application reported in the literature include cache coherence protocols [Girault et al. 87], communication protocols [Diaz 87], computer aided software engineering [Baldassari & Bruno 88], credit authorization systems [Hildebrand 85], databases [Voss 87a], distributed systems [Sanders & Meyer 87], fault tolerant systems [Lu et al. 87], finite state machines [Peterson 81], flexible manufacturing systems [Alla et al. 85], high-performance computer architecture [Baer 87], human-machine interaction [Oberquelle 87], the λ-calculus [Meijer 87], local area networks [Ajmone Marsan et al. 87], logic [Genrich et al. 80], music description and processing [Haus & Rodriguez 88], natural language representation [Koseska-Toszewa & Mazurkiewicz 88], office automation [Voss 87b], parallel systems [Vautherin 87], production systems [Valette 87], programming language semantics [Jensen & Schmidt 86], project planning and scheduling [Peterson 81], real-time control systems [Barke 90], replicated file systems [Dugan & Ciardo 87], scheduling problems [Carlier et al. 85], software engineering [Reisig 87], and telecommunication systems [Lopez & Palaez 88].

The preceding 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 may be found in [Peterson 77, Peterson 81, Reisig 85, Murata 89].

2.7.2 Modeling Examples

In this section we demonstrate the use of PPNs in the modeling of systems via four examples. The PPNs we present are named CentralServer, CommChannel, Multiprocessor, DB All-At-Once, and DB One-At-A-Time, respectively.

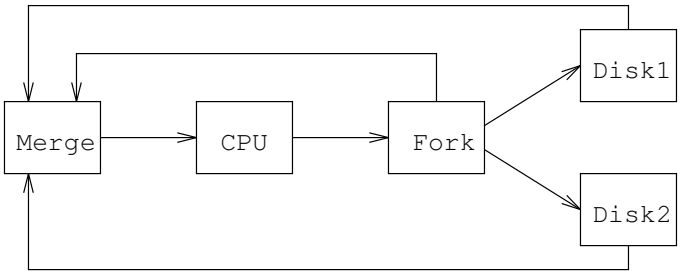

Figure 2.10: Central server model.

To reduce the clutter of the nets, we replace two arcs by one two-headed arc between place p and transition t when p is both an input and output place of t or, more precisely, when $p \in I_{PR}(t) \wedge p \in O_{PU}(t) \wedge U((p,t)) = U((t,p))$.

Central Server

Figure 2.10 shows a model of a central server system useful for the performance evaluation of a simple time-shared computer system. This is an application often addressed using queueing network models. We include it here because much of the work on parallel simulation techniques has used queueing network models (and especially this one) as an application [Reed et al. 88, Nicol 88, Wagner & Lazowska 89, Greenberg & Lubachevsky 90] and because queueing network models are easily represented by PPNs. In this model, after a job receives service at `CPU` it moves with equal probability to `Disk1`, `Disk2`, or back to `CPU`; after being serviced by a disk, a job returns to `CPU`.[4] It may be used, for example, to answer questions about the effect of configuration on job performance.

A PPN that represents the central server model is depicted in Figure 2.11. Each (node, output arc) pair in the model is realized by a (place, transition) pair and associated arcs in the net. The places `CPUFree`, `Disk1Free`, and `Disk2Free` are required to model sequential access to the respective devices. Tokens repre-

[4] The `Fork` and `Merge` nodes are required by the RESQ [Sauer et al. 80] queueing network model specification language used in [Reed et al. 88].

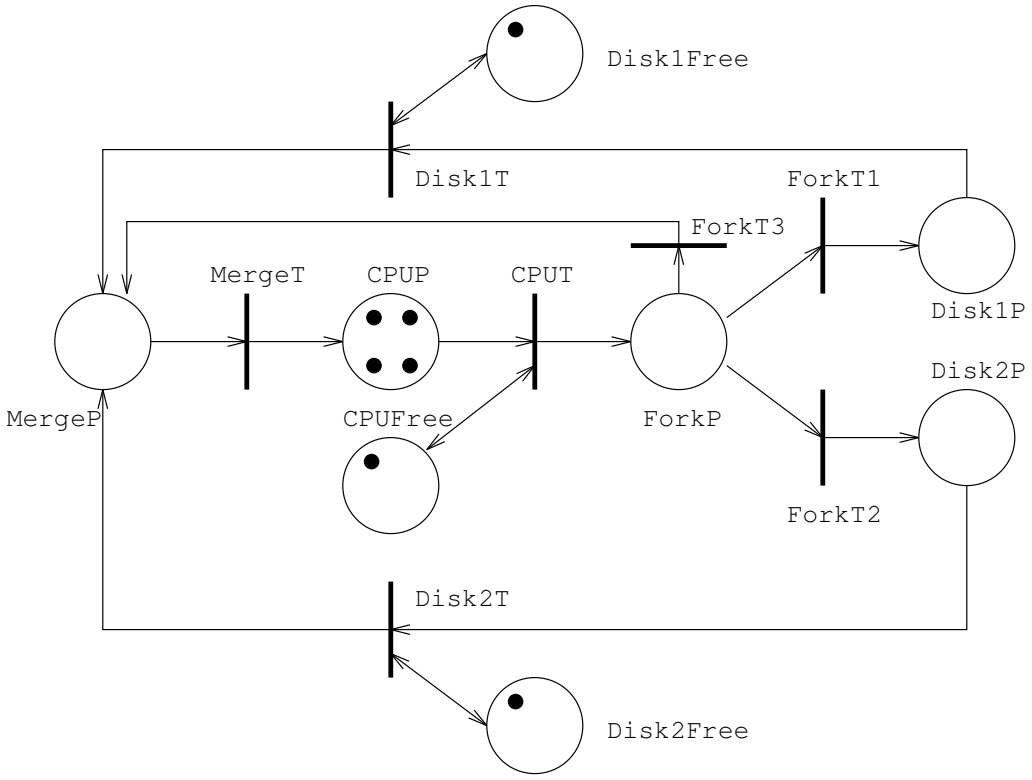

Figure 2.11: Central server net.

sent jobs, with the initial marking determined by the initial distribution of the job population. Transition delays correspond to the service times of the nodes.

Communication Channel

Consider a system in which users on one host (`HostA`) generate messages to be sent to another host (`HostB`). The messages are decomposed into transmission units called *frames* and then transmitted over an errorless full-duplex communication channel. `HostB` replies to each frame it receives by transmitting an acknowledgement (ACK) to `HostA`. At most one frame and one ACK can be on the channel at one time. A *window flow control* protocol is used; a window of size w allows `HostA` to send up to w unacknowledged frames. Such a model might be used to evaluate how the window size (which is related to the number of buffers

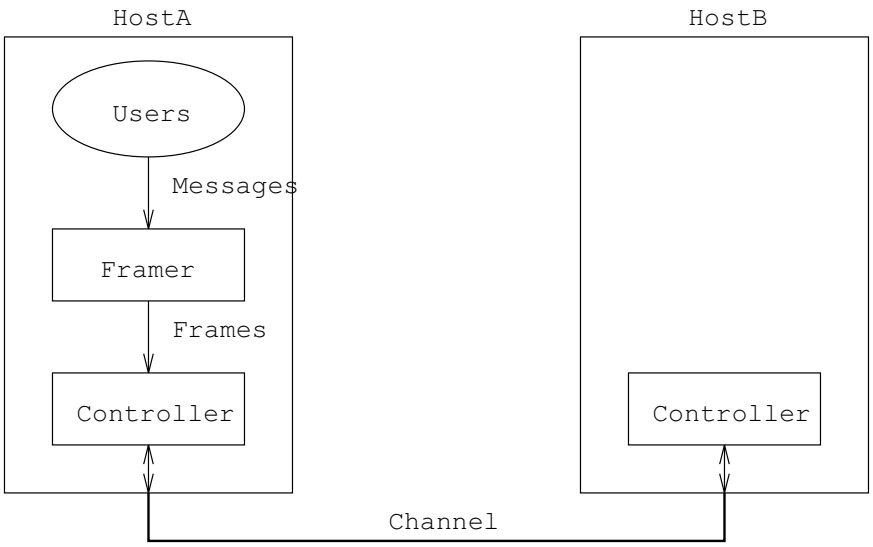

Figure 2.12: Communication channel model.

in the receiver) affects performance.

An abstraction of such a system is shown in Figure 2.12. With respect to an OSI-type model of communication [Tanenbaum 88], `Framer` represents everything above the data link layer and `Controller` represents the data link and physical layers. `Framer` transforms messages from `Users` into frames and presents the frames to `Controller` for transmission. The `Controller`s of `HostA` and `HostB` exchange frames over `Channel` and implement the flow control protocol.

The PPN in Figure 2.13 is one possible portrayal of the system in Figure 2.12. Each user produces messages at a rate determined by the delay of `Msg`. Each message consists of three frames. If there is a frame enqueued (token in `FrameQ`), a window slot available (token in `Window`), and no frame is on the channel (token in `FwdFree`), then a frame is sent. After `FrameSent`'s delay, the frame arrives at its destination and generates an ACK. If there is an ACK enqueued (token in `AckQ`) and no ACK is on the channel (token in `BwdFree`), then an ACK is sent. After `AckSent`'s delay, the ACK arrives at its destination and increments the number of available window slots by one.

The requirement that at most one frame and one ACK be on the channel

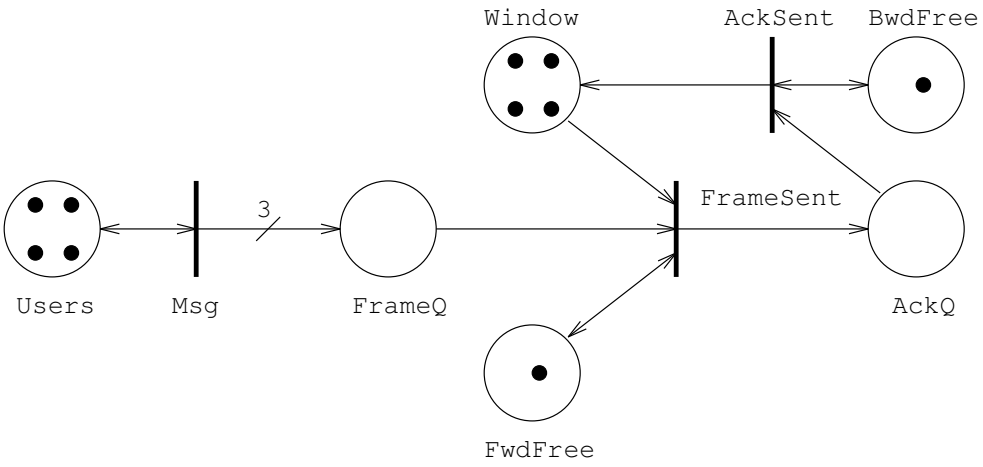

Figure 2.13: Communication channel net.

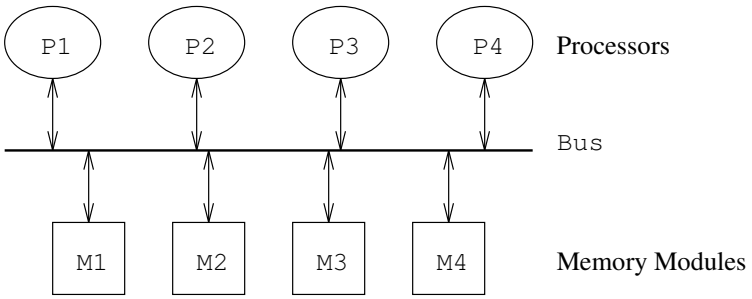

Figure 2.14: Multiprocessor model.

at one time precludes concurrent firings of `FrameSent` and of `AckSent`. The mechanism used here to prevent concurrent firings—`FwdFree` for `FrameSent` and `BwdFree` for `AckSent`—is often used in Petri net modeling.

Multiprocessor

Figure 2.14 shows a model of a multiprocessor in which processors communicate with each other via shared memory over a single common bus. A processor p computes locally in its registers and cache until it issues a request r to one of the memory modules m (each is equally likely), at which point p stalls until r is satisfied. After acquiring the bus, r acquires m, is serviced, and generates a reply r'. r' acquires the bus and returns to p, which resumes computing.

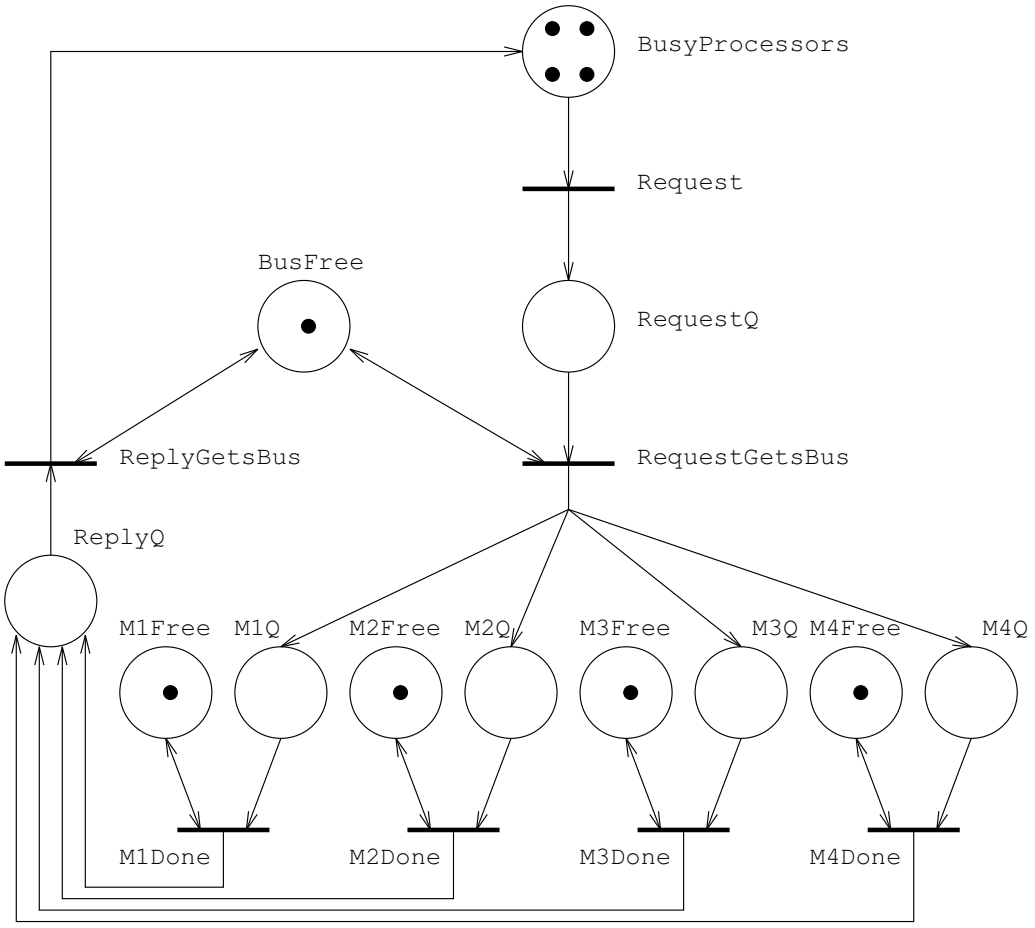

Figure 2.15: Multiprocessor net.

A PPN which models a system having four memory modules appears in Figure 2.15. The initial markings of `Processors` and `BusFree` are the number of processors and the number of buses, respectively. Tokens at `Processors` correspond to processors computing locally. With an interrequest time given by the delay of `Request`, a processor makes a request and stalls (a token moves from `Processors` to `RequestQ`). If there is a request enqueued (token in `RequestQ`) and the bus is available (token in `BusFree`), then the request acquires the bus (a token is removed from `RequestQ` and from `BusFree`), arrives at a memory module—`M1`, for example—after the bus delay (a token is deposited at `M1Q` after the delay of `RequestGetsBus`), and releases the bus (a token is deposited at

`BusFree`). At `M1`, if there is a request enqueued (token in `M1Q`) and no other request is being serviced (token in `M1Free`), then the request acquires the memory module (a token is removed from `M1Q` and from `M1Free`), generates a reply after the memory delay (a token is deposited at `ReplyQ` after the delay of `M1Done`), and releases the memory module (a token is deposited at `M1Free`). Replies acquire and release the bus in exactly the same fashion as requests. The return of a reply to `Processors` completes the cycle, allowing the processor to resume computing.

By increasing the number of tokens in place `BusyProcessors`, this net can be used to evaluate how bus contention grows with the number of processors in a multiprocessor. Similarly, changing the number of tokens in `BusFree` could reflect adding additional buses to the machine.

Database Locking

As a final example, consider a database system in which transactions acquire locks so that the integrity of the system is preserved in the face of concurrent accesses. Let the number of lockable units (called *granules*) be G, and let them be numbered $1, 2, \ldots, G$. For the sake of simplicity, assume that each transaction acquires a sequence of consecutively numbered granules; the sequence starts with granule g, $1 \leq g \leq G$, with probability $1/G$; and that a transaction requiring granule g also requires granule $g+1$ (if it exists) with probability 0.5.

The set of all transactions can be partitioned into disjoint sets such that the members of a set S require the same sequence of granules; call each S a *class* of transactions. Let $N(G)$ denote the number classes for G granules.

$$\begin{aligned} N(G) &= \sum_{\ell=1}^{G} \text{number of sequences of length } \ell \\ &= \sum_{\ell=1}^{G} G - \ell + 1 \\ &= \frac{G^2 + G}{2} \end{aligned}$$

As an example, for three granules there are $N(3) = 6$ classes; the sequences of

granules they correspond to are (1), (2), (3), (1,2), (2,3), and (1,2,3).

It is likely that the locking strategy—the procedure for obtaining locks adhered to by transactions—has a significant influence on the system's performance. One view of the spectrum of strategies has endpoints "All-At-Once" (ALL) and "One-At-A-Time" (ONE). In the ALL strategy, a transaction obtains at once all the granules it needs; it must wait until all the desired granules are available simultaneously. In the ONE strategy, a transaction obtains the granules it needs one by one in ascending order (but still must obtain all of its granules before proceeding to compute).

PPNs modeling each of these strategies for a three-granule system are given in Figures 2.16 and 2.17. The two PPNs operate similarly. A token at `G1`, `G2`, or `G3` means the respective granule is available. `RequestP` and `RequestT` implement a source of transactions (requests). Each request falls into one of the six classes; a token at `P_`x represents a request of class x, e.g., a token at `P_23` represents a transaction which needs granules 2 and 3. The firing of transition `T`x`_` signifies that the sequence of granules x has been obtained. After the delay of `T`x`_`, the granules that have been obtained are released, e.g., the firing of `T23_` releases granules 2 and 3. The delay of a granule-releasing transition (i.e., `T`x`_`) is taken to be the product of the number of granules released and a uniform with mean 1, because we expect that transactions which require more granules will compute longer before releasing them.

The PPN in Figure 2.17 is almost a "superset" of the PPN in Figure 2.16 because locking procedure ONE embodies more steps than locking procedure ALL. The additional steps are represented by the intermediate nodes for each multi-granule request. A token at place `P`x`_`y indicates that granules x have been acquired already and granules y have yet to be acquired, e.g., a token at `P12_3` indicates that granules 1 and 2 have been acquired already and granule 3 still needs to be acquired. The firing of transition `T`x`_`y indicates that granules x have been acquired already and granules y have yet to be acquired. The delay of an intermediate transition in Figure 2.17 is zero. A nonzero delay would model the case where transactions could proceed to compute each time they acquired

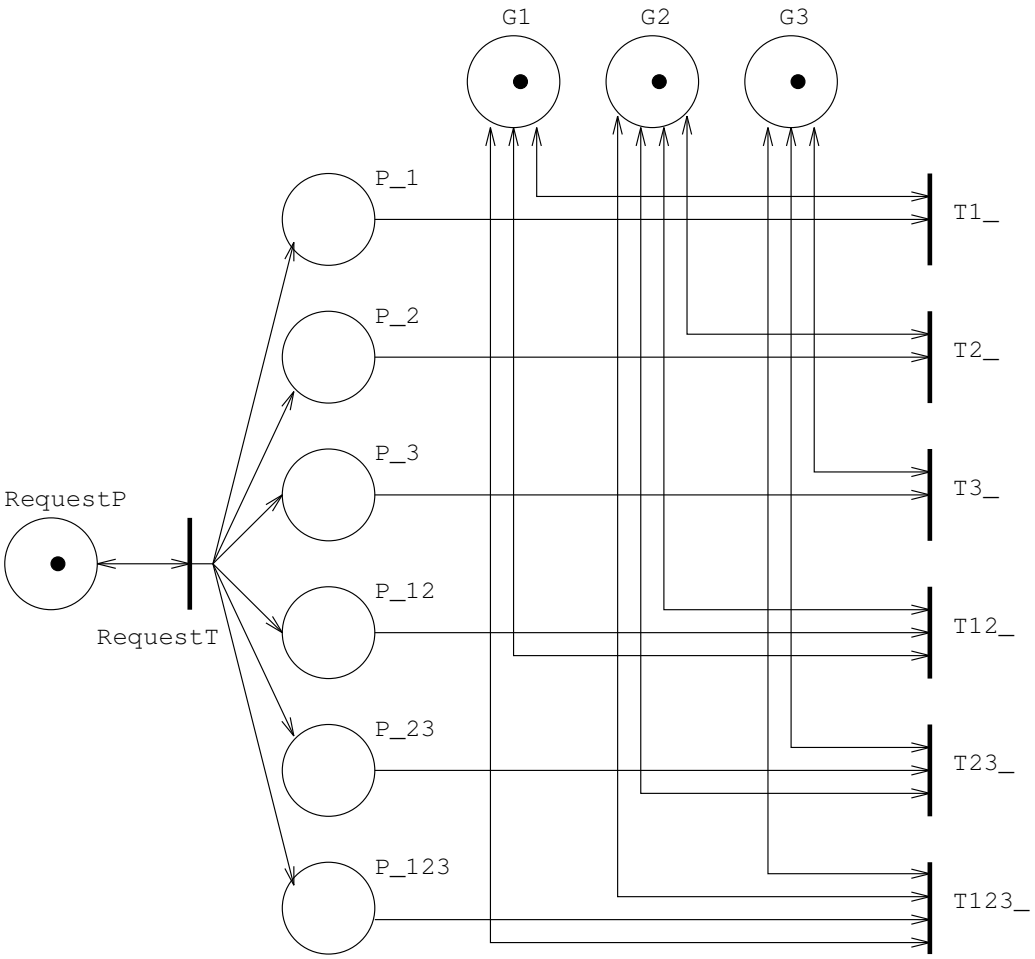

Figure 2.16: Net for database locking strategy ALL.

another granule.

Finally, the arrival rate of each class of transactions is computed as a weighted sum of probabilities which are generated by a model of the probabilistic behavior of sequences of granule lock requests, multiplied by a scale factor to yield a reasonable utilization. The rate is implemented by a combination of the delay of `RequestT` and the deposit branching weights on `RequestT`'s output arcs.

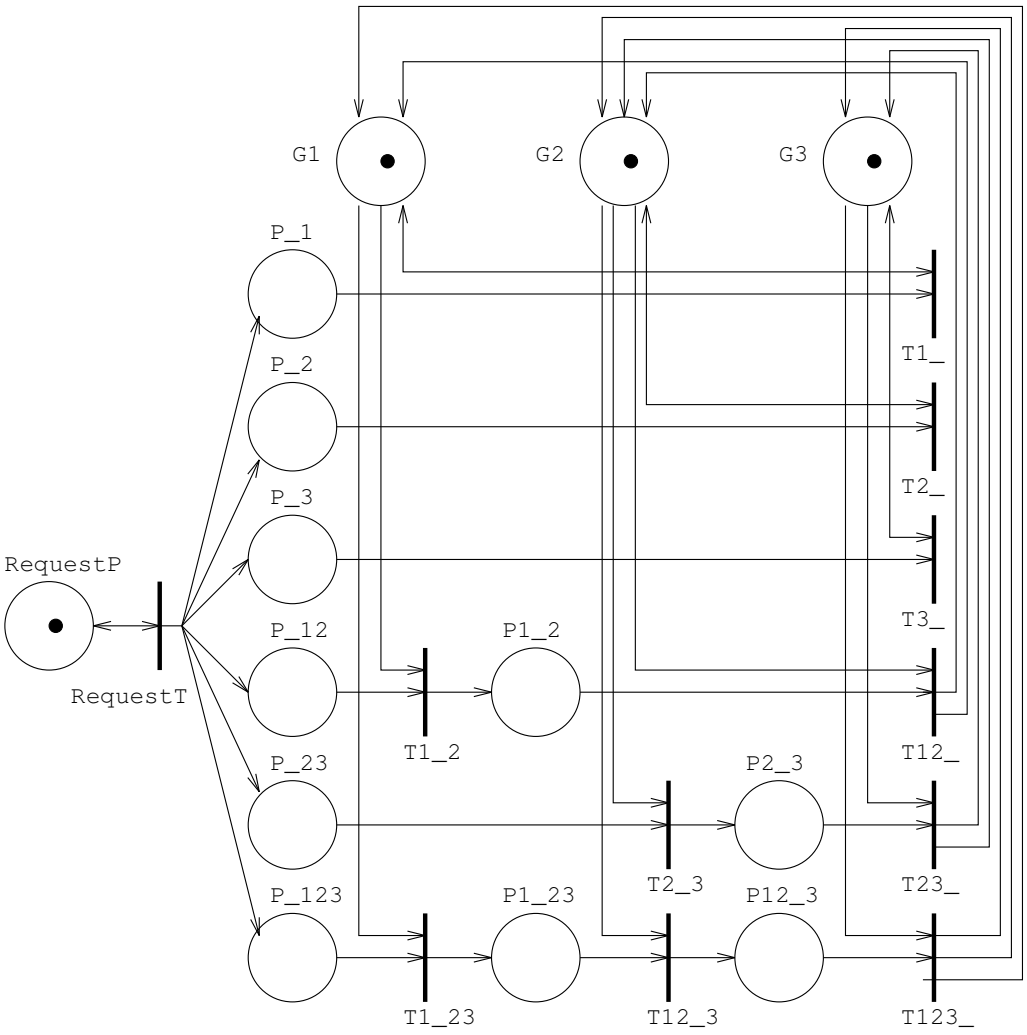

Figure 2.17: Net for database locking strategy ONE.

Chapter 3

Parallel Simulation

In this chapter we briefly characterize different simulation paradigms, discuss various possible decompositions of nets for the purpose of parallel simulation, and explain why the standard paradigm of parallel simulation does not work for the most promising decompositions.

3.1 Simulation

In the following introduction, we adopt several of the definitions in [Law & Kelton 82].

A *system* is a collection of entities acting together toward the accomplishment of some goal. To study a system, a *model* of it is constructed. The model consists of assumptions about the relationships between the entities of the system. If the model is simple enough, mathematical techniques may be employed to obtain *analytic* (exact) solutions to questions of interest. The many models too complex to admit analytic solutions must be studied through *simulation*, wherein a computer is used to numerically evaluate the model for some duration to obtain estimates of the true values of solutions to questions of interest.

A simulation model can be *static*, representing the system at a particular time, or *dynamic*, representing the system as it evolves over time. A dynamic simulation model whose state variables change continuously with respect to time

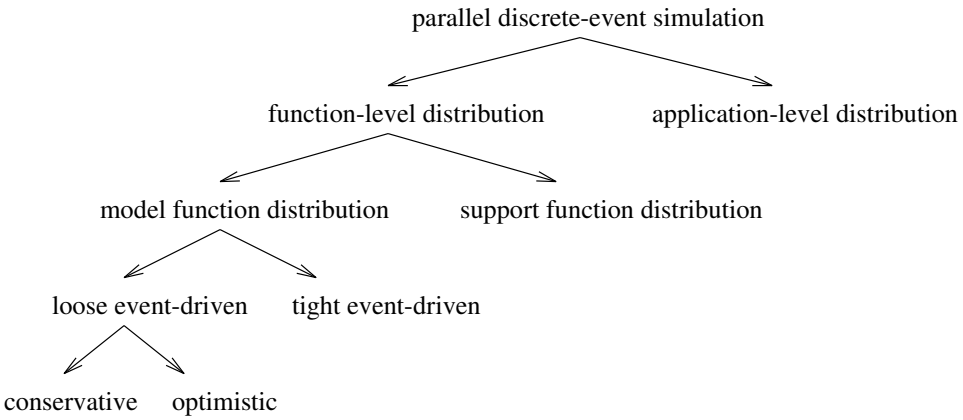

Figure 3.1: Taxonomy of parallel discrete-event simulation.

is called *continuous*, while one whose state variables change only at a countable number of points in time is called *discrete*. An *event* is an instantaneous occurrence which may change the state of the system. A simulation using a discrete model is often called a *discrete-event* simulation. This thesis is concerned exclusively with discrete-event simulation.

3.2 Parallel Simulation

Because simulation often requires substantial computational resources, the use of parallel and/or distributed computing has been actively investigated as a means of accelerating the simulation process. There are a number of strategies for incorporating parallelism in the simulation process. An abbreviated taxonomy of parallel discrete-event simulation, based upon the presentations in [Kaudel 87, Lin 90], appears in Figure 3.1.

In application-level distribution, different replications of the simulation are distributed across the available processors, and the replications execute in parallel. This strategy is useful for exploring large parameter spaces. In function-level distribution, on the other hand, multiple processors cooperate on a single replication, the goal being to reduce the execution time of the replication. This strategy is useful in situations in which the results of replication i are used to

select parameter values for replication $i + 1$.

Support function distribution is one type of function-level distribution. It uses different processors to perform different support functions, e.g., statistics collection, random number generation, and event list manipulation. In general, support function distribution provides little parallelism. In model function distribution, different events are processed concurrently by different processors. There are several strategies for doing this. The tight event-driven strategy processes concurrently only those events occurring at the same simulation time (i.e., events having identical *timestamps*). The loose event-driven strategy permits events having different timestamps to be processed concurrently, and thus yields greater potential parallelism.

In loose event-driven parallel simulation, the system to be simulated is commonly referred to as the *physical system*; the interacting components constituting it are called *physical processes*. A physical system is mapped to a *logical system*, a network of *logical processes* (LPs) that communicate (schedule events) by sending timestamped messages through unidirectional *channels* [Chandy & Misra 81]. For an LP to correctly simulate the system entity it represents, it must process messages in timestamp order, regardless of the real time order in which messages may arrive.

In the conservative approach to loose event-driven parallel simulation, an LP does not process a message until it knows that no message with an earlier timestamp can ever arrive. In the optimistic approach, an LP can process a message as soon as it arrives—it optimistically assumes that no message with an earlier timestamp will arrive later. If a message with an earlier timestamp *does* arrive later, the LP initiates a rollback computation to restore the simulation to a correct state.

This thesis concentrates on the conservative approach.

3.3 Parallel Simulation and PPNs

There are a number of ways to map a PPN into a network of LPs. In choosing one, the basic tradeoff of parallel computing applies: increasing the number of LPs working on the problem can decrease the elapsed execution time (a benefit) but can also increase inter-LP communication overhead (a cost).

Perhaps the most natural PPN to LP mapping is an isomorphism: one LP per PPN node and one channel per PPN arc. This node-based decomposition maximizes the potential parallelism of the simulation, and, since the number of LPs scales directly with the number of PPN nodes, has the potential to address very large problems running on massively parallel machines. However, this decomposition has two drawbacks. First, because the amount of work each LP has to do "per message" (e.g., updating the marking of a place when tokens are deposited) is very small, the relative cost of message passing overhead is very large, leading to potentially poor performance. Second, neither the standard conservative nor optimistic method of parallel simulation can be used for this decomposition of PPNs.

In standard parallel simulation, interaction among physical processes occurs exclusively via explicit messages, with messages in the logical system corresponding to messages in the physical system [Chandy & Misra 81]. The node-based decomposition of PPNs violates this paradigm because the behavior of a node of the net may be determined by state that is global with respect to that node rather than through explicit message exchange. For example, this is always the case for decision places. As illustrated in Figure 2.15, if (decision) place `BusFree` has a token, it cannot on its own decide whether to send it to transition `RequestGetsBus` or transition `ReplyGetsBus` since it cannot determine whether either is enabled.

Note that this problem exists for optimistic as well as conservative parallel simulation. One might imagine optimistically sending the token both directions in the case above, and relying on the rollback feature of optimistic simulation to later undo the incorrect decision. However, this will not work. Under optimistic simulation the rollback mechanism is triggered by the arrival of a message

timestamped in an LP's past. No such message will arrive in the case of PPNs. Further, even if messages conveying "I could not use the token you sent me at time τ" could be sent, this would not solve the problem. If no such messages were sent from any of the receiving LPs, what would this mean? At most one of them should be allowed to ultimately consume the single token that existed, but there is no way to coordinate their actions so that this happens.

An alternative mapping of a PPN into a network of LPs is one in which each LP represents a static conflict set of the net, with channels corresponding to arcs between conflict sets. (Such arcs will always be directed from a transition to a place.) This mapping has characteristics that are just the opposite of the node-based mapping discussed above. In particular, because each LP represents a potentially large number of nodes, the number of events that occur internally to an LP per inter-LP message can be high, minimizing the relative cost of communication overhead. Also, this conflict set-based decomposition of the net allows the standard methods of parallel simulation to be used, since there is no interaction among conflict sets other than the depositing of tokens as a result of transition firings. On the other hand, the potential parallelism of this decomposition is poor, as the number of LPs scales with the number of conflict sets rather than with the number of nodes, and there is no direct relationship between the size of a PPN and the number of conflict sets it contains. (The entire net, or nearly the entire net, can be a single conflict set, for instance. In fact, the five-granule version of the PPN in Figure 2.16 has two conflict sets, one containing two nodes and the other containing thirty-five nodes.)

In the end, the most appropriate decomposition of a PPN into a network of LPs is probably some combination of these two approaches. In those cases where a conflict set-based decomposition yields parallelism commensurate with the number of physical processors available to run the simulation, its low overhead makes it an attractive alternative. However, for PPNs with only a very few conflict sets or in situations where the simulation will be run on a massively parallel machine, the node-based decomposition described above must be adopted for at least part of the network.

With this understanding, in the remainder of this thesis we concentrate on the node-based decomposition, since it presents new challenges for parallel simulation. We first consider the problem of simply correctly simulating a PPN using this decomposition. Subsequently, we examine its performance using a prototype implementation. It should be kept in mind, though, that while for simplicity of exposition we assume that the entire PPN is decomposed as an LP per node, in a practical situation a more sophisticated combination of node-based and conflict set-based decompositions is probably the best choice.

Chapter 4

The Transition Firing Protocol

A protocol for the conservative parallel simulation of PPNs, which we call the *Transition Firing Protocol* (TFP), is presented in this chapter. We chose to follow a conservative approach because we expected its performance characteristics to be more stable (and often better) than those of an optimistic approach. The basic reason for this is the behavior of decision places in the net. As explained in the remainder of this chapter, it is the problems caused by decision places that render invalid standard conservative parallel simulation.

Recall from Section 3.3 our chosen decomposition: each place and each transition is represented by an LP, and each arc in the net is represented by a channel. We augment this simulation model by introducing additional channels: there is a channel from LP t_j to LP p_i if there is an arc from place p_i to transition t_j in the PPN. To distinguish these additional channels from the "ordinary" channels (along which tokens may flow) we refer to them as *control* channels.

The graphical representation of the simulation model is identical to that of the PPN, except that we include control channels (drawn as dashed arcs) when appropriate. The net fragment in Figure 4.1 contains place LPs `p1` and `p2`, transition LP `t1`, ordinary channels (`p1`, `t1`) and (`t1`, `p2`), and control channel (`t1`, `p1`) (as well as additional channels connecting with other, unspecified LPs).

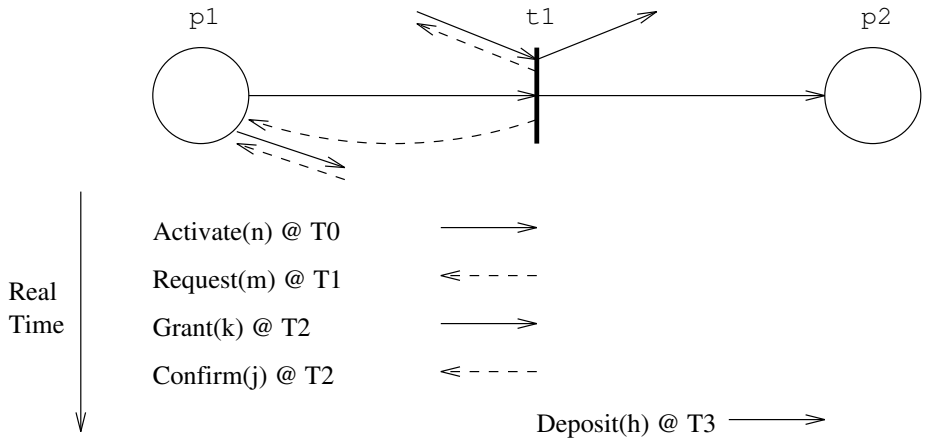

Figure 4.1: Overview of TFP.

TFP is basically a double handshake between transitions and their input places, as indicated in Figure 4.1. The notation $Message(n)@Ti$ means *Message*, passing parameter n, is sent at simulation time Ti. Ti is the *timestamp* of *Message*; every message has a timestamp.

TFP's double handshake is motivated by decision places.[1] When a decision place receives tokens, it may cause many transitions to become enabled simultaneously. Because the decision place cannot know which transitions are now enabled without interacting with them, it cannot make an immediate decision about which transition should be sent tokens. The first of the two handshakes determines which transitions are enabled. A decision about which transitions this decision place would like to fire is then made, and in the first portion of the second handshake those transitions are offered tokens. Because each of these transitions may have other decision places as input places, and under the conflict resolution scheme we use (see Chapter 6) those places may make different firing decisions, in the final portion of the second handshake the transitions inform their input places whether or not they have actually fired.

[1]There are several cases in which TFP can be streamlined. For example, a place with a single noninhibitor output arc can send a *Grant* as soon as it has tokens. Additional special case optimizations are possible to further reduce the number of messages that need to be exchanged.

In what follows, we describe the four steps in the handshake without explicitly considering how they are integrated with the messages corresponding to the token deposits due to transition firing. This integration is provided in Chapter 5.

For the sake of simplicity in the following discussion we assume regular (i.e., noninhibitor) arcs of multiplicity one. TFP handles the general case—regular and inhibitor arcs of any multiplicity.

- *Activate*(n)@T0

  The first handshake starts when place `p1` receives a deposit of n tokens at time T0,[2] causing it to send an *Activate*(n)@T0 message to each of its output transitions, indicating that it has n tokens available at time T0.

- *Request*(m)@T1

  Once `t1` has received an *Activate* from each of its input places it calculates T1 $\geq$ T0, the latest timestamp of any of the *Activate* messages it has received, and $m \leq n$, the smallest of the *Activate* parameters. `t1` then updates its (local) simulation clock to T1 and responds to each input place with a *Request*(m)@T1 message. `t1` thereby asserts that it is ready to fire m times at time T1 if all of its input places (still) contain sufficient tokens at that time. This completes the first handshake.

- *Grant*(k)@T2

  `p1` waits to receive a *Request* from each output transition it has sent an *Activate*.[3] Let T2 $\leq$ T1 be the earliest timestamp among these *Request*s. (In Figure 4.1, T2 = T1.) When `p1` has received all *Request*s timestamped T2, it executes a conflict resolution algorithm (explained in Chapter 6) to choose among competing earliest *Request*s, and then replies to each such *Request* with a *Grant*(k)@T2 message, indicating that $k \leq m$ tokens are

[2] In general, a deposit is not needed to initiate the sequence—tokens already residing at `p1`, perhaps in the initial marking, can initiate it. We present it this way because it is the easiest situation to understand.

[3] This is why our approach is conservative.

available to the sender of the *Request* for firing. (Note that k could be zero.)

- *Confirm*(j)@T2

  `t1` collects a *Grant* message from each of its input places. Let j be the minimum parameter of these *Grant*s. (Thus, j could be zero.) `t1` ends the second handshake by sending a *Confirm*(j)@T2 to each input place, indicating that `t1` has fired j times (and so has consumed a corresponding number of tokens from each input place).

  If appropriate, `t1` schedules one or more *Deposit*(h)@T3 messages, where $0 < h \leq j$ and T3 $\geq$ T2, to be sent (when the corresponding firing delays have elapsed) to one or more of its output places.

The receipt of a *Confirm*(j) from `t1` by `p1` completes the second handshake between `p1` and `t1`. `p1` subtracts the j tokens from its marking that were consumed by the firing of `t1` and begins the next cycle of the protocol.

Chapter 5

Selective Receive and TFP

The development of TFP led us to introduce a new technique, Selective Receive, which relaxes one of the fundamental restrictions imposed by the standard conservative approach to parallel simulation. In this chapter we describe Selective Receive, discuss its integration into TFP, and present the finite automata that implement the place and transition LPs.

A basic tenet of the conservative approach to parallel simulation is that messages on each channel must be sent in nondecreasing timestamp order. It is this property that distinguishes conservative from optimistic simulation.

The standard conservative algorithm achieves this output ordering goal by imposing an input ordering restriction. Associated with a channel from LP_i to LP_j is a *channel clock value*, c_{ij}, which is equal to the earliest timestamp of any undelivered message pending on that channel, if one exists, or else the time of the last message sent on it. LP_j is prevented from receiving a message m having timestamp τ_m unless τ_m is equal to H_j, the *message acceptance horizon* of LP_j, where $H_j = \min_i c_{ij}$.

This input ordering restriction is a major source of performance problems for conservative simulations, including the possibility of deadlock. Consequently, a great deal of work has focused on minimizing its impact (e.g., [Reed et al. 88, Nicol 88, Wagner & Lazowska 89, Fujimoto 90]).

The problem of deadlock is particularly extreme for TFP. Because transitions send messages along their control channels only in response to messages received from the input place connected there, no messages can be received by places at the beginning of a TFP cycle. For instance, imagine that a place has no tokens in its current marking and so has not sent out any *Activate*s. At this point it would like to receive any *Deposit* messages that its input transitions might send it. However, the semantics of the message receive operation under conservative simulation prevents it from doing so, even if there is a pending *Deposit* message from each of its input transitions, because its message acceptance horizon cannot proceed beyond the time of the last TFP cycle (which is the channel clock value on the channels from its output transitions).

Our solution to this problem is to change the rules governing the receipt of messages by LPs. In particular, we allow each LP to specify that certain channels should be ignored in computing that LP's message acceptance horizon, thus allowing the receipt of messages that would otherwise have to remain pending. We call this feature *Selective Receive*.

Selective Receive can be applied when an LP can deduce that the next message it will receive (in simulation timestamp order) cannot arrive on one or more of its input channels. Under TFP in particular, a place LP knows statically that any output transition not sent an *Activate* message by it will not send it any messages. Thus, the LP is free to receive the earliest message from the remaining channels.

Note that Selective Receive is distinct from approaches based on null messages, lookahead, and futures. Unlike null messages, Selective Receive does not involve the transmission of any extra control messages. Unlike lookahead and futures, Selective Receive can be used to avoid performance problems involving message channel loops even when the lower bound on the message propagation delay around the loop is zero. In fact, our use of Selective Receive in TFP serves exactly this purpose.

Under TFP, a place LP never ignores a channel from an input transition LP.

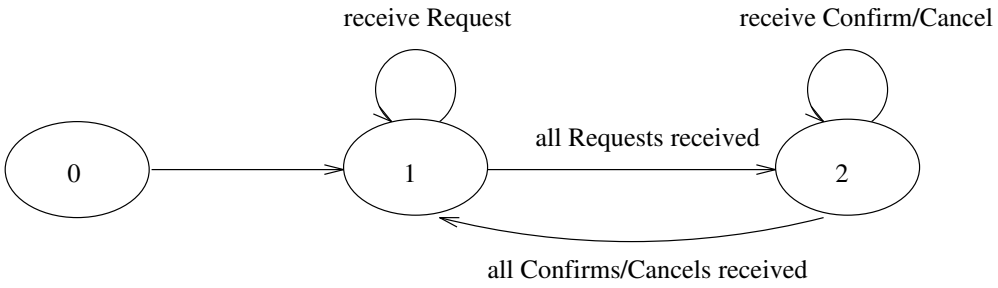

Figure 5.1: State diagram for place LP.

It ignores a channel from output transition LP_j except between the time at which it sends an *Activate* to LP_j and the time at which it receives the corresponding *Confirm*. A transition LP ignores a channel from input place LP_i after it receives an *Activate* from LP_i until it sends *Request*s, and after it receives a *Grant* from LP_i until it sends *Confirm*s.

To better understand the use of Selective Receive in TFP, and the integration of *Deposit* messages into TFP, we now summarize the activity of the place and transition LPs. For simplicity, we again assume regular arcs of multiplicity one.

5.1 Place LP

An LP that represents a place may be thought of as a finite automaton. A state diagram for this automaton appears in Figure 5.1. In the discussion below, MRT is the *minimum Request time*—the timestamp of the earliest *Request* received.

A place LP, P, starts out in state 0, a transient boot state. It sends *Activate*s to its output transitions as warranted by its initial marking, then goes to state 1.

While in state 1, P collects *Request* responses to previously sent *Activate*s and accepts new *Deposit*s, sending out any new *Activate*s that they may allow. To do this, P uses Selective Receive, allowing input only on channels from its input transitions (messages received will be *Deposit*s) and from any *Activate*d output transitions—channels from non-*Activate*d output transitions are ignored. If such a channel from an output transition T is not ignored, the progress of P

is impeded because no messages can arrive along this channel until after P sends T an *Activate* and so the message acceptance horizon cannot advance.

P remains in state 1 until all earliest timestamped *Request*s are received and it knows its marking at MRT.[1] It then applies a conflict resolution scheme (explained in Chapter 6) to decide which of the earliest *Request*s to grant, and sends out an appropriate *Grant* in response to each earliest *Request*. P now moves to state 2, where it receives information about which transitions eventually do fire. Note that *Request*s timestamped later than MRT remain pending.

P remains in state 2 until it receives a *Confirm* in response to each *Grant*(k), $k > 0$, it sent. (The response to a *Grant*(0) will be a *Confirm*(0), and thus will not affect the marking of P, so P need not wait for responses to *Grant*(0) messages before returning to state 1.[2]) P ignores the input channel from output transition T after P receives a *Confirm* from T. Once again, if the channel is not ignored, the progress of P is impeded because no further message can arrive from T until after P sends an *Activate* to T.

When all required *Confirm*s are received, P updates its marking, subtracting tokens for each transition that fired. If tokens remain after this update, P sends an *Activate* to each inactive output transition if allowed by the updated marking. Finally, P returns to state 1.

5.2 Transition LP

A state diagram for the transition LP automaton appears in Figure 5.2. In the discussion below, MAT is the *maximum Activate time*—the timestamp of the latest *Activate* received.

---

[1] This involves sophisticated *lookahead* functions. Details are given in Section 7.3.2.

[2] Transitions that were sent *Grant*(0) messages are of secondary importance at this point in the protocol, and failure to receive *Confirm*s from them will not prevent P from returning to state 1. Thus, a *Confirm*(0) from some LP T may be received while in state 1. When this happens, an *Activate* is immediately sent to T if allowed by the current marking.

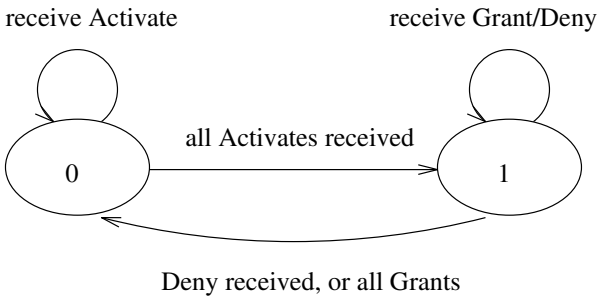

Figure 5.2: State diagram for transition LP.

A transition LP, T, starts out in state 0 and remains there until it has received an $Activate(n_i)$ from each input place P_i. It then sends a $Request(\min_i n_i)$@MAT to each input place and goes to state 1. Between the time T receives an *Activate* from P_i and the time it sends *Request*s, T ignores P_i because P_i cannot send another message to T until after T sends a *Request* to P_i.

T remains in state 1 until it receives a $Grant(j_i)$@MAT from each input place P_i. It then sends a $Confirm(\min_i j_i)$@MAT to each input place, possibly schedules one or more *Deposit*s to be sent to its output places after suitable firing delays have elapsed, and returns to state 0. Between the time T receives a *Grant* from P_i and when T sends *Confirm*s, T ignores P_i because P_i cannot send another message to T until after T sends a *Confirm* to P_i.

Chapter 6

Conflict Resolution

In this chapter we discuss different methods for resolving conflicts in PPNs, including our method, which was designed to exploit the properties of parallel simulation.

When one or more decision places in a conflict set contain tokens, a decision must be made about which set of transitions to fire. As mentioned in Section 2.2, a number of proposals have been made for this decision procedure, and there is no standard agreement on this aspect of PPNs. However, one of the more flexible approaches is offered by [Holliday & Vernon 87]. They allow the user to specify weights that are the basic parameters of a function that assigns probabilities to each maximal set. (Recall that a maximal set is a firable set of transitions that is not a subset of any other firable set.) A single maximal set is then chosen for firing according to this probability distribution.

While Holliday and Vernon proposed this scheme for use in analytic approaches to PPN analysis, their technique is easily applied in a sequential simulation: after an event is processed the new PPN marking is examined, the maximal sets are enumerated and assigned probabilities, one is selected, and a new set of (transition firing) events are scheduled. This is a relatively straightforward procedure in a sequential simulation.

In a parallel simulation, however, applying this technique is much more complicated. The reason for this is that determining maximal sets requires a global

view of the net at a particular simulation time. Since in a parallel simulation each LP may have a different simulation time, determining the global state at a particular simulation time is not a simple matter. Further, because each LP has information about only its local state, no LP is naturally in a position to compute maximal sets. To do so requires the creation of a new LP for this purpose, and cooperation from place and transition LPs in registering state information with this LP. Needless to say, this is a complicated protocol to implement, and tends to serialize execution of the simulation.[1]

When constructing a parallel simulation of PPNs, a decentralized approach is more natural. The one we use is based on "trial-and-error": each decision place p selects at random according to the user supplied arc weights one or more transitions in $O_{TR}(p)$ that it would like to fire, and offers them tokens via a *Grant* message. Additionally, all transitions t for which $t \in O_{TI}(p)$ and $M(p) < U((p,t))$ are sent *Grant*s.[2] If a transition t is lucky enough to receive *Grant*s from all $p \in I_P(t)$, it then fires. Otherwise, it replies that it cannot use any of the tokens and its input places try again.

This procedure, which is incorporated in TFP, is completely decentralized: each place makes decisions based solely on information available to it either locally or through communication with its own output transitions only. Thus, we might expect the decentralized approach to have better performance than the more serial maximal set-based approach.

While this reduction in serialization may have important performance implications, experience with our simulator shows that an even more important effect is the change in computational complexity that results from the decentralized approach. Enumerating maximal sets is of exponential complexity

[1]Note, however, that with such an implementation the *Confirm* messages of TFP could be eliminated since it would be guaranteed that the firing decisions of all decision places would be in agreement.

[2]Since the semantics of inhibitor arcs differ from those of regular arcs, and the firing of $t \in O_{TI}(p)$ does not consume any tokens from p, p always sends a *Grant* to t if permitted by M(p). This is why inhibitor arcs do not have weights.

[Holliday & Vernon 87], both in terms of time and space. Thus, if conflicts involving a large number of transitions or tokens occur with any frequency, conflict resolution by this approach can be extremely slow.[3]

We note that the decentralized conflict resolution scheme has slightly different semantics than the maximal set-based approach, reflecting the difference between the "repeated trials" approach of the former and the one-time enumeration of the latter. In general, neither scheme is able to simulate the other, i.e., no choice of weights for the decentralized scheme results in a distribution identical to that of the maximal set scheme and vice versa. However, this is not considered a significant problem since there is no standard for conflict resolution semantics and neither scheme provides a significantly more convenient way for the user to express the desired behavior of the model.

We also note that while the decentralized scheme employed in TFP could be emulated in a sequential simulation, resulting in performance gains comparable to those we observed in the parallel simulations, this would be somewhat unnatural. It therefore seems unlikely that the new scheme would have been developed in a sequential environment.

In Chapter 8, which concentrates on the speedup characteristics of our parallel simulation and its performance relative to the analytic techniques, we also provide some empirical information about the performance advantages of decentralized over maximal set-based conflict resolution.

[3] On the other hand, the time requirement of the decentralized scheme depends on the arc weights. Since each decision place selects transitions independently, an unfortunate set of arc weights could lead to a very large expected time to determine a firing. Thus, while the decentralized approach has better average time characteristics in our experience, it is easy to generate artificial models for which the expected conflict resolution time exceeds any bound. This rather disconcerting aspect of this approach is the topic of current work.

Chapter 7

Implementation

The subject of this chapter is Persephone, a prototype implementation of TFP. After describing the hardware and software comprising the environment in which Persephone runs, we give a high level sketch of the execution of Persephone, followed by a compendium of selected issues involved in the implementation.

7.1 Environment

Our hardware platform is a Sequent Symmetry S81 shared-memory multiprocessor, configured with 32 Mbytes of primary memory and 20 16-MHz Intel 80386 32-bit CPUs. The CPUs are connected to the primary memory by a 64-bit system bus with a maximum sustainable data transfer rate of 53.3 Mbytes per second. Each CPU has a 64 Kbyte two-way set associative copy-back cache.

The Symmetry runs the DYNIX operating system, Sequent's multiprocessor version of UNIX.

The bulk of Persephone is written in C++ [Stroustrup 86]. (Part of the front end, because it uses a scanner and a parser generated by the UNIX tools *lex* and *yacc*, respectively, is written in C [Kernighan & Ritchie 78].)

Persephone is built on top of a modified version of Synapse [Wagner 89], a library of C++ classes for conservative parallel simulation. Synapse exports a basic LP class that provides the message delivery mechanism and ensures that

messages are received in nondecreasing timestamp order, and a scheduler that handles the scheduling of LP objects. Synapse exploits the shared memory of the Symmetry in an attempt to achieve good performance [Wagner & Lazowska 89].

The implementation of TFP required several modifications to Synapse, primarily to its lookahead, deadlock detection and recovery, and message delivery mechanisms. Selective Receive—the major enhancement made to Synapse—violates the normal message delivery semantics of conservative simulation (and thus of Synapse), so many of the changes are directly related to Selective Receive.

As DYNIX provides only standard heavyweight UNIX processes, applications seeking to exploit medium- or fine-grained parallelism require some kind of support for lightweight processes (also called *threads*). PRESTO [Bershad et al. 88] is a user-level threads package, written in C++, which provides such support. Synapse runs on top of PRESTO.

The layering of these systems is beneficial because it separates concerns and abstracts away details, but it makes determining where the bottlenecks are in a Persephone simulation a significant challenge. This is the focus of continuing work.[1]

7.2 Execution

Persephone is an application-level program that takes a description of a PPN as input, performs a parallel simulation of the given PPN, and produces statistics from the simulation as output, as illustrated by Figure 7.1.

A PPN is described by a text file in which nodes, arcs, and various attributes (e.g., the initial marking) are specified. A modeler may create this text file by using either a standard text editor or *xsim* [Thomas 90], an interactive X Window System-based graph editor/compiler that can be configured to generate input suitable for Persephone from a PPN drawn by the modeler.

[1] The three systems—Persephone, Synapse, and PRESTO—together comprise 40,000 lines of C++ code.

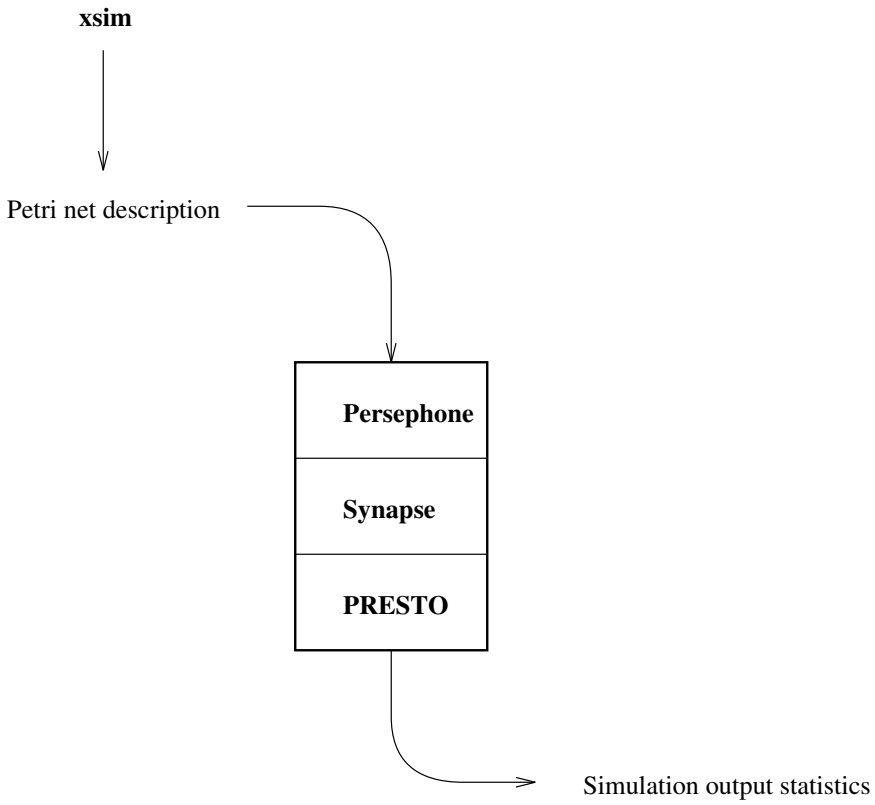

Figure 7.1: Use of Persephone.

Persephone reads the input file, creates an internal representation of the input PPN, constructs the network of LPs to which the PPN maps, and spawns one thread per LP. The simulation is performed by the threads executing TFP on a number of processors specified by the user. When the simulation completes, statistics collected during the simulation are processed. Examples of statistics computed are the *resource-time product* [Murata 89] for each place and the firing rate for each transition. After the statistics are reported, Persephone halts.

Details of interest to a user of Persephone—the format of a PPN description, the command line interface, and output statistics—appear in [Thomas 91].

7.3 Issues

7.3.1 Firing Delay Distributions

While Definition 2.1 does not restrict the distributions from which transition firing delays are drawn, an implementation can support only some specific set of distributions. Persephone supports the following distributions:

- constant(c)
- exponential(β)
- geometric(p)
- uniform(a, b)

Definitions of these distributions appear in [Law & Kelton 82]. As mentioned in Section 2.3, arbitrary distributions are acceptable; the four listed above were chosen because it was felt they would be the ones most often used in models.

7.3.2 Lookahead

Lookahead refers to the ability of an LP to guarantee that it will not send a message before a certain time in the future. This is crucial for good performance of a conservative parallel simulation because an LP blocks, unable to process pending messages, unless it can determine that no messages with earlier timestamps can arrive later [Fujimoto 90].

The implementations of place and transition LPs in Persephone exploit the properties of TFP to provide lookahead capability. The reasoning involved is similar to that used for Selective Receive. The next sections sketch the ideas used to provide this capability. In what follows, define $L_{x,y}$ to be the lookahead from LP_x to LP_y, i.e., the maximum simulation time τ such that LP_x can guarantee that it will not send a message to LP_y that is timestamped earlier than τ.

Transition LP

A transition t in state 0 (refer to Figure 5.2) will not send a *Request* to any $p \in I_P(t)$ until t has received an *Activate* from each $p \in I_P(t)$, so $L_{t,p}$ is the latest time at which t could receive such an *Activate*. Let MDT (*minimum Deposit time*) be the earlier of (a) the next completion of a firing already in progress, and (b) the earliest time at which a firing not yet in progress could begin and end; t will not send a *Deposit* to any $q \in O_P(t)$ before MDT, so $L_{t,q}$ is equal to MDT.

In state 1, t *will* send a *Confirm* to all $p \in I_P(t)$ at the current simulation time τ, so $L_{t,p}$ is equal to τ. t will not send a *Deposit* to any $q \in O_P(t)$ before MDT, where MDT here is computed as in state 0, so $L_{t,q}$ is again equal to MDT.

Place LP

The lookahead function defined for a place LP is analogous to the one described above for a transition LP, except that the case analysis for a place LP is more extensive. In particular, there are 24 cases to consider, according to the state of the LP, the type of arc, and the point in the protocol the LP has reached.

Probabilistic Minimum

The computation of MDT above depends on the minimum possible firing delay, ϵ, of the transition. The value of ϵ is equal to the lower bound of the distribution from which a transition's firing delays are selected. Some distributions, e.g., the exponential, have a lower bound of zero. This adversely affects lookahead because it reduces MDT and other quantities similar to MDT.

To help alleviate this problem, we use a *probabilistic minimum* for ϵ rather than the true lower bound of the distribution. We associate with each transition a *risk* r, $0 \leq r < 1$. If F is the cumulative distribution function of the transition's firing delay distribution, we define the probabilistic minimum $pmin$ as $pmin(r) = F^{-1}(r)$.

Intuitively, r is the probability that the distribution returns a sample that is less than $pmin(r)$, which may result in an eventual out-of-order message and abortion of the simulation. Larger values of r increase the risk of the simulation aborting, but they also increase lookahead and thereby (hopefully) improve the performance of the simulation.

If the simulation does fail, it can be rerun with smaller values of r. In the long run, this procedure of trying positive values for r and rerunning the simulation if it fails can be faster than always using $r = 0$.

7.3.3 Termination

Persephone can produce both transient and steady-state solutions. The type of solution determines the termination criteria. Specifying a particular simulation stop time τ yields a transient solution—when all LPs have simulated up to time τ, the simulation terminates.

Computing a steady-state solution is more difficult, because it must be decided when the estimate for the parameter of interest has reached steady-state or, in other words, when the simulation has run long enough to derive an estimate that is "reliable" in some sense.

For computing steady-state solutions we use the sequential stopping procedure of Law and Carson, as described in [Law & Kelton 82],[2] to terminate the simulation when it reaches a $100(1-\alpha)\%$ confidence interval of relative precision γ for an estimate of the average time spent by a token at a selected place. The user may specify α and γ.

[2]However, in addition to correcting several minor errors in the description, we modified the procedure to correctly handle the situations in which all samples are identical or the true value of the parameter being estimated is zero or near zero.

Chapter 8

Performance

We have two goals in this chapter. The first is to examine how well a parallel simulation using a node-based decomposition of a PPN is able to exploit available processors. We address this by measuring the speedups obtained for simulations of a number of realistic PPNs. For these measurements we use Persephone, our prototype implementation of TFP.

Our second goal is to evaluate the growth in the running time of our PPN simulation as the size of the PPN increases, and to compare these times with those obtained using analytic approaches to PPN evaluation. In this case, we take a PPN model and increase its size in a natural way (as explained below), yielding a series of models. We compare the elapsed time to evaluate each model using Persephone and GTPNA [Holliday & Vernon 86], which, as mentioned in Section 2.6, employs analytic techniques to evaluate PPNs. Our purpose here is to determine whether or not simulation can extend the applicability of PPN modeling to larger systems by facilitating the evaluation of systems larger than those amenable to analytic techniques.

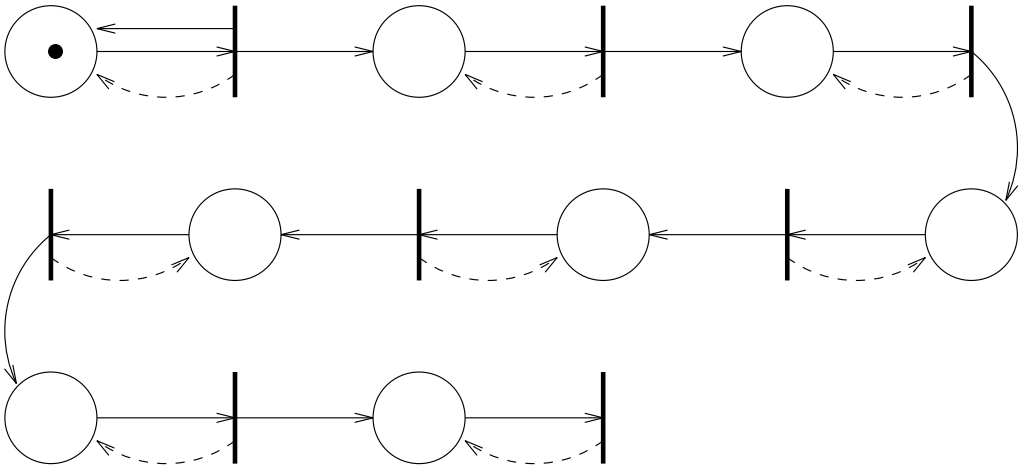

Figure 8.1: Instruction pipeline net.

8.1 Speedup Versus Number of Processors

In this section we examine the speedups achieved by Persephone on six distinct PPN models—the five presented in Section 2.7.2,[1] plus another one introduced here. Each of the PPN models represents an interesting computer system, and while certain aspects of these models have been given less attention here than would be appropriate if our purpose were in fact to model these computer systems, their basic structures could be used to answer performance questions about those systems.

Figure 8.1 shows a new PPN model, called Pipe, and its initial marking. Pipe represents an instruction pipeline in a CPU. There is a token source, representing instruction issue, followed by a number of pipeline phases, each represented as a single place and transition pair. All transition firing times are deterministic with duration 1.0, reflecting the deterministic nature of pipeline stages. Control channels are shown explicitly, as dashed arcs, to emphasize the fact that the net is not truly feed-forward. This is discussed later in this section.

[1]The PPN models DB All-At-Once and DB One-At-A-Time used in these measurements are five-granule systems; the PPNs in Figures 2.16 and 2.17 represent three-granule systems, because the smaller systems are easier to draw.

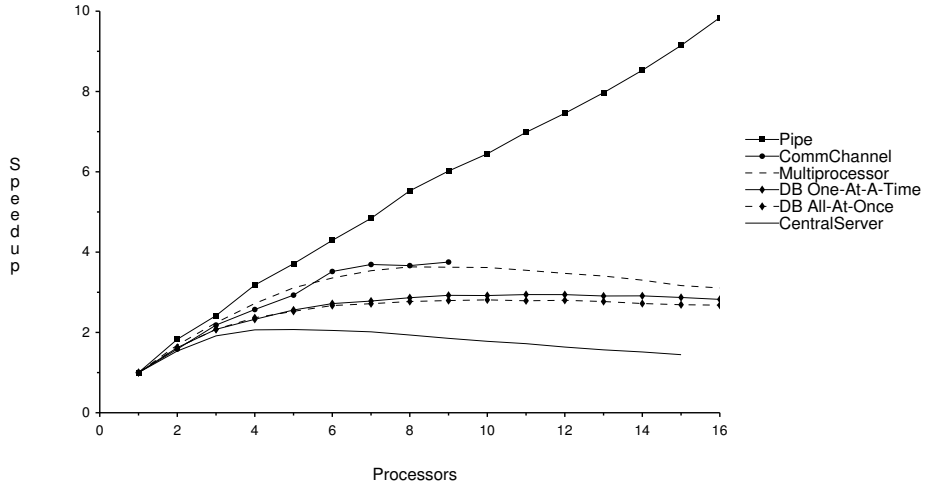

Figure 8.2: Measured speedups for Persephone on the PPN models.

The speedup curves for the aforementioned PPNs are given in Figure 8.2. Following the example of [Reed et al. 88], we define speedup relative to a parallel simulation on one processor rather than to an equivalent sequential (uniprocessor) simulation, i.e., the speedup for a Persephone simulation using p processors is the execution time for Persephone on one processor divided by the execution time for Persephone on p processors.[2] Note that the CentralServer and CommChannel models have fewer than sixteen data points because they have fewer than sixteen LPs.

The linear speedup exhibited by Pipe would not ordinarily be surprising, since the net is largely feed-forward. The presence of the control channels, however, introduces a large amount of feedback into the topology of the simulation model, which makes the linear speedup somewhat unexpected. We hypothesize that the feedback due to the control channels is what prevents the speedup from being

[2]Unlike [Reed et al. 88], however, the speedups we present are not upper bounds. Persephone on one processor can be *faster* than an equivalent sequential simulation, as shown in Section 8.2. In these cases, speedups reported relative to Persephone on one processor are *smaller* than speedups reported relative to an equivalent sequential simulation.

Table 8.1: Number of conflict sets and maximum speedups of PPN models.

Model	*Conflict Sets*	*Speedup*
Pipe	8	9.8
CommChannel	3	3.8
Multiprocessor	6	3.6
DB One-At-A-Time	6	2.9
DB All-At-Once	2	2.8
CentralServer	5	2.1

perfectly linear, i.e., of slope one.

The remaining models each exhibit speedups flattening out at between four and eight processors and, not coincidentally, contain large amounts of feedback in the PPN itself. There are several factors limiting these speedups. First, a model may have poor speedup potential regardless of the degree of concurrency in the system being modeled [Wagner 89]. As an example, Wagner shows that the queueing network model from which CentralServer is derived has a theoretical limit of 3.67 on its achievable speedup despite the fact that the queueing network simulation contains five LPs. Thus, for some models the limit on speedups is intrinsic to the problem and cannot be fixed by tuning.

Second, there are theoretical limits on the performance of conservative parallel simulation that are related to the topology of the simulation model [Lin 90]. Lin shows, for example, that under certain conditions the speedup of a conservative parallel simulation cannot exceed the number of strongly connected components in the simulation model. In the TFP simulation model used by Persephone, the strongly connected components are precisely the static conflict sets of the PPN. Table 8.1 gives the number of static conflict sets in each of the models and the maximum speedup achieved by Persephone on the model. Since Persephone does not meet Lin's criteria exactly,[3] in some cases it does achieve speedup greater than the number of strongly connected components, but overall these results can be viewed as experimental evidence that supports Lin's work.

[3]In particular, Persephone is not a system without lookahead.

We hypothesize that another factor limiting speedup is serialization due to critical sections in the underlying software systems on which Persephone is built, i.e., Synapse and PRESTO. For example, Synapse's scheduling of LPs on processors, which happens in concert with PRESTO, requires a queue of ready LPs. Access to this queue must be serialized to ensure correct execution. While we have been unable to verify directly (because of lack of appropriate measurement tools) that contention for this queue is limiting speedup, this hypothesis is consistent with our experience with other applications using the PRESTO system. If our hypothesis is correct, this limit on speedup could be addressed by tuning of the run time systems.

8.2 Execution Time Versus Model Size

For these measurements we compare the execution times of Persephone and GTPNA on four different series of PPN models. Each series is obtained by starting with an initial version of either CentralServer or Multiprocessor and increasing its size in two ways: first, by increasing the number of tokens in the initial marking, then by adding nodes and arcs (and possibly tokens) to the net. Specifically:

1. In a CentralServer with two disks, vary the number of jobs from 1 to 32 (by varying the initial marking of place `CPUP`).

2. In a CentralServer with eight jobs, vary the number of disks from 1 to 8 (by adding the necessary nodes and arcs to the net).

3. In a Multiprocessor with four memory modules, vary the number of processors from 1 to 32 (by varying the initial marking of place `BusyProcessors`).

4. In a Multiprocessor with three processors, vary the number of memory modules from 1 to 8 (by adding the necessary nodes and arcs to the net).

Analyzing our other benchmark nets with GTPNA is infeasible. GTPNA requires bounded nets, as do most of the stochastic analysis methods, and neither

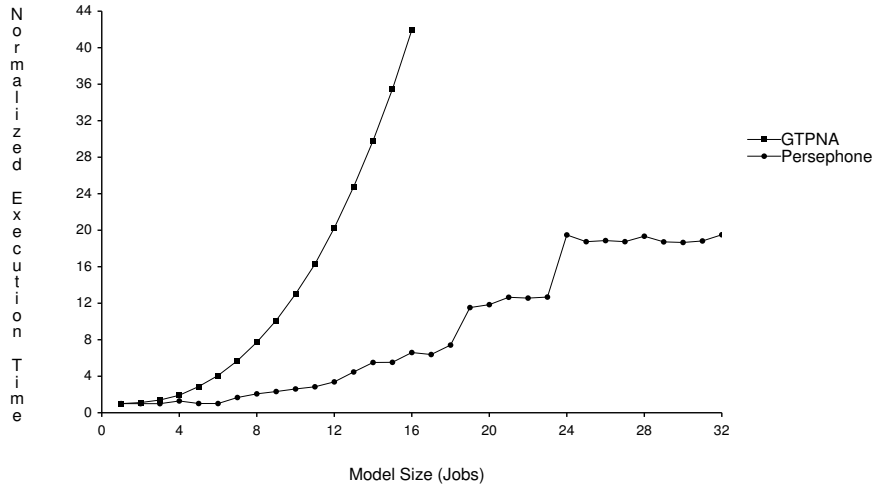

Figure 8.3: Normalized execution time on CentralServer, varying jobs.

CommChannel nor the DB PPNs are bounded. Furthermore, aside from being too large for GTPNA, the DB PPNs use exponential and uniform firing delay distributions, neither of which is supported by GTPNA.

All measurements reported here are for Persephone running on eight physical processors of the Sequent Symmetry described in Section 7.1 and GTPNA running on a DEC VAXstation 3500 with 16 Mbytes of memory. The execution times from the different systems cannot meaningfully be compared directly, so the metric we use is normalized execution time—the time for model size s divided by the time for model size 1.

Each Persephone run was terminated when it achieved a 99% confidence interval of relative precision 0.2 for an estimate of the average time spent by a token at a selected place, using the procedure discussed in Section 7.3.3.

The normalized execution times for Persephone and GTPNA on the four series of models described above are plotted in Figures 8.3, 8.4, 8.5, and 8.6. These figures illustrate several interesting phenomena.

- The exponential time complexity from which the analytic techniques all suffer is evident in the curves for GTPNA. In Figure 8.5, for example,

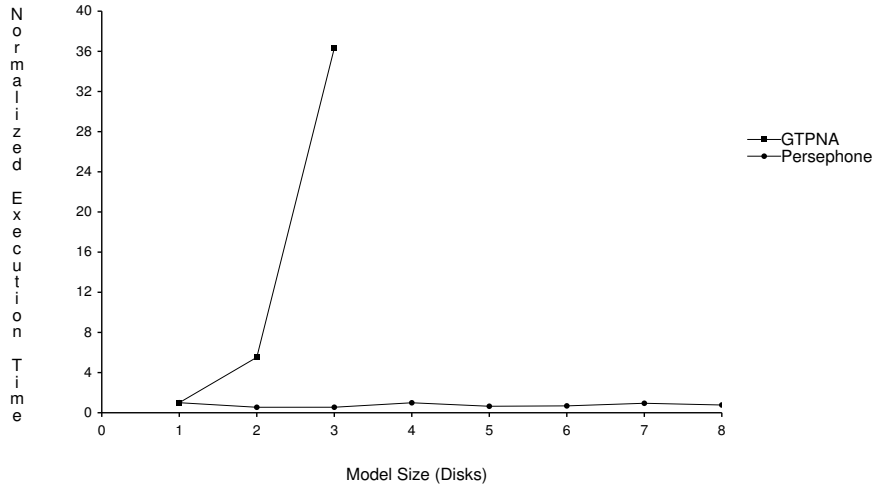

Figure 8.4: Normalized execution time on CentralServer, varying disks.

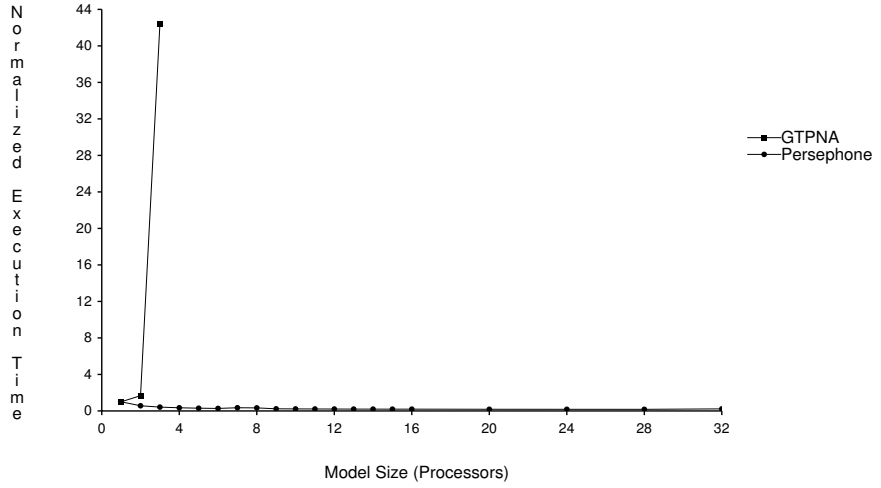

Figure 8.5: Normalized execution time on Multiprocessor, varying processors.

we see that GTPNA requires over 42 times as long to evaluate the three-processor model as it does the one-processor model.

- The exponential space complexity that plagues the analytic techniques is also shown by the GTPNA curves: three of them are missing data points

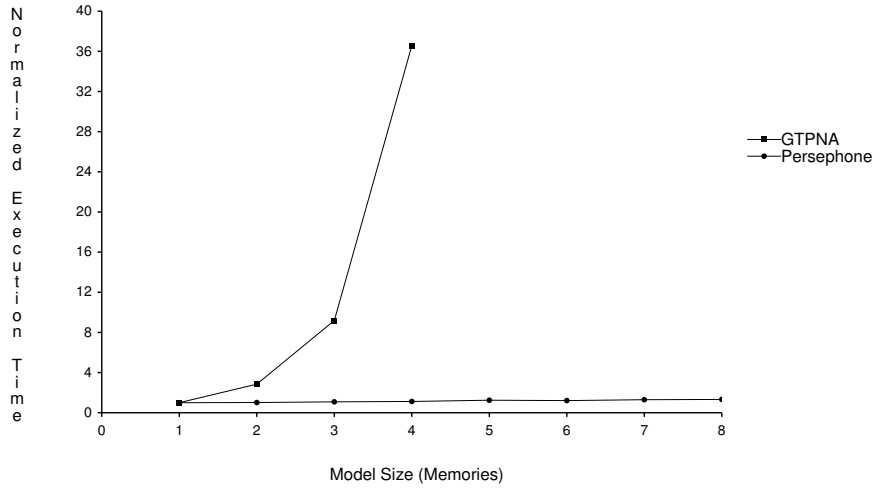

Figure 8.6: Normalized execution time on Multiprocessor, varying memories.

for the larger model sizes because GTPNA exhausts its available memory for these models and terminates abnormally.[4] The remaining curve (Figure 8.5) is missing data points because we manually aborted GTPNA when it seemed unlikely that it would terminate.[5]

Using a larger machine (a VAX 8550 with 96 Mbytes of memory) for these experiments gave qualitatively identical results, both in terms of the shapes of the curves and the sizes of the models GTPNA could successfully solve.

- In stark contrast to GTPNA, the curves for Persephone are nearly constant or exhibit very slow linear growth across the entire range of model sizes in these experiments.

- The Persephone curve in Figure 8.4 actually *decreases* as model size increases. In this case, this behavior is due to an interesting relationship

[4]GTPNA uses a set of statically sized tables. We recompiled GTPNA a number of times to increase these table sizes, but beyond a certain limit the run time system refused to execute it.

[5]We observed runs that consumed more than 150 CPU hours without finishing, probably due to enormous paging overheads or numerical convergence problems.

between the simulation and the system being simulated.

Consider a physical CentralServer system. The CPU is likely to be much faster than the disks. If we have only a single disk, the disk is probably the bottleneck of the system, severely limiting its throughput (the rate at which jobs flow through the system). As disks are added, offering parallel I/O to the CPU, each disk becomes less of a bottleneck, and the system's throughput improves.

Now consider the simulation of the CentralServer system. To achieve a specified confidence level for the estimate of a parameter, a number of observations of values of the parameter must be collected. For the CentralServer PPN, an observation is the length of time a token resides at `CPUP`; therefore, the greater the rate at which tokens cycle through the net, the sooner (in simulation time) a given number of observations can be made.

The single disk in the 1-disk CentralServer PPN is a bottleneck: tokens arrive from the CPU faster than the disk can route them back to the CPU. Since there are a fixed number of tokens in the PPN, the CPU is essentially starved. The simulation must run to a large simulation time to generate enough observations to achieve a specified confidence level, and simulating a large period of time can increase the execution time of the simulation.

Increasing the number of disks in the PPN alleviates the bottleneck. The token throughput is increased, so a given number of observations can be generated in a smaller amount of simulation time, reducing the execution time of the simulation.

- The Persephone curve in Figure 8.5 also decreases as model size increases. The explanation is that for some nets, larger numbers of tokens can generate greater numbers of events in a given period of simulation time, so the simulation can achieve a specified confidence level in a smaller amount of simulation time when it has more tokens, and simulating a smaller period of time can reduce the execution time of the simulation.

Table 8.2: Execution times (seconds) of Persephone (1 processor) and Seq on DB models.

Model	Persephone	Seq
DB All-At-Once	61.5	658.2
DB One-At-A-Time	104.4	129.6

Although we cannot claim, based on this small set of examples, that simulation will always dominate the analytic methods so convincingly, we are encouraged by these results because they demonstrate that such domination is at least possible.

One of the advantages Persephone has over GTPNA is its decentralized conflict resolution scheme, presented in Chapter 6, which allows it to avoid computing maximal sets. Note, however, that *any* PPN evaluation technique that resolves conflicts in the standard centralized way is at a similar disadvantage. To illustrate this concretely, we constructed a sequential PPN simulator (call it "Seq") that provides a subset of the functionality of Persephone but that resolves conflicts by computing maximal sets. Seq is written in C++ and runs on the same machine as Persephone, so its execution times are directly comparable. Table 8.2 lists the execution time in seconds for Persephone (running on a single processor) and Seq to simulate two of the PPN models. Persephone running on one processor is *faster* than Seq, a simulator that provides *less* functionality, and the gap in execution times widens as the sizes of the conflict sets grow.

Chapter 9

Summary

In this concluding chapter we summarize the contributions made by this thesis and suggest areas for further work.

9.1 Contributions

We have examined the use of parallel simulation for the analysis of performance Petri nets. Because the actions of the "physical processes" of PPNs require global information, these networks present new challenges for parallel simulation methodologies, which have assumed that physical processes interact only through the explicit exchange of messages.

We have developed a parallel simulation protocol for PPNs, the Transition Firing Protocol, that copes with the global nature of their actions. This protocol is based on the conservative approach to parallel simulation. We have introduced a new technique to conservative simulation, Selective Receive, that relaxes the traditional message receipt rules by allowing a logical process to sometimes ignore specific input channels when attempting to receive messages. Using information about the PPN simulation known statically, we use Selective Receive to allow receipt of messages that would normally have to remain pending, leading quickly to deadlock of the simulation.

We have also introduced a new conflict resolution procedure for PPNs that

was inspired by the parallel simulation approach. This conflict resolution procedure has been observed to be much faster in practice than the exponential procedures used previously.

Finally, we have created a prototype implementation of our parallel PPN simulator and measured its performance. Its speedup characteristics are similar to those obtained in other applications of parallel simulation, despite the apparently sequential flavor of PPNs induced by their global decision making procedures. More importantly, we have demonstrated that simulation can in fact be used to evaluate PPN models which are too large for the analytic PPN evaluation techniques, thereby extending the applicability of PPN modeling to larger systems.

9.2 Areas for Further Work

This work could be extended in a number of ways.

- *Optimize TFP for special cases.* By taking advantage of the local topology of part of a PPN, some steps can be eliminated from TFP for the nodes in the part, as noted in Chapter 4. In particular, nodes having single inputs and/or outputs are prime candidates for streamlined protocols.

- *Prove the correctness of TFP.* A formal proof of correctness would be interesting from a theoretical point of view and desirable from a practical point of view.

- *Improve the implementation of TFP.* A simple improvement that could yield significant performance gains would be to incorporate the future list technique [Nicol 88] into the lookahead computation employed by Persephone. Opportunities also exist for modifying the data structures and algorithms used by Persephone or by the run time systems underlying it to improve performance. Work has already been done on PRESTO toward this end [Faust & Levy 90].

- *Experiment with different conflict resolution strategies.* We have not quantified the effect of our "repeated trials" conflict resolution strategy, i.e., we have not studied how the number of trials necessary to resolve a conflict is related to the topology of the PPN. If such a study were performed, it might suggest how to redefine the conflict resolution strategy to ameliorate performance degradation in the presence of pathological topologies. Static or dynamic analysis of the PPN might be required to identify such topologies.

- *Experiment with different partitionings.* As noted in Section 3.3, the optimal decomposition of a PPN into LPs is probably a combination of node- and conflict set-based decompositions. Determining a good combination will require study of various combinations.

- *Investigate NUMA parallel simulation.* Non-uniform memory access (NUMA) platforms present new challenges for parallel simulation. Recent work has addressed parallel programming [Chase et al. 89] and the reimplementation of Synapse [Brown 90] on a network of shared-memory multiprocessors. Achieving performance on a NUMA platform similar to or better than the performance currently achieved by Persephone on a Sequent Symmetry (a uniform memory access platform) will probably require a different PPN decomposition, as well as heuristics for determining how to distribute LPs among processors.

- *Investigate optimistic parallel simulation of PPNs.* We suspect that, at least for certain PPN topologies, an optimistic parallel simulation would suffer greatly from excessive rollback. It would be interesting to characterize topologies for which excessive rollback does and does not occur, and to assess the overall performance of optimistic parallel simulation.

Glossary

It is not our intent to provide precise definitions of the terms listed herein, but rather to give intuitive explanations as a convenience for the reader.

arc multiplicity An integer associated with an arc. If the arc is from a place p to a transition t, the multiplicity specifies the number of tokens required at p to enable t (if the arc is a regular arc) or to inhibit t (if the arc is an inhibitor arc). In the former case, it also is the number of tokens removed from p each time t fires. If the arc is from t to p, the multiplicity specifies the number of tokens deposited at p each time t fires.

arc weight A real number associated with an arc that is used in conflict resolution (if the arc's source is a place) or deposit branching (if the arc's source is a transition).

channel A communication path in a parallel simulation from one LP to another.

classical Petri net A Petri net of the "original" variety, without added features. In particular, classical Petri nets do not incorporate time or branching probabilities.

conservative parallel simulation A type of parallel simulation in which an LP does not process a message timestamped τ until it is certain that no messages timestamped earlier than τ will arrive.

control channel A channel introduced by TFP that does not correspond to an arc in the PPN.

CPU Central processing unit.

decision place A place that has more than one regular output arc.

deposit branching A mechanism whereby a transition randomly chooses a place to deposit tokens at, instead of depositing tokens at each of its output places.

enable A transition is enabled when all of its input places have appropriate markings.

generalized inhibitor arc An inhibitor arc with an associated multiplicity. A generalized inhibitor arc of multiplicity m from a place p to a transition t inhibits t from firing if there are m or more tokens at p.

Generalized Timed Petri Net Analyzer A software tool that performs a stochastic analysis of a Generalized Timed Petri Net (a type of PPN).

GTPNA Generalized Timed Petri Net Analyzer.

inhibitor arc An inhibitor arc from a place p to a transition t inhibits t from firing if there are any tokens at p. Equivalent to a generalized inhibitor arc of multiplicity 1.

logical process An entity in a logical system. It represents a physical process.

logical system A network of LPs representing a physical system in a parallel simulation.

lookahead The length of simulation time (starting from the current simulation time) during which an LP can guarantee that it will send no message.

LP Logical process.

marking The number of tokens at a place. Also, the marking of a net is the vector of markings of its places.

optimistic parallel simulation A type of parallel simulation in which an LP processes messages as soon as they arrive. If a message timestamped τ is processed and later a message timestamped earlier than τ arrives, a "rollback" of the simulation occurs.

parallel simulation A single simulation employing multiple processors.

performance Petri net A Petri net augmented with time (firing delays) and branching probabilities.

Persephone Our prototype implementation of TFP: a conservative parallel simulator for PPNs.

Petri net A modeling tool that is both graphical and mathematical. Named after C. A. Petri, whose dissertation was the seminal work in the field.

physical process An entity in a physical system.

physical system A system being simulated.

place One of the two types of nodes in a Petri net. Drawn as a circle.

PPN Performance Petri net.

PRESTO A user-level threads package used by Synapse.

regular arc An arc that is not an inhibitor arc. See "arc multiplicity."

Selective Receive Our extension to conservative parallel simulation whereby an LP can at times ignore certain of its input channels when determining how far it may progress.

Synapse A conservative parallel simulation toolkit used by Persephone.

TFP Transition Firing Protocol.

thread A lightweight process.

timestamp The simulation time associated with a message.

token Tokens reside at places, and are consumed and produced by transition firings.

transition One of the two types of nodes in a Petri net. Drawn as a bar.

transition firing A firing of transition t removes tokens from the input places of t and deposits tokens at the output places of t.

transition firing delay The length of time between the start of a transition firing (when tokens are consumed) and its end (when tokens are deposited).

Transition Firing Protocol Our conservative parallel simulation protocol for PPNs.

untimed Petri net See "classical Petri net."

Bibliography

[Ajmone Marsan & Chiola 87] M. Ajmone Marsan and G. Chiola. On Petri nets with deterministic and exponentially distributed firing times. In [Rozenberg 87], pages 132–145.

[Ajmone Marsan et al. 87] M. Ajmone Marsan, G. Chiola, and A. Fumagalli. An accurate performance model of CSMA/CD bus LAN. In [Rozenberg 87], pages 146–161.

[Alla et al. 85] H. Alla, P. Ladet, J. Martinez, and M. Silva-Suarez. Modelling and validation of complex systems by coloured Petri nets; application to a flexible manufacturing system. In [Rozenberg 85], pages 15–31.

[Baer 87] J.-L. Baer. Modelling architectural features with Petri nets. In [Brauer et al. 87b], pages 258–277.

[Baldassari & Bruno 88] M. Baldassari and G. Bruno. An environment for object-oriented conceptual programming based on PROT nets. In [Rozenberg 88], pages 1–19.

[Barke 90] A. R. Barke. Use of Petri nets to model and test a control system. Master's thesis, University of Washington, Seattle, WA, 1990.

[Bershad et al. 88] B. N. Bershad, E. D. Lazowska, H. M. Levy, and D. B. Wagner. An open environment for building parallel programming systems. In *Symposium on Parallel Programming: Experience With Applications, Languages, and Systems*, pages 1–9, July 1988.

[Brauer 80] W. Brauer, editor. *Net Theory and Applications*, volume 84 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY, 1980.

[Brauer et al. 87a] W. Brauer, W. Reisig, and G. Rozenberg, editors. *Advances in Petri Nets 1986, Part I*, volume 254 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY, 1987.

[Brauer et al. 87b] W. Brauer, W. Reisig, and G. Rozenberg, editors. *Advances in Petri Nets 1986, Part II*, volume 255 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY, 1987.

[Brown 90] R. L. Brown. Conservative parallel discrete-event simulation in a non-uniform memory access system. Master's thesis, University of Washington, Seattle, WA, 1990.

[Carlier et al. 85] J. Carlier, P. Chretienne, and C. Girault. Modelling scheduling problems with timed Petri nets. In [Rozenberg 85], pages 62–82.

[Chandy & Misra 81] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(4):198–206, April 1981.

[Chase et al. 89] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 147–158, December 1989.

[Ciardo 87] G. Ciardo. Toward a definition of modeling power for stochastic Petri net models. In [PNP 87], pages 54–62.

[Diaz 87] M. Diaz. Petri net based models in the specification and verification of protocols. In [Brauer et al. 87b], pages 135–170.

[Drees et al. 87] S. Drees, D. Gomm, H. Plünnecke, W. Reisig, and R. Walter. Bibliography of Petri nets. In [Rozenberg 87], pages 309–451.

[Dugan & Ciardo 87] J. B. Dugan and G. Ciardo. Stochastic Petri net analysis of a replicated file system. In [PNP 87], pages 84–92.

[Faust & Levy 90] J. E. Faust and H. M. Levy. The performance of an object-oriented threads package. In *Proceedings of the 4th ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 278–288, October 1990.

[Feldbrugge & Jensen 87] F. Feldbrugge and K. Jensen. Petri net tool overview 1986. In [Brauer et al. 87b], pages 20–61.

[Feldbrugge 86] F. Feldbrugge. Petri net tools. In [Rozenberg 86], pages 203–223.

[Fujimoto 90] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):31–53, October 1990.

[Genrich & Stankiewicz-Wiechno 80] H. J. Genrich and E. Stankiewicz-Wiechno. A dictionary of some basic notions of net theory. In [Brauer 80], pages 519–535.

[Genrich et al. 80] H. J. Genrich, K. Lautenbach, and P. S. Thiagarajan. Elements of general net theory. In [Brauer 80], pages 21–163.

[Girault et al. 87] C. Girault, C. Chatelain, and S. Haddad. Specification and properties of a cache coherence protocol model. In [Rozenberg 87], pages 1–20.

[Greenberg & Lubachevsky 90] A. G. Greenberg and B. D. Lubachevsky. Unboundedly parallel simulations via recurrence relations. In *Proceedings of the 1990 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 1–12, May 1990.

[Hartung 88] G. Hartung. Programming a closely coupled multiprocessor system with high level Petri nets. In [Rozenberg 88], pages 154–174.

[Haus & Rodriguez 88] G. Haus and A. Rodriguez. Music description and processing by Petri nets. In [Rozenberg 88], pages 175–199.

[Hildebrand 85] T. Hildebrand. Design and programming of interfaces for monetic applications using Petri nets. In [Rozenberg 85], pages 197–214.

[Hoel et al. 72] P. G. Hoel, S. C. Port, and C. J. Stone. *Introduction to Stochastic Processes.* Houghton Mifflin, Boston, MA, 1972.

[Holliday & Vernon 86] M. A. Holliday and M. K. Vernon. The GTPN analyzer: Numerical methods and user interface. Technical Report 639, Computer Sciences Department, University of Wisconsin at Madison, Madison, WI, April 1986.

[Holliday & Vernon 87] M. A. Holliday and M. K. Vernon. A generalized timed Petri net model for performance analysis. *IEEE Transactions on Software Engineering*, SE-13(12):1297–1310, December 1987.

[Jantzen 87] M. Jantzen. Complexity of place/transition nets. In [Brauer et al. 87a], pages 413–434.

[Jensen & Schmidt 86] K. Jensen and E. M. Schmidt. Pascal semantics by a combination of denotational semantics and high-level Petri nets. In [Rozenberg 86], pages 297–329.

[Jensen 87] K. Jensen. Computer tools for construction, modification and analysis of Petri nets. In [Brauer et al. 87b], pages 4–19.

[Kaudel 87] F. J. Kaudel. A literature survey on distributed discrete event simulation. *ACM Simuletter*, 18(2):11–21, June 1987.

[Kernighan & Ritchie 78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Koseska-Toszewa & Mazurkiewicz 88]
V. Koseska-Toszewa and A. Mazurkiewicz. Net representation of sentences in natural languages. In [Rozenberg 88], pages 249–265.

[Law & Kelton 82] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis.* McGraw-Hill, New York, NY, 1982.

[Lin 90] Y.-B. Lin. *Understanding the Limits of Optimistic and Conservative Parallel Simulation.* PhD dissertation, University of Washington, Seattle, WA, 1990. Available as Department of Computer Science and Engineering Technical Report 90-08-02.

[Lopez & Palaez 88] I. Lopez and M. C. Palaez. Experiences in the use of Galileo to design telecommunication systems. In [Rozenberg 88], pages 283–306.

[Lu et al. 87] M. Lu, D. Zhang, and T. Murata. Stochastic net model for self-stability measures of fault tolerant clock synchronization. In [PNP 87], pages 104–110.

[Meijer 87] E. Meijer. Petri net models for the λ-calculus. In [Rozenberg 87], pages 162–180.

[Molloy 85] M. K. Molloy. Discrete time stochastic Petri nets. *IEEE Transactions on Software Engineering*, SE-11(4):417–423, April 1985.

[Murata 89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[Nicol 88] D. M. Nicol. Parallel discrete-event simulation of FCFS stochastic queueing networks. *SIGPLAN Notices*, 23(9):124–137, September 1988.

[Oberquelle 87] H. Oberquelle. Human-machine interaction and role/function/action-nets. In [Brauer et al. 87b], pages 171–190.

[Peterson 77] J. L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, September 1977.

[Peterson 81] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Petri 62] C. A. Petri. *Kommunikation mit Automaten.* PhD dissertation, Schriften des Institutes für Instrumentelle Mathematik, Bonn, Germany, 1962. Also, English translation by Clifford F. Greene, Jr. *Communication with Automata.* Technical Report RADC-TR-65-377, volume 1, supplement 1, Rome Air Development Center, Griffiss Air Force Base, New York, NY, 1966.

[Petri 80] C. A. Petri. Introduction to general net theory. In [Brauer 80], pages 1–19.

[Petri 87] C. A. Petri. "Forgotten topics" of net theory. In [Brauer et al. 87b], pages 500–514.

[PNP 87] *Proceedings of the 1987 International Workshop on Petri Nets and Performance Models*, Madison, WI, August 1987. IEEE Computer Society Press.

[Reed et al. 88] 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, April 1988.

[Reisig 85] W. Reisig. *Petri Nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, New York, NY, 1985.

[Reisig 87] W. Reisig. Petri nets in software engineering. In [Brauer et al. 87b], pages 63–96.

[Rozenberg 85] G. Rozenberg, editor. *Advances in Petri Nets 1984*, volume 188 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY, 1985.

[Rozenberg 86] G. Rozenberg, editor. *Advances in Petri Nets 1985*, volume 222 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY, 1986.

[Rozenberg 87] G. Rozenberg, editor. *Advances in Petri Nets 1987*, volume 266 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY, 1987.

[Rozenberg 88] G. Rozenberg, editor. *Advances in Petri Nets 1988*, volume 340 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY, 1988.

[Sanders & Meyer 87] W. H. Sanders and J. F. Meyer. Performability evaluation of distributed systems using stochastic activity networks. In [PNP 87], pages 111–120.

[Sauer et al. 80] C. H. Sauer, E. A. MacNair, and S. Salza. A language for extended queueing network models. *IBM Journal of Research and Development*, 24(6):747–755, November 1980.

[Sifakis 80] J. Sifakis. Performance evaluation of systems using nets. In [Brauer 80], pages 307–319.

[Stroustrup 86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.

[Tanenbaum 88] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall, Englewood Cliffs, NJ, second edition, 1988.

[Taubner 88] D. Taubner. On the implementation of Petri nets. In [Rozenberg 88], pages 418–439.

[Thomas 90] G. S. Thomas. XSIM: An X11-based configurable graph editor. Technical Note 163, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 1990.

[Thomas 91] G. S. Thomas. *Persephone User's Manual.* Department of Computer Science and Engineering, University of Washington, Seattle, WA, 1991.

[Valette 87] R. Valette. Nets in production systems. In [Brauer et al. 87b], pages 191–217.

[Vautherin 87] J. Vautherin. Parallel systems specifications with coloured Petri nets and algebraic specifications. In [Rozenberg 87], pages 293–308.

[Vernon et al. 86] M. K. Vernon, J. Zahorjan, and E. D. Lazowska. A comparison of performance Petri nets and queueing network models. Technical Report 86-09-09, Department of Computer Science and Engineering, University of Washington, Seattle, WA, September 1986.

[Voss 87a] K. Voss. Nets in data bases. In [Brauer et al. 87b], pages 97–134.

[Voss 87b] K. Voss. Nets in office automation. In [Brauer et al. 87b], pages 234–257.

[Wagner & Lazowska 89] D. B. Wagner and E. D. Lazowska. Parallel simulation of queueing networks: Limitations and potentials. In *Proceedings of the 1989 SIGMETRICS Conference and Performance '89*, pages 146–155, May 1989.

[Wagner 89] D. B. Wagner. *Conservative Parallel Discrete-Event Simulation: Principles and Practice.* PhD dissertation, University of Washington, Seattle, WA, 1989. Available as Department of Computer Science and Engineering Technical Report 89-09-03.