DISTRIBUTED GENERATION OF STATE SPACE FOR TIMED PETRI NETS

IRINA RADA

Distributed generation of state space for timed Petri nets

by

Irina Rada

A thesis submitted to the School of Graduate Studies

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Memorial University of Newfoundland

May 2000

St. John's Canada

Abstract

Development of complex systems is usually preceded by detailed studies of their models. For concurrent systems, Petri nets have proved to be a convenient modeling formalism because of their ability to express concurrency, synchronization, precedence constraints and nondeterminism. Timed Petri nets also take into account the durations of modeled activities, facilitating qualitative as well as quantitative analysis of models. The behavior of Petri nets is represented by their state spaces, which are Markov (or embedded Markov) chains. For large models these state spaces easily exceed the resources of a single computer system. Readily available networks of computers provide an attractive alternative to complex methods of state space reduction or aggregation.

The main objective of this project is to use a cluster of PC's or workstations for the state space generation of timed Petri nets. The distributed algorithm uses a divide and conquer technique: disjoint regions of the state graph are constructed on different machines. On each machine the communication is separated from the computation part, and is performed by two specialized concurrent processes: one receiving, and one sending messages. The implementation is based on PVM (Parallel Virtual Machine) using a modified version of TPN–tools, a software package for the analysis of timed Petri nets. Experiments performed on a cluster of 32 PC's connected via a 100 Mbps Ethernet show almost linear speedup for some classes of timed Petri nets.

Acknowledgements

I would like to express my sincere thanks to my supervisor, Dr. Wlodek Zuberek, for his guidance, help, and thoughtfulness throughout my program.

I am grateful to the School of Graduate Studies and to the Department of Computer Science for financial support.

Many thanks go to my friend Ulf Schünemann for our discussions and his helpful comments, and to Nolan White for technical assistance.

Finally, I want to thank all my friends for making my stay here enjoyable, and especially my family for moral support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Development of complex, real-world systems is usually preceded by detailed studies conducted on formal models. Formal, mathematical models are used for the verification of system's properties and for the derivation of its performance characteristics [16, 20, 22].

For systems which exhibit concurrent activities, Petri nets are a good choice of modeling formalism, because of their ability to express concurrency, synchronization, precedence constraints and non-determinism. Moreover, Petri nets "with time" (stochastic or timed) include the durations of modeled activities into the system's description and this allows the study of performance aspects of the modeled system. The analysis of a Petri net model of a system provides many useful insights; the net's qualitative properties characterize the system's behavioral properties [1, 26], while the ability to incorporate time into the description allows the derivation of that system's quantitative characteristics [4, 20, 39].

Three basic approaches to the analysis of Petri net models are known as structural analysis, reachability analysis and, for time-augmented nets, discrete-event simulation [32, 38]. Structural methods predict the properties of net models on the basis of their

structure (i.e., connections between elements). Structural analysis is usually rather simple, but it can be applied only to nets with special properties. Net simulation [45] is based on the fact that a (timed or stochastic) Petri net is a discrete event system, where the events are related to the net transition firings (occurrences). Simulation can be applied to a larger class of nets, but may sometimes not capture events which occur very rarely.

Reachability analysis is the most suitable method when a detailed analysis of the model's behavior is needed. Based on the exhaustive generation of all model's states and transitions between the states, reachability analysis answers questions about reachable states, liveness, boundedness, persistence, deadlock existence, etc. [26, 32]. The first and most memory consuming step in reachability analysis is to determine all the states of the net and the possible relations among them. This information is organized in a directed graph, called the reachability graph (in which the nodes are the net's states and the directed arcs represent the possible state-transitions). The reachability graph is used for checking the properties mentioned above. For timed and stochastic Petri nets (with deterministic or exponentially distributed firing times), this graph is a Markov chain, whose steady state behavior can be determined using known numerical methods [22, 34]. The steady state probabilities are used to derive performance measures of the net, from which performance aspects of the system can be obtained [4, 11].

The power of reachability analysis lies in its ability to characterize the exact behavior of the system. However, while yielding good results for simple models, this method cannot be applied to nets with very large state spaces. For such nets, the memory and computational requirements can be too large for a single machine. There are two basic methods to cope with this problem [10]: avoidance methods, which use net properties to obtain a smaller state space, and tolerance methods, which accept that the state

space is large and use various techniques (in particular parallel/distributed algorithms) to generate it. The current availability of clusters of workstations and portable libraries for distributed computing makes the second approach very attractive: the state space can be constructed in a distributed manner, using a collection of processors.

While there have been several papers published on distributed generation of state spaces of systems [28, 30] and on parallel and distributed state space generation for *stochastic* Petri nets [8, 6, 7, 9, 23], very little information is available for distributed analysis of *timed* Petri nets.

This thesis proposes a distributed algorithm for the generation of state space for timed Petri nets. The algorithm has been implemented in C++ using the TPN–tools [38], STL [36], and PVM [18] libraries, and then tested on the network of PC's and workstations in the Department of Computer Science, Memorial University of Newfoundland. Experimental results show almost linear speedup for some classes of timed Petri nets.

This thesis is organized as follows: Chapter 2 presents the theoretical background of the problem and an overview of the literature. Chapter 3 introduces the proposed distributed algorithm. Chapter 4 presents experimental results. Performance analysis, limitations, and possible extensions are discussed in Chapter 5.

Chapter 2

Petri nets and state space generation

The first two sections of this chapter provide a short introduction to place/transition Petri nets and the generation of their markings (in the case of basic and stochastic Petri nets, markings are often called states; for timed nets, markings and states are two different concepts). Section 2.3 presents Petri nets augmented with the durations of activities and discusses the generation of their state space. The final section reviews the current literature on distributed generation of the state space for stochastic Petri nets.

The presented definitions are similar to those in [39]. The notation follows [39, 38].

2.1 Introduction to basic Petri nets

All basic place/transition Petri nets are characterized by their structure, their current marking, and execution rules defining their behavior. Basic concepts of Petri nets are introduced in the following section.

2.1.1 Basic Petri nets

Definition 2.1 A Petri net is a triple $\mathbf{N} = (P, T, A)$ where:

- P is a finite set of elements called places,
- T is a finite set of elements called transitions,
- A is a set of directed arcs connecting places with transitions and transitions with places, i.e., $A \subseteq P \times T \cup T \times P$. □

Definition 2.2 Let $\mathbf{N} = (P, T, A)$ be a Petri net, t a transition, $t \in T$, and p a place, $p \in P$. The **input set**, Inp, and the **output set**, Out, of a transition t or a place p are defined as follows:

$$Inp(t) = \{p \mid (p,t) \in A\},\ Out(t) = \{p \mid (t,p) \in A\},$$
$$Inp(p) = \{t \mid (t,p) \in A\},\ Out(p) = \{t \mid (p,t) \in A\}.\ \square$$

The dynamic behavior of the net is represented by the distributions of the so-called tokens associated with places of the net. This association is called a marking of a net. A net with a marking is called a marked net.

Definition 2.3 A **marking** of a Petri net $\mathbf{N} = (P, T, A)$ is a function $m : P \to \mathbb{N}$ which assigns a non-negative number of tokens to each place of net $\mathbf{N}$. A place p is **marked** by the marking m if it contains at least one token, $m(p) > 0$, otherwise it is **unmarked** by m. A **marked net** is a pair $\mathbf{M} = (\mathbf{N}, m_0)$, where $\mathbf{N}$ is a Petri net and m_0 is a marking of $\mathbf{N}$, called the initial marking. □

A basic Petri net is a bipartite graph, usually drawn with circles representing places and rectangles representing transitions. The tokens are represented as black dots inside the circles.

Example 2.1 [38] The Petri net in Figure 2.1 models a consumer-producer bounded-buffer system. The subnet (t_1, p_1, t_2, p_2) represents the producer process which produces an item (t_1) and stores it in the buffer (t_2) provided that there is space for it (condition p_5). The subnet (t_3, p_3, t_4, p_4) represents the consumer process, which fetches an item from the buffer (t_3) provided that the buffer is not empty (condition p_6) and consumes it (t_4). □

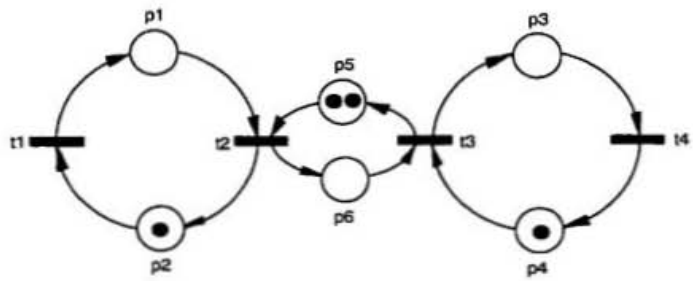


Figure 2.1: Producer-consumer bounded buffer model.

The behavior of a basic net is reflected by the changes of the marking function. A change of a marking function is performed by an **occurrence** (or a **firing**) of an enabled transition. A transition is enabled if all its input places contain at least one token. A transition occurs by simultaneously removing one token from all its input places and adding one token to all its output places.

Definition 2.4 Let $\mathbf{N} = (P, T, A)$ be a Petri net, t a transition, and m a marking of $\mathbf{N}$. The transition t is **enabled** by m iff:

$$\forall p \in Inp(t): \; m(p) \geq 1.$$

The set of all transitions enabled by a marking m is denoted $E(m)$. □

A marking m' is directly reachable from a marking m if m' can be obtained from m by an occurrence of an enabled transition.

Definition 2.5 Let $\mathbf{N} = (P, T, A)$ be a Petri net, and m and m' be two markings. m' is **directly reachable** from m iff there exists a transition $t \in T$ enabled by m such that:

$$\forall p \in P: \; m'(p) = \begin{cases} m(p) + 1, & \text{if } p \in Out(t) \text{ and } p \notin Inp(t); \\ m(p) - 1, & \text{if } p \in Inp(t) \text{ and } p \notin Out(t); \\ m(p), & \text{otherwise. } \square \end{cases}$$

The notation $m \stackrel{t}{\mapsto} m'$ indicates that m' is directly reachable from m by firing the transition t, and the notation $m \mapsto m'$ indicates that m' is directly reachable from m by firing some transition.

The general reachability relation between markings is defined as the reflexive transitive closure of the direct reachability relation.

Definition 2.6 A marking m' is (**generally**) **reachable** from a marking m ($m \stackrel{*}{\mapsto} m'$) if there exists a sequence of markings $m_0, \ldots, m_n$ such that $m_0 = m$, $m_n = m'$, and

$$\forall 0 < i \leq n : m_{i-1} \mapsto m_i. \; \square$$

Definition 2.7 The **reachability set**, $\mathcal{R}(\mathbf{M})$, of a marked Petri net $\mathbf{M} = (\mathbf{N}, m_0)$ is the set of all possible markings reachable from the initial marking m_0, i.e.,

$$\mathcal{R}(\mathbf{M}) = \{m \mid m_0 \stackrel{*}{\mapsto} m\}.$$

If the set $\mathcal{R}(\mathbf{M})$ of a marked net $\mathbf{M} = (\mathbf{N}, m_0)$ is finite, the net is **bounded**, otherwise it is **unbounded**. $\square$

The reachability set of a marked net, together with the direct reachability relation, form the reachability graph, which is a complete description of a marked net's behavior. For bounded nets this graph is finite.

Definition 2.8 The **reachability graph** of a marked Petri net $\mathbf{M} = (\mathbf{N}, m_0)$ is a labeled directed graph $G(\mathbf{M}) = (V, D, l)$ where:

- V is the set of vertices, $V = \mathcal{R}(\mathbf{M})$,
- D is the set of directed arcs, $D = \{(m_i, m_j) \mid m_i, m_j \in V \ \wedge \ m_i \mapsto m_j\}$,
- l is the arc labeling function, $l : D \rightarrow 2^T$; for each arc $(m_i, m_j) \in D$, $l(m_i, m_j)$ contains all those transitions whose firing transforms m_i into m_j:

$$l(m_i, m_j) = \{t \mid t \in E(m_i) \ \wedge \ m_i \stackrel{t}{\mapsto} m_j\}. \ \square$$

Example 2.2 Figure 2.2 shows the reachability graph for the net in Figure 2.1. $\square$

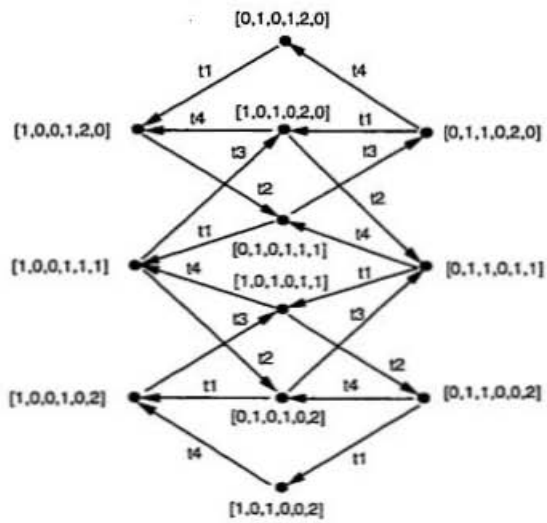


Figure 2.2: Reachability graph for the producer-consumer bounded-buffer model.

2.1.2 Extensions of basic Petri nets

Several extensions of basic Petri nets have been proposed in the literature. The most common one is attaching weights to arcs [26, 32]. Petri nets with weights have the

same expressive power as the basic nets, so the weights are used only as a modeling convenience.

An extension of basic Petri nets which significantly increases the modeling power of the basic model is the addition of the so-called inhibitor arcs [2]. Nets with inhibitor arcs are called inhibitor nets.

Definition 2.9 An **inhibitor** Petri net is a quadruple $N = (P, T, A, B)$ where (P, T, A) is a basic net, and B is a set on inhibitor arcs, $B \subseteq P \times T$, which is disjoint with A, $A \cap B = \emptyset$. The set of places connected by inhibitor arcs with a transition t is called the **inhibitor set** of t, and is denoted $Inh(t)$, $Inh(t) = \{p \in P \mid (p,t) \in B\}$. In inhibitor nets, a transition t is enabled by a marking m if all its input places are marked and all places in its inhibitor set are unmarked:

$$(\forall p \in Inp(t) : m(p) > 0) \wedge (\forall p \in Inh(t) : m(p) = 0). \square$$

Another important extension of basic Petri nets introduces the durations of modeled activities (Section 2.3).

There are several important structural properties of inhibitor nets.

Definition 2.10 Let $\mathbf{M} = (\mathbf{N}, m_0)$ be a marked inhibitor net. A place is **shared** if it belongs to the input set of more than one transition. A shared place is **guarded** if for each pair of transitions sharing it, there is another place which is in the input set of one transition, and in the inhibitor set of the other transition. A place is **free–choice** if the input sets and inhibitor sets of all transitions sharing it are identical. All transitions sharing a free–choice place are in a free–choice relation. A place is a **conflict place** if it is shared but it is neither guarded nor free–choice. Transitions sharing a conflict place are in **potential conflict** (conflicting transitions). An inhibitor net is **free–choice** iff each shared place is either guarded or free–choice. $\square$

In free–choice nets, the free–choice relation is an equivalence relation in the set of transitions, T, and therefore determines a partition of the set of transitions into free–choice equivalence classes:

$$Free(T) = \{T_1, T_2, ..., T_k\}.$$

2.1.3 Selection of firings for conflicting transitions

For the net shown in Figure 2.3 , the transitions sharing place p_1, i.e., t_2, t_4, and t_6, are in potential conflict. Transitions t_2 and t_4 are both enabled but only one can occur. A systematic approach is needed to determine all possible combinations of transition occurrences for nets with conflicting transitions.

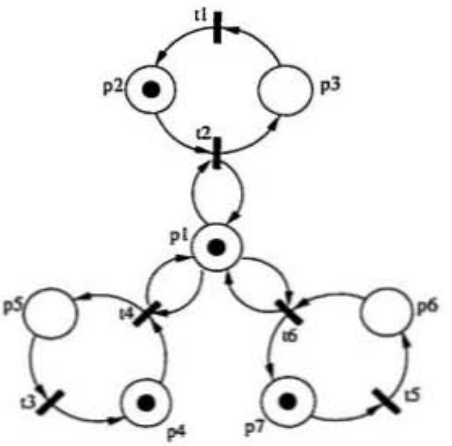


Figure 2.3: Central server model.

Definition 2.11 Let $\mathbf{N} = (P, T, A)$ be a net, and m a marking. For each transition $t \in T$, enabled by m, its **conflict class** $CC(m, t)$, is defined as follows:

$CC(m, t) = \{t' \in E(m) \mid Inp(t) \cap Inp(t') \neq \emptyset \vee$

$$\exists t'' \in E(m) : Inp(t) \cap Inp(t'') \neq \emptyset \ \wedge \ t'' \in CC(m, t'). \ \square$$

The notion of choice of the firing transition can be expressed formally as a **choice function** $c : T \mapsto [0,1]$, which assigns free–choice probabilities to free–choice equivalence classes and relative frequencies of firings to the conflicting transitions.

The different combinations of transitions which can start their firings for a given marking are described by the selection set, a set of selection functions which describe different "selections" of firings.

Definition 2.12 [44] Let $\mathbf{N} = (P, T, A)$ be a Petri net, and m a marking. A **selection** of the marking m is a function $g : T \to \mathbb{N}$, describing a possible combination of transitions which can start their firings for m, i.e., g is any function such that:

1. There exists a sequence of markings, $\sigma = (m_0, m_1, \ldots, m_k)$, and a corresponding sequence of transitions, $(t_1, \ldots, t_k)$, such that $m = m_0$, $t_j \in E(m_{j-1})$ for $j = 1, \ldots, k$, and:

$$\forall p \in P : \; m_j(p) = m_{j-1}(p) - \begin{cases} 1, & \text{if } p \in Inp(t_j); \\ 0, & \text{otherwise.} \end{cases}$$

2. The set of transitions enabled by the final marking m_k is empty, i.e., $E(m_k) = \emptyset$.

3. For each transition t, $g(t)$ is the number of occurrences of t in the sequence σ.

The set of all selections of a marking m is denoted by $Sel(m)$. □

Definition 2.13 Let $\mathbf{N} = (P, T, A)$ be a Petri net, c a choice function for $\mathbf{N}$, and m a marking of $\mathbf{N}$. A **selection graph** of the marking m is a rooted, directed, labeled, (acyclic) graph $G = (V, U, v_0, f, q, q_n)$ where:

- V is a finite set of vertices, which are pairs of functions (m_i, n_i), $m_i : P \to \mathbb{N}$, $n_i : T \to \mathbb{N}$, such that:

$$\forall p \in P : m_i(p) + \sum_{t \in Out(p)} n_i(t) = m(p);$$

- U is a set of directed arcs, $U \subset V \times V$, such that:

$$((m_i, n_i), (m_j, n_j)) \in U \iff \exists t_k \in E(m_i) : m_j = \text{sub}(m_i, t_k) \wedge n_j = \text{add}(n_i, t_k),$$

where:

$$\forall p \in P : \text{sub}(m_i, t_k)(p) = \begin{cases} m_i(p), & \text{if } p \notin Inp(t_k); \\ m_i(p) - 1, & \text{if } p \in Inp(t_k); \end{cases}$$

$$\forall t \in T : \text{add}(n_i, t_k)(t) = \begin{cases} n_i(t), & \text{if } t \neq t_k; \\ n_i(t) + 1, & \text{if } t = t_k; \end{cases}$$

- v_0 is the root, $v_0 = (m, n_0)$, where $n_0(t) = 0$ for all $t \in T$;

- f is an arc-labeling function which associates a transition $t \in T$ with each arc $(v_i, v_j) \in U$:

$$f((m_i, n_i), (m_j, n_j)) = t_k \iff t_k \in E(m_i) \wedge m_j = \text{sub}(m_i, t_k) \wedge n_j = \text{add}(n_i, t_k);$$

- q is another arc-labeling function which assigns, to each arc $(v_i, v_j) \in U$, the probability of transforming v_i into v_j:

$$\forall (v_i, v_j) \in U : q(v_i, v_j) = \begin{cases} 1.0, & \text{if } f(v_i, v_j) \text{ is conflict-free,} \\ c(f(v_i, v_j)), & \text{if } f(v_i, v_j) \text{ is free–choice,} \\ \frac{c(f(v_i, v_j))}{\sum_{t \in CC(m_i, f(v_i, v_j))} c(t)} & \text{otherwise.} \end{cases}$$

- q_n is a node labeling function, $q_n : V \mapsto [0, 1]$, which assigns a probability $q(x)$ to each node x of the selection graph such that $q_n(v_0) = 1$ and:

$$\forall x \in V - \{v_0\} : q(x) = \sum_{y \in Pred(x)} \frac{q_n(y) * q(y, x)}{\sum_{z \in Succ(y)} q(y, z)}. \square$$

Example 2.3 Let c be a choice function for the net in Figure 2.3 such that $c(t_i) = 0.1$ for $i = 1, 3, 5, 6$, $c(t_2) = 0.3$, and $c(t_4) = 0.2$. Figure 2.4 shows the selection graph for the initial marking $(1, 1, 0, 1, 0, 0, 1)$. The probabilities $q(v_i)$ are shown in brackets. $\square$

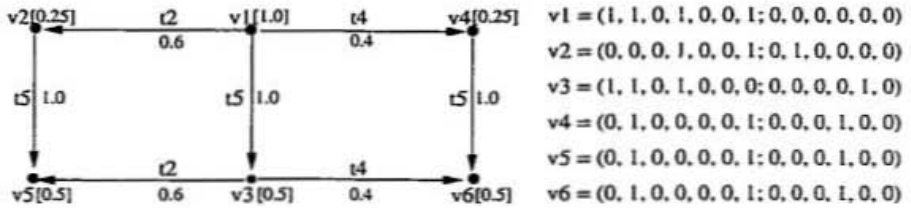


Figure 2.4: Selection graph for Figure 2.3 .

2.2 Generation of the reachability graph

A typical algorithm for the generation of the reachability graph of a (bounded) net is given below. There are several variations of this algorithm, but the differences are rather insignificant (e.g., using a stack instead of the queue [7]).

2.2.1 Sequential algorithm

```
1. algorithm sequential_reachability_graph_generation;
2. var m_0;                              (* initial marking *)
3.     rset := {m_0};                    (* set of markings *)
4.     arcs := ∅;                        (* set of arcs *)
5.     unexplored := ∅;                  (* queue of unexplored markings *)
6.     search_set := ∅;                  (* search tree *)
7. begin
8.     insert(search_set, m_0);
9.     insert(unexplored, m_0);
10.    while nonempty(unexplored) do
11.        m := remove(unexplored);
12.        for all m' ∈ successors(m) do
13.            if m' ∉ search_set then
14.                rset := rset ∪ {m'};
15.                insert(unexplored, m');
16.                insert(search_set, m')
17.            endif;
18.            arcs := arcs ∪ {(m, m')}
19.        endfor
20.    endwhile
21. end.
```

This algorithm constructs the reachability graph $G = (rset, arcs)$ for a Petri net $\mathbf{N}$

with an initial marking m_0. It uses a queue, *unexplored*, for the unexplored markings, and an auxiliary search data structure, *search_set*, for efficient checking of whether a marking has already been generated. The function *successor*(m) returns the markings directly reachable from m.

The algorithm terminates for nets having a finite reachability graph. It does not terminate, however, for nets with infinite state spaces (i.e., for unbounded nets).

The infinite state space of an unbounded net can be compressed to its coverability tree or to its coverability graph [26]. A marking in the coverability tree uses a special symbol to express that the number of tokens in a place can grow infinitely. The coverability graph can be obtained from the coverability tree by collecting together the nodes with the same marking and redirecting the arcs correspondingly.

Research has been conducted on handling the case of unbounded nets [14, 17, 35]. In [17], a solution is given for a special class of unbounded stochastic Petri nets (nest with exactly one unbounded place). Other approaches are based on using the coverability graph as a compressed representation of the reachability graph. Several method for constructing coverability graphs are given in [14, 35].

2.2.2 Net properties based on the reachability graph

Important net properties related to the reachability graphs include boundedness, reachability, coverability, persistence, conservativeness, liveness, etc. These properties are very useful in the modeling of systems because they can be directly related to the modeled systems' qualitative properties [20, 26, 31, 32].

Definition 2.14 Let $\mathbf{M} = (\mathbf{N}, m_0)$ be a marked net and k a natural number, $k \in \mathbb{N}$. A place p of the net is **k-bounded** iff the number of tokens assigned to p by any reachable marking does not exceed k. The net is **k-bounded** iff the number of tokens

assigned to any place by any reachable marking does not exceed k:

$$\forall m \in \mathcal{R}(\mathbf{M})\ \forall p \in P : m(p) \leq k.$$

A net is **bounded** if it is k–bounded for some $k \in \mathbb{N}$. A 1-bounded Petri net is called **safe**. □

Bounded nets are useful in modeling systems with finite capacity resources; finite capacity buffers, for instance, are usually represented by bounded places. The safeness property must usually be satisfied by nets in which places model conditions: the true or false value of the condition is reflected by the existence or absence of a token in the corresponding place.

Definition 2.15 A marking m of Petri net $\mathbf{N}$ is **dead** if no transition is enabled by m, i.e., $E(m) = \emptyset$. A marked net $\mathbf{M} = (\mathbf{N}, m_0)$ contains a **deadlock** if its set of reachable markings contains a dead marking:

$$\exists m \in \mathcal{R}(\mathbf{M}) : E(m) = \emptyset.$$

A marked net $\mathbf{M} = (\mathbf{N}, m_0)$ has a **livelock** if there exists a proper subset $\mathcal{S}$ of its reachability set, $\mathcal{S} \subset \mathcal{R}(\mathbf{M})$, such that once a marking from $\mathcal{S}$ is reached, no other element from $\mathcal{R}(\mathbf{M}) - \mathcal{S}$ can be reached:

$$\exists \mathcal{S} \subset \mathcal{R}(\mathbf{M})\ \forall m \in \mathcal{S}\ \forall m' \in \mathcal{R}(\mathbf{M}) : m \stackrel{*}{\mapsto} m' \Rightarrow m' \in \mathcal{S}.\ \square$$

Definition 2.16 Let $\mathbf{M} = (\mathbf{N}, m_0)$ be a marked net. The net is **live** iff for any reachable marking m and for any transition $t \in T$ there exists a marking reachable from m which enables t:

$$\forall m \in \mathcal{R}(\mathbf{M})\ \forall t \in T\ \exists m' \in \mathcal{R}(\mathbf{M}) : m \stackrel{*}{\mapsto} m' \wedge t \in E(m').\ \square$$

Net models of operating systems are usually required to be live; the property of liveness implies the absence of deadlocks.

Definition 2.17 Let $\mathbf{M} = (\mathbf{N}, m_0)$ be a marked net. A marking $m \in \mathcal{R}(\mathbf{M})$ is **coverable** iff there exists another marking, m', reachable from m, such that every place has at least the same number of tokens in m' as in the marking m:

$$\exists m' \in \mathcal{R}(M) : m \stackrel{*}{\mapsto} m' \ \wedge \ (\forall p \in P : m'(p) \ \geq m(p)). \ \square$$

Definition 2.18 [26] Let $\mathbf{M} = (\mathbf{N}, m_0)$ be a marked net. The net is **persistent** if, for any reachable marking m, $m \in \mathcal{R}(\mathbf{M})$, and for any two transitions enabled by m, the firing of one transition does not disable the other:

$$\forall m \in \mathcal{R}(\mathbf{M}) \ \forall t_1, t_2 \in E(m) : m \stackrel{t_1}{\mapsto} m' \ \Rightarrow t_2 \in E(m'). \ \square$$

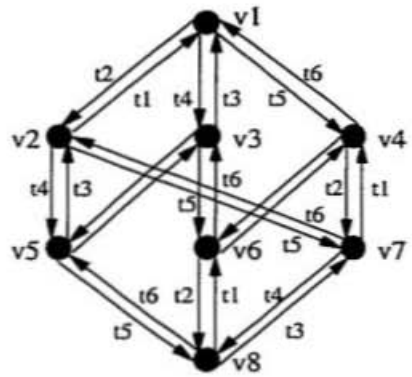
Definition 2.19 Let $\mathbf{M} = (\mathbf{N}, m_0)$ be a marked net. The net is **conservative** iff for any marking m reachable from m_0 the total number of tokens in m is the same as in m_0:

$$\forall m \in \mathcal{R}(M) : \sum_{p \in P} m(p) = \sum_{p \in P} m_0(p). \ \square$$

For nets in which tokens represent resources, the property of conservation reflects the preservation of resources in a system.

Because bounded nets have finite reachability graphs, all their behavioral properties can be verified by the exhaustive analysis of the reachability graph. For unbounded nets (which have infinite reachability sets) some of these properties, such as persistence and coverability, can be analyzed using the coverability tree [26, 31, 32].

Example 2.4 The net in Figure 2.3 models a central server with three kinds of jobs. From its state graph (shown in Figure 2.5) it can be seen that the net is live, safe, and conservative. $\square$



v1 = (1, 1, 0, 1, 0, 0, 1)
v2 = (1, 0, 1, 1, 0, 0, 1)
v3 = (1, 1, 0, 0, 1, 0, 1)
v4 = (1, 1, 0, 1, 0, 1, 0)
v5 = (1, 0, 1, 0, 1, 0, 1)
v6 = (1, 0, 1, 1, 0, 1, 0)
v7 = (1, 1, 0, 0, 1, 1, 0)
v8 = (1, 0, 1, 0, 1, 1, 0)

Figure 2.5: Graph of reachable markings for the net in Figure 2.3.

2.3 Time-augmented Petri nets

While basic Petri nets are useful for the analysis of qualitative properties of systems, they cannot be used for performance evaluation because they do not represent the durations of modeled activities. Several Petri nets "with time" have been proposed by introducing temporal descriptions in different ways [3, 4, 13, 5, 11, 24, 25, 27, 39, 41, 44].

There are three main aspects with regard to the addition of temporal information to Petri nets: time can be associated with places or with transitions, timed activities can be deterministic or stochastic, and different "firing execution policies" can be used.

Two classes of nets in which time is associated with transitions (timed transitions) are known as stochastic Petri nets and timed Petri nets. In stochastic nets, the time is introduced in terms of a delay before the (instantaneous) firing of a transition occurs; in timed nets, the time determines the duration of the transition's firings. For both these classes of nets the graphs of reachable states are Markov chains. The steady state probabilities of the states of a Markov chain can be determined using known techniques [34], and can be used for determining quantitative properties of the net models.

This thesis is concerned with the generation of the state space of timed Petri nets. An overview of similar research that has been conducted for stochastic Petri nets, is

given in the last section of this chapter. A brief introduction to stochastic Petri nets follows.

2.3.1 Stochastic Petri nets

In stochastic Petri nets [4, 11, 13, 24, 25, 27] there is a time delay from the moment when a transition becomes enabled to the moment when it fires. This time is a random variable with an exponential distribution.

Definition 2.20 A **stochastic Petri net** (SPN) is a pair $\mathcal{S} = (\mathbf{M}, d)$, where:

- $\mathbf{M} = (\mathbf{N}, m_0)$ is a marked net,
- d is a function which, for each transition $t \in T$, specifies the rate of the firing delay associated with it, $d : T \rightarrow \mathbb{R}^+$. The firing delay of a transition $t \in T$ is an exponentially distributed random variable X_t with the rate $d(t)$; the probability that the delay is greater than y, $y > 0$, is:

$$Prob(X_t > y) = e^{-y*d(t)}. \ \square$$

In stochastic nets the firing delays associated with transitions can be marking dependent.

A stochastic net has the following firing behavior: once a transition t is enabled, the tokens must remain in t's input places for the time described by the firing delay function. When this time has elapsed, the tokens are removed from the input places of the firing transition and added to the output places of this transition.

Similarly to basic Petri nets, a state of the net is completely described by the token distribution in places. The state space of stochastic nets is therefore the reachability set of basic Petri nets.

Molloy has shown [24] that due to the memoryless property of the exponential distribution, the reachability graph of an SPN is a continuous–time Markov chain [16, 20]. For ergodic continuous-time Markov chains (i.e., for Markov chains which have a steady–state solution), the steady–state probabilities can be determined by solving a system of linear equations [34]. The steady–state probabilities can be used for determining the mean number of tokens in a place, the mean number of a transition's firings in the time unit, the throughput of a transition, and many other properties [4, 11].

A popular generalization of stochastic Petri nets is known as generalized stochastic Petri nets (GSPN). In GSPNs [3, 5, 13], there are two classes of transitions: transitions with exponentially distributed firing times (timed transitions), and transitions having the firing delay equal to zero (immediate transitions). The reachability graph of a GSPN is an embedded Markov chain [3, 4].

2.3.2 Timed Petri nets

In timed Petri nets [39, 41, 44], the firing of a transition is a non-instantaneous activity; the transition starts the firing by removing the tokens from the input places, it continues the firing for a specified period of time, and then finishes the firing by adding tokens to the output places. The firing of a transition starts as soon as the transition is enabled (although some enabled transitions do not start their firings because of conflicts). Several concurrent occurrences of a transition's firing can take place if the transition remains enabled after starting a firing.

Timed Petri nets whose transitions have deterministic firing times are known as D–timed Petri nets, while those whose transitions have exponentially distributed firing times are called M–timed Petri nets (Markovian nets).

2.3.3 M–timed Petri nets

In M–timed Petri nets [39, 41, 42, 44], the transitions' firing times are exponentially distributed random variables.

Definition 2.21 [44] An **M–timed Petri net** is a triple $\mathcal{T}_M = (\mathbf{M}, c, f)$ where:

- $\mathbf{M} = (\mathbf{N}, m_0)$ is a marked Petri net,
- $c : T \rightarrow [0, 1]$ is a choice function which assigns free–choice probabilities to free–choice equivalence classes and relative frequencies of firings to the conflicting transitions,
- $f : T \rightarrow \mathbb{R}^+$ is the firing-rate function, which assigns the rate of firings, $f(t)$, to each transition t of the net. The firing time of a transition t is an exponentially distributed random variable $X(t)$, with the rate $f(t)$; the probability that the firing time is greater than y, $y > 0$, is:

$$Prob(X(t) > y) = e^{-y*f(t)}. \square$$

Example 2.5 Figure 2.6 shows an M–timed net for the problem of three dining philosophers. Places A, B, and C represent the forks, places $p1b$, $p2b$, and $p3b$ represent, respectively, philosopher "1", "2" and "3" wanting to eat, and places $p1a$, $p2a$, and $p3a$ represent the state of a philosopher after eating. There are three "eat" transitions and three "think" transitions. An "eat" transition (for instance eat_1) is enabled if both forks are available and the philosopher is hungry (i.e. places A, B and $p1b$ are marked). Firing times associated with "eat" and "think" transitions are exponentially distributed random variables with the rates 5 and 3, respectively. $\square$

A state description of a timed Petri net must specify the distribution of tokens over net's places and also the numbers of (active) firings of transitions.

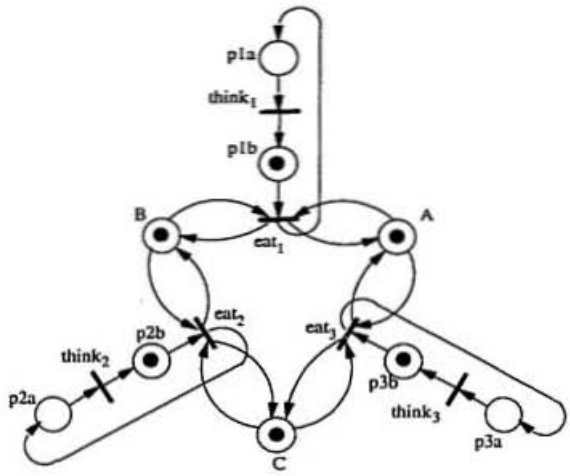


Figure 2.6: Three dining philosophers.

A state (or an "instantaneous description") of an M–timed net is a pair of functions: a marking function specifying the distribution of tokens in places, and a firing function which describes the numbers of the active firings of all transitions.

Definition 2.22 [44] A **state** of an M–timed Petri net $\mathcal{T}_M = (\mathbf{M}, c, f)$ is a pair $s = (m, n)$ where:

- m is a marking function, $m : P \to \mathbb{N}$,
- n is a firing function, $n : T \to \mathbb{N}$, where $n(t)$ is the number of active firings (firings which have been initiated but not finished) of transition t. □

Definition 2.23 An **initial state** of an M–timed Petri net $\mathcal{T}_M = (\mathbf{M}, c, f)$ is a pair $s = (m, n)$ where n is a selection function for m_0, $n \in Sel(m_0)$, and the marking m is defined as:

$$\forall p \in P : m(p) = m_0(p) - \sum_{t \in Out(p)} n(t). \; \square$$

An M–timed Petri net can have several initial states.

Example 2.6 States are often represented using vector notation for functions m and n, i.e., assuming some ordering of places and transitions. For this example, the ordering is:

$$s = [A, B, C, p1a, p1b, p2a, p2b, p3a, p3b; think_1, think_2, think_3, eat_1, eat_2, eat_3].$$

For the given initial marking (places A, B, C, $p1b$, $p2b$, and $p3b$ are marked, i.e., initially all philosophers are hungry and all forks are available), there are three enabled conflicting transitions (eat_1, eat_2, and eat_3), but only one of them can start its firing. The net has thus three initial states:

$s_1 = [0, 0, 1, 0, 0, 0, 1, 0, 1; 0, 0, 0, 1, 0, 0]$, if eat_1 is selected to fire, or

$s_2 = [1, 0, 0, 0, 1, 0, 0, 0, 1; 0, 0, 0, 0, 1, 0]$, if eat_2 is selected to fire, or

$s_3 = [0, 1, 0, 0, 1, 0, 1, 0, 0; 0, 0, 0, 0, 0, 1]$, if eat_3 is selected to fire. □

For M–timed nets the "direct reachability" relation is an extension of that for marked nets (Section 2.1).

Definition 2.24 [44] Let $\mathcal{T}_M = (\mathbf{M}, c, f)$ be an M–timed Petri net. A state $s_j = (m_j, n_j)$ is **directly reachable** (or (t_k, g_l)-reachable) from a state $s_i = (m_i, n_i)$ iff:

1. $n_i(t_k) > 0$;

2. $g_l \in Sel(m_i')$;

3. $\forall p \in P : m_j(p) = m_i'(p) - \sum_{t \in Out(p)} g_l(t)$;

4. $\forall t \in T : n_j(t) = n_i(t) + g_l(t) - \begin{cases} 1, & \text{if } t = t_k; \\ 0, & \text{otherwise;} \end{cases}$

5. $\forall p \in P : m_i'(p) = m_i(p) + \begin{cases} 1, & \text{if } p \in Out(t_k), \\ 0, & \text{otherwise. } \square \end{cases}$

State s_i is transformed into state s_j when one of the firing transitions (in this case t_k) ends its firing (1) and deposits tokens into its output places (5), transforming the marking m_i into a marking m', and m' enables new firings, which are described by the selection function g_l (2, 3, 4).

Example 2.7 The state $s_4 = [1,0,0,0,0,0,0,0,1;1,0,0,0,1,0]$ is directly reachable from state $s_1 = [0,0,1,0,0,0,1,0,1;0,0,0,1,0,0]$; when transition eat_1 ends its firing, the tokens are deposited into A, B, and $p1a$, the new marking ([1,1,1,1,0,0,1,0,1]) enables transition $think_1$, which can immediately start its firing, and transitions eat_2 and eat_3, which are in conflict, so only one can fire. There are two possible selection functions, one selecting $think_1$ and eat_2 to fire, the other selecting $think_1$ and eat_3. If the first selection function is used, the next state is s_4. □

The relation $s_i \mapsto s_j$ denotes that s_j is directly reachable from s_i, while $s_i \overset{t_k,g_l}{\mapsto} s_j$ indicates that s_j is (t_k, g_l)-reachable from s_i.

As in the case of basic Petri nets, the general reachability relation is defined as the reflexive transitive closure of the direct reachability relation.

Definition 2.25 Let $\mathcal{T}_M = (\mathbf{M}, c, f)$ be an M–timed Petri net. A state s_j is (**generally**) **reachable** from a state s_i ($s_i \overset{*}{\mapsto} s_j$) if there is a sequence of states $s_{i_0}, \ldots, s_{i_k}$ such that $s_{i_0} = s_i$, $s_{i_k} = s_j$, and s_{i_l} is directly reachable from $s_{i_{l-1}}$ for $l = 1, \ldots, k$. □

Definition 2.26 The **set of reachable states**, $\mathcal{R}(\mathcal{T}_M)$ of an M–timed net $\mathcal{T}_M = (\mathbf{M}, c, f)$ is the set of all states which are (generally) reachable from any initial state of $\mathcal{T}_M$. □

Definition 2.27 [44] A **state graph** of an M–timed Petri net $\mathcal{T}_M$ is a labeled directed graph $G(\mathcal{T}_M) = (V, D, h, q)$ where:

- V is a set of vertices, $V = \mathcal{R}(\mathcal{T}_M = (\mathbf{M}, c, f))$,

- D is a set of directed arcs: $D = \{(s_i, s_j) \mid s_i, s_j \in V \wedge s_i \mapsto s_j\}$,
- h is a node labeling function, $h : V \to \mathbb{R}^+$, which specifies the average holding times of states:

$$\forall s = (m, n) \in S : h(s) = 1/\sum_{t \in T} f(t) * n(t),$$

- q is an arc labeling function, $q : D \to [0, 1]$, which assigns the probability of state transition from $s_i = (m_i, n_i)$ to s_j to each arc (s_i, s_j), where s_j is (t_k, g_l)-reachable from s_i; $q(s_i, s_j) = q' * q''$ where q' is the probability that t_k terminates its firing in state s_i:

$$q' = \frac{n(t_k) * f(t_k)}{\sum_{t \in T} n(t) * f(t)},$$

and q'' is the probability of the selection g_l after the end of the firing of t_k, $q'' = q(m_{i,k}, g_l)$ where $m_{i,k}$ is the marking of the net after the end of t_k's firing:

$$\forall p \in P : m_{i,k}(p) = \begin{cases} m_i(p) + 1, & \text{if } p \in Out(t_k), \\ m_i(p), & \text{otherwise;} \end{cases}$$

and $q(m_{i,k}, g_l)$ is the probability of the node corresponding to g_l in the selection graph for $m_{i,k}$ (Section 2.1.3). □

A state graph of an M–timed net is a continuous-time Markov chain whose stationary probabilities of states, $x(s)$, $s \in \mathcal{R}(\mathcal{T}_M)$, are determined from the set of equilibrium equations [44]:

$$\begin{cases} \sum_{1 \le j \le K} q(s_j, s_i) * x(s_j)/h(s_j) = x(s_i)/h(s_i); \ i = 1, ..., K-1; \\ \sum_{1 \le j \le K} x(s_i) = 1; \end{cases}$$

where K is the number of reachable states.

Many performance measures can be derived from these probabilities [44].

Example 2.8 Table 2.1 shows the state space of the net shown in Figure 2.6. □

Table 2.1: State space for the net in Figure 2.6.

s	$[m;n]$	$h(s)$	next states	transition prob.
s_1	$[0,0,1,0,0,0,1,0,1;0,0,0,1,0,0]$	2.0	$\{s_4, s_5\}$	$\{0.5, 0.5\}$
s_2	$[1,0,0,0,1,0,0,0,1;0,0,0,0,1,0]$	2.0	$\{s_6, s_7\}$	$\{0.5, 0.5\}$
s_3	$[0,1,0,0,1,0,1,0,0;0,0,0,0,0,1]$	2.0	$\{s_8, s_9\}$	$\{0.5, 0.5\}$
s_4	$[1,0,0,0,0,0,0,0,1;1,0,0,0,1,0]$	1.42	$\{s_2, s_{10}\}$	$\{0.29, 0.71\}$
s_5	$[0,1,0,0,0,0,1,0,0;1,0,0,0,0,1]$	1.42	$\{s_3, s_{11}\}$	$\{0.29, 0.71\}$
s_6	$[0,0,1,0,0,0,0,0,1;0,1,0,1,0,0]$	1.42	$\{s_1, s_{10}\}$	$\{0.29, 0.71\}$
s_7	$[0,1,0,1,0,0,0,0,0;0,1,0,0,0,1]$	1.42	$\{s_3, s_{12}\}$	$\{0.29, 0.71\}$
s_8	$[0,0,1,0,0,0,1,0,0;0,0,1,1,0,0]$	1.42	$\{s_1, s_{11}\}$	$\{0.29, 0.71\}$
s_9	$[1,0,0,0,1,0,0,0,0;0,0,1,0,1,0]$	1.42	$\{s_2, s_{12}\}$	$\{0.29, 0.71\}$
s_{10}	$[0,1,0,0,0,0,0,0,0;1,1,0,0,0,1]$	1.11	$\{s_7, s_5, s_{13}\}$	$\{0.22, 0.22, 0.56\}$
s_{11}	$[1,0,0,0,0,0,0,0,0;1,0,1,0,1,0]$	1.11	$\{s_9, s_4, s_{13}\}$	$\{0.22, 0.22, 0.56\}$
s_{12}	$[0,0,1,0,0,0,0,0,0;0,1,1,1,0,0]$	1.11	$\{s_8, s_6, s_{13}\}$	$\{0.22, 0.22, 0.56\}$
s_{13}	$[1,1,1,0,0,0,0,0,0;1,1,1,0,0,0]$	1.66	$\{s_{12}, s_{11}, s_{10}\}$	$\{0.33, 0.33, 0.33\}$

2.3.4 D–timed Petri nets

In D–timed Petri nets [40, 44], the transitions' firing times are constant (positive real numbers).

Definition 2.28 [44] A **D–timed Petri net** is a triple $\mathcal{T}_D = (\mathbf{M}, c, f)$ where:

- $\mathbf{M} = (\mathbf{N}, m_0)$ is a marked Petri net,
- $c : T \to [0, 1]$ is a choice function which assigns free–choice probabilities to free–choice equivalence classes and relative frequencies of firings to the conflicting transitions,
- $f : T \to \mathbb{R}^+$ is a firing-time function which assigne the firing time $f(t)$ to each transition $t \in T$. □

Because D–timed nets do not have the memoryless property, in addition to the token distribution over places of the net and the number of firings of transitions, a state of

a D–timed net must also specify the remaining-firing-time for each occurrence of each active transition.

Definition 2.29 [44] A **state** of a D–timed Petri net $\mathcal{T}_D$ is a triple $s = (m, n, r)$ where:

- m is a marking function, $m : P \rightarrow \mathbb{N}$,
- n is a firing function (as for M–timed nets), $n : T \rightarrow \mathbb{N}$,
- r is a remaining-firing-time function $r : T \mapsto \mathbb{N} \mapsto \mathbb{R}^+$, which assigns the remaining firing time to each independent firing (if any) of each transition. Function r is partial; if $n(t) = k$ and $k > 0$, then $r(t)$ is a vector of k nonnegative nondecreasing real numbers denoted by $r(t)[1], r(t)[2], ..., r(t)[k]$; if $n(t) = 0$, $r(t)$ is undefined. □

Definition 2.30 [44] An **initial state** s of a D–timed Petri net $\mathcal{T}_D = (\mathbf{M}, c, f)$ is a triple $s = (m_i, n_i, r_i)$ where:

- m_i is a marking function,

$$\forall p \in P : m_i(p) = m_0(p) - \sum_{t \in Out(p)} n_i(t),$$

- n_i is a selection function of m_0, $n_i \in Sel(m_0)$,
- r_i is a remaining-firing-time function defined as:

$$\forall t \in T : r_i(t)[k] = \begin{cases} f(t), & \text{if } n_i(t) > 0 \text{ and } 1 \leq k \leq n_i(t); \\ \text{undefined}, & \text{otherwise. } \square \end{cases}$$

A D–timed Petri net can have several initial states.

Definition 2.31 [44] Let $\mathcal{T}_D = (\mathbf{M}, c, f)$ be a D–timed Petri net. A state $s_j = (m_j, n_j, r_j)$ is **directly reachable** (or g_k-reachable) from a state $s_i = (m_i, n_i, r_i)$ iff the following conditions are satisfied:

1. $g_k \in Sel(m'_i)$;

2. $\forall p \in P : m_j(p) = m'_i(p) - \sum_{t \in Out(p)} g_k(t)$;

3. $\forall t \in T : n_j(t) = n_i(t) - d_i(t) + g_k(t)$;

4. $$\forall t \in T : r_j(t)[l] = \begin{cases} r_i(t)[l + d_i(t)] - h_i, & \text{if } 1 \leq l \leq n_i(t) - d_i(t); \\ f(t), & \text{if } n_i(t) - d_i(t) < l \leq n_j(t); \end{cases}$$

where:

5. $\forall p \in P : m'_i(p) = m_i(p) + \sum_{t \in Inp(p)} d_i(t)$;

6. $$\forall t \in T : d_i(t) = \begin{cases} z, & \text{if } n_i(t) \geq z \text{ and } r_i(t)[l] = h_i \text{ for } 1 \leq l \leq z, \\ 0, & \text{otherwise;} \end{cases}$$

7. $h_i = \min_{t \in T \wedge n_i(t) > 0} (r_i(t)[1])$. □

The general reachability relation between states and the set of reachable states $\mathcal{R}(()\mathcal{T}_D)$ are defined in a similar manner as for M–timed nets.

Definition 2.32 [44] A **state graph** of a D–timed Petri net is a labeled directed graph $G(\mathcal{T}_D) = (V, D, h, q)$ where:

- V is the set of vertices, $V = \mathcal{R}(\mathcal{T}_D)$,
- D is the set of directed arcs, $D = \{(s_i, s_j) \mid s_i, s_j \in V \wedge s_i \mapsto s_j\}$,
- h is a node labeling function, $h : V \to \mathbb{R}^+$, which specifies the holding times of states:

$$\forall s_i = (m_i, n_i, r_i) \in S : h(s_i) = \min_{t \in T \wedge n_i(t) \geq 0} (r_i(t)[1]),$$

- q is an arc labeling function, $q : D \to [0, 1]$, which assigns the probability of transition from s_i to s_j to each arc (s_i, s_j) where s_j is g_k-reachable from s_i:

$$q(s_i, s_j) = q(m', g_k)$$

and m' is the marking after the termination of the firings with the smallest remaining firing time (as determined in Definition 2.31), and $q(m', g_k)$ is the probability of the node corresponding to g_k in the selection graph for m'. □

Example 2.9 For the net shown in Figure 2.1 with firing times $f(t_1) = 1$, $f(t_2) = f(t_3) = 0.5$, and $f(t_4) = 2.5$, the only initial state is $s_0 = [0, 0, 0, 1, 2, 0; 1, 0, 0, 0; 1, 0, 0, 0]$. The reachability set of the net is shown in Table 2.2. The net is conflict–free, so there is only one next state for each reachable state. □

Table 2.2: State space for the net in Figure 2.1.

s	$[m;$	$n;$	$r]$	$h(s)$	next state	transition prob.
s_1	$[0,0,0,2,2,0;$	$1,0,0,0;$	$1,0,0,0]$	1.0	s_2	1.0
s_2	$[0,0,0,1,1,0;$	$0,1,0,0;$	$0,0.5,0,0]$	0.5	s_3	1.0
s_3	$[0,0,0,0,1,0;$	$1,0,1,0;$	$1,0,0.5,0]$	0.5	s_4	1.0
s_4	$[0,0,0,0,2,0;$	$1,0,0,1;$	$0.5,0,0,2.5]$	0.5	s_5	1.0
s_5	$[0,0,0,0,1,0;$	$0,1,0,1;$	$0,0.5,0,2]$	0.5	s_6	1.0
s_6	$[0,0,0,0,1,1;$	$1,0,0,1;$	$1,0,0,1.5]$	1.0	s_7	1.0
s_7	$[0,0,0,0,0,1;$	$0,1,0,1;$	$0,0.5,0,0.5]$	0.5	s_8	1.0
s_8	$[0,0,0,0,0,1;$	$1,0,1,0;$	$1,0,0.5,0]$	0.5	s_9	1.0
s_9	$[0,0,0,0,1,1;$	$1,0,0,1;$	$0.5,0,0,2.5]$	0.5	s_{10}	1.0
s_{10}	$[0,0,0,0,0,1;$	$0,1,0,1;$	$0,0.5,0,2]$	1.0	s_{11}	1.0
s_{11}	$[0,0,0,0,0,2;$	$1,0,0,1;$	$1,0,0,1.5]$	0.5	s_{12}	1.0
s_{12}	$[1,0,0,0,0,2;$	$0,0,0,1;$	$0,0,0,0.5]$	0.5	s_{13}	1.0
s_{13}	$[1,0,0,0,0,1;$	$0,0,1,0;$	$0,0,0.5,0]$	0.5	s_{14}	1.0
s_{14}	$[0,0,0,0,0,1;$	$0,1,0,1;$	$0,0.5,0,2.5]$	0.5	s_{15}	1.0
s_{15}	$[0,0,0,0,0,2;$	$1,0,0,1;$	$1,0,0,2]$	1.0	s_{16}	1.0
s_{16}	$[1,0,0,0,0,2;$	$0,0,0,1;$	$0,0,0,1]$	1.0	s_{13}	1.0

The state graph of a free–choice D–timed net is a discrete-time discrete-state semi-Markov process [16] whose embedded stationary probabilities $y(s), s \in \mathcal{R}(\mathcal{T}_D)$, are determined by solving a system of linear equations [44]:

$$\begin{cases} \sum_{1 \le j \le K} q(s_i, s_j) = y(s_i);\ i = 1, ..., K-1; \\ \sum_{1 \le j \le K} y(s_i) = 1; \end{cases}$$

The stationary probabilities of states, $x(s), s \in \mathcal{R}(\mathcal{T}_D)$, are determined from the embedded stationary probabilities [44]:

$$\forall\ s \in \mathcal{R}(\mathcal{T}_D):\ x(s_i) = y(s_i) * h(s_i) / \sum_{1 \le j \le K} y(s_j) * h(s_j).$$

where K is the number of states in the reachability set $\mathcal{R}(\mathcal{T}_D)$.

Detailed information on timed Petri nets, their analysis and applications can be found in [39, 40, 41, 42, 44]. A software package, TPN–tools [38], has been developed for the analysis of timed Petri nets.

2.4 Distributed state space generation for stochastic Petri nets

State space generation for nets with large numbers of states is a difficult task because of the large memory requirements. This impediment can be avoided by using the (combined) memory available in a cluster of computers. Research in this direction has been conducted in the last few years [8, 6, 7, 9, 19, 21, 23, 28, 30]. Most of the authors emphasize that the main advantage of the distributed algorithm is the possibility of generating state spaces which were too large for the memory of a single workstation. However, such an approach introduces a communication overhead induced by the necessary coordination between the processes cooperating in the distributed algorithm.

This section reviews several aspects of distributed state space generation for stochastic Petri nets.

2.4.1 General framework

A natural approach to distributed state space generation is to use a "divide and conquer" technique, i.e., to construct disjoint state subgraphs on different computers, and then integrate them to obtain the entire state graph.

Requirements: In order to minimize the communication overhead and to achieve a good speedup, a distributed algorithm for state space generation should satisfy (at least) the following requirements: **balance** (the states should be equally distributed among processors, called **spatial balance** (or memory balance), and all processors should be busy almost all the time, called **temporal balance**) and **locality** (whenever possible, a successor state should be processed by the same processor as its parent state).

General approach: The reachability graph is partitioned into disjoint regions which are constructed separately on different processors. This partitioning should be done in such a way that all processors are assigned approximately the same number of states. However, it is rather difficult to determine a mechanism to do such a partitioning without knowing the state space.

The algorithm used by each processor is based on the sequential algorithm, but some additional aspects must be taken into consideration.

The first modification to the sequential algorithm is related to the initial state (there is only one initial state for stochastic nets). Each process is provided with the same initial state, but only the process which is responsible for it adds this initial state to the working list *unexplored*.

Secondly, when a state is generated, it must be decided whether it has already been generated carlier or not (i.e., whether it is a new state or not). This question can be answered if the processor which processes this state is known. This leads to the

idea of defining a partitioning mechanism (partitioning function) which can be used by all processors in order to determine the processor responsible for a state *without additional communication*. In other words, the state space is "split" into regions before the computation.

Whenever a processor generates a new state, it checks, using the partitioning mechanism, whether the state is local or not. If the state is local (which is the desired case, according to the *locality* requirement, so that the communication is minimized) then lines 19-27 in the following algorithm are executed. Otherwise the state is sent to the processor responsible for its further processing.

The third important issue is that the termination condition from the sequential algorithm is not sufficient anymore: the process cannot terminate when its queue of unexplored states is empty, because it can still receive states generated by other processors. Therefore, a global termination detection method is needed. Dijkstra's "circulating probe" [15] is used in [19, 30]. In [23] termination is detected using barrier synchronization. In [28] the authors use the non-committal barrier synchronization algorithm [29].

In general, all processors follow the algorithm given below. It is assumed that upon the detection of the global termination, a "termination message" is sent to all processes engaged in the computation.

```
1. algorithm distributed_state_space_generation;
2. var s0;
3.     states := ∅;
4.     arcs := ∅;
5.     search_set := ∅;
6.     unexplored := ∅;
7.     extern_states := ∅;
8.     this_process_id := process_id(processor);
9. begin
10.    if partition(s0) = this_process_id then
11.        states := states ∪ {s0};
```

```
12.        insert(search_set, s0);
13.        insert(unexplored, s0)
14.    endif;
15.    while not_received_termination_message do
16.        while nonempty(unexplored) do
17.            s := remove(unexplored);
18.            for all s' ∈ successor(s) do
19.                if partition(s') ≠ this_process_id then
20.                    send_state(s', partition(s'));
21.                    send_arc((s, s'), partition(s'))
22.                else
23.                    if s' ∉ search_set then
24.                        states := states ∪ {s'};
25.                        insert(search_set, s');
26.                        insert(unexplored, s')
27.                    endif;
28.                    arcs := arcs ∪ {(s, s')}
29.                endif
30.            endfor
31.        endwhile;
32.        extern_states := receive_states − states;
33.        states := states ∪ extern_states;
34.        arcs = arcs ∪ receive_arcs;
35.        for all s'' ∈ extern_states do
36.            insert(unexplored, s'')
37.        endfor
38.    endwhile
39. end.
```

Lines 32 and 34 from the above algorithm describe the receiving of states and corresponding arcs from other processors, and their integration into the current processor data structures. The new external states (i.e., states which do not belong already to the set *states*) are inserted in the reachability graph. All the arcs are added to the set *arcs*.

Architecture: In the distributed approach, the state space is generated by several processors performing the same algorithm. In [30] several processes following the above algorithm are used. The state space is partitioned into classes. Each processor is assigned a number of classes. The search data structure, *search_set*, consists of several

search trees, one per class.

Additionally, a central processor can be used to provide global coordination. A master-slave architecture is used in [7]. The slave processors are organized in a chain and they actually construct the reachability graph. Each of the slave processors uses a balanced search tree as the search structure. The master processor controls the process of load balancing.

Partitioning mechanism: A partitioning mechanism must ensure a uniform distribution of states. A partitioning mechanism which does not distribute the states evenly among processors can have two negative consequences: (a) if a processor is assigned too many states to process it may run out of memory and the entire computation will stop; (b) uneven distribution of states can result in large differences in execution times, and, therefore, low speedup.

Moreover, the locality and the memory balance of the method depend on the chosen partitioning mechanism:

- locality can be achieved if the partitioning mechanism maps the successor states on the same processor as their parent states;
- balance can be achieved if the partitioning mechanism uniformly distributes the states among processors. This is highly model-dependent and especially difficult because the state space is not known in advance, so the strategy of partitioning can only be based on structural properties of the model.

Hash Function: A first choice for the partitioning mechanism is a hash function [23, 30]. This function typically depends on a (well chosen) set of places of the Petri net, $C = \{p_1, p_2, ..., p_{|C|}\}$, $C \subseteq P$, called *control set.* The general form of the hash function is [23]:

$$partition(m) = (\sum_{i=0}^{|C|} \alpha_i \; m(p_i)) \bmod (n).$$

In the formula above, n is the number of processors, m is a marking function (or the state in this case), and the coefficients α_i are integer (prime) numbers which are determined experimentally for the analyzed net.

The efficiency of the hash function depends on the selection of the control set C and the coefficients α_i.

Finding a good hash function, which ensures a uniform distribution of states among processors, requires some knowledge of the model. This is why there is no general rule on how a hash function should be determined.

Balanced search tree: A method to automatically construct a partitioning mechanism is given in [28]. The state space is partitioned into classes. Each class is assigned to a processor but a processor can have several classes assigned to it. The classes are implemented as balanced search trees. Class 0 (actually, the corresponding balanced tree) is used as a partitioning mechanism. It will reside on all processors.

In order to determine the class of a state, the tree is searched. If the state is found, then it belongs to class 0. Otherwise, the terminal node (and, implicitly, the class) is determined where the state should be inserted. For a given size of the class 0, this tree is "automatically" constructed before the state space generation starts: all processors generate a "random walk" through the state space determining a set of states. Then these sets of states are combined into the balanced search tree. Without going into further details, this requires communication and cooperation. This process is not fully automated because experimental work is still necessary in order to determine the best size of class 0.

The advantage of such a mechanism over a hash function is that it is (partially) automatically constructed, and it does not require (much) model–dependent information. However, the process of constructing this partitioning mechanism is rather complicated.

Remapping of states to processors: Because it is very difficult to define a good partitioning mechanism, memory imbalances can be ameliorated by changing the state assignment to processors. This operation is called **remapping**.

The following "mixed" approach has been found successful [8, 28]: a partitioning function is defined and used during the state space generation. However, from time to time, the memory usage of each processor is checked. If large differences are found, a remapping process is initiated and some states from "overloaded" processors are reassigned to "underloaded" processors.

Remapping can be done for different purposes: for achieving memory balance, or to prevent overloaded processors from the danger of exhausting their memory (**memory-balance-oriented remapping**) or to improve the execution time (**temporal-balance-oriented remapping**).

In [28] remapping is done by reassigning whole classes to different processors. Two remapping strategies are proposed; one is an attempt to balance memory utilization, and the other is an attempt to minimize the execution time. The data transfer is done between processors with high load differences, i.e., overloaded processors send data to underloaded processors.

A different approach is used in [8]. The master process checks the slave workloads from time to time. In case of differences in memory utilization higher than some predetermined value, a load balancing is initiated, but the data are transferred only between neighboring processors (a chain topology is used).

2.4.2 Conclusions

All the distributed approaches to state space generation use a partitioning mechanism which divides the state space before generating it.

The distributed versions of algorithms bring an important benefit over their sequential counterparts: they can handle large nets whose memory requirements are unmanageable for a single computer. Their performance (memory balance, execution time, speedup), however, is influenced by the (static) partitioning mechanism employed. This influence can be reduced if the partitioning mechanism can be modified at run-time, according to the current workload distribution among the processors. For this purpose, some algorithms use dynamic load distributing techniques, improving the performance.

Chapter 3

Distributed state space generation for timed Petri nets

This chapter describes a distributed algorithm for the generation of the state graph of timed Petri nets.

Section 3.1 outlines the method used. Section 3.2 describes the temporal organization of the system. Section 3.3 presents the top-level design of the distributed system (the components' functions and inter-component cooperation). Finally, Section 3.4 describes each component in detail with PVM–pseudo-code algorithms.

3.1 General considerations

Let $\mathcal{T} = (\mathbf{M}, c, f)$ be a timed Petri net, where $\mathbf{M} = (\mathbf{N}, m_0)$ is a marked Petri net. The state graph of $\mathcal{T}$ can be generated using a "divide and conquer" technique as follows: the (yet unknown) state space is partitioned into n disjoint regions $R_1, R_2, ..., R_n$, which are constructed independently, and then integrated in one state graph if needed. The construction of these regions can be distributed to n identical processes running concurrently on different machines. The entire distributed generation has three phases:

1. the *initialization phase* during which the system is set up by creating the cooperating processes and exchanging the information necessary for inter-process

communication,

2. the *computational phase* during which the regions of the state space are constructed,

3. the (optional) *integration phase*, during which all the states and arcs of the regions are collected, and integrated into the complete state graph.

This approach requires the existence of three kinds of logical processes (as shown in Figure 3.1): a process starting the distributed system and initiating the computations, called *Spawner*; several processes constructing the regions graphs, called *Generators*, and a process collecting and integrating the results, called *Collector* [1]. Section 3.3.3 discusses the messages exchanged between these processes.

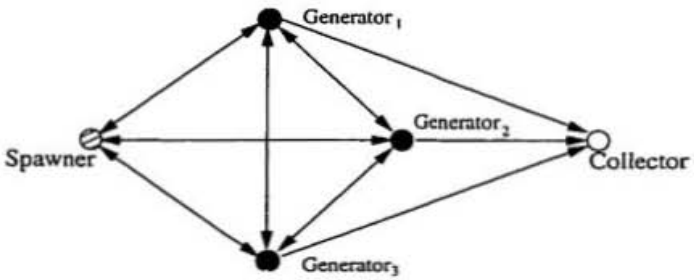


Figure 3.1: Distributed generation system 3 processors.

Physical processes corresponding to these logical processes constitute a "virtual machine." This virtual machine runs on a cluster of computers.

State graph partitioning

The first problem is to determine, for a given net, the disjoint regions in which the (not yet constructed) state graph should be partitioned. The solution is a partitioning

[1] Technically, as a process, the *Collector* can be the same as the *Spawner*, because they exist in disjoint periods of time: the *Spawner* performs the initialization phase, while the *Collector* works in the integration phase. The distinction between them is made for clarity only.

mechanism residing on each processor, which determines, for each state, the region to which it belongs. In this way all processors are aware of the structure of this partition.

This mechanism is a hash function which assigns states to processors according to the distribution of tokens in places and the numbers of firing transitions:

$$partition : \mathcal{R}(\mathcal{T}) \rightarrow \{1, ...n\},$$

$$partition(s) = (\sum_{i=0}^{|P|} \alpha_i \; m(p_i) + \sum_{i=0}^{|T|} \beta_i \; f(t_i)) \bmod (n).$$

where the coefficients α_i and β_i are integer numbers and $s = (m, f, r)$ if $\mathcal{T}$ is a D-timed net (Section 2.3.4), or $s = (m, f)$ if $\mathcal{T}$ is an M-timed net (Section 2.3.3). This function implicitly partitions the graph into n regions $R_1, \ldots, R_n$ such that for each region $R_i = (States_i, Arcs_i)$:

$$States_i = \{s \mid s \in \mathcal{R}(\mathcal{T}) \wedge partition(s) = i\}$$

and

$$Arcs_i = \{(s, s') | s' \in States_i\}.$$

The partitioning function is similar to the one used in [23], with the difference that the components of the state corresponding to the firing transitions are also taken into account for determining the region of the state.

Process $Generator_i$ is *responsible* for the states in region R_i and for the arcs directed to these states. Practically, if a *Generator* is responsible for a state, it determines its successors. A successor state can be in the same region (in this case the connecting arc is an **internal arc**) or in a different region (in which case the connecting arc is called a **cross–arc**).

3.2 System temporal organization

The distributed generation of the state space is composed of the following steps:

1. System startup (Section 3.2.1), which includes:

   (a) setting up the virtual machine (i.e., starting all processes of the distributed system);

   (b) providing all processors with the addresses of the processes they need to interact with.

2. Computational phase (Section 3.2.2), which includes:

   (a) generation of all the states and arcs starting from the initial states; this includes sending states to their appropriate processors;

   (b) transfer of remaining cross-arcs to the processors responsible for them.

3. Result integration (Section 3.2.3).

3.2.1 System startup

During the startup phase, all processes are created, and they exchange the information that is needed for cooperative construction of the state graph.

The program has two input files: one containing the $n + 1$ available hosts and the other containing the Petri net description.

The distributed state space generation starts with the execution of the *Spawner*. The *Spawner* creates the *Collector* and spawns n *Generators* on the other hosts, providing them with the addresses of itself and of the *Collector*, so that they can direct messages to them.

The communication address of each *Generator* must be known to all processes which want to send that *Generator* messages, i.e., the *Spawner* and other *Generators*. For this purpose, each *Generator* sends its communication address back to the *Spawner* as soon

as it is ready. The *Spawner* collects all addresses into the process table, and broadcasts it back to all *Generators*.

3.2.2 Construction of the state subgraphs

The state graph is constructed in a manner similar to the sequential algorithm described in Section 2.2.1, with some differences due to the workload distributed among a set of processors.

A state s created by $Generator_i$ is **local** to it if $Generator_i$ is responsible for s, and it is **non-local** otherwise. A state s is **external** to $Generator_i$ if $Generator_i$ is responsible for s, but s has been created by another *Generator*. A non-local state can be generated many times by *Generators*. Non-local states generated for the first–time are called **first–time non-local states**, and the corresponding cross-arcs directed to them are called **first–time cross-arcs**.

Because a *Generator* is responsible only for the states in one region of the graph, it sends all non-local states with the appropriate cross-arcs to the appropriate *Generators*. Also, each *Generator* must be able to receive the states and arcs sent to it. Therefore, each *Generator* must have primitives to send and receive messages. In order to be able to send messages directly to other *Generators*, the processes must know each other's communication addresses. All necessary addresses are kept in a process table, which is an array of process identifiers.

When an external state does not already exist in the region of the destination processor, it is inserted there and then processed. External cross-arcs are treated differently because the insertion of cross-arcs into their appropriate regions is not critical for the state space generation. This leads to the idea that the sending the cross-arcs to the *Generators* responsible for them can be postponed to the moment when all the states have been generated in all regions reducing the communication during the state generation

phase.

A two-stage algorithm for generating the state graph is used: in the first stage all the states and all the arcs are generated and all non-local states are sent to their corresponding regions. During this stage only the internal arcs, and the first–time cross–arcs are stored in the appropriate regions, all other cross–arcs are stored in the *Generators* which have created them. During the second stage, the remaining cross-arcs are transferred to their corresponding regions.

In this distributed algorithm, some *Generators* may not be provided with start states. Instead, they wait to receive non-local states to be processed. The *Spawner* sends the initial state(s) of the Petri net to the *Generators* responsible for them. The other *Generators* must wait until they receive their first states from the initialized *Generators*.

An important aspect of distributed generation of the state space is the termination condition. The *Generator* cannot simply halt when its working queue is empty, because it may still receive external states from other *Generators* later. A *Generator* may halt only when all other *Generators* have finished as well. Therefore, each *Generator* must know if all other *Generators* have finished the computation of their respective state spaces.

3.2.3 Termination detection

When a *Generator* runs out of states which need to be processed, it waits for states from other processes. In order to prevent a deadlock situation in which all *Generators* are idle and wait for external data, a global termination detection algorithm is interleaved with the computation. This termination algorithm checks if all processors have finished their first stage computations. In the following description, the term termination detection refers to the termination detection of the first stage.

Global termination detection is a classical problem in distributed computing. The algorithm used here is the one proposed by Dijkstra et al. [15]. It assumes that the cluster of processors has a ring topology, $P_1 \rightarrow P_2$, $P_2 \rightarrow P_3$, ..., $P_{n-1} \rightarrow P_n$, $P_n \rightarrow P_1$.

The method is based on the use of a token, which, transmitted over the ring, checks whether all processors have terminated their tasks. It uses two colors, black and white, to represent two states of the distributed system: the white color corresponds to the situation when all processors are found idle; the black color corresponds to the situation where it has been found that some activity existed prior to the moment of checking, and, therefore, it cannot be concluded that the system is idle. A processor remembers that its state is idle or active by making its color white or black. Whenever a processor induces activity in the system by sending a data message, it also sets its color to black.

The process of determining if all processes are idle is started by a designated initiator processor (it can be assumed, without loss of generality, that this process is P_1) by marking itself white and sending a white token to processor P_2. The token message is further propagated over the ring according to the following rules [15]:

1. Upon the receipt of the token, processor P_i holds the token if it is not idle, or it propagates the token to P_{i+1}, if it is idle.

2. When P_i propagates the token, it sends a black token if it is black itself, otherwise, it sends the token that it has received (without changing its color).

3. Upon propagating the token to P_{i+1}, the processor P_i makes its own color white.

4. If P_1 is black when it receives the token back, or it receives a black token, then it initiates a new termination detection process because it is not sure that all processors have completed their work.

If the initiator receives a message containing a white token while it is marked as white, it concludes that all the processors are idle, so it informs the other processors about this by sending a termination notification message.

If the initiator does not receive back the token message, the token must have been lost when it has reached an active processor. In this case the initiator continues its execution and, upon becoming idle, it starts again the process of checking for termination.

3.2.4 Integration of results

When the construction of all regions is completed, each *Generator* sends the states and arcs to the *Collector* and then terminates. The *Collector* combines the received information creating the entire graph, i.e., the union of the regions.

The *Collector* inserts the states in the final graph, and creates arcs between these states. For inserting an arc (as a link from a state to its successor state), the *Collector* determines the two states in the final graph, and then creates the connection.

For optimization reasons each state is given an identifier, so that an arc can be represented by a pair of identifiers. For each state, the *Generator* sends a complete description of the state while for each arc it sends a triple containing the parent state id, the successor state id and the transition probability.

In the state graph, all state identifiers must be unique. Their uniqueness is ensured by the following approach: each *Generator* keeps a counter of states, *cnt*, which is increased each time a state is inserted in the set of states of its region; a state id is determined upon its insertion in the set of states of region R_i using the following formula:

$$id = cnt * K + i$$

where K is the maximum number of *Generators*.

This approach of creating state identifiers makes it impossible for a *Generator* to insert locally the cross-arcs it creates (a *Generator* creating a non-local state s' does not know in advance the id which will be given to s' by the *Generator* responsible for it). Therefore each cross-arc $\langle s, s', prob \rangle$ is sent to the region to which s' belongs.

It should be noted that the ids of non-local states cannot be assigned by the generating processes, because different processes generating the same (non-local) state could assign different ids to it.

3.3 System architecture

This section provides an overview of the distributed program, by discussing the components and their functions, as well as the way in which they interact. In order to keep this description at a system level, component implementation details are omitted here; they are discussed in Section 3.4.

3.3.1 The components

The *Spawner* is responsible for:

- establishing the system's physical configuration,
- spawning other processes,
- determining initial states and sending them to the appropriate *Generators*,
- constructing the "process table",
- broadcasting the "process-table" to all processes which need it.

The *Collector* gathers all the states and arcs sent by *Generators* and combines them into the complete state graph.

Each *Generator* constructs its region of the graph and has the following responsibilities:

- determining the successor states for each state in its working queue,
- creating globally unique ids for the states in its region,
- sending non-local states and cross-arcs to the *Generators* responsible for them,
- receiving external states,
- constructing its region and sending it to the *Collector*,
- cooperating in termination detection.

One *Generator* plays the role of initiator in the process of termination detection.

Decomposition of a Generator

In order to perform some tasks concurrently, each *Generator* is composed of three processes (Figure 3.2): the *Worker*, responsible for the generation of the state space, the *Sender*, responsible for sending messages to other processes, and the *Receiver*, responsible for receiving messages from other processes and for the termination detection. When the *Spawner* creates the *Generators*, it actually creates *Worker* processes. As its first steps, *Worker* creates the *Receiver* and *Sender* processes.

It is the *Sender*'s responsibility to keep track of states sent, and to send, in the first stage, only the first–time cross–arcs.

The *Receiver* responsible for the initiation of the termination detection is called the "initiator" *Receiver*. The others are just "ordinary" *Receivers*. In the description that follows, the terms "Generator" and "triple" are used interchangeably.

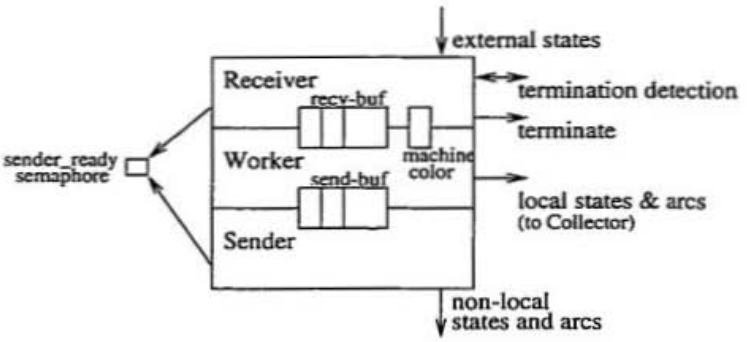


Figure 3.2: The structure of a *Generator*.

Processes $Receiver_i$, $Sender_i$, and $Worker_i$ constitute $Generator_i$. Each $Generator_i$ has a logical identifier i and a task identifier *tid*, which is used as the communication address.

3.3.2 Local communication

The *Worker*, *Receiver*, and *Sender* of each *Generator* reside on the same processor. Their communication is based on shared variables.

Inside the *Generator*, the *Worker* communicates with the *Receiver* using a shared memory segment *recv-buffer*. The *Receiver* adds states into this buffer and the *Worker* retrieves them. This is a standard producer–consumer scheme with the buffer *recv-buffer*. Similarly, the *Worker* and the *Sender* communicate via a shared memory segment *send-buffer* in a producer–consumer fashion where the *Worker* is the producer and the *Sender* is the consumer.

The mutual exclusion for accesses to the buffer inside the typical producer–consumer scheme is based on three semaphores, em not_empty, *not_full* and *mutex*; the operation *put(state, buffer)* stores the state *state* in the buffer *buffer* and the function *get(buffer)* retrieves a state from the buffer *buffer*, and returns it.

The *Sender* and the *Receiver* share the semaphore *sender_ready*. In the initialization

phase, the *Receiver* waits at this semaphore until the *Sender* signals it. This ensures that the *Receiver* indicates to the *Spawner* to broadcast the process table only when the *Sender* is ready to receive it.

Also the machine color (see Section 3.2.3) needs to be accessed by both *Worker* and *Receiver*. Therefore, it is stored in a memory segment shared by the *Worker* and the *Receiver*.

3.3.3 Message based communication

System components residing on different hosts need to communicate during the execution of the program. They communicate by message passing using the PVM (Parallel Virtual Machine) package [18]. PVM is an integrated set of software tools that emulate a concurrent computing environment using a collection of interconnected computers. The PVM system is composed of two parts: the first part is a demon residing on each of the cooperating computers; the second part is a library of routines for typical operations needed for parallel computing (message passing, spawning processes, coordinating tasks, etc.).

In the startup phase, the *Spawner* and *Generators* must communicate to determine the addresses of all processes they need to cooperate with. Each *Generator* provides its "communication address" by sending to the *Spawner* a message containing the triple's id and the task id (tid) of its *Receiver*. The *Spawner* collects all these ids into the process table and broadcasts it back to all *Generators* (actually to their *Senders* and their *Receivers*). For this purpose, the *Senders* and the *Receivers* join a PVM process group, called *generators*. The *Senders* need to know where to send non-local states to, while the *Receivers* need to know their successors in the processor–ring used for termination detection. To ensure that all *Senders* are ready to receive, each *Receiver* only sends its address after its *Sender* is ready. Two kinds of messages are used:

EXIST(i, tid) - message containing the triple's id i and the task id *tid* of the $Receiver_i$; it is sent by each *Receiver* to the *Spawner*;

INFO$(recv\_tids)$ - message containing an array with the ids of all n *Receivers*; sent by *Spawner* to all *Senders* and *Receivers*.

In the computation phase, the *Generators* exchange non-local states and cross-arcs by using DATA messages. A DATA message contains one or more items, each item containing state and arcs directed to that state. The arc information contains the parent state id, and the probability associated with the state transition.

DATA$(\langle s_j, \langle parent_{j,i}, prob_{j,i} \rangle_{i=1..n_j} \rangle_{j=1,...,k})$ - message containing k items, each of them comprising a state s_j and information about n_j incoming arcs; the message is sent to the *Generator* responsible for the states and arcs.

During the computation phase, the *Generators* must also communicate to determine when the first stage is globally finished. First token messages are sent between *Receivers* for the termination detection as described in Section 3.2.3. If the detection is successful, the initiator *Receiver* sends a special message indicating this situation to all other *Receivers*. The messages used are:

CHECK_TERM$(token)$ - message propagated to check the global termination; this message is transmitted ring-wise between *Receivers*;

TERMINATE - message sent by the initiator *Receiver* to inform all processes that the end of the first stage has been globally reached; upon reception of the TERMINATE message, the *Receiver* interrupts its *Sender*, and puts a special item into the *Worker*'s state buffer, which makes the latter finish the first stage.

The *Generators* need also to communicate to start and to determine the termination of the second stage. The *Sender* starts the second stage immediately after being interrupted by the *Receiver*, by sending the remaining cross-arcs packed in DATA messages. After this, each *Sender* broadcasts a message **DONE_ARCS_EXPORT**, indicating that its triple will not send any more data to other *Generators*.

During the result collection phase, each *Generator* sends three kinds of messages to the *Collector*:

STATE$(\langle s_i \rangle_{i=1..k})$ - message containing k items, each item comprising a state description and a state id;

ARC$(\langle successor\_state\_id_i, state\_id_i, prob_i \rangle_{i=1..k})$ - message containing k items, each item contains arc information: the identifiers of two states, and the probability of transition between states;

FINISHED - message indicating that the *Generator* has finished sending the results; upon receiving such messages from all *Generators*, the *Collector* knows that it will not receive any subsequent data.

Figure 3.3 shows a summary for inter-component communication for a system with three *Generators*, the initiator being $Generator_1$.

The *Spawner* receives EXIST messages and sends out INFO messages and DATA messages.

Each *Generator* receives INFO messages from *Spawner*, DATA messages from the *Spawner* and from all other *Generators*, and CHECK_TERM messages from the previous neighbor in the ring; all *Generators* except of $Generator_1$ receive TERMINATE messages from $Generator_1$. Each *Generator* sends EXIST messages to the *Spawner*,

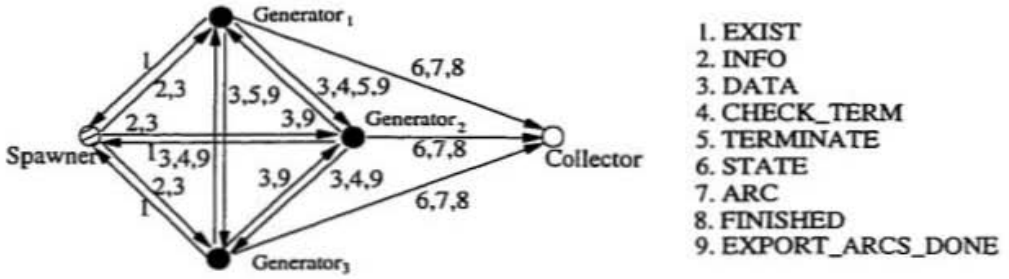


Figure 3.3: Inter-components communication summary.

DATA messages to all other *Generators*, CHECK_TERM messages to its next neighbor in the ring, and STATE, ARC, and FINISHED messages to the *Collector*.

The *Collector* receives STATE, ARC, and FINISHED messages from the *Generators*.

The messages are distinguished by marking them with distinct tags.

The communication between processes is done using standard message passing primitives [18].

3.4 Algorithms

This section presents the algorithms for the six kinds of processes (*Spawner*, *Collector*, *Worker*, *Sender*, initiator *Receiver*, and ordinary *Receiver*) in an untyped pseudo-code.

Some of the algorithms use the generic semaphore routines *wait*(*semaphore*) and *signal*(*semaphore*).

3.4.1 The Spawner

The *Spawner* performs initialization tasks:

```
1. algorithm Spawner:
2. var hostfile;              (* file containing the virtual machine configuration*)
3.     tpnet_file;            (* file containing the description of the Petri net *)
4.     net;                   (* the Petri net *)
5.     hosts[];               (* array with the hosts in the virtual machine *)
6.     n;                     (* number of processors in the virtual machine *)
7.     init_states_list;      (* a list containing the initial states of the Petri net *)
8.     id;                    (* identifier *)
```

```
9.    initiator_id;          (* identifier of the initiator *)
10.   tid;                   (* PVM task identifier *)
11.   tid_collector;         (* PVM task identifier of the Collector *)
12.   generators;            (* group of all Senders and Receivers*)
13. begin
14.   (n, hosts) := read(hostfile);
15.   tid_collector := spawn Collector(n) on host_name[0];
16.   net := read(tpnet_file);
17.   init_states_list := get_initial_states(net);
18.   initiator_id := partition(head(init_states_list));
19.   for i := 1 to n do
20.       spawn Worker(initiator_id, tid_collector, my_tid, net, i, n) on hosts[i]
21.   endfor;
22.   for i := 1 to n do
23.       (id, tid) := receive(EXIST);
24.       process_table[id] := tid
25.   endfor;
26.   broadcast(generators, INFO(process_table));
27.   while not_empty(init_states_list) do
28.       state := head(init_states_list);
29.       init_states_list := tail(init_states_list);
30.       send(process_table[partition(state)],DATA(state, 0, state.prob))
31.   endwhile
32. end.
```

First the *Spawner* reads (line 14) the configuration of the virtual machine from the file *hostfile*: n is the number of *Generators*, and *hosts* is an array with $n + 1$ names of the hosts on which to create the other processes (n *Generators* and the *Collector*).

Next, the *Spawner* creates the *Collector* on the first host, by calling the function *spawn* (line 15). This function has as parameters the host on which to spawn the process, the process to be spawned, and its program arguments. The *Collector* receives only one argument, the number of *Generators*, n. The *Collector* is spawned before the *Generators* because, in this way, the *Generators* can be given the *Collector*'s address at spawning time. If the *Collector* is not created as a separate process, the *spawn* function is not needed, and the *Collector* function is called after the termination of the *Spawner*. In this case $tid\_collector = tid\_spawner$.

The *Spawner* then reads the net description from the file *tpnet_file* (line 16), and determines the net's initial states by invoking the function *get_initial_states* (line 17). Each of these states has a probability associated with it. The termination detection initiator is then determined as the *Receiver* process of the *Generator* responsible for the first initial state (line 18). It is important that the initiator is not a processor which is idle all the time, because, otherwise, it would burden the computation with unnecessary termination checking messages.

The n *Workers* are spawned by calling the function *spawn* (line 20). As arguments each *Worker* receives the initiator's logical identifier, (*initiator_id*), the *Spawner*'s address, the *Collector*'s address (*tid_collector*), the net description (*net*), and the total number of *Generators* (n). The tid of the *Spawner* is determined by calling the function *my_tid*, which returns the tid of the calling process.

After spawning the *Workers*, the *Spawner* waits for n $EXIST$ messages from the *Generators* (lines 22-25). In general, the syntax used for a message receive function is *message* := *receive*. If a processor wants to receive only a specific type of messages, then *receive* has one of the message types (described in the previous subsection) as an argument; in line 23, an EXIST message is expected.

Each EXIST message contains the *Generator*'s *Receiver* logical id and the task id. Upon receipt of such message, the task id contained in the message is added to the process-table. When the process table is complete, it is broadcast back to all *Generators* (line 26).

When the entire virtual machine is set up, the *Spawner* initiates the process of state space generation by determining and sending the initial states of the net to their corresponding *Generators* (lines 27-31). As in [38], a dummy state, with id 0, is used as a common parent of all initial states. Therefore, each initial state is packed into

a DATA message containing the state representation, it's parent id (i.e., 0) and the corresponding transition probability.

3.4.2 The Worker

Process $Worker_i$ constructs the region $R_i = (arcs_i, states_i)$ of the state graph. The set of generated states, $states_i$, is implemented using the C++ standard class *set* [36], which guarantees logarithmic search time. Each element of the set is a tuple $\langle state, id \rangle$ containing a state description and a state id. The set of arcs is a linked list of triples $\langle id_1, id_2, prob \rangle$, where id_1 and id_2 are state identifiers and $prob$ is the state transition probability.

The outline of each Worker is as follows:

```
1.  algorithm Worker_i (initiator_id, tid_collector, tid_spawner, net, n);
2.  var states_i := ∅; arcs_i := ∅;              (* states and arcs of region R_i*)
3.      machine_color shared with Receiver_i;   (* the state of the generator *)
4.      send_buffer shared with Sender_i;       (* buffer with incoming states *)
5.      recv_buffer shared with Receiver_i;     (* buffer with outgoing states *)
6.      state;                                  (* state *)
7.      next_state;                             (* state *)
8.      id;                                     (* identifier *)
9.      parent_id;                              (* identifier *)
10.     parent_i;                               (* identifier *)
11.     prob;                                   (* arc probability *)
12.     cont;                                   (* loop continuation flag *)
13. begin
14.     if i = initiator_id then
15.         spawn InitiatorReceiver_i(tid_collector, tid_spawner, n) on this_host
16.     else
17.         spawn OrdinaryReceiver_i(tid_collector, tid_spawner, n) on this_host
18.     endif;
19.     spawn Sender(n) on this_host;
20.     execute main_loop;                      (* main loop is shown on the next page *)
21.     cont := true;
22.     while cont do
23.         ⟨state, ⟨parent_i, prob_i⟩_{i=1..k}⟩ := get(recv_buffer);
24.         if state = null then
25.             cont := false
26.         else
```

```
27.            id := states_i.find(state);
28.            arcs := arcs ∪ {(parent_i, id, prob_i)_{i=1..k}}
29.        endif
30.    endwhile;
31.    send(tid_collector, states_i);
32.    send(tid_collector, arcs_i);
33.    send(tid_collector, FINISHED)
34. end.
```

First, the $Worker_i$ creates its $Receiver_i$ (InitiatorReceiver or OrdinaryReceiver) and its $Sender_i$ (lines 15, 17, 19), on the same host as itself (*this_host*). Then it enters the main loop in which all reachable states and their descendants are determined. This loop is described below.

When the main loop is finished, all *Workers* have generated all the states of their regions, but if a *Generator* has created more arcs leading to non local states, only the first arc has been sent to the *Generator* responsible for it. Therefore, the *Worker* loops again (lines 22 to 30) to collect all the remaining arcs leading to its region, which are stored in *recv_buffer* by its *Receiver*. This loop continues until a null item is retrieved from the *recv_buffer* (lines 24 and 25). When a non-null item $(state, \langle parent_i, prob_i \rangle_{i=1..k})$ is retrieved, the id of the state *state* is looked up in the region (the state already exists there, because it has been inserted during the main loop), and the arcs are inserted in the list of arcs.

After completing its region, the *Worker* sends the states and arcs to the *Collector*. States and arcs are sent clustered in two messages, one for states, and the other for arcs. A final message FINISHED notifies the *Collector* that the entire region has been sent.

The *Worker* deals with states generated locally and states received from other processes. It maintains two queues, *w_que*, the working queue for local states, and *recv_buffer*, the queue for external states.

The main loop is as follows:

```
1. main_loop: cont := true;
2.     while cont do
3.         if send_buffer.empty ∧ w_que.not_empty then
4.             ⟨state, id⟩ := w_que.remove
5.         else
6.             ⟨state, parent_id, prob⟩ := get(recv_buffer);
7.             if state = null then
8.                 cont := false
9.             else
10.                id := states_i.find(state);
11.                if id ≠ noID then
12.                    arcs := arcs ∪ {⟨parent_id, id, prob⟩}
13.                else
14.                    id = states_i.create_local_id;
15.                    states_i := states_i ∪ ⟨state, id⟩;
16.                    arcs := arcs ∪ {⟨parent_id, id, prob⟩}
17.                endif
18.            endif
19.        endif;
20.        if cont ∧ id ≠ noID then
21.            for all next_state in successors(state) do
22.                if partition(next_state) = i then
23.                    id := states_i.find(next_state);
24.                    if id ≠ noID then
25.                        arcs := arcs ∪ {⟨state.id, id, next_state.prob⟩}
26.                    else
27.                        id = create_local_id;
28.                        states_i := states_i ∪ {⟨next_state, id⟩};
29.                        arcs := arcs ∪ {⟨state.id, next_state.id, next_state.prob⟩};
30.                        w_que.insert(⟨next_state, id⟩)
31.                    endif
32.                else
33.                    processor_color := Black;
34.                    put(⟨next_state, parent_id, next_state.prob⟩, send-buffer)
35.                endif
36.            endfor
37.        endif
38.    endwhile;
```

If *recv_buffer* is empty but the working queue is not empty, then the state to process is taken from the working queue (line 4); otherwise, the function *get* is called to get an item from *recv_buffer* (line 6), which will involve waiting if the buffer is empty. This loop

is discontinued when a null item is retrieved from the buffer, i.e., when the function *get* returns a null state (line 8); the *Receiver* puts this null item in the buffer after receiving the message that the first stage's global termination has been detected.

If function *get* returns a non-null item from the buffer (line 6), a state *state*, the id of its parent on the remote processor, *parent_id*, and the probability of the transition between the two states are retrieved from this item. The *Worker* checks whether the state already exists in this region by calling the method *find* (line 10). If this state exists, *find* returns the id of the state, and the state is not processed any further, only the arc $\langle parent\_id, id, prob \rangle$ is added to the set of arcs (line 12). If the function returns $noID$, the state is new to the *Worker*, and it must be inserted into the graph with a new, unique id (line 14). The corresponding arc from the parent state is inserted in the set of arcs (line 16).

For each new *state*, the *Worker* creates the successor states (line 21) with their probabilities, and analyzes them as follows: first, it determines if a successor state is local or non-local (line 22) using the function *partition*. Each successor state in the region i, (i.e., local successor state) is looked up in the set of already generated states $states_i$ (line 23). If it is found, only the arc from the current state to the child state is inserted into the set of arcs (line 25); otherwise the state is inserted into $states_i$ (line 28) and in *w_que* (line 30) for further processing, and the arc is inserted into $arcs_i$ (line 29).

Non-local successor states are deposited, together with their arcs, into *send_buffer* by calling the function *put* (line 34). From there they will be subsequently extracted and sent away by $Sender_i$. The *Worker* also changes the color of the *Generator* to Black (line 33) as the processing of states is not finished yet.

3.4.3 The Sender

The *Sender* is mainly responsible for sending the non-local states and cross-arcs to their corresponding *Generators*.

```
1. algorithm Sender_i(n);
2. var state;                                   (* state of the Petri net *)
3.     sent_states;                             (* states already sent *)
4.     sender_ready semaphore shared with Receiver_i;
5.     send_buffer shared with Worker_i;
6.     process_table;                           (* communication addresses *)
7.     cont := true;                            (* loop continuation flag *)
8. begin
9.     join_group(generators);
10.    signal(sender_ready);
11.    process_table := receive(INFO);
12.    while cont do
13.        ⟨state, parent_id, prob⟩ := get(send_buffer);
14.        if state = null then
15.            send_rest(process_table, sent_states, cluster_size);
16.            cont := false
17.        else
18.            if state ∉ sent_states then
19.                send(process_table[partition(state)], DATA(state, parent_id, prob));
20.                sent_states.insert(state)
21.            else
22.                state.add_arc_link(parent_id, prob)
23.            endif
24.        endif
25.    endwhile;
26.    bcast(generators, ARCS_EXPORT_DONE)
27.end.
```

First, the $Sender_i$ joins the group of *Generators* (line 9) in order to receive the table of process identifiers from the *Spawner*. The $Sender_i$ signals to $Receiver_i$ that it is ready to receive a message containing the process table from the *Spawner* (line 10). It then receives this table and stores it in the *process_table* (line 11).

The *Sender* invokes the routine *get* (line 13) to obtain the data which it must send. In the case of non-null state (lines 18 to 23), *get* returns an item put by the *Worker* into this buffer, which comprises: a state (*state*), its parent id (*parent_id*), and the

probability *prob* of transition between the parent state and s. The item is sent to its destination only if the state s has not been sent before (line 19).

The *Sender* uses a structure *sent_states* to keep track of the states sent. This structure is organized as a search tree, using the state representation as the key. Each node is linked with a list of arcs. Whenever an item is retrieved from *send_buffer*, the state *state* is searched in *sent_states* (line 18). If *state* is found, then a new arc $\langle parent\_id, prob \rangle$ is added to its list of arcs (line 22), otherwise the state is inserted into the tree and then sent together with its arc to the appropriate *Generator* (lines 19 and 20).

Sender's loop is terminated when the function *get* returns *null*, as a consequence the global termination of the first stage. The *Sender* then sends away the remaining arcs, together with their states (states are needed to determine their ids at the destination), by calling the function *send_rest* (line 15). All arcs directed to the same state are already clustered together (from the way the tree and the arcs list are constructed).

After sending this data, the *Sender* broadcasts a message ARCS_EXPORT_DONE and terminates.

3.4.4 The ordinary Receiver

All *Receivers* except the one used for initiating termination detection implement the following algorithm.

```
1. algorithm OrdinaryReceiver (tid_collector, tid_spawner, n);
2. var message;
3.     token;
4.     finished := false;
5.     processor_color shared with Worker_i;
6.     sender_ready semaphore shared with Sender_i;
7.     recv_buffer shared with Worker_i;
8.     process_table[];
9.     k := 0;
10. begin
```

```
11.   wait(sender_ready);
12.   join_group(generators);
13.   send(tid_spawner, EXIST(i, my_tid));
14.   process_table := receive(INFO);
15.   while k < n − 1 do
16.         message := receive;
17.         case message.type of
18.             DATA:
19.                 put(recv_buffer, message.S);
20.             CHECK_TERM:
21.                 if Worker_is_idle ∧ Sender_is_idle then
22.                     if processor_color = Black then
23.                         token := Black
24.                     endif;
25.                     send(Receiver_{i⊕1}, CHECK_TERM(message.token));
26.                     processor_color := White
27.                 endif;
28.             TERMINATE:
29.                 begin
30.                     interrupt_Sender;
31.                     put(recv_buffer, null)
32.                 end;
33.             DONE_EXPORT_ARCS:
34.                 begin
35.                     k := k + 1;
36.                     if k = n − 1 then put(recv_buffer, null) endif
37.                 end
38.         endcase
39.   endwhile
40. end.
```

The initialization part is common for all *Receivers*: before sending an $EXIST$ message to the *Spawner*, the *Receiver* waits for its *Sender* to be ready (the *Sender* should be able to receive messages when the process table is sent to it by the *Spawner*, otherwise the message with the process table is lost). For this purpose, the *Receiver* and the *Sender* share a semaphore *sender_ready*, and the *Receiver* waits at this semaphore until it receives a signal from the *Sender*, and then it sends to the *Spawner* an EXIST message containing the triple's id and the *Receiver*'s address (line 13). After this the *Receiver* receives the process table, storing it in the *process_table*.

The *Receiver* reacts to different messages as follows.

For *DATA* messages, with the contents $\langle s_j, \langle parent_{j,i}, prob_{j,i} \rangle_{i=1,\ldots,n_j} \rangle_{j=1,\ldots,k}$, the enclosed k state descriptions are extracted from the message and stored in *recv_buffer*.

In the case of a CHECK_TERM message, the *Receiver* checks whether its triple is idle by calling the routines *Worker_is_idle* and *Sender_is_idle*. The *Worker* is idle when it is waiting with empty *recv_buffer*, i.e., it is suspended on the semaphore *not-empty*. Similarly, the *Sender* is idle when it is also suspended on a semaphore *not-empty*. If the triple is idle, the $Receiver_i$ propagates the token (line 25), ensuring to change the token's color in the case when processor's color is Black (line 23). $Receiver_{i \oplus 1}$ is the next *Receiver* in the ring, where $i \oplus 1 = i + 1$ if $i < n$, and $i \oplus 1 = 0$ if $i = n$.

For a TERMINATE message, the *Receiver* must make the *Worker* end its first stage and the *Sender* send the remaining arcs. Therefore, the *Receiver* adds a special null item to *recv_buffer* (line 31); upon retrieving this item, the *Worker* finishes the first stage. The *Receiver* also discontinues its *Sender*'s loop (line 30). Finally, the *Worker* receives all the remaining cross–arcs, which come as DATA messages.

All DONE_EXPORT_ARCS messages are counted (line 35). After $n - 1$ such messages, the *Receiver* knows that it will not receive any subsequent data (PVM ensures that the order of sent messages is preserved at the receiver side) and puts a null item into *recv_buffer* (line 36). When the *Worker* retrieves this message, it terminates its second stage.

3.4.5 The initiator Receiver

The initiator *Receiver* has a slightly different algorithm then the others, because it has the special responsibility of starting the process of termination detection.

```
1. algorithm InitiatorReceiver (tid_collector, tid_spawner, n);
2. var message;
3.     token;
```

```
4.    processor_color shared with Worker_i;
5.    sender_ready semaphore shared with Sender_i;
6.    recv_buffer shared with Worker_i;
7.    process_table[];
8.    k := 0;
9. begin
10.     wait(sender_ready);
11.     join_group(generators);
12.     process_table := receive(INFO);
13.     while k < n − 1 do
14.           message := timeout_receive;
15.           if message = null then
16.               if Worker_is_idle ∧ Sender_is_idle then
17.                   token := White;
18.                   send(Receiver_{i⊕1}, CHECK_TERM(token));
19.                   processor_color := White
20.               endif
21.           else
22.               case message of
23.                   DATA:
24.                       put(recv_buffer, message.S);
25.                   CHECK_TERM:
26.                       if message.token = White ∧ processor_color = White then
27.                           interrupt_sender;
28.                           put(recv_buffer, null);
29.                           for j = 1 to n do
30.                               if j ≠ initiator_id then
31.                                       send(process_table[j], TERMINATE)
32.                               endif
33.                           endfor
34.                       endif;
35.                   DONE_EXPORT_ARCS:
36.                       begin
36.                           k := k + 1;
37.                           if k = n − 1 then put(recv_buffer, null) endif
38.                       end
39.               endcase
40.           endif
41.     endwhile
42. end.
```

The initialization part (lines 10-12) is the same as for other *Receivers*.

However, the initiator *Receiver* has a slightly different loop. First, it performs a receive with timeout (line 14). If, within a timeout period of time, no message has

been received, the *Receiver* checks whether the entire triple is idle, i.e., whether the *Worker* and the *Sender* are also idle (line 16). If this is the case, all other *Generators* can also be idle. So, the *Receiver* initiates the termination detection process by sending a CHECK_TERM message to the next *Receiver* in the ring (line 18).

On the other hand, if a message is received, an action is performed according to the message type. DATA messages are treated in the same way as by the ordinary *Receivers*. A CHECK_TERM message indicates that a token message sent before is back. Upon its receipt, the *Receiver* checks the processor color and the color of the token in the message. If both colors are White (line 26), all *Generators* have finished computations, the *Receiver* notifies the *Worker* and the *Sender* about this (lines 27 and 28) and sends a TERMINATE message to all other *Receivers* (lines 29 to 33).

3.4.6 The Collector

The *Collector* gathers all the results constructing the complete state graph.

```
1. algorithm Collector(n);
2. var graph := ∅;
3.     arcs := ∅;
4.     finished := 0;
5. begin
6.     while finished ≠ n do
7.         message := receive;
8.         case message of
9.             ARC(S):
10.                arcs_set.insert(S);
11.            STATES(S):
12.                begin
13.                    graph.insert(S);
14.                    arcs_set.insert_arc(state.id, pid)
15.                end;
16.            FINISHED:
17.                finished := finished + 1
18.        endcase
19.    endwhile;
20.    while nonempty(arcs_set) do
21.        arc := arcs_set.remove_arc;
```

22. *graph.insert_arc*
23. **endwhile**
24. **end.**

The *Collector* loops until it receives n *FINISHED* messages, treating the other incoming messages as follows: if an ARC message is received, then the arcs from the message are inserted in the set *arcs* for later use; the arcs are not inserted into the *graph* right when they are received, because it may happen that an arc arrives earlier than its corresponding states. When a STATE message is received, all its states (set S) are inserted into the graph.

Only at the end of the algorithm are all received arcs inserted into the graph (lines 20 to 23).

Chapter 4

Examples

This chapter presents some experimental results obtained for input nets with a complex structure and a fairly large state space size. Experiments are conducted for D–timed and well as M–timed nets because it is anticipated that the performance results for these two classes of nets can be quite different; usually D–timed nets generate less states than comparable M–timed nets, but D–timed nets are more computationally–demanding for state processing that M–timed nets. Consequently, the computation–to–communication ratios for these two classes of nets are quite different.

The performance measure used in the experiments is the speedup of the program. The speedup S of a distributed program is a function $S : \mathbb{N} \mapsto \mathbb{R}^+$ defined as the ratio of the program's execution time on one processor, $T(1)$, to the program's execution time on n processors, $T(n)$:

$$S(n) = \frac{T(1)}{T(n)}.$$

The program's speedup is influenced by the partitioning function. The locality of a partitioning function is defined as follows: the locality of each processor is the ratio between its number of local states and its region size. The average locality of the distributed program is defined as the arithmetic average of the localities of all the processors involved.

All experiments have been performed on a cluster of 32 diskless Linux PCs, each with 128MB RAM, connected via a 100 Mbps Ethernet.

Section 4.1 shows the program's behavior for D-timed nets, and Section 4.2 presents some results for M-timed nets. Section 4.3 summarizes the results obtained in these experiments.

4.1 D-timed nets

4.1.1 Example 1

The first example is taken from [12]. The net models a parallel MIMD architecture.

The description of Petri nets follows the syntax used in TPN-tools [38]. Nets are described as collections of transitions, and each transition contains all parameters associated with it (firing rate or time, choice probability, input and output places). Two different initial markings are used to control the size of the state space.

The description of the net is as follows:

```
Dnet(#T1=P2,P1/P3;
     #T2=P17,P4/P5;
     #T3=P28,P4/P6;
     #T4*1.0=P4,P3/P7,P4,P2;
     #T5*1.0=P5/P10,P4;
     #T6*1.0=P6/P21,P4;
     #T7,0.5=P7/P9;
     #T8,0.5=P7/P8;
     #T9=P11,P10/P12;
     #T10=P13,P9/P14;
     #T11=P37,P13/P15;
     #T12*1.0=P13,P12/P16,P13,P11;
     #T13*1.0=P14/P13,P1;
     #T14*1.0=P15/P30,P13;
     #T15,0.5=P16/P17;
     #T16,0.5=P16/P18;
     #T21=P22,P21/P23;
     #T22=P38,P24/P25;
     #T23=P24,P8/P26;
     #T24*1.0=P24,P23/P27,P24,P22;
     #T25*1.0=P25/P30,P24;
     #T26*1.0=P26/P24,P1;
     #T27,0.5=P27/P28;
     #T28,0.5=P27/P29;
     #T29=P31,P30/P32;
```

```
      #T30=P33,P18/P34;
      #T31=P33,P29/P35;
      #T32*1.0=P33,P32/P36,P33,P31;
      #T33*1.0=P34/P33,P10;
      #T34*1.0=P35/P33,P21;
      #T35,0.5=P36/P38;
      #T36,0.5=P36/P37)

mark(P1:4,P2,P4,P10:2,P11,P13,P21:1,P22,P24,P30:1,P31,P33);
mark(P1:4,P2,P4,P10:4,P11,P13,P21:1,P22,P24,P30:1,P31,P33);
```

For the first marking, the state graph has 14487 states and 26675 arcs (Example 1 (a)), while for the second marking it has roughly three times as many states (46729) and 92253 arcs (Example 1 (b)).

Figure 4.1 shows the execution times for the generation of the state graphs for these two initial markings. The plots show that the distributed algorithm's execution time decreases when the number of processors increases.

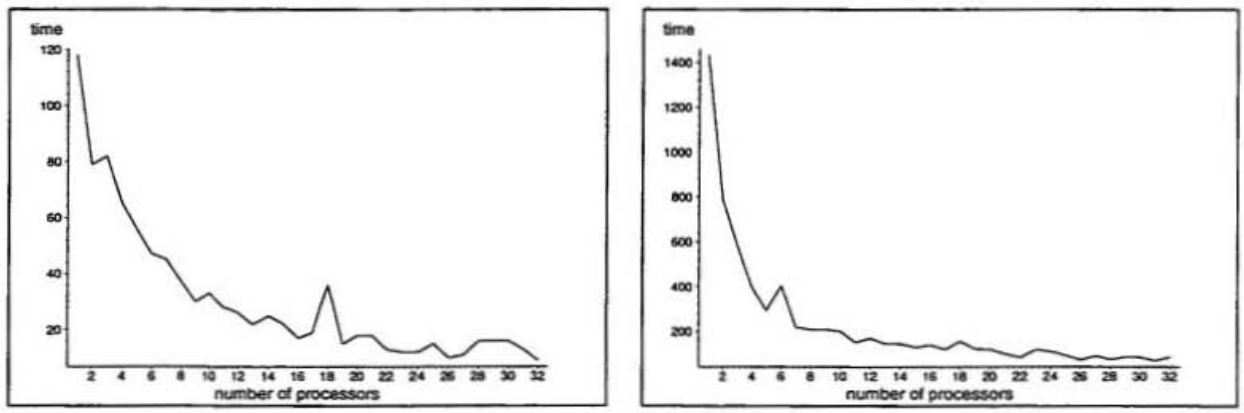


Figure 4.1: Execution time for Example 1 (a) and Example 1 (b).

An irregularity can be observed in Figure 4.1(a) for 18 processors; the execution time grows from 18 seconds for 17 processors to 36 seconds for 18 processors, and then decreases back to 15 seconds for 19 processors.

The growth of the execution time from 17 to 18 processors in Example 1(a) is due to a lower average locality in the later case (5 % compared to 6 %) which results in a large

difference in the (total) waiting time (27 seconds compared to 10 seconds). This waiting time "destroys" the general advantage of using more processors (on average there are less states to process, and less external messages per processor). For 19 processors the locality is the same as for 18, i.e. 5 %; because the regions are smaller for 19 processors, the total execution time decreases from 15 seconds to 8 seconds.

Figure 4.2 shows the speedup for the two cases, for the number of processors from 1 to 32.

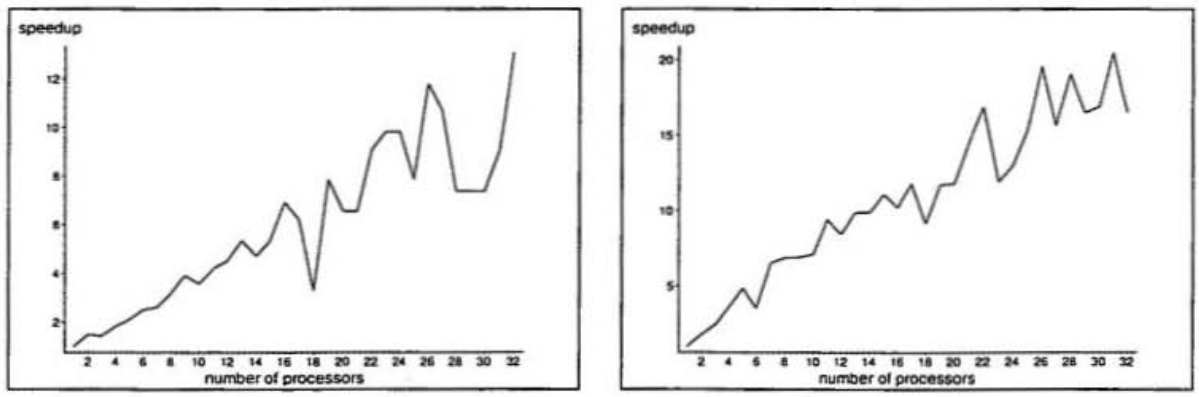


Figure 4.2: Speedup for Example 1 (a) and Example 1 (b).

In order to "smooth" the speedup curves shown in Figure 4.2, an approximation of the speedup by a best fitting polynomial of degree 3 is shown in Figure 4.3.

It can be observed that the character of the speedup curves for the two cases is very similar, however, the speedup is better for the larger state graphs. Figure 4.4 compares the two cases with the ideal speedup $S(n) = n$; line (i) represents the ideal speedup, curve (a) corresponds to the speedup for Example 1(a), and curve (b) corresponds to results of Example 1(b).

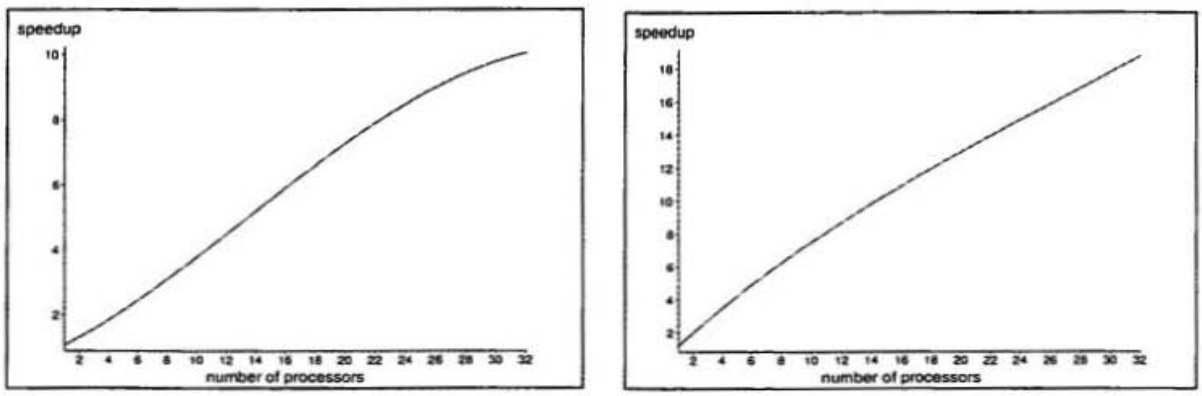


Figure 4.3: Speedup curves for Example 1 (a) and Example 1 (b).

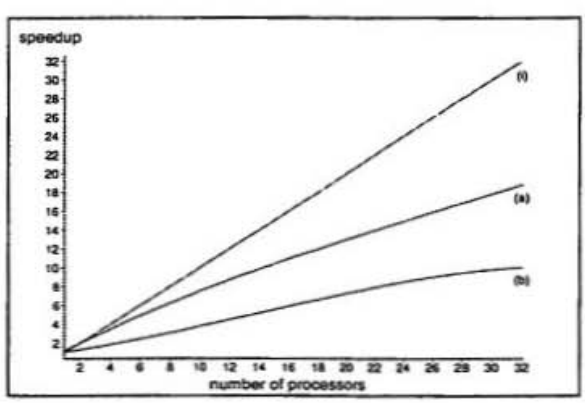


Figure 4.4: Speedup comparison for Example 1 (a) and Example 1 (b).

4.1.2 Example 2

Figure 4.5 shows the execution time for the problem of the 12 dining philosophers. The net has 705 states and 901 arcs. The relatively small number of states is due to the fact that all eating times and all thinking times are equal. If these times are different for different philosophers, the state space grows very quickly. The net description and the initial marking are as follows:

```
Dnet(#think1*5=p1a/p1b;
     #think2*5=p2a/p2b;
     #think3*5=p3a/p3b;
     #think4*5=p4a/p4b;
     #think5*5=p5a/p5b;
     #think6*5=p6a/p6b;
     #think7*5=p7a/p7b;
     #think8*5=p8a/p8b;
     #think9*5=p9a/p9b;
     #think10*5=p10a/p10b;
     #think11*5=p11a/p11b;
     #think12*5=p12a/p12b;
     #eat1*2=p1b,A,B/A,B,p1a;
     #eat2*2=p2b,B,C/B,C,p2a;
     #eat3*2=p3b,C,D/C,D,p3a;
     #eat4*2=p4b,D,E/D,E,p4a;
     #eat5*2=p5b,E,F/E,F,p5a;
     #eat6*2=p6b,F,G/F,G,p6a;
     #eat7*2=p7b,G,H/G,H,p7a;
     #eat8*2=p8b,H,I/H,I,p8a;
     #eat9*2=p9b,I,J/I,J,p9a;
     #eat10*2=p10b,J,K/J,K,p10a;
     #eat11*2=p11b,K,L/K,L,p11a;
     #eat12*2=p12b,L,A/L,A,p12a)

mark(A,B,C,D,E,F,G,H,I,J,K,L,p1b,p2b,p3b,p4b,p5b,p6b,p7b,p8b,p9b,p10b,
     p11b,p12b);
```

Similarly to the previous example, the execution time is improving very quickly at the beginning, from 1 to 4 machines. It continues decreasing up to 19 machines, where a "saturation" is reached. For 19 processors the average region size is only 39 states in this case, so the advantages of distributed computing are becoming less significant because the utilization of processors is decreasing; processors are spending an increasing proportion of time waiting for data from other processors; further increase of the number of processors actually increases the execution time.

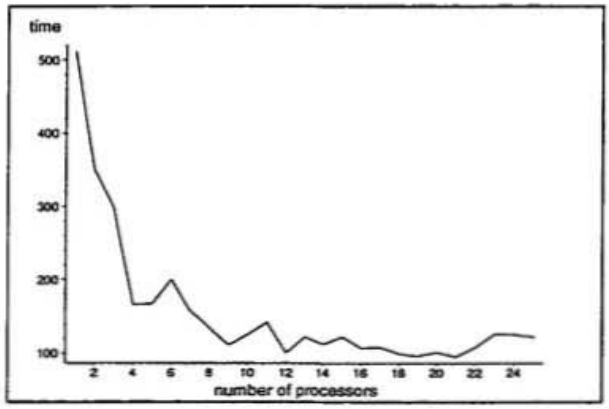


Figure 4.5: Execution time for Example 2.

Figure 4.6 shows the speedup and the speedup fitting curve for this example.

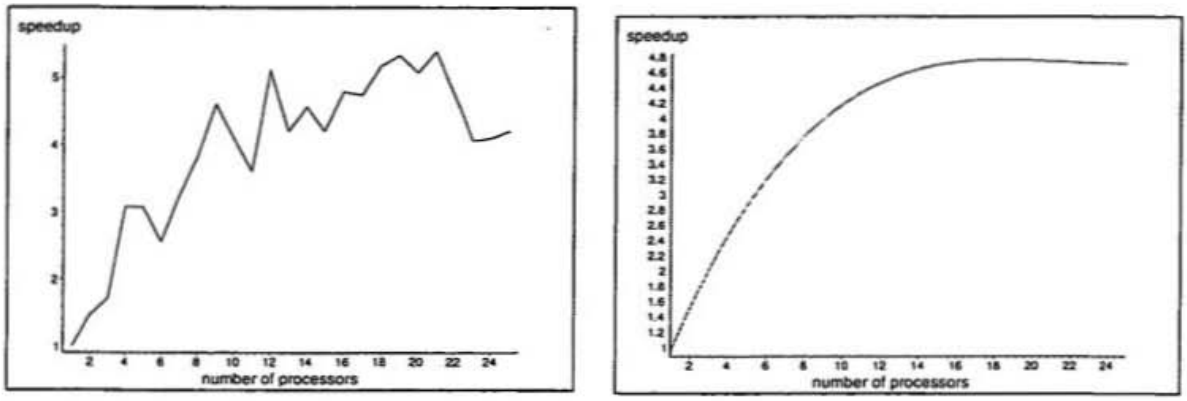


Figure 4.6: Speedup for Example 3.

4.2 M-timed nets

4.2.1 Example 3

This example is taken from [12]. The net models a parallel MIMD architecture. The state graph has 27399 states and approximately 7 times more arcs (197337).

The program has the following input:

```
Mnet(#T1=P4,P1/P2;
```

```
        #T2*1.0=P2/P7,P4;
        #T3=P4,P3/P6;
        #T4*1.0=P6/P9,P4;
        #T5=P5,P4/P8;
        #T6*1.0=P8/P21,P4;
        #T7,0.5=P7/P12;
        #T8,0.5=P7/P24;
        #T9=P11,P9/P10;
        #T10*1.0=P10/P14,P11;
        #T11=P12,P11/P15;
        #T12=P13,P11/P16;
        #T13*1.0=P15/P11,P1;
        #T14*1.0=P16/P29,P11;
        #T15,0.5=P14/P3;
        #T16,0.5=P14/P31;
        #T21=P25,P21/P22;
        #T22*1.0=P22/P28,P25;
        #T23=P25,P23/P26;
        #T24=P25,P24/P27;
        #T25*1.0=P26/P29,P25;
        #T26*1.0=P27/P25,P1;
        #T27,0.5=P28/P5;
        #T28,0.5=P28/P33;
        #T29=P32,P29/P30;
        #T30*1.0=P30/P35,P32;
        #T31=P32,P31/P34;
        #T32=P33,P32/P36;
        #T33*1.0=P34/P32,P9;
        #T34*1.0=P36/P32,P21;
        #T35,0.5=P35/P23;
        #T36,0.5=P35/P13)
mode=E;
mark(P1:2,P4,P9:3,P11,P21:3,P25,P29:2,P32);
```

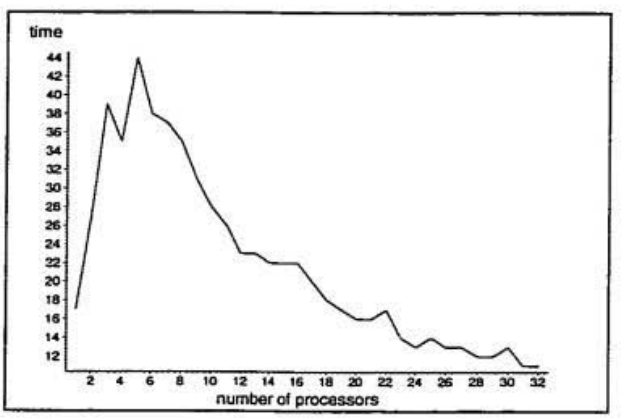


Figure 4.7: Execution time for Example 3.

The execution time, for the number of processors between 1 and 5, changes irregularly, it first increases, then decreases, increases again and only for more than 5

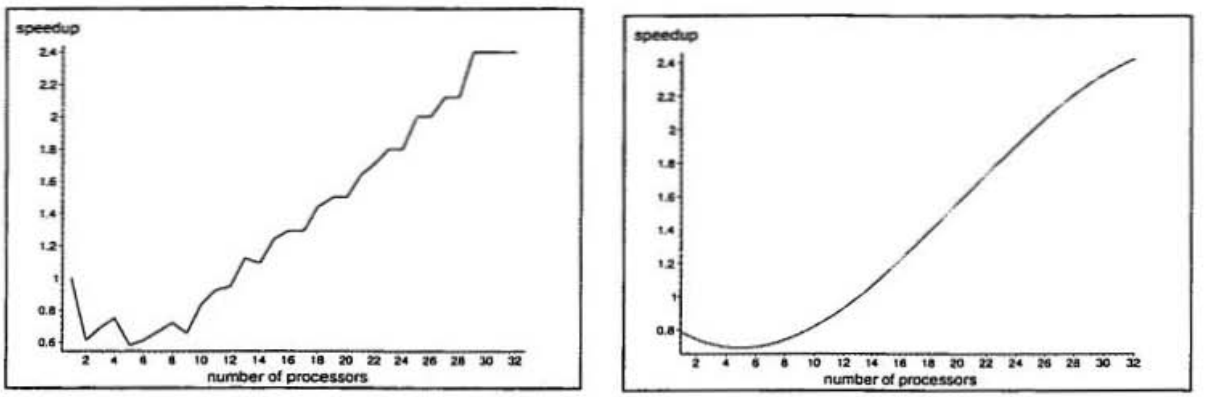


Figure 4.8: Speedup for Example 3.

processors shows a more regular trend. The increase in the execution time from 1 to 2 processors is not surprising. For 1 processor there is no communication, therefore the processor does not wait for data. When 2 processors are used, 52959 messages are exchanged; each processor's execution time contains a waiting time due to this additional communication.

When the number of processors changes from 2 to 3, it appears that the locality decreases (from 72 % to 59 %), the number of messages in the system nearly doubles (from 52959 to 93801), and the execution time increases from 27 seconds to 39 seconds. The change from 3 to 4 processors improves the execution times because the average locality increases to 71%. For 5 processors, the locality drops again to 46%, and the execution time becomes larger once more.

The execution time changes more regularly from 5 to 32 processors. However, the execution time is improving rather slowly. Although the region size decreases with increasing number of processors, the number of searches due to external data per region is quite large, so the computational time does not decrease significantly.

The speedup obtained in this example is shown in the Figure 4.8.

4.3 Concluding remarks

The experiments indicate that the program's execution time depends on: (1) the average time needed for processing a single state, (2) the average number of arcs per state, and (3) the locality of the partitioning function.

- If the average state processing time is high, the time a processor spends in creating the successor states is significantly reduced with each additional processor added to the virtual machine.
- If the average number of arcs per state is large, the number of messages in the system grows with each additional processor, thus affecting processor's computation time (the searching of external states), and its waiting time.
- If the locality decreases with additional processors, the number of messages in the system increases, and, on average, each processor's waiting time increases.

For nets with large number of arcs per state and small state processing time, the communication time seems to dominate the execution time, so only moderate speedups can be obtained. Faster communication medium might improve the results.

Distributed processing is quite efficient for nets with high state processing time and small number of arcs per state (like D–timed nets).

Chapter 5

Conclusions

Distributed algorithms for the generation of state graphs of Petri nets have evolved as an alternative to sequential methods, which, due to their high computational and memory demands, become insufficient for nets with large state spaces. However, creating efficient distributed algorithms for this problem is difficult because the irregularity of the state space induces a (large) inherent communication overhead (non-local states must be sent to their corresponding processors), and a computation overhead (a processor may need to deal with the same external state coming from several processors).

This thesis proposes a distributed algorithm for the generation of state space for *timed* Petri nets. The algorithm is built on top of the software package TPN-tools [38]. It is the first distributed algorithm for this class of nets. Similar research has been conducted for another class of Petri nets with time, stochastic Petri nets. There are significant differences between these two formalisms, resulting in very different models of the same systems, but many issues related to distributed implementation of the sequential algorithms are similar, so a comparison with this other work is instructive. The algorithm proposed in this thesis distinguishes itself from the others in the following aspects:

- The proposed algorithm totally separates the computational aspect from the com-

munication, by the use of the three concurrent processes per machine (Section 3.3).

- Each Generator gives priority to external states over local ones preventing therefore redundant incoming states from accumulating in the memory. All other algorithms give priority to local states. They wait to receive external states only after the available local states have been processed.

- The construction of the state graph is decomposed into two consecutive stages: during the first stage all the states and arcs are created, and all states are sent to the processors responsible for them. During this stage, cross–arcs directed to already sent non-local states are not sent, but stored. In the second stage, all cross-arcs directed to the same non-local state are sent in one message. This delayed sending has two consequences: (1) it reduces the traffic network, so that the external *states*, which are needed to complete the construction of the region, can be transferred with reduced delays, improving the performance, and (2) no matter how many cross–arcs a processor creates for one non-local state, all arcs will be sent in at most two messages. Also, the processor responsible for them needs to search them at most twice.

The primary objective of this work, implementation of distributed state space generation, has been successfully achieved; the program is able to generate state spaces up to a fairly large size on a cluster of medium memory sized machines (128 MB RAM diskless Linux PC's).

Experimental results suggest that the performance of the algorithm is influenced by both the structure of the model (the average state processing time, the average number of successors per state), and the choice of the partitioning function (which establishes the number of cross–arcs).

For the class of D–timed Petri nets with a high average state processing time and a small average number of arcs per state, the distributed implementation gives almost linear speedup. The algorithm allows the generation of relatively large state spaces of this class (an order of 10^6 states) in very reasonable run times. For this class of nets, a "better" behavior can be noted when increasing the state space size, which can be attributed to large and more uniformly distributed numbers of states assigned to each of the processors.

The proposed system can be extended in several directions:

- At present, the maximum problem size which can be handled is restricted by PVM memory demands and resource limitations; e.g., PVM uses dynamically allocated memory for messages en route between processes. Messages sent but not yet received accumulate in PVM's local daemon's memory. If messages are sent faster than the receiving processor consumes them, the diskless PC runs out of memory. This limitation will disappear if a protocol with message acknowledgments is implemented. This is a straightforward extension; each Generator would not send data to another one before having a notice specifying that the latter is ready for receiving. It would be interesting to note how this overhead for acknowledging messages would affect the program's speedup.

- The experiments presented in Chapter 4, as well as the examples in the literature, corroborate the observation that the performance of the distributed state space generation algorithms is influenced by the partitioning function. The partitioning function used here achieves a very good memory balance. However, a function with a better locality would reduce the traffic in the system, and therefore, the communication overhead. Unfortunately, finding a good partitioning function from the net's structure is a non-trivial task, which clearly needs more research.

As an alternative, the program's performance could be improved by using dynamic partitioning mechanisms. Dynamic partitioning remaps states to processors according to two criteria: memory balance, and execution time.

Finally, the developed distributed implementation of the state space generation can be used as a starting point for a distributed steady state solution of the state graph, a "natural" next step in quantitative analysis of timed net models.

Bibliography

[1] T. Agerwala, "Putting Petri Nets to Work"; *IEEE Computer*, vol. 12, no. 12, pp. 85–94, 1979.

[2] T. Agerwala and M. Flynn, "Comments on Capabilities, Limitation, and 'Correctness' of Petri Nets"; *Proc. of the First Annual Symp. on Computer Architecture*, pp. 81–86, 1973.

[3] M. Ajmone Marsan, G. Balbo, and G. Conte, "A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Systems"; *ACM Transactions on Computer Systems*, vol. 2, no. 2, pp. 93–122, 1984.

[4] M. Ajmone Marsan, G. Balbo, and G. Conte, *Performance Models of Multiprocessor Systems.* MIT Press, 1986.

[5] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis, *Modelling with Generalized Stochastic Petri Nets.* Wiley Series in Parallel Computing, Wiley, New York, 1995.

[6] S. C. Allmaier and G. Horton, "Parallel Shared-Memory State-Space Exploration in Stochastic Modeling"; *Solving Irregularly structured Problems in Parallel (IRREGULAR'97), Lecture Notes in Computer Science*, vol. 1253, pp. 207–218, Springer-Verlag, 1997.

[7] S. C. Allmaier, M. Kowarschik, and G. Horton, "State Space Construction and Steady State Solution of GPSN on a Shared Memory Multiprocessor"; *Proc. IEEE Int. Workshop Petri Nets and Performance Models (PNPM '97)*, pp. 112–121, IEEE Computer Society, June 1997.

[8] S. Allmaier, S. Dalibor, and D. Kreische, "Parallel Graph Generation Algorithms for Shared and Distributed Memory Machines"; *Parallel Computing: Fundamentals, Applications and New Directions (Proc. of the Conference ParCo'97)* (E. D'Hollander, G. Joubert, F. Peters, and U. Trottenberg, eds.), *Advances in Parallel Computing*, vol. 12, pp. 581–588, Elsevier, North-Holland, 1998.

[9] S. Allmaier and D. Kreische, "Parallel Approaches to the Numerical Transient Analysis of Stochastic Reward Nets"; *Application and Theory of Petri Nets 1999 (Proc. 20th International Conference, IACTPN'99)* (S. Donatelli and J. Klejn, eds.), *Lecture Notes in Computer Science*, vol. 1639, pp. 147–167, Springer-Verlag, 1999.

[10] G. Balbo, "On the Success of Stochastic Petri Nets"; *Proc. of the IEEE Conf. on Petri Nets and Performance Models (PNPM'95)*, Durham, NC, pp. 2–9, October 1995.

[11] F. Bause and P. Krinzinger, *Stochastic Petri Nets - An Introduction to the Theory.* Vieweg, Wiesbaden, 1996.

[12] S. Caselli and G. Conte, "GSPN Models of Concurrent Architectures with Mesh Topology"; *Fourth Int. Workshop on Petri Nets and Performance Models (PNPM 91)*, pp. 280–289, IEEE Computer Society, 1991.

[13] G. Chiola, M. Ajmone Marsan, G. Balbo, and G. Conte, "Generalized Stochastic Petri Nets: A Definition at the Net Level and its Implications"; *IEEE Transactions on Software Engineering*, vol. 19, pp. 89–107, February 1993.

[14] C. Coves, D. Crestani, and F. Prunet, "How to Manage Coverability Graphs Construction: an Overview"; *Proc. IEEE Int. Conf. on Systems, Man, and Cybernetics (SMC'98)*, pp. 541–546, 1998. San Diego, California.

[15] E. Dijkstra, W. Feijen, and A. van Gasteren, "Derivation of a Termination Detection Algorithm for Distributed Computations"; *Information Processing Letters*, vol. 16, no. 5, pp. 217–219, 1983.

[16] D. Ferrari, *Computer Systems Performance Evaluation.* Prentice-Hall, Inc., 1978.

[17] G. Florin and S. Natkin, "One-Place Unbounded Stochastic Petri Nets: Ergodic Criteria and Steady-State Solutions"; *Journal of Systems and Software*, vol. 6, no. 1-2, pp. 103–115, 1986.

[18] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[19] B. Haverkort, A. Bell, and H. Bohnenkamp, "On the Efficient Sequential and Distributed Generation of Very Large Markov Chains from Stochastic Petri Nets"; *Proc. IEEE Int. Workshop Petri Nets and Performance Models (PNPM'99)*, pp. 12–21, IEEE Computer Society, September 1999.

[20] K. Kant, *Introduction to Computer Systems Performance Evaluation.* McGraw Hill, Inc., 1992.

[21] W. Knottenbelt, M. Mestern, P. Harrison, and P. Kritzinger, "Probability, Parallelism and the State Space Exploration Problem"; *Computer Performance Evaluation (TOOLS'98)*, *Lecture Notes in Computer Science*, vol. 1469, pp. 165–179, Springer-Verlag, 1998.

[22] H. Kobayashi, *Modeling and Analysis: An Introduction to System Performance Evaluation Methodology.* Addison-Wesley Publishing Co., 1978.

[23] P. Marenzoni, S. Caselli, and G. Conte, "Analysis of Large GSPN Models: a Distributed Solution Tool"; *Proc. IEEE Int. Workshop Petri Nets and Performance Models (PNPM'97)*, pp. 122–131, IEEE Computer Society, June 1997.

[24] M. K. Molloy, *On the Integration of Delay and Throughput Measures in Distributed Processing Models.* PhD thesis, University of California, 1981.

[25] M. K. Molloy, "Performance Analysis Using Stochastic Petri Nets"; *IEEE Transactions on Computers*, vol. C-31, no. 9, pp. 913–917, 1982.

[26] T. Murata, "Petri Nets: Properties, Analysis, and Applications"; *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[27] S. Natkin, *Des Reseaux de Petri Stochastiques et leur Application a l'Evaluation des Systemes Informatiques.* PhD thesis, CNAM, Paris, 1980.

[28] D. M. Nicol and G. Ciardo, "Automated Parallelization of Discrete State-Space Generation"; *Journal of Parallel and Distributed Computing*, vol. 47, no. 2, pp. 153–167, 1997.

[29] D. Nicol, "Noncommital Barrier Synchronization"; *Parallel Computing*, vol. 21, no. 4, pp. 529–549, 1995.

[30] D. Nicol and G. Ciardo, "Distributed State Space Generation of Discrete State Stochastic Models"; *INFORMS Journal on Computing*, vol. 10, no. 1, pp. 82–92, 1998.

[31] J. Peterson, "Petri Nets"; *ACM Computing Surveys*, vol. 9, no. 3, pp. 223–252, 1977.

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

[33] C. Snow, *Concurrent Programming.* Cambridge University Press, 1992.

[34] W. Stewart, *Introduction to the Numerical Solution of Markov Chains.* Princeton University Press, 1994.

[35] P. Stotts and T. Pratt, "Coverability Graphs For a Class of Synchronously Executed Unbounded Petri Nets"; *Journal of Parallel and Distributed Computing*, vol. 10, no. 4, pp. 253–260, 1990.

[36] B. Stroustrup, *The C++ Programming Language (3-rd ed.).* Addison-Wesley Publishing Co., 1997.

[37] G. Tel, *Topics in Distributed Algorithms.* Cambridge University Press, 1991.

[38] W. M. Zuberek, "Timed Petri Nets: an Introduction – lecture notes for course "Modeling and Analysis of Computer Systems""; Memorial University of Newfoundland, Canada, 1998.

[39] W. Zuberek, "M-timed Petri Nets, Priorities, Preemptions, and Performance Evaluation of Systems"; *Advances in Petri nets 1985* (G. Rozenberg, ed.), *Lecture Notes in Computer Science*, vol. 222, pp. 478–498, Springer-Verlag, 1986.

[40] W. Zuberek, "On D–Timed Petri Nets, Timeouts, Protocols, and Modelling of Communicating Systems"; Tech. Rep. # 8609, Department of Computer Science, Memorial University of Newfoundland, Canada, November 1986.

[41] W. Zuberek, "On M–Timed Petri Nets, Priorities, Preemptions, and Performance Evaluation of Systems"; Tech. Rep. # 8606, Department of Computer Science, Memorial University of Newfoundland, Canada, August 1986.

[42] W. Zuberek, "On Modelling and Evaluation of Multiprocessor Systems Using Extended M–Timed Petri Nets"; Tech. Rep. # 8605, Department of Computer Science, Memorial University of Newfoundland, Canada, August 1986.

[43] W. Zuberek, "On Generation of State Space for Timed Petri Nets"; *Proc. of ACM 16th Annual Computer Science Conference*, Atlanta, Georgia, pp. 239–248, ACM, Feb. 1988.

[44] W. Zuberek, "Timed Petri Nets, Definitions, Properties, and Applications"; *Microelectronics and Reliability*, vol. 31, no. 4, pp. 627–644, 1991.

[45] W. Zuberek, "Modeling with Timed Petri Nets - Event-Driven Simulation"; Tech. Rep. #9602, Department of Computer Science, Memorial University of Newfoundland, Canada, September 1996.