UNIVERSITÀ DEGLI STUDI DI TORINO

Dipartimento di Informatica
C.so Svizzera, 185 - 10149 Torino (Italia)

DOTTORATO DI RICERCA IN INFORMATICA
(CICLO XIX)

TITOLO DELLA TESI

Timed and Stochastic Model Checking of Petri Nets

TESI PRESENTATA DA:

Davide D'Aprile

TUTOR:

Prof.ssa Susanna Donatelli

COORDINATORE DEL CICLO:

Prof. Piero Torasso

ANNI ACCADEMICI
'03 - '04 / '04 - '05 / '05 - '06

To *Francesca*,
my wife, for her encouraging love.

To *Rocco*,
my dear Grandfather, because he taught me the pride of being a mild man.

Acknowledgements

I would like to thank:

Susanna Donatelli and *Jeremy Sproston* for their support;

Gian Luca, *Claudio*, *Rossano*, *Livio* and *Arianna* for the time we spent together;

Enrico and *Katia* for their psychological aid!

Sante, *Maria Teresa*, *Fabio*, *Stefania*, *Francesca*, *Anna*, *Rita*, and *Dora* because they love and believe me.

Contents

List of Figures

List of Tables

Part I

Introduction

0.1 Introduction

Electronic commerce, telephone switching networks, highway and air traffic control systems, medical instruments are examples of systems that, provided with a proper integration with the Internet and embedded systems (considering both hardware and software implementations), surely are improved with respect to the tasks they have been designed for. Clearly, the need for reliable hardware and software systems is critical, and even if failure is not life-threatening, the consequences can be economically devastating, so it has been becoming more and more important to develop methods that increase our confidence in the correctness of such systems. For these reasons, over the last past decades we have assisted to a growth of interest around the so called *formal methods*, which are defined as mathematically based techniques for the specification, development and verification of software and hardware systems.

The *model checking* technique is an automatic approach for verifying finite state concurrent systems, having a number of advantages over traditional formal methods that are based on simulation, testing and deductive reasoning. It is basically a procedure that checks, for every reachable state of a specific system, if a given property of interest holds, or does not. The properties under investigation, which formalize the specification, are usually given in terms of some appropriate logics, often provided with an explicit or an implicit notion of time, while the system specification is described using either a high level (e.g. Petri Nets, Process Algebra, etc.) or a low level (e.g. Timed Automata, Continuous Time Markov Chain, etc.) formalism.

In this work we are primarily interested in two classes of systems:

- the *real-time* systems, so called because the correctness of an operation depends not only upon the logical correctness of the operation but also upon the time at which it is performed. In a real-time system the activities occur (within) by a given (period of) time duration. The standard classification is that in a *hard* real-time system the completion of an operation after its deadline is considered useless (or even a catastrophic event), while a *soft* real-time system will tolerate such lateness, and for example respond with a decreased service quality (e.g. skipping images in video). Example of real-time systems are: a car engine control system, medical systems and industrial process controllers, etc.

- the *stochastic systems*, in which the activities have a duration specified by a stochastic probability distribution; typical examples of such systems are: stock market and exchange rate fluctuations, signals such as speech, audio and video; medical data such as a patient's EKG, EEG, blood pressure or temperature; and random movements such as Brownian motion or random walks. On these systems the attention is usually focused on the verification of performance and dependability properties.

Given our level of expertise in the use of Petri nets, a mathematical formalism which is well suited for modeling concurrent *discrete event dynamic systems* (DEDS) and for all those systems exhibiting complex behaviors due to the presence of synchronization and resource sharing mechanisms, we have chosen the class of Time Petri nets and two classes of stochastic Petri nets (named Generalized Stochastic Petri nets and Stochastic Well-formed nets) to model and analyze real-time systems and stochastic systems, respectively. In fact, the mathematical foundations of the Petri net formalism allow both correctness (i.e., logical) and efficiency (i.e., performance) analysis, while its graphical approach lets to produce self documented specification.

For these and the above motivations we have decided to investigate about the possibility to add model checking capabilities into our Petri nets tool, named GREATSPN. Of course, as pointed out before, this implied the utilization of appropriate temporal logics, to specify the requirements of the specific real-time or stochastic systems under investigation; we have utilized the Timed Computational Tree Logic for the former and the Continuous Stochastic Logic for the latter.

This thesis is organized as follows. Chapter 1 and Chapter 2 are introductory chapters providing the basics about Petri nets and about model checking, respectively. In Chapter 3 we present our first main contribute, consisting in a technique for model checking Time Petri nets against the Timed Computational Tree Logic. Chapter 4 is devoted to our second main contribute, illustrating our solutions for model checking Generalized Stochastic and Stochastic Well-formed nets against the Continuous Stochastic Logic. Chapter 5 concludes the thesis, recalling the contributes, within the advantages, the open problems and the future directions of the proposed solutions.

Part II

Backgrounds

Chapter 1

The PN formalisms

Petri nets (PN) [53, 58, 59] are a mathematical formalism which is well suited for modeling concurrent *discrete event dynamic systems* (DEDS) and for all those systems exhibiting complex behaviors due to the presence of synchronization and resource sharing phenomena; it has been satisfactorily applied to fields such as communication networks, computer systems, discrete part manufacturing systems, etc. [32].

The mathematical foundations of the formalism allow both correctness (i.e., logical) and efficiency (i.e., performance) analysis, while its graphical approach let to produce self documented specification.

To be noticed, in addition, is the fact that, rather than a single formalism, PN are a family of formalisms, ranging from low to high level, each of them best suited for different purposes [66].

A PN model of a dynamic system consists of two parts:

- A *net structure*, i.e., an inscribed bipartite directed graph, that represents the static part of the system. There are two kinds of nodes: *places*, corresponding to state variables, and *transitions*, representing transformations of state variables, pictorially represented as circles and boxes, respectively. We may have arcs from place to transition (in this case, we have an *input place* for the transition, or an *output transition* for the place) or *viceversa* (in this case, we have an *output place* for the transition, or an *input transition* for the place). The inscriptions may be very different, leading to various families of nets: if the inscriptions are simply natural numbers associated with the arcs, named weights or multiplicities, *Place/Transition* (P/T) nets are obtained [53, 58, 59], while more elaborate inscriptions, associated with places, transitions, and arcs, lead to the so called *High Level Petri Net* formalisms [41].

- A *marking*, pictorially represented by tokens inside the places, that represents a distributed overall state on the structure. The marking of a place, called *state variable*, is its state value.

A *net system* is a net structure together with an initial marking. The system dynamics (i.e., the system behavior) is given by the evolution rules for the marking: a transition occurs when the input state values fulfill some condition expressed by the arc inscriptions. The occurrence of a transition changes the values of its adjacent state variables, according again to the arc inscriptions.

In Figure 1.1 we show a simple PN model with five places, $idle$, $working_1$, $working_2$, $waiting_1$, and $waiting_2$, differently linked *to* and *from* transitions named $start$, end_1, end_2, $synchronized$. The arc inscriptions, when not explicitly indicated, indicate that one token is required (for every arcs connecting places to transitions) or produced (for every arcs connecting transitions to places). The initial marking is given by the presence of two tokens in place $idle$. Figure 1.2 shows the net evolution after the firing of the transition called $start$: one token is deleted from place $idle$, and two tokens are deposed in place $working_1$ and $working_2$, respectively.

The *interpretation* of a model precises the semantics of objects and their behavior: so, an interpretation may give a physical meaning to the net entities (places, transitions, tokens), evolution conditions and, possibly, will define the actions generated by the evolutions.

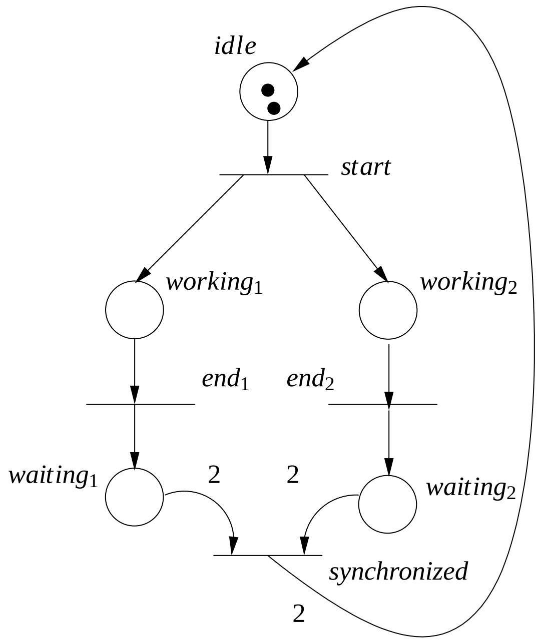


Figure 1.1: A PN model

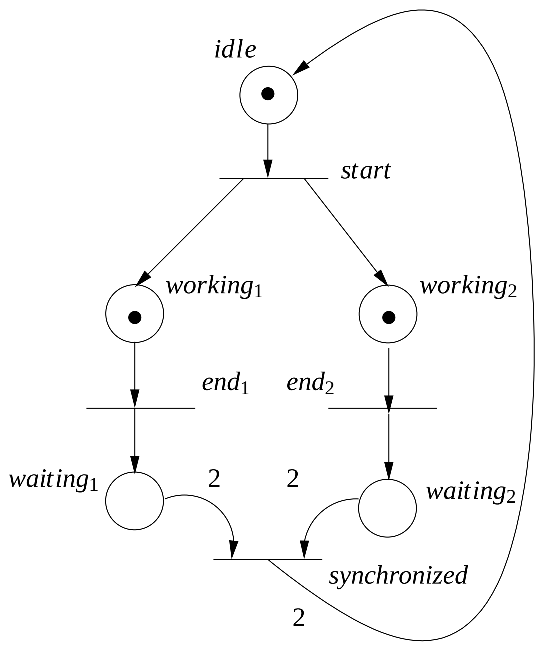


Figure 1.2: The PN model of Figure 1.1, after the firing of the transition *start*

Considering the example depicted in Figure 1.1, we could give the following interpretation to the static and dynamic aspects of the net.

- place *idle*: contains tasks of a given job; with the given marking, we have two tasks, indicated by two tokens within it.
- transition *start*: lets one task be removed (arc from place *idle* to transition *start*) and divided into two subtasks that are assigned to two different processing systems (arcs from transition *start* to places $working_1$ and $working_2$, respectively), in order to be processed .
- places $working_1$ and $working_2$: indicates the two subtasks are working (in parallel) on the two assigned processing systems.
- transitions end_1 and end_2: depict the ending of computational activities of the above cited subtasks on the two processing systems.
- places $waiting_1$ and $waiting_2$: indicates each subtask is waiting for the other respective subtasks of the second task.
- transition *synchronized*: finally, the two subcomponents of both two the processing systems may synchronized each others (arcs, with inscription equals to 2, from places $waiting_1$ and $waiting_2$ to transitions *synchronized*), and the processing cycle is initialized again (arc, with inscription equals to 2, from transition *synchronized* to place *idle*).

Taking into account the purposes of this thesis, a particularly interesting family of net interpretations is obtained when time and probabilities are associated with the model, as will be illustrated lated.

The chapter is organized as follows. In Section 1.1 we introduce what is considered the basic PN formalism, and explore its abilities for the modeling of systems. Section 1.2 is dedicated to the introduction of time into the basic PN formalism.

1.1 The PN basic model

A *Place/Transition* (P/T) net [53, 58, 59] is a tuple $S = (P, T, I, O, M^0)$, where P is the set of places, T is the set of transitions, $I, O : P \to T \to 2^{\mathbb{N}}$ define the input and output arcs with associated multiplicity, and $M^0 : P \to \mathbb{N}$ describes the initial marking.

The structure of a net is something static. Assuming that the behavior of a system can be described in terms of the state and its changes, the dynamics on a net structure is created by defining its initial state and the state evolution rule.

The marking of a net N is a place indexed vector $\mathbf{m} \in \mathbb{N}^P$ which assigns a non negative integer (number of tokens) to each place. *A P/T net system* is the pair $S\langle N, m_0\rangle$, where N is a P/T net and m_0 is its initial marking. The number of tokens at a place represents the local state of the place, i.e., the value of the state variable represented by that place, which in the P/T formalism is an integer. The state of the overall net system is defined by the collection of local states of the places. Therefore, the vector $\mathbf{m}$ is the state vector described by the net system. Pictorially, we put $\mathbf{m}(p)$ black dots tokens in the circle representing place p. The marking in a net system evolves as follows:

1. A transition is said to be *enabled* at a given marking when each input place has at least as many tokens as the weight of the arc joining them. The number of simultaneous enabling of a transition t at a given marking. $\mathbf{m}$ is called its *enabling degree*.

2. The *occurrence*, or *firing*, of an enabled transition is an atomic operation that removes from (adds to) each input (output) place a number of tokens equal to the weight of the arc joining the place (transition) to the transition (place).

Let $\mathcal{S}$ be a P/T system:

- An occurrence or *firing sequence* from $\mathbf{m}$ is a sequence $\sigma = t_1 \cdots t_k \cdots$ such that $\mathbf{m} \xrightarrow{t_1} \mathbf{m_1} \cdots \xrightarrow{t_k} \mathbf{m_k} \cdots$. If the firing of sequence σ yields the marking m', this is denoted by $\mathbf{m} \xrightarrow{\sigma} \mathbf{m'}$.

- The *language* of $\mathcal{S}$, denoted by $L(\mathcal{S})$, is the set of all the occurrence sequences from $\mathbf{m_0}$.
- The *reachability set* (RS) of $\mathcal{S}$, denoted by $RS(\mathcal{S})$, is the set of all the markings reachable from $\mathbf{m_0}$ by firing some sequences in $L(\mathcal{S})$.
- The *reachability graph* (RG) of $\mathcal{S}$, denoted by $RG(\mathcal{S})$, is a labeled graph where the vertices are the reachable markings and there is an edge labeled t from vertex $\mathbf{m}$ to vertex $\mathbf{m'}$ if and only if $\mathbf{m} \xrightarrow{t} \mathbf{m'}$.

Taking again our running example, as shown in Figure 1.1, we can provide the following examples to what was theoretically exposed in this section.

- the set P is given by the elements $idle$, $working_1$, $working_2$, $waiting_1$, and $waiting_2$;
- the set T is given by the elements $start$, end_1, end_2, $synchronized$;
- $m_0 = 2 \cdot idle$ (using a vector-like notation to indicate markings).
- the function I is so defined:
  $\{idle \rightarrow synchronized \rightarrow 2,$
  $waiting_1 \rightarrow end_1 \rightarrow 1,$
  $waiting_2 \rightarrow end_2 \rightarrow 1,$
  $working_1 \rightarrow start \rightarrow 1,$
  $working_2 \rightarrow start \rightarrow 1\};$
- the function O is so defined:
  $\{idle \rightarrow start \rightarrow 1,$
  $waiting_1 \rightarrow synchronized \rightarrow 2,$
  $waiting_2 \rightarrow synchronized \rightarrow 2,$
  $working_1 \rightarrow end_1 \rightarrow 1,$
  $working_2 \rightarrow end_2 \rightarrow 1\};$

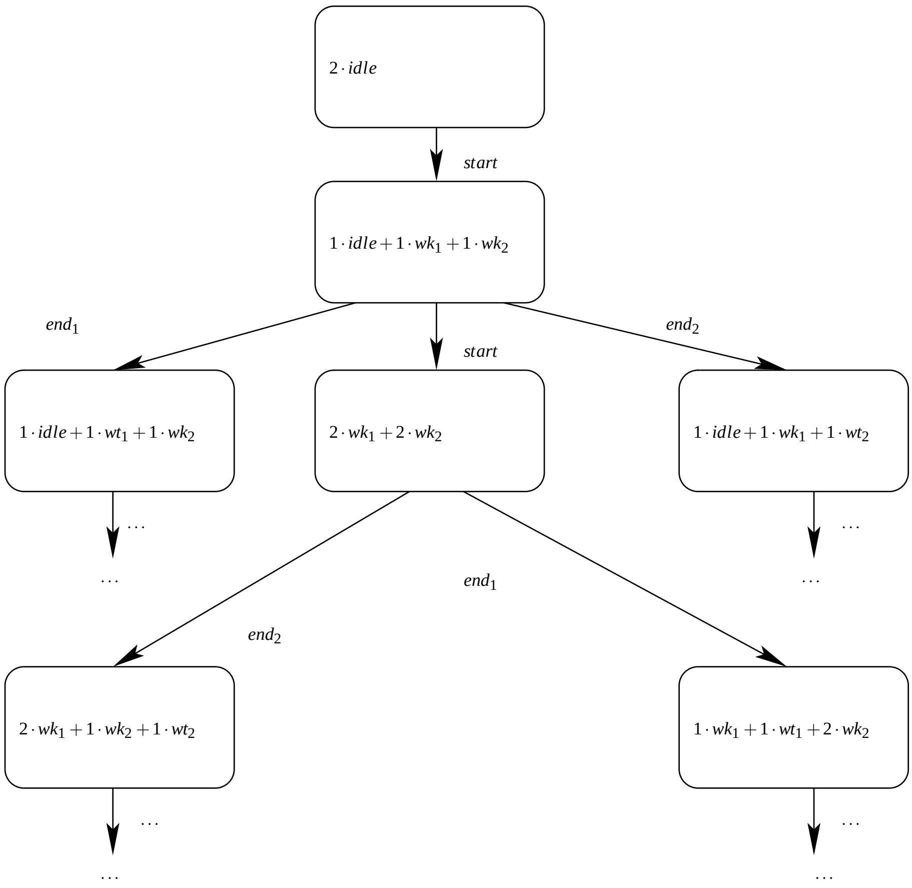


Figure 1.3: A portion of the RG of the PN model of Figure 1.1

- $\sigma_1 = start \cdot start \cdot end_1 \cdot end_1 \ldots$ and $\sigma_2 = start \cdot end_1 \cdot end_2 \ldots$ are two (partially listed) allowed firing sequences;

- a portion of the RG is given in Figure 1.3, in which every node is an element of the RS; starting from the root node, corresponding to the initial marking, we obtain a set of paths, given by all possible firing sequences. Note that, for readability reasons, in Figure 1.3 we abbreviated $working_i$ with wk_i and $waiting_i$ wt_i.

Observe that, in the firing rule, enabled transitions are never forced to fire: this is a form of *non determinism*. It must also be noticed that it is not specified whether the occurrence of a transition takes some time, since time has not been introduced yet. This is again matter of the interpretation we give to the model.

Properties of interest

The properties usually considered with this basic formalism are called *qualitative*, and they may be synthesized as follows:

- *boundedness*: i.e., finiteness of the state space; every place in the net has a bounded number of token within it.
- *liveness*: related to potential fireability in all reachable markings;
- *deadlock-freeness* is a weaker condition w.r.t. liveness, in which only global infinite activity (i.e., fireability) of the net system model is requested, even if some parts of it do not work at all.
- *reversibility*: characterizing recoverability of the initial marking from any reachable marking;
- *mutual exclusion*: dealing with the impossibility of simultaneous sub-markings (*p-mutex*) or firing concurrency (*t-mutex*).

Our running example (Figure 1.1) is a bounded, live (then deadlock-free), and reversible system. An example of p-mutex (t-mutex) property is given by the fact that is not possible having simultaneously the markings (enabled transitions) $2 \cdot idle$ and $2 \cdot waiting_1 + 2 \cdot waiting_2$ ($start$ and *synchronized*). Figure 1.4, derived by Figure 1.1 by labeling with one token (i.e., by an empty inscription), instead of two tokens, the input arcs of the transition *synchronized* , is affected by unboundedness, because every task generates two new tasks at the end of every computational cycle. Figure 1.5, obtained by Figure 1.1 simply deleting the arc connecting transition *synchronized* with place $idle$, depicts a non-live (then, also a prone-to-deadlock) system (after the second firing of transition *synchronized* there are no more tokens in the net, so none of the transitions can be able to fire); obviously such system is also not reversible.

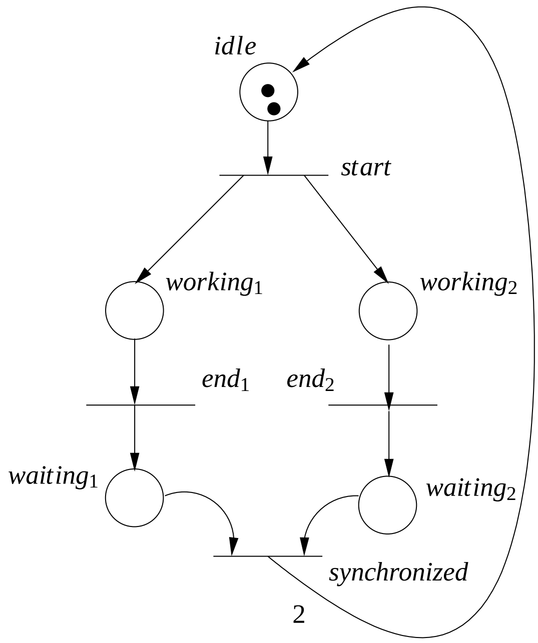


Figure 1.4: A PN model, derived from Figure 1.1, affected by unboundedness of the state space

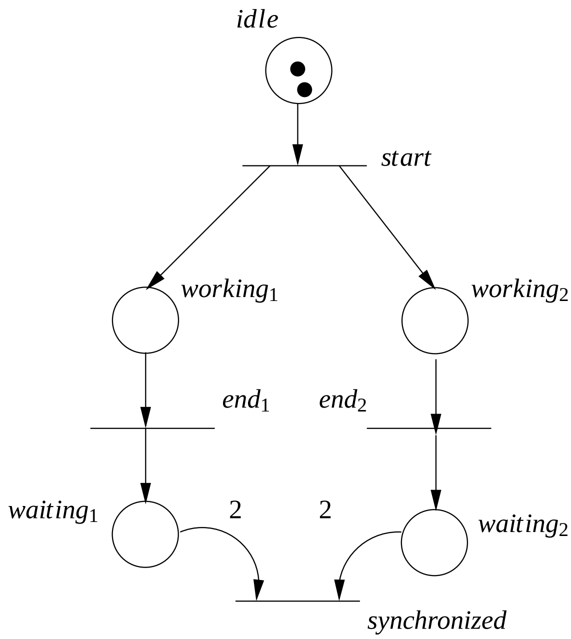


Figure 1.5: A PN model, derived from Figure 1.1, prone to deadlock and non-reversibility

Analysis techniques

Conventionally, analysis techniques for PN are classified as:

- *Enumeration*, which are based on the construction of a RG which represents, individually, the net markings and single transition firings between them. If the net system is bounded, the RG is finite and the different qualitative properties can be easily verified. If the net system is unbounded, the RG is infinite and its construction is not possible. In this case, graphs known as *coverability graphs* can be constructed [44]. Despite of its power, enumeration is often difficult to apply, even in small nets, due to its computational complexity (it is strongly combinatorial).
- *Transformation*. It proceeds transforming a net system S into S', preserving the set of properties to be verified, in such a way the state space of S' may be bigger than that of S, but S' may belong to a subclass for which state enumeration can be avoided.
- *Reduction* methods are a special class of transformation methods in which a sequence of net systems preserving the properties to be studied is constructed, and for which the respective RG s appear as decreasing in the cardinality of nodes or edges.
- *Structural*. Investigate the relationships between the behavior of a net system and its structure, while the initial marking acts, basically, as a parameter. In this last class of analysis techniques, we can distinguish two subgroups:
  - *Linear algebra*-based techniques, which permit a fast diagnosis without the need for enumeration.
  - *Graph based techniques*, in which the net is seen as a bipartite graph and some *ad-hoc* reasoning (frequently derived from the firing rule) is applied.

Simulation methods have also been applied to study systems modeled with P/T nets. It proceed playing the *token game* (firing enabled transitions) on the net system model under certain strategies. In general, simulation methods do not allow to prove properties, but they might be of great help for understanding the modeled system or to fix the manifested problems during simulation.

1.2 Introducing temporal information into PN

Petri nets were originally proposed as a causal model, explicitly neglecting time. Subsequently, the need to face with timing aspects in the modeling activity, for example for describing stochastic or real time systems, leaded to different approaches for introducing time aspects into the PN formalisms: for example, time concerns may be associated to places, token, transitions, and so on.

We will consider PN formalisms in which timing is associated with transitions.

The introduction of time into the basic net model entails many subtle difficulties in the definition of a coherent model. In particular, the specification of the timed behavior, in order to evaluate performance, requires a great deal of interpretation to be added over the basic net system model.

First of all, it is needed to specify what happens when the enabling degree of a transition is greater then one; since a timed transition may be thought as a server performing the required task, we could taking into account the following alternatives, when multiple instances of the activity are requesting a service to the hypothetical server associated to the the enabled transition:

- 1-**server** semantics: they are queued and served 1-at-time;
- $infinite$-**server** semantics: they are served in parallel (like if there are infinite servers);
- k-**server** semantics: they are served k-at-time in parallel (this approach coincides with the previous one, when the enabling degree is less or equal to k).

The first alternative suggests the need to specify also *queuing policies*.

It is necessary, in addition, to associate with the model an *execution policy*, comprising two specifications: a rule to choose the next transition to fire in any marking (the *firing policy*) and a criterion to account for the past history of the model whenever a transition fires (the *memory policy*).

As regards the firing policy, two alternatives are basically possible: either using the delays associated with transitions to decide which one will fire next, or adding a specific metrics for this purpose. In this thesis, we will use classes of PN in which the transition with the minimum remaining delay is the one that fires first; this approach is called *race policy*.

As regards the memory policy, again two basic alternatives are possible at every change of marking:

- **continue**: the timers[1] associated with transitions hold their present values and will continue being decremented later on.
- **restart**: the timers associated with transitions are restarted, i.e. their present values are discarded, and new values will be generated when needed.

The memory policy affects transitions that fire as well as transitions that lose their enabling due to the change of marking, and transitions that keep their enabling in the new marking. The memory of transitions that fire is irrelevant, since in this case a new delay instance must alway be generated. The memory of transitions that do not fire is often assumed to be of the following types:

- **re-sampling**: the timer of the transition is reset to a new value at any change of marking.
- **enabling memory**: if in the new marking the transition is still enabled, the value of the timer is kept; otherwise, it is reset to a new value.

[1]we can associate a timer with each transition, in order to describe the evolution of a timed PN model: timers are decremented at constant speed while transitions are enabled, and when a timer runs down to zero the corresponding transition may fire.

- **age memory**: the timer is kept at any change of marking.

Given a specific class of systems to be described, an appropriate PN formalism, given form the combination of the above depicted different solutions, may be chosen.

In the following we will concentrate on two (classes of) formalisms useful for describing real-time (Section 1.2.1) or stochastic (Section 1.2.2) systems.

1.2.1 The TPN formalism

The two main timed extensions of PN are Merlin and Faber 's time Petri nets (TPN) [51] and Ramchandani's timed Petri nets [64]. Whereas in the former the transitions have to fire in a given temporal interval, in the latter system's timing characteristics are represented by minimal durations between the firing of transitions ("as soon as" semantics).

Since the class of timed Petri nets is included in the class of TPN [60], we are mainly interested in this latter formalism.

As an example of TPN model, look at Figure 1.6, in which we consider again our running model, adding to it a transition *initialize*, with input and output arcs linked to place *idle*, for taking into account an initialization procedure for the tasks accessing the system; as it can be observed we provide timing information, by means of temporal intervals depicted near to every transitions.

In the original TPN specification non-deterministic behavior is not associated with a measure of probability over the variety of feasible behaviors. Moreover, an enabled transition, can be disabled before firing and before reaching its maximum enabling time: none of memory, server semantics or queue policies is defined *a priori*.

In the following we will consider the so called *strong semantics* (the elapsing of time must not disable transitions), as specified in [8] as T-TPN, referring to it simply as TPN.

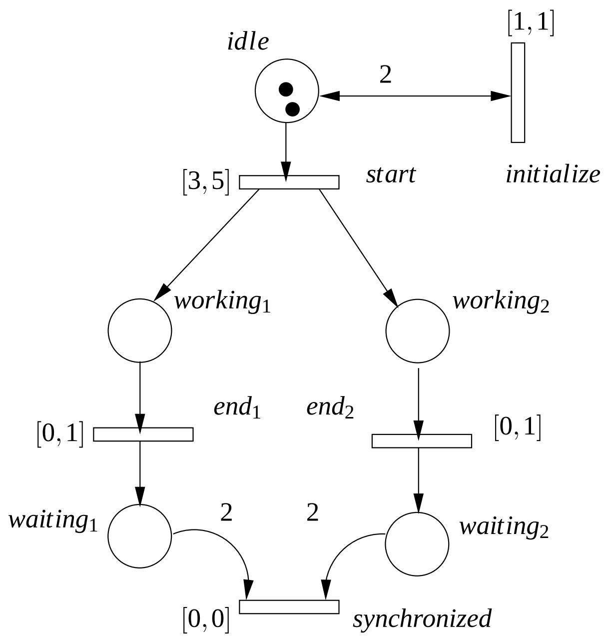


Figure 1.6: A TPN model, derived by the PN model of Figure 1.1

Finally, it is important to notice that non interpreted net systems may be redefined as TPN, putting the minimum and maximum bounds equal to zero, without changing their non deterministic behavior. On the contrary, the behavior of a timed model may be different from the one of the uninterpreted net system. For example, taking the example in Figure 1.6, it is possible to understand that the firing sequences starting from $start$ are interdicted by the timing information provided with transition $initialize$.

In the following we provide the formal syntax and semantics of TPN (Section 1.2.1), but first we need to recall, in Section 1.2.1, the timed transition system notation, useful for a rigorous behavioral description of TPN.

The TTS **formalism**

A *timed transition system* (TTS) is a state-transition graph, where the labels of transitions can belong to a finite set of events Σ or can be real numbers. A TTS S can be represented by a tuple $\langle Q, Q^O, \Sigma, \rightarrow \rangle$ where Q is a set of the states, $Q^0 \in Q$ is an initial state, Σ is the set of the events (or labels), and $\rightarrow \subseteq Q \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times Q$ is the transition relation (i.e. the set of transitions). When $(q, \sigma, q') \in \rightarrow$, it is denoted $q \xrightarrow{\sigma} q'$, and denotes that when the state of the system is q, it can change to q' upon event a. The transitions that are labeled with an element of Σ are called *discrete transitions* and the transitions that are labeled with a real positive number are called *continuous transitions*.

TPN syntax and semantics

Syntax of TPN

A *time Petri net* (TPN) $\mathcal{T}$ [51, 8] is a tuple $\langle P, T, W^-, W^+, M_0, (\alpha, \beta) \rangle$ where :

- $P = \{p_1, ..., p_m\}$ is a finite set of places;
- $T = \{t_1, ..., t_n\}$ is a finite set of transitions;
- $W^- \in (\mathbb{N}^P)^T$ is the backward incidence mapping;
- $W^+ \in (\mathbb{N}^P)^T$ is the forward incidence mapping;
- $M^0 \in \mathbb{N}^P$ is the initial marking;
- $\alpha \in (\mathbb{Q}_{\geq 0})^T$ and $\beta \in (\mathbb{Q}_{\geq 0} \cup \{\infty\})^T$ are the earliest and latest firing time mappings.

Semantics of TPN

The semantics of a TPN $\mathcal{T}$ can be represented with a TTS $S_{\mathcal{T}}$. Before defining the semantics of $\mathcal{T}$, we first introduce the following definitions. A *marking* is an element of $\mathbb{N}^P$. A *valuation* is a vector $v \in (\mathbb{R}_{\geq 0})^n$ such that each value v_i represents the elapsed time since the last time t_i was enabled or since the launching of the system if t_i was never enabled. The initial valuation $\overline{0} \in (\mathbb{R}_{\geq 0})^n$ is the valuation with $\overline{0}_i = 0$ for all $i \in \{1, ..., n\}$. A transition t is said to be *enabled* for a marking M if and only if $M \geq W^-(t)$. A transition t_k is said to be *newly enabled* after the firing of a transition t_i from a marking M if t_k is not enabled for the marking $M - W^-(t)$ and it is enabled for the marking $M' = M - W^-(t) + W^+(t)$. The function $\uparrow enabled : T \times \mathbb{N}^P \times T \mapsto \{true, false\}$ is defined by $\uparrow enabled(t_k, M, t_i) = true$ if t_k is newly enabled after the firing of t_i from M. Formally, for all $(t_k, M, t_i) \in T \times \mathbb{N}^P \times T$,we have $\uparrow enabled(t_k, M, t_i) = (M - W^-(t_i) + W^+(t_i) \geq W^-(t_k)) \wedge ((M - W^-(t_i) < W^-(t_k)) \vee (t_k = t_i))$.

The TTS $S_{\mathcal{T}} = \langle Q, q_0, T, \rightarrow \rangle$ associated to a TPN $\mathcal{T} = \langle P, T, W^-, W^+, M_0, (\alpha, \beta) \rangle$ is defined by:

- $Q = \mathbb{N}^P \times (\mathbb{R}_{\geq 0})^n$;
- $q_0 = (M_0, \overline{0})$;
- $\rightarrow \in Q \times (T \cup \mathbb{R}_{\geq 0}) \times Q$ is the transition relation defined by:
  - (discrete transitions) for all $t_i \in T$, we have:

$$(M, v) \xrightarrow{t_i} (M', v') \Leftrightarrow \begin{cases} M \geq W^-(t_i) \wedge M' = M - W^-(t_i) + W^+(t_i) \\ \alpha(t_i) \leq v_i \leq \beta(t_i) \\ v'_k = \begin{cases} 0 & \text{if } \uparrow enabled(t_k, M, t_i) \\ v_k & otherwise \end{cases} \end{cases}$$

  - (continuous transitions) for all $\delta \in \mathbb{R}_{\geq 0}$, we have:

$$(M, v) \xrightarrow{\delta} (M, v') \Leftrightarrow \begin{cases} v' = v + \delta \\ \forall k \in \{1, \ldots, n\}, (M \geq W^-(t_k) \Rightarrow v'_k \leq \beta(t_k)) \end{cases}$$

The last condition on continuous transitions ensures that the time that elapses in places cannot increase to a value which would disable transitions that were enabled by the marking.

When defining the semantics of TPN, three kinds of policies must be fixed:

- **The choice policy** concerning the next event to be fired. For our TPN models the choice is non deterministic.

- **The service policy** concerning the possibility fo simultaneous instances of a same event to occur. Here we adopt the *single server* policy.
- **The memory policy** concerning the updating of timing information when a discrete step occurs.The key issue in the semantics is to define when we reset the clock measuring the time since a transition was enabled. Here we will use the *enabling memory* policy.

1.2.2 The SPN, GSPN, and SWN formalisms

A Stochastic Petri Net [52] (SPN) is a tuple $S = (P,T,I,O,W,m_0)$, where P is the set of places, T is the set of transitions, $I,O : P \to T \to 2^{\mathbb{N}}$ define the input and output arcs with associated multiplicity, $W : T \to \mathbb{R}_{\geq 0}$ defines the rate of the exponential distributions associated to transitions, and $m_0 : P \to \mathbb{N}$ describes the initial marking.

It is well-known that the stochastic process underlying an SPN is a Continuous Time Markov Chain (CTMC) which is isomorphic to RG of the SPN built disregarding the timing aspects.

Generalized SPN (GSPN) [1] are an extension of SPN in which the set T is split into immediate and stochastic transitions. Immediate transitions fire in zero time, with priority over timed ones, and conflicts are resolved probabilistically.

As a consequence the RG includes also some *vanishing* states in which zero time elapses. We distinguish the RG from the Tangible RG (TRG) in which only tangible (non-vanishing) states are preserved.

The stochastic process associated to a GSPN is a semi-Markov process from which a CTMC can be produced considering only the tangible states.

Figure 1.7, in its right-most path, describes the solution process usually performed for the GSPN models: the RG is built first, and each arc is labeled with the name of the transition that causes that change of state; then the vanishing states are eliminated, thus creating the TRG, in which each arc is labeled with either

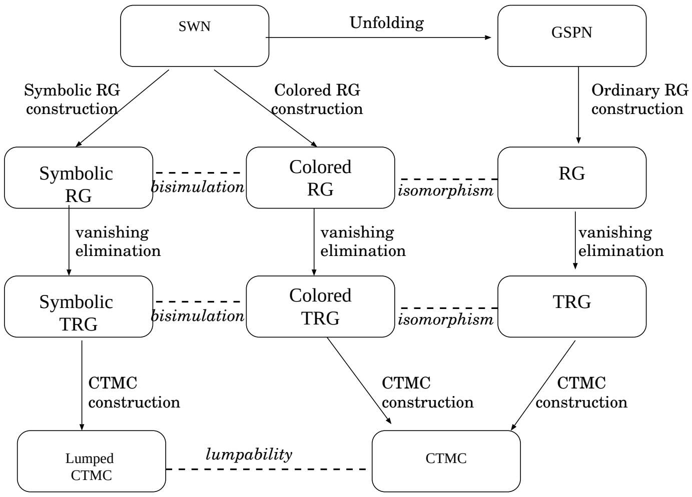


Figure 1.7: Solution approaches for SWN

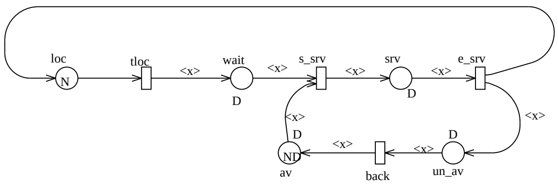


Figure 1.8: The SWN model of a simple computing system

the name of a timed transition or of a timed followed by a sequence of immediate transitions. The CTMC is then built using the TRG and the rates and probabilities associated to the transitions of the net.

Stochastic Well-formed Nets [15] are a colored extension of GSPN. The peculiarity of SWN is that the *color domain* of places is built as the Cartesian product of a limited number of *basic color classes*, and that functions on arcs are expressed as linear combination of a few basic functions (projection to select an element, sum function to select the whole color class).

We do not provide a formal definition (see [15]) of SWN in this context, but we recall the main points by using an example.

The net of Figure 1.8 shows an SWN model of a simple computing system:

there is a central server, modelled by place loc and transition tloc, at which there are initially N jobs.

Once the computation local to the central server terminates, a job chooses one of the peripherical devices, and a request for that device is put in place wait. Different devices are modelled as different colors of the color class D. Hence $D = \{d_1, d_2, \ldots, d_K\}$ indicates that there are K different devices. A device d can be in one of the following states: available (one token of color d in place av), being used by a job (one token of color d in place srv), and unavailable (one token of color d in place un_av). The choice of a device by a job is modelled by the single server transition tloc that, due to the free variable x associated to the arc out of tloc, puts in place wait a token of color d, randomly chosen with equal probability amongst all K colors of the color class D.

A function $\langle x \rangle$, such as the one on the arc from wait to s_srv, is called a projection function and evaluates to one of the elements of D: if in a marking there is a token in place loc then there are K *instances* of transition tloc enabled, one per each possible color assignment to x.

Transition s_srv can fire for an assignment to variable x of color d only if there is at least one token of color d in place wait and at least one token of color d in place av. The firing of s_srv for $x = d$ puts a token of the same color in srv. Observe that, once the device has provided the required service, transition e_srv fires, and puts an uncolored token (the job) back to the server place loc, and a colored d token (the device) into place un_av. Place un_av models some reset time of the device (a period in which the device is unavailable). We assume that all devices have the same speed and that each device can work on a single job at a time (the service policy of transition s_srv and e_srv is “single server per color”).

The other basic function of SWN is the constant function S: if we change the function on the arc out of tloc to $\langle S \rangle$, then each firing of tloc adds K tokens to place wait, one token per color in D, thus representing a fork of a job into K threads, one per device. If the function $\langle S \rangle$ is associated also to the arc from srv to e_srv then the transition can fire only if there is in srv at least one token per color: therefore

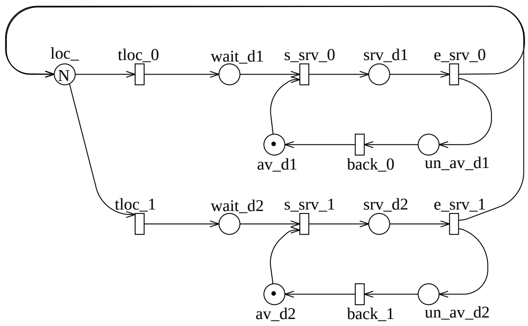


Figure 1.9: Unfolding of the SWN of Figure 1.8

it expresses a synchronization in the elaboration at the n peripherical devices.

From an SWN an equivalent GSPN can be generated, which has the same structure, replicated as many times as there are colors (in the example above there are K replicas of all colored places). The process of generating a GSPN equivalent to a given SWN is called *unfolding* [42]: each place is duplicated as many times as there are distinguished colored tokens assigned to that place, transitions are duplicated as many times as there are transition instances, and arcs are computed according to the functions associated to the arcs of the SWN. Figure 1.9 shows the GSPN model obtained through unfolding of the SWN model of Figure 1.8.

Figure 1.7, in its central path, describes the *colored* solution process of an SWN: the Colored Reachability Graph (CRG) is built first, followed by a colored TRG, from which the CTMC is computed. While a state in a GSPN is an assignment of tokens to places, in SWNs, as in all colored nets, a state is an assignment of colored tokens to places. Listed in the last three lines of Table 1.1 are three colored markings: state C_{1a} corresponds to a situation in which there is a job waiting for device $d1$, while a job is using device $d2$, and another one is using $d3$.

Figure 1.7 depicts also the relationship between an SWN and its unfolded GSPN: the CRG and the RG are isomorphic, and the same Markov chain is produced.

State	State Description	Z Cardinality
S_1	$\mathrm{M}(\mathsf{wait}) = 1 \cdot Z_{D,1}$, $\mathrm{M}(\mathsf{srv}) = 1 \cdot Z_{D,2}$	$\lvert Z_{D,1}\rvert = 1$, $\lvert Z_{D,2}\rvert = 2$
S_2	$\mathrm{M}(\mathsf{wait}) = 1 \cdot Z_{D,1}$, $\mathrm{M}(\mathsf{srv}) = 1 \cdot Z_{D,1} + 1 \cdot Z_{D,2}$	$\lvert Z_{D,1}\rvert = 1$, $\lvert Z_{D,2}\rvert = 1$
C_{1a}	$\mathrm{M}(\mathsf{wait}) = 1 \cdot d_1$, $\mathrm{M}(\mathsf{srv}) = 1 \cdot d_2 + 1 \cdot d_3$	n.a.
C_{1b}	$\mathrm{M}(\mathsf{wait}) = 1 \cdot d_2$, $\mathrm{M}(\mathsf{srv}) = 1 \cdot d_1 + 1 \cdot d_3$	n.a.
C_{1c}	$\mathrm{M}(\mathsf{wait}) = 1 \cdot d_3$, $\mathrm{M}(\mathsf{srv}) = 1 \cdot d_1 + 1 \cdot d_2$	n.a.

Table 1.1: Symbolic and colored states for the SWN of Figure 1.8

The real advantage of colored nets in general, and of SWN in particular, is the ability to exploit the symmetries described by the color. This exploitation is fully automatic for SWN. Figure 1.7, in its left-most path, describes the *symbolic* solution process of an SWN: the Symbolic Reachability Graph (SRG) is built first, followed by a symbolic TRG, from which the CTMC is computed. We do not enter in the details of the SRG construction [15] here: however, we recall that a state of the SRG (*symbolic marking*) is an equivalence class of colored markings. The equivalence classes are represented in terms of a partition of a color class into dynamic subclasses $Z_{ColorClass,index}$, characterized simply by the subclass cardinality.

Examples of symbolic markings for the central server model are given in the first two lines of Table 1.1. State S_1 is the equivalence class of all markings that have a job waiting for a device and two other jobs in service using the two other devices: indeed state S_1 corresponds to the three colored markings C_{1a}, C_{1b} and C_{1c}. State S_2 corresponds to all colored markings that have a job waiting for a device that is already in use by another job (the same dynamic subclass $Z_{D,1}$ is associated to places wait and srv), and a different device (one of the remaining two) in use by another job.

An interesting feature of SWN is that the equivalence classes are directly computed from the initial marking of the SWN, thanks to a definition of *symbolic transition firing*: a firing in which dynamic subclasses, instead of specific colors, are assigned to variables. Therefore in the SRG, out of state S_1 there is an arc in which x is associated to $Z_{D,1}$: the single arc represents the three arcs corresponding to the possible assignment to x of a color in D.

The SRG is a bisimulation of the CRG (although the bisimulation relation is never computed), while the CTMC built from the SRG is a lumped CTMC with respect to the CTMC generated from the CRG, as summarized by Figure 1.7. Observe that, for each symbolic marking, it is possible to build the list of corresponding colored markings.

Chapter 2

Model checking temporal logics

Electronic commerce, telephone switching networks, highway and air traffic control systems, medical instruments are examples of systems that, provided with a proper integration with the Internet and embedded systems (considering both hardware and software implementations), surely are improved with respect to the tasks they have been designed for. Clearly, the need for reliable hardware and software systems is critical, and even if failure is not life-threatening, the consequences can be economically devastating, so it is becoming more and more important to develop methods that increase our confidence in the correctness of such systems.

Taking into account the deeply heterogeneous purposes of the wide different application fields, we could reasoning about which kind of properties and about which idea of correctness we are interested in. Stated that a system is correct when *it does what it was designed for*, one of the basic axioms of the software and hardware engineering is that the concept of the "correctness of a system " has always to be referred to a specific set of requirements; therefore, we can not say *the system is correct*, but we will say *the system is correct with respect the designed specifications*, since it is not possible to verify the abstract view of the complete absence of errors that can not be derived from the unfulfillment of the specifications.

Given that, we can roughly distinguish among the following correctness meanings:

- *functional*: e.g., the system output is correct with respect to the purposes it is designed for (e.g., “the mainframe processes the jobs without errors”);
- *performance*: e.g., the system is able to execute a task within a temporal deadline (e.g., “the mainframe processes the jobs within 10 time units”);
- *dependability*: e.g., the system is able to react and to recover from internal or external failure (e.g., “the mainframe is able to recover from a deadlocked job”);
- *fairness*: e.g., the system is able to act, taking into account specific service policies (e.g., “the mainframe does not process indefinitely the same jobs”);

According to the given list of different items, a wide number of different property classes may be obtained from the combination of them; for example, the *performability* class is given when performance and dependability properties are quite undistinguishable.

In this chapter, we will introduce the *model checking* technique: an automatic approach for verifying finite state concurrent systems, having a number of advantages over traditional procedures that are based on simulation, testing and deductive reasoning. It is basically a procedure that checks, for every reachable state of a specific system, if a property of interest holds, or does not. The properties under investigation, which formalize the specification, are given in terms of some appropriate logics, usually provided with an explicit or an implicit notion of time [21].

This chapter is organized as follows. In Section 2.1, we provide a panoramic *excursus*, taken from [21] on the main model checking motivations, history, and application fields. In Section 2.2 and in Section 2.3 we illustrate how the model checking techniques was applied to two specific logics acting an important rule in the formal specification of real-time and performability-oriented systems.

2.1 The model checking verification technique

In this section we provide an overview about the conceptual basis of the model checking technique, specifically by delineating the framework in which the model checking may be contextualized (Section 2.1.1), by listing the most important model checking techniques, according to the different temporal logics it was applied for (Section 2.1.2), and by furnishing an informal description about the model checking process (Section 2.1.3). This section is adapted from [21].

2.1.1 Formal methods for hardware and software verification

The widest validation methods for hardware and software systems are *simulation*, *testing*, *deductive verification* and *model checking*.

Simulation and testing [54] both involve making experiments before deploying the system in the field, but while the former is performed on an *abstraction* or a *model* of the system, testing is performed on the actual product; in both cases, these methods are typically based on the coherence checking between the injected inputs and the obtained outputs, and they can be a cost-efficient way to find many errors, even if checking all of the possible interactions and potential pitfalls is quite impossible.

With *deductive verification* it is intended the use of axioms and proof rules to prove the correctness of systems [40]. The importance of such technique is widely recognized by computer scientists; however, deductive verification is a time-consuming process that can be performed only by experts who are educated in logical reasoning and have considerable experience. Consequently, the uses of deductive methods are rare, and are primarily applied to highly sensitive systems, such as security protocols, where enough resources need to be invested to guarantee their safe usage [21]. An important limitation of such approach is highlighted by the theory of computability: it shows that there cannot be an arbitrary algorithm that decides the termination of an arbitrary computer program; thus, most proof systems cannot be completely automated. Despite this, an advantage of deductive verification is that it can be used for reasoning about infinite state systems, and this can be automated to a limited extent. However, even if the property to be verified is true, no limit can be placed on the amount of time and memory that may be needed to find a proof.

Model checking is a technique for verifying finite state concurrent systems; one benefit of this restriction is that verification can be performed *automatically*. The procedure normally uses an exhaustive search of the state space of the system to determine if some specification is true or not. Given sufficient resources, the procedure always terminate with a **yes/no** answer. Although the restriction to finite state systems may seem be a major disadvantage, model checking is applicable to several very important classes of systems (for example, hardware controller or communication protocol are finite state systems); in addition, systems that are not finite state may be verified using model checking in combination with various abstraction and *induction* principles. Finally, in many cases errors can be found by restricting unbounded data structures to specific instances that are finite state. Because model checking can be performed automatically, it is preferable to deductive verification, whenever it can be applied; however, there will always be some critical applications in which theorem proving is necessary for complete verification.

2.1.2 Temporal logics and model checking

Temporal logics are useful for specifying concurrent systems, because they can describe the ordering of events in time without introducing time explicitly, and they are often classified according to whether time is assumed to have a linear or branching structure: in the first case, we consider only one possible sequence of possible events, while in the latter case, we analyze all possible futures (i.e., computations).

Several researchers have proposed using temporal logics for reasoning about computer programs; among all, Pnueli [61] was the first to use them for reasoning about concurrency: his approach involved proving properties of the program under investigation from a set of axioms that described the behavior of the individual statements in the program.

The introduction of temporal logic model checking algorithms by Clarke and Emerson [17] in the early 1980s allowed this type of reasoning be automated, because checking that a single model satisfies a formula is much easier that proving the validity of a formula for all models. The algorithm developed by Clarke and Emerson for the branching time Computational Tree logic (CTL) was polynomial in both the size of the model and in the length of its specification in temporal logic; they also showed how fairness properties could be handled without changing the complexity of the algorithm. This was an important step, because the correctness of many concurrent programs depends on some type of fairness assumption; for example, absence of starvation in a mutual exclusion algorithm may depend on the constraint that each process makes progress infinitely often. Later, Clarke, Emerson and Sistla [18] devised an improved algorithm that was linear in the product of the length of the formula and the size of the state transition graph.

Sistla and Clarke [67] analyzed the model checking problem for a variety of temporal logics and showed, in particular, that for the Linear Temporal logic (LTL) the problem was PSPACE-complete. Pnueli and Lichtenstein [47] reanalyzed the complexity of checking linear time formulae and discovered that although the complexity appears exponential in the length of the formula, it is linear in the size of the global state graph: based on this observation, they argued that the high complexity of linear time model checking might still be acceptable for short formulae.

CTL* is a very expressive logic that combines both branching time and linear time operators. The model checking problem for this logic was first considered by Clarke, Emerson and Sistla in [18], where it was shown to be PSPACE-complete; this result can be exploited to show that CTL* and LTL model checking are of the same algorithmic complexity in both size of the state graph and of the size of the formula. Thus, for purposes of model checking, there is no practical complexity advantage to restricting oneself to a linear time logic [29].

Computers are frequently used in critical applications where predictable response times are essential for correctness. Such systems are called real-time systems. The above cited temporal logics are not able to express the quantitative temporal specifications usually required to hold in real time systems, for example " event *p* will happen within at most *n* time units", despite of their expressive power, which let them to express qualitative temporal specifications, for example " event *p* will happen in the future". Real-time CTL (RTCTL) [30] and and Timed CTL (TCTL) [2] are two examples of logics for which model checking algorithms were introduced for the above cited goals; where the former is a simple extension of the CTL logic, providing a syntactical sugar for expressing bounds, and it is based on a discrete view of time (useful for describing synchronous systems), the latter was developed for specifying asynchronous systems, in which the notion of time is necessary continuous, and for which the consequent potentially non-finitness of the state space has to be handled.

An important class of logics is related to the capability to express properties connected to the performance and dependability evaluation activities, in the field of the stochastic model checking. Continuous Stochastic logic (CSL) [5, 6] and Continuous Stochastic Reward logic (CSRL) [35], are two examples of such approaches, letting to specify probabilistic specification (the former), even combined with purely performance requirements (the latter).

In the original implementation of the model checking algorithms, transition relations were represented explicitly by adjacency lists. For concurrent systems with small numbers of processes, the number of states was usually fairly small, and the approach was often quite practical: the EMC tool for CTL model checking, developed by Clarke, Emerson and Sistla [18] was able to check transition graphs with between 10^4 and 10^5 states at a rate about 100 states per second for typical formulae. In systems with many concurrent parts, however, the number of states was too large to handle. For facing with that problem, McMillian [49] realized that by using a symbolic representation for the state transition graph, much larger systems could be verified and implemented the SMV tool using the ordered binary decision diagrams; using the *symbolic model checking*, the original EMC tool could verify systems that had more than 10^{20} states [43]. Since then, various refinements of such approach let to manage up to more than 10^{120} states [28].

Verifying software causes some problem for model checking, because software tends to be less structured then hardware and, in addition, concurrent software is usually asynchronous. For these reasons, the *state space explosion phenomenon* is a particularly serious problem for software. Consequently, model checking has been used less frequently for software verification than for hardware verification. The most successful techniques for dealing with this problem are based on the *partial order reduction* [33], which let to reduce the state space by selecting only a subset of the way one can interleave independently executed transition (two transitions are called *independent* of each other when executing them in either order results in the same global state).

Although symbolic representation and the partial order reduction have greatly increased the size of the systems that can be verified, many realistic systems are still too large to be handled: so some other approaches was developed.

The first technique exploits the *modular structure* of complex circuits and protocols: if it is possible to show that the system satisfies each local property (i.e. a property involving only a small part of the whole system), and if the conjunction of the local properties implies the overall specification, then the complete system must satisfy the specification as well (see, for example, [34]).

The second technique involves the use of *abstraction*, usually applied by giving a mapping between the actual data values in the system and a small set of abstract data values; by extending the mapping to states and transitions , it is possible to produce an abstract version of the system under consideration, often much smaller then the original system, and usually much simpler to verify (see, for example, [20]).

Symmetry can also be used to reduce the state explosion problem (see, for example, [68]): having symmetry in a system implies the existence of nontrivial permutation group that preserves the state transition graph, and that can be exploited to define an equivalence relation on the state space of the system and to reduce the state space.

Induction involves the use of parameterization to reason about families of systems and it is called in this way, because an inductive argument is used to verify that an invariant process (a representative of a family of systems) is an appropriate representative (see, for example, [19]).

2.1.3 The model checking process

The *model checking problem* is easy to describe. Given a Kripke sructure $M = (S, L, R)$, that represents a finite state concurrent system (where S is the set of states, R the transition relation between the states and L is a function labeling the set of states with assertions that hold in such states) and a temporal logic formula f expressing some desired specification, find the set of all states in S that satisfy f:
$\{s \in S \mid M, s \models f\}$.

Normally, some states of the concurrent system are designated as initial states. The system satisfies the specification provided that all of the initial states are in the set. Applying model checking consists of several tasks, in the following synthesized.

- **Modeling**: it is performed by converting a design into a formalism accepted by a model checking tool. In some cases this is simply a compilation task; in other cases , due to memory and time limitations, it may require the use of abstractions to eliminate irrelevant details.
- **Specification**: useful to express the properties the system has to satisfy, it is usually given in some logical formalism, and it is common to use temporal logic, which can assert how the system evolves over time. An important issue in specification is *completeness*: model checking provides means for evaluating if a given model of the design satisfies a given specification, but it is impossible to determinate whether the given specification covers all the properties that the system should satisfy.

- **Verification**: ideally, the verification is completely automatic. However, in practice it often involves human assistance, for example for the analysis of the verification results. In case of negative result, the user is usually provided with an error trace, that can be used as a counterexample for the checked property and can help the designer in tracking down where the error occurred. In this case, analyzing the error trace may require a modification to the system and the reapplication of the model checking algorithm. An error trace can also result from incorrect modeling of the system or from an incorrect specification (often called a *false negative*). The error trace can also be useful for identifying and fixing these two problems. A final possibility is that the verification task will fail to terminate normally, due to the size of the model, which is too large to fit into the computer memory. In this case it may be necessary to redo the verification after changing some of the parameters of the model checker or by adjusting the model (e.g. by using additional abstractions).

2.2 Model checking TCTL formulae

The TCTL logic was defined to specify the real-time requirements that a user-specified system model by mean of the TA formalism must exhibit. In this section we provide an informal introduction to TA (Section 2.2.1), and to the model checking problem of TCTL formulae (Section 2.2.2).

2.2.1 The TA formalism

TA [4] are automata extended with clocks, which are real-valued variables, which increase at the same rate as real time. Let X be a set of *clocks*, and $\Phi(X)$ be the set of the *clock constraints* over X, which are defined by the following grammar: $\varphi := x \leq c | x \geq c | x < c | x > c | \varphi_1 \wedge \varphi_2$, where $x \in X$, $c \in \mathbb{Q}_{\geq 0}$. A *timed automaton* A is a tuple $\langle L, L^0, \Sigma, X, I, E \rangle$ where: L is a finite set of *locations*; $L^0 \subseteq L$ is a set of the *initial locations*; I is a (total) function $L \mapsto \Phi(X)$ that associates to each location an *invariant condition* (i.e. a clock constraint); $E \subseteq L \times \Sigma \times \Phi(X) \times 2^X \times 2^{X^2} \times L$ represents the set of the *switches*. The switch $(s, \sigma, \varphi, \lambda, \rho, s') \in E$ represents a switch from s to s' on the event σ; φ is the clock constraint (or guard) associated to this switch, $\lambda \subseteq X$ gives the set of clocks that are reset when the switch is executed, and $\rho \subseteq X^2$ is a clock renaming function. In [4] the semantics of TA is described by means of a TTS, and the problems and the possible solutions regarding the infinite number of states and symbols of such a TTS are also illustrated. This leads to the use of abstraction methods, for example the *region graph* and the *zone graph*. The semantics of TA is standard (in particular, see [10, 48] for the semantics of the variant of TA with clock renaming), then we omit it here.

An example of TA

As an example of TA consider the simple system, as shown in Figures 2.1, depicting a trivial communication sub-system. The considered TA has two locations, one of them is labeled *sending*, to indicate a message is going to be sent by the sub-system, and it is the initial location, the other is labeled *receiving*, for describing it is waiting for an acknowledgment to the previous message. It is also defined a clock named x. The *sending* location has got the invariant $0 \leq x \leq 10$, suggesting that it is not possible to stay in that state for more than 10 time units. There is an arc from *sending* to *receiving*, named *send* and provided with the clock constraint $8 \leq x \leq 10$, that establishes the action may be performed not before 8 and not after 10 time units. In addition, the clock x is reset, because it will be used to count the time elapsing in the arrival location. The *receiving* location has got the invariant $0 \leq x \leq 15$, suggesting that it is not possible to stay in that state for more than 15 time units. There is an arc from *receiving* to *sending*, named *receive* and provided with the clock constraint $12 \leq x \leq 15$, that establishes the action may be performed not before 12 and not after 15 time units. In addition, the clock x is reset, for the just before cited motivation.

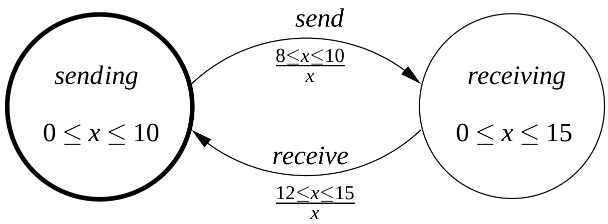


Figure 2.1: An example of TA, modeling a simple communicating subsystem

2.2.2 The TCTL logic

The TCTL syntax

Let A be a TA, AP the set of atomic propositions, and D the non-empty set of clocks, disjoined by the clocks in A, i.e., $C \cap D = \varnothing$. $z \in D$ is often called *formula clock*. For $\mathsf{a} \in AP$, $z \in D$, and $\alpha \in \Psi(C \cup D)$, the syntax of TCTL [2] is defined as follows:

$$\psi ::= \mathsf{a} \mid \alpha \mid \neg\psi \mid \psi \vee \psi \mid z\mathbf{in}\psi \mid \mathbf{E}[\psi\mathbf{U}\psi] \mid \mathbf{A}[\psi\mathbf{U}\psi].$$

The boolean operators, $true$, $false$, $\wedge$, $\Rightarrow$ e $\Leftrightarrow$ may be easily derived, as the operators **EF** (*eventually in the future)*, **EG** *(*eventually globally), **AF** (*always in the future) e* **AG** *(*always globally). The z in $z\mathbf{in}\psi$ is named *freeze identifier*, and it limits the formula clock scope in ψ; informally speaking, $z\mathbf{in}\psi$ holds in the state s if ψ holds in s, when z starts in s with the value 0; it is usually used toghether with the **U** (*Until*) operator, for specifying temporal requirements.

For example, a property like ‘starting from the current state, and considering all possible computations (paths), the property ϕ holds continuously until, within 7 time units, ψ will hold’, may be expressed in this way:

$$z\mathbf{inA}[(\phi \wedge z \leq 7)\mathbf{U}\psi].$$

The TCTL semantics

The semantics of TCTL is defined by a satisfaction relation (written as $\models$), which relates the transition system M, a state s (i.e., a *location* l plus a clock evaluation v, on the automata clocks), a clock evaluation w, on the clocks occurring in

ψ, and a formula ψ. It may be written $(M,(s,w),\psi) \in \models$, or $M,(s,w) \models \psi$, if and only if ψ holds in the state s of the M, being considered a formula clock evaluation w. The state $s = (l,v)$ satisfies ψ, if the 'extended' state (s,w) satisfies ψ, where w is a clock evaluation, so that $w(z) = 0$ for all formula clock z. The formal semantics of TCTL can be found in [2]. Here we give an informal description. A state s satisfies the atomic proposition a if s is labelled with a, the operators $\wedge$ and $\neg$ have the usual interpretation. The clock constraint α holds in (s,w), if the clock value in v, the evaluation in s and w satisfy α. The formula $z\mathbf{in}\psi$ holds in (s,w), if ψ holds in (s,w'), where w' is obtained by w resetting z. The formula $\mathbf{E}[\phi\mathbf{U}\psi]$ holds in (s,w), if exists a path σ time-diverging, which starts from s, and that satisfies ψ in some point in the future, satisfying $\psi \wedge \phi$ in all previous positions. The formula $\mathbf{A}[\phi\mathbf{U}\psi]$ holds in (s,w), if all the paths σ time-diverging start from s, satisfy ψ in some point, and satisfy $\psi \vee \phi$ in all previous positions.

The TCTL model checking problem

Model checking a TA against a TCTL formula Φ consists of computing the set of states $Sat(\Phi)$ such that $s \in Sat(\Phi)$ if and only if Φ is true in s. The set $Sat(\Phi)$ is constructed by computing recursively the sets of states satisfying the sub-formulae of Φ.

An examples of property specification using TCTL

We take the example in Figures 2.1. We could ask the following class of properties on it:

1. *Reactivity*: specifies a maximal delay between an event occurrence and a reaction. For example: 'every transmission of a message is followed by an acknowledgment within 5 time units'. Formally,

$$\mathbf{A}G[receiving \Rightarrow \mathbf{AF}^{<5} sending]$$

2. *Punctuality*: specifies an exact delay between two events. For example: 'it exists a computation so that the delay between waiting for an acknowledgment and sending of a message is exactly 11 time units'. Formally,

$$\mathbf{EG}[receiving \Rightarrow \mathbf{AF}^{=11} sending]$$

3. *Periodicity*: specifies periodical occurences. For example: ‘The system is waiting for an acknowledgment every 25 time units’. Formally,

$$\mathbf{AG}[\mathbf{AF}^{<25} receiving]$$

4. *Minimal delay*: specifies a maximal delay between two events. For example: “After a message is sent, the system is sending again a message not before 15 time units”. Formally,

$$\mathbf{AG}[sending \Rightarrow \neg sending \mathbf{U}^{\geq 15} sending]$$

5. *Interval delay*: specifies that, since an event occurs, the subsequent event has to occur in a specific timing interval. For example: ‘After a message is sent, the system is sending again a message between 15 and 30 time units’. Formally,

$$\mathbf{AG}[sending \Rightarrow \neg sending \mathbf{U}^{\geq 15} sending \wedge \neg sending \mathbf{U}^{\leq 30} sending]$$

2.3 Model checking CSL formulae

The CSL logic was defined to specify properties that are typical in those systems characterized by a stochastic evolution of the dynamics; in fact it is interpreted over models specified by mean of CTMC. In this section we provide a brief recall of the CTMC formalism (Section 2.3.1), and to the model checking problem of CSL formulae (Section 2.3.2).

2.3.1 The CTMC formalism

Consider a CTMC as an ordinary finite transition system (Kripke structure) where the edges are equipped with probabilistic timing information. Let AP be a fixed, finite set of atomic propositions. A (labeled) CTMC M is a tuple $M = (S, \mathbf{R}, L)$ with S as a finite set of states, $R : S \times S \to \mathbb{R}_{\geq 0}$ as the rate matrix, and $L : S \to 2^{AP}$

as the *labeling function*. Intuitively, function L assigns to each state $s \in S$ the set $L(s)$ of atomic propositions $a \in AP$ that are valid in s. In the traditional interpretation, at the end of a sojourn in state s, the system will move to a different state. Instead, we consider the definition in according to [6], in which self-loops at state s are possible and are modeled by having $\mathbf{R}(s,s) > 0$. We thus allow the system to occupy the same state before and after taking a transition. The inclusion of self-loops neither alters the transient nor the steady-state behavior of the CTMC, but allows the usual interpretation of linear-time temporal operators like *next step* and *until*, and it will be exploited when we address the semantics of the logic CSL in 2.3.2. A state s is called *absorbing* if and only if $\mathbf{R}(s,s') = 0$ for all states s'. Intuitively, $\mathbf{R}(s,s') > 0$ if and only if there is a transition from s to s'. Furthermore, $1 - e^{-\mathbf{R}(s,s')\cdot t}$ is the probability that the transition $s \rightarrow s'$ can be triggered within t time units. Thus, the delay of transition $s \rightarrow s'$ is governed by the exponential distribution with rate $\mathbf{R}(s,s')$. If $\mathbf{R}(s,s') > 0$ for more than one state s', a competition between the transitions originating in s exists, known as the *race condition*. The probability to move from a non-absorbing state s to a particular state s' within t time units, i.e., the transition $s \rightarrow s'$ wins the race, is given by: $\mathbf{P}(s,s',t) = \frac{\mathbf{R}(s,s')}{E(s)} \cdot (1 - e^{-E(s)\cdot t})$, where $E(s) = \sum_{s' \in S} \mathbf{R}(s,s')$ denotes the *total rate* at which any transition outgoing from state s is taken. More precisely, $E(s)$ specifies that the probability of taking a transition outgoing from state s within t time units is $1 - e^{-E(s)\cdot t}$, due to the fact that the minimum of two exponentially distributed random variables is an exponentially distributed random variable with as rate the sum of their rates. Consequently, the probability of moving from a non-absorbing state s to s' by a single transition, denoted $\mathbf{P}(s,s')$, is determined by the probability that the delay of going from s to s' finishes before the delays of other outgoing edges from s; formally, $\mathbf{P}(s,s') = \mathbf{R}(s,s')/E(s)$. For an absorbing state s, the total rate $E(s)$ is 0. In that case, we have $\mathbf{P}(s,s') = 0$ for any state s'. The matrix $\mathbf{P}$ is usually known as the *transition matrix* of the embedded discrete-time Markov chain of M (except that usually $P(s,s) = 1$ for absorbing s). An initial distribution on M, is a function $\alpha : S \rightarrow [0,1]$, such that $\sum_{s \in S} \alpha(s) = 1$.

For a CTMC, two major types of state probabilities are distinguished: *steady-*

state probabilities where the system is considered "on the long run", i.e., when an equilibrium has been reached, and *transient-state* probabilities where the system is considered at a given time instant t.

An example of a CTMC model

We take the example illustrated in [6]. Consider the *triple modular redundant system*, consisting of three processors and a single (majority) voter. We model this system as a CTMC where state $s_{i,j}$ models that i ($0 \leq i \leq 3$) processors and j ($0 \leq j \leq 1$) voters are operational. As atomic propositions, we use $AP = \{up_i | 0 \leq i < 4\} \cup \{down\}$. The processors generate results and the voter decides upon the correct value by taking a majority vote. Initially, all components are functioning correctly, i.e., $\alpha = \alpha^1_{s_{3,1}}$. The failure rate of a single processor is λ and of the voter ν failures per hour. The expected repair time of a processor is $1/\mu$ and of the voter $1/\delta$ hours. It is assumed that one component can be repaired at a time. The system is operational if at least two processors and the voter are functioning correctly. If the voter fails, the entire system is assumed to have failed and, after a repair (with rate δ), the system is assumed to start as good as new. The details of the CTMC modeling this system are shown in Figures 2.2 (with a clockwise ordering of states for the matrix/vector-representation, starting with $s_{3,1}$). States are represented by circles and there is an edge between state s and state s' if and only if $\mathbf{R}(s,s') > 0$. The labeling is defined by $L(s_{i,1}) = \{up_i\}$, for $0 \leq i < 4$, by $L(s_{0,0}) = \{down\}$, and it is indicated, in the depicted example, near the states. For the transition probabilities, we have, e.g., $\mathbf{P}(s_{2,1}, s_{3,1}) = \mu/(2\lambda + \mu + \nu)$ and $\mathbf{P}(s_{0,1}, s_{0,0}) = \nu/(\mu + \nu)$.

2.3.2 The CSL logic

The CSL syntax

Let *AP* be a set of atomic propositions, and let $\mathbb{R}_{\geq 0}$ be the set of non-negative real numbers. The syntax of CSL [5, 6] is defined as follows:

$$\Phi ::= \mathsf{a} \mid \Phi \wedge \Phi \mid \neg\Phi \mid \mathcal{P}_{\bowtie\lambda}(X^I \Phi) \mid \mathcal{P}_{\bowtie\lambda}(\Phi U^I \Phi) \mid \mathcal{S}_{\bowtie\lambda}(\Phi)$$

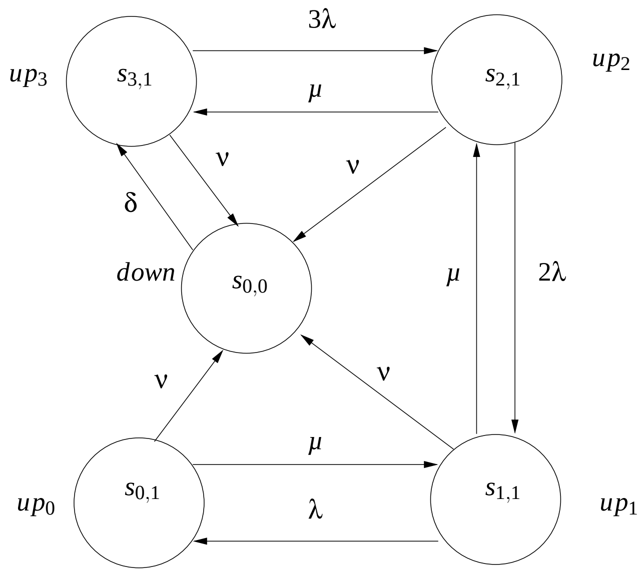


Figure 2.2: An example of CTMC, modeling a triple modular redundant system, taken from [6]

where $\mathsf{a} \in AP$ is an atomic proposition, $I \subseteq \mathbb{R}_{\geq 0}$ is a nonempty interval, interpreted as a time interval, $\bowtie \in \{<, \leq, \geq, >\}$ is a comparison operator, and $\lambda \in [0,1]$ is interpreted as a probability.

The CSL semantics

Formulae of CSL are evaluated over continuous-time Markov chains (CTMC s), each state of which is labeled with the subset of atomic propositions *AP* that hold true in that state.

The logic CSL can be divided into *state formulae*, which are true or false in a state, and *path formulae*, which are interpreted as being true or false for a path of the system (where a path is a sequence of transitions which represents a computation of the CTMC). The interpretation of the state formulae is as follows: a state s satisfies the atomic proposition a if s is labelled with a, the operators $\wedge$ and $\neg$ have the usual interpretation, while $\mathcal{S}_{\bowtie\lambda}(\Phi)$ is true in s if, assuming s as initial state, the sum of the steady state probabilities of the states that satisfy Φ is $\bowtie \lambda$. For a path formula $\varphi \in \{X^I\Phi, \Phi_1 U^I \Phi_2\}$, the state formula $\mathcal{P}_{\bowtie\lambda}(\varphi)$ is true in s if the probability of the paths leaving s which satisfy φ is $\bowtie \lambda$. We say that

$\mathcal{P}_{\bowtie\lambda}(\varphi)$ is a *probabilistic formula*, whereas $\mathcal{S}_{\bowtie\lambda}(\Phi)$ is a *steady-state formula*.

The interpretation of the path formulae $X^I\Phi$ and $\Phi_1 U^I \Phi_2$ is as follows. The Next formula $X^I\Phi$ is true for a path if the state reached after the first transition along the path satisfies Φ, and the duration of this transition lies in the interval I. The Until formula $\Phi_1 U^I \Phi_2$ is true along a path if Φ_2 is true at some state along the path, possibly also the initial state of the path under consideration, the time elapsed before reaching this state lies in I, and Φ_1 is true along the path until that state. The formal semantics of CSL can be found in [6]. The usual abbreviations for propositional and temporal logic will be used throughout the paper; for example, $\Diamond^I\Phi \equiv \mathtt{true} U^I \Phi$, $\mathcal{P}_{\geq\lambda}(\Box^I\Phi) \equiv \mathcal{P}_{<1-\lambda}(\Diamond^I \neg\Phi)$.

The CSL model checking problem

Model checking a CTMC against a CSL formula Φ consists of computing the set of states $Sat(\Phi)$ such that $s \in Sat(\Phi)$ if and only if Φ is true in s. The set $Sat(\Phi)$ is constructed by computing recursively the sets of states satisfying the sub-formulae of Φ. When probability bounds are present in the CSL formula ($\mathcal{P}_{\bowtie\lambda}$ or $\mathcal{S}_{\bowtie\lambda}$), the model-checking algorithm requires the computation of transient or steady-state probabilities of the original CTMC as well as, depending on the formula, a number of additional CTMC s built through manipulation of the original one: see [6] for more details.

An example of property specification using CSL

As an example of a CSL property specification of a CTMC model, we take again the example illustrated in [6]. Consider the *triple redundant system*, as in Figure 2.2, provided with a set of atomic propositions as defined by its own labeling function. We could check the following requirements, considering state $s_{3,1}$ as initial state:

- *Steady-state availability*, e.g.:

$$\mathcal{S}_{\geq 0.99}(up_2 \vee up_3)$$

is valid in $s_{3,1}$, if in the long run, for at least 99 percent of time, at least two processors are operational (starting in $s_{3,1}$).

- *Instantaneous availability at time t*, e.g.:

$$\mathcal{P}_{\geq 0.99}(\diamond^{t,t}(up_2 \vee up_3))$$

is valid in $s_{3,1}$, if the probability to have at least two processors running at time t exceeds 0.99 (starting in $s_{3,1}$).

- *Conditional interval availability*, e.g.:

$$\mathcal{P}_{\leq 0.01}((up_2 \vee up_3)U^{[0,10]}down)$$

is valid in $s_{3,1}$, if the probability of the system being *down* within 10 time units, after having continuously operated with at least two processors is at most 0.01 (starting in state $s_{3,1}$).

- *Steady-state interval availability*, e.g.:

$$\mathcal{S}_{\geq 0.9}(\mathcal{P}_{\geq 0.8}\Box^{[0,10]}\neg down)$$

is valid in those states guaranteeing that, in equilibrium with probability at least 0.9, the probability that the system will not go down within 10 time units is at least 0.8 (starting in state $s_{3,1}$).

- *Conditional time-bounded steady-state availability*, e.g.:

$$\mathcal{P}_{\geq 0.5}((\neg down)U^{[10,20]}\mathcal{S}_{\geq 0.8}(up_2 \vee up_3))$$

is valid in those states guaranteeing that, with probability at least 0.5, will reach a state s between 10 and 20 time units, which assures the system to be operational with at least two processors when the system is in equilibrium (starting in state $s_{3,1}$).

Part III

Model Checking Petri Nets

Chapter 3

TCTL model checking of TPN models

3.1 Introduction

Time Petri Nets (TPN) [51] and Timed Automata (TA) [4] are widely used formalisms to model and analyze timed systems. TPN and TA are dense-time formalisms, which implies that their underlying state-space is infinite, and therefore verification techniques which enumerate exhaustively the state-space cannot be applied. In general, this difficulty is addressed by applying symbolic methods or by partitioning the infinite state-space. With regard to TA, the well-known region graph [4] or zone-based graph [3] techniques are two such methods, the latter of which forms the basis of the techniques implemented in tools such as UPPAAL [26] and KRONOS [70]. With regard to TPN, in [8, 50] an approach based on the so-called state class graph (SCG) construction, is presented. In the SCG the nodes are sets of states, represented by a pair (marking, firing domain), where the firing domain represents the set of times at which a transition can be fired. The SCG construction allows the verification of untimed reachability and LTL properties [8, 50], while variants of this method allow the verification of CTL, and a subset of TCTL [2] properties [9, 72].

A different approach to allow TCTL model checking of TPN is to produce from a TPN a timed bisimilar TA which maintains TCTL properties, and verifying it using some well-known tools for model checking (for example, the above cited UPPAAL and KRONOS). In literature there are two different techniques derived from it: the first is based on the Petri net (PN) structure [12], and is generally characterized by a potentially high number of clocks in the produced TA; the second is based on exploration of the timed state space, for example in [48], in which a method based on an extended version of the SCG is used to compute the so-called state class timed automaton (SCTA), and in [31], where zone-based timed reachability analysis (see [3]) allows the construction of the so-called marking timed automaton, that in the following will be named zone-based marking timed automaton (ZBMTA). The ZBMTA always has a less or equal number of locations and edges than the SCTA, while the latter is optimal in the number of clocks with respect to the former. Finally, it should be noticed that, in [31, 48], the reachability techniques for the generation of a TA is generally employed again subsequently to analyze the produced TA; this fact could increase the verification time of the TPN under investigation.

In this chapter we present a different technique (presented in [24]) for the translation of a TPN into a (strong) timed bisimilar TA, by using the reachability graph of the underlying *untimed* Petri Net to build what we have called the *marking class timed automaton* (MCTA). We will show that the SCTA, obtained by applying [48], and the MCTA, obtained by applying our approach, are incomparable in the number of locations and edges, while the MCTA produces a greater or equal number of locations and edges with respect to the ZBMTA approach, obtained by applying [31]); finally, the number of clocks may be equal to those of the SCTA, and less or equal to those of the ZBMTA. From these considerations it may be deduced that our approach represents a competitive choice for a number of classes of systems, especially when a trade-off is needed between the number of the produced locations and clocks; we will present experimental evidences to show this. The main disadvantage of our method is the requirement of boundedness of the underlying untimed PN, while [31, 48] are less restrictive, needing only TPN boundedness. In order to address this problem, we give some suggestions to partially bound specific PN subnets of the TPN under investigation. In addition, because our method may explore some paths in the untimed Petri net which are unreachable in the TPN, resulting in a greater number of locations than necessary, we consider an adjustment to the MCTA construction algorithm which, for some TPN, can alleviate this problem.

This chapter is organized as follows: Section 3.2 explains our approach to verify TPN by translation to TA using the reachability graph of the untimed Petri net. Our method is also compared with the above cited SCTA and ZBMTA approaches. Section 3.3 presents some optimization techniques: a simple method to partially resolve the above cited unreachable path problem, a variant for minimizing the location cardinality of the produced TA, and some ideas to address the boundedness requirements of our approach. Section 3.4 describes how we have interfaced the new tool, GREATSPN2TA with the KRONOS model checker, in order to perform verification, and reports some testing results, obtained on a set of case studies, also comparing them against the results of the tool ROMEO [65], which implements the SCTA and ZBMTA approaches. Section 4.5 concludes the chapter.

3.2 From TPN to TA

Inspired by [31, 48], here we will show our approach (presented in [24]) for translating a TPN model into a TA, that we term marking class timed automaton (MCTA), in order to subsequently perform analysis on the TA. Section 3.2.1 is devoted to this technique, also providing a proof that the timed transition system (TTS) of the TPN and of the TA are timed bisimilar, while in Section 3.2.2 our approach will be compared with the similar ones, based on the SCTA and the ZBMTA, as illustrated in [48] and [31] respectively.

3.2.1 MCTA **of a** TPN

In this section we present the MCTA construction, where the constructed TA has an equivalent (timed bisimilar) behavior to that of a TPN. We will consider the TPN $\mathcal{T} = \langle P, T, W^-, W^+, M^0, (\alpha, \beta)\rangle$. We will "untime" the TPN $\mathcal{T}$ in order to obtain a Place/Transition PN $\mathcal{P} = \langle P, T, W^-, W^+, M^0\rangle$. We denote by $\mathcal{R}_{un}(M^0) \subseteq \mathbb{N}^P$ the reachability set of $\mathcal{P}$ (the set of markings that $\mathcal{P}$ can reach from its initial marking M^0). When bounded (i.e. $(\exists k \in \mathbb{N})(\forall p \in P)(\forall M \in \mathcal{R}_{un}(M^0))(M(p) \leq k)$), the behavior of this PN can be represented by the *reachability graph*, which is an untimed finite-state transition system $S_{\mathcal{T}} = \langle Q, q_0, T, \rightarrow\rangle$ where $Q = \mathcal{R}_{un}(M^0)$, $q_0 = M^0$, and the transition relation $\rightarrow$ is defined by, for all $M, M' \in \mathcal{R}_{un}(M^0)$, for all $t \in T$:

$$M \xrightarrow{t} M' \Leftrightarrow \begin{cases} M \geq W^-(t) \text{ and} \\ M' = M + W^+(t) - W^-(t)\ . \end{cases}$$

From this reachability graph, we will build a TA which will have the same behavior as the TPN up to timed bisimulation.

The MCG **construction.**

We now present the algorithm which builds the *marking class graph* (MCG) $\Gamma(\mathcal{T})$ of the TPN $\mathcal{T}$, which is a transition system $\Gamma(\mathcal{T}) = \langle MC, Mc_0, T, \rightarrow_{mc}\rangle$. The states MC of $\Gamma(\mathcal{T})$ are called *marking classes*. Each marking class is a triple of the form $\langle M, \chi, trans\rangle$, comprising a marking M of $\mathcal{T}$, a set χ of clocks, and a function $trans : \chi \rightarrow 2^T$ associating a set of transitions to each clock in χ. The initial marking class $Mc_0 = \langle M^0, \{x_0\}, trans_0\rangle$ is such that M^0 is the initial marking of $\mathcal{T}$, the set of clocks χ_0 of Mc_0 is composed of a single clock x_0, and $trans_0$ is defined by $trans_0(x_0) = \{t \in T \mid t \text{ is enabled for } M^0\}$. To build the graph, we also need the notion of clock similarity (adapted from [48]), in order to group certain marking classes together.

Definition Two marking classes $C = \langle M, \chi, trans\rangle$ and $C' = \langle M', \chi', trans'\rangle$ are *clock-similar*, denoted $C \approx C'$, if and only if they have the same markings, the same number of clocks and their clocks are mapped to the same elements, written formally as:

$$C \approx C' \Leftrightarrow \begin{cases} M = M' \text{ and} \\ |\chi| = |\chi'| \text{ and} \\ \forall x \in \chi, \exists x' \in \chi', trans(x) = trans(x') \ . \end{cases}$$

To build the MCG we use a breadth-first graph generation algorithm, shown in Algorithm 1. It begins by computing the children of the initial marking class, then it proceeds by computing progressively the children of each computed marking class. The algorithm terminates when it cannot compute new marking classes anymore. In line 1.3, the first marking class of the queue *New* is taken (i.e. $C := \langle M_C, \chi_C, trans_C \rangle$), and the computation of the marking class $C' = \langle M_{C'}, \chi_{C'}, trans_{C'} \rangle$, a child of C, is performed from line 1.4 to line 1.20. Line 1.5 regards the $M_{C'}$ portion, while $\chi_{C'}$ and $trans_{C'}$ are computed from line 1.6 to line 1.14. Lines 1.15 to 1.19 are devoted to check if the newly computed child C' was already reached in a previous iteration, modulo a renaming application. We remark that the construction of this graph can be done by following the different paths in the reachability graph of the underlying PN adding a clock set χ' and a relation $trans'$, and possibly "unlooping" some loops of the reachability graph when a marking is reached many times with associated marking classes which are not clock-similar.

The MCTA **Construction.**

From the MCG defined previously, it is possible to build a TA $A(\mathcal{T})$ which has the same behavior as the TPN $\mathcal{T}$, as we will show in the next section. Let $\mathcal{T} = \langle P, T, W^-, W^+, M^0, (\alpha, \beta) \rangle$ be a TPN and $\Gamma(\mathcal{T}) = \langle MC, Mc_0, T, \rightarrow_{mc} \rangle$ its associated marking class graph. The *marking class timed automaton* (MCTA) $A(\mathcal{T})$ associated to $\mathcal{T}$ is the TA $\langle L, L^0, \Sigma, X, I, E \rangle$ defined by:

- $L = MC$ is the set of the marking classes ;
- $L^0 = \{Mc_0\}$, where Mc_0 is the initial marking class ($Mc_0 = \langle M^0, \{x_0\}, trans_0 \rangle$);

input : The initial marking class Mc_0 of a TPN $\mathcal{T}$

output: The MCG of $\mathcal{T}$

$MCG := \emptyset$; $New := Mc_0$;

while *New is NOT empty* **do**

  $C :=$ `remove`(New);

  for *all transitions t enabled for* M_C **do**

    $M_{C'} := M_C + W^+(t) - W^-(t)$;

    For each clock $x \in \chi_C$, remove from $trans_C(x)$ all the transitions t_k such that t_k is enabled in M_C and is not in $M_C - W^-(t)$, to obtain a relation $trans'$;

    The clocks whose image by $trans'$ is empty are removed from χ_C, to obtain a set of clocks χ';

    for *all transitions* t_k *which verify* $\uparrow enabled(t_k, M_C, t) = True$ **do**

      if *a clock x has already been created for the computation of* C' **then**

        t_k is added to $trans'(x)$;

      else

        a new clock x_n is created; n is the smallest available index among the clocks of χ' and $trans'(x_n) = t_k$;

      end

    end

    if *there is a marking class* C'' *in MCG such that* $C' \approx C''$ **then**

      $MCG := MCG \cup \{C \xrightarrow{t}_{mc} C''\}$;

    else

      $MCG := MCG \cup \{C \xrightarrow{t}_{mc} C'\}$ and `add`(New, C');

    end

  end

end

Algorithm 1: MCG construction

- $X = \bigcup_{\langle M,\chi,trans\rangle \in MC} \chi$;
- $\Sigma = T$;
- E is the set of switches defined by:

$$\forall C_i = \langle M_i, \chi_i, trans_i\rangle \in MC, \forall C_j = \langle M_j, \chi_j, trans_j\rangle \in MC$$
$$\exists C_i \xrightarrow{t_i}_{mc} C_j \Leftrightarrow \exists (l_i, a, \phi, \lambda, \rho, l_j) \in E \text{ such that}$$
$$\begin{cases} l_i = C_i, l_j = C_j, a = t_i, \\ \phi = (trans_i^{-1}(t_i,) \geq \alpha(t_i)), \lambda = \{trans_j^{-1}(t_k) | \uparrow enabled(t_k, M_i, t_i) = True\}, \\ \forall x \in \chi_i, \forall x' \in \chi_j, \text{ such that } trans_j(x') \subseteq trans_i(x), x' \notin \lambda, \rho(x') = x \end{cases}$$

- $\forall C_i = \langle M_i, \chi_i, trans_i\rangle \in MC$, $I(C_i) = \bigwedge_{x \in \chi_i, t \in trans_i(x)} (x \leq \beta(t))$.

Remark In order to build the MCTA of a TPN, the number of marking classes has to be bounded, otherwise the construction of the MCG will not terminate. Note that a TPN has a bounded number of marking classes if and only if the underlying PN (i.e. the PN obtained by untiming the TPN) is bounded. We recall that in contrast to the case of the boundedness of TPN [8], the boundedness of a PN is decidable. In Section 3.3.3 we will introduce an empirical approach that allows, by an iterative technique, to build the MCTA for those TPN that have an unbounded RG but a bounded set of reachable markings when time is taken into consideration.

Bisimulation.

In this section, we define a binary relation between the states of the TPN $\mathcal{T}$ and the states of its associated MCTA, and we will prove that this relation is a timed bisimulation. Our results are analogous to those of in the context of the SCTA [48] and the ZBMTA [31].

First, we recall the definition of timed bisimulation. Let $S_1 = \langle Q_1, Q_1^0, \Sigma_1, \rightarrow_1 \rangle$ and $S_2 = \langle Q_2, Q_2^0, \Sigma_2, \rightarrow_2\rangle$ be two TTS. Let $\approx \subseteq Q_1 \times Q_2$ be an equivalence relation on Q_1 and Q_2. The equivalence relation $\approx$ is a *timed bisimulation* if and only if, for all $a \in \Sigma \cup \mathbb{R}_{\geq 0}$:

- if $s_1 \approx s_2$ and $s_1 \xrightarrow{a} s'_1$ then there exists $s_2 \xrightarrow{a} s'_2$ such that $s'_1 \approx s'_2$;
- if $s_1 \approx s_2$ and $s_2 \xrightarrow{a} s'_2$ then there exists $s_1 \xrightarrow{a} s'_1$ such that $s'_1 \approx s'_2$.

Let $\mathcal{T} = \langle P, T, W^-, W^+, M^0, (\alpha, \beta)\rangle$ be a TPN and $A(\mathcal{T})$ its associated MCTA. We consider $\overline{Q}_{\mathcal{T}}$ the set of reachable states of $\mathcal{T}$ and Q_A the set of states of $A(\mathcal{T})$. We define the relation $\simeq_{mc} \subseteq \overline{Q}_{\mathcal{T}} \times Q_A$ by the following rule. For all $s = (M, \nu_{\mathcal{T}}) \in \overline{Q}_{\mathcal{T}}$, for all $r = (Mc, v_A) \in Q_A$ (with $Mc = \langle M_r, \chi_r, trans_r\rangle$):

$$s \simeq_{mc} r \Leftrightarrow \begin{cases} M = M_r \text{ and} \\ \forall t \in T \text{ such that } t \text{ is enabled in } M, \\ \qquad \nu_{\mathcal{T}}(t) = v_A(x) \text{ with } x \in \chi_r \text{ such that } t \in trans_r(x) \end{cases}$$

Given the definition of the relation $\simeq_{mc}$, we have the following result.

Theorem 3.2.1 *The binary relation $\simeq_{mc} \subset \overline{Q}_{\mathcal{T}} \times Q_A$ is a timed bisimulation.*

The proof of Theorem 3.2.1 relies on the following two lemmas.

Lemma 3.2.2 *For all $(s, r) \in \overline{Q}_{\mathcal{T}} \times Q_A$, if $s \simeq_{mc} a$ then:*

1. *for all $\delta \in \mathbb{R}_{\geq 0}$, if $s \xrightarrow{\delta} s'$, then there exists $r \xrightarrow{\delta} r'$ such that $s' \simeq_{mc} r'$;*
2. *for all $t \in T$, if $s \xrightarrow{t} s'$, then there exists $r \xrightarrow{t} r'$ such that $s' \simeq_{mc} r'$.*

Proof. Let $s = (M, \nu_{\mathcal{T}})$ be a state in $\overline{Q}_{\mathcal{T}}$ and $r = (Mc, v_A)$ with $Mc = \langle M_r, \chi_r, trans_r\rangle$ be a state in Q_A such that $s \simeq_{mc} r$.

We first consider part 1. Let $\delta \in \mathbb{R}_{\geq 0}$ and suppose that there exists $s' \in \overline{Q}_{\mathcal{T}}$ such that $s \xrightarrow{\delta} s'$. Then $s' = (M, \nu'_{\mathcal{T}})$ with $\nu'_{\mathcal{T}} = \nu_{\mathcal{T}} + \delta$. For all $t \in T$, such that t is enabled from M, we have $\nu_{\mathcal{T}}(t) + \delta \leq \beta(t)$ (by the definition of continuous transitions). Since $s \simeq_{mc} r$, we can deduce that for all $t \in T$ such that t is enabled from M, and for the unique $x \in \chi_r$ such that $t \in trans_r(x)$, we have $v_A(x) + \delta \leq \beta(t)$. For all $x \in \chi_r$, for all $t \in T$, if $t \in trans_r(x)$ then t is enabled from $M_r = M$ (by construction of the relation *trans*). Then we have $\bigwedge_{x \in \chi_r, t \in trans_r(x)} (v_A(x) + \delta \leq$

$\beta(t))$, which means that $v_A + \delta$ satisfies $I(Mc) = \bigwedge_{x\in\chi_r, t\in trans_r(x)}(x \leq \beta(t))$. We deduce that there exist $r' \in Q_A$ such that $r \xrightarrow{\delta} r'$ with $r' = (Mc, v'_A)$ and $v'_A = v_A + \delta$.

The markings which appear in r' and s' are equal because the marking class of r' is the same as the one of r, the marking of s' is the same marking as the one from s and the marking of s and the marking of the marking class of r are equal (due to the fact that $s \simeq_{mc} r$). We consider $t \in T$ such that t is enabled from M, and denote x the clock in χ_r such that $t \in trans_r(x)$; then $\nu_{\mathcal{T}}(t) = v_A(x)$, and consequently $\nu_{\mathcal{T}}(t) + \delta = v_A(x) + \delta$. Since $\nu'_{\mathcal{T}} = \nu_{\mathcal{T}} + \delta$ and $v'_A = v_A + \delta$, and since the marking classes in r and in r' are the same, we can deduce that for all $t \in T$ such that t is enabled from M, for $x \in \chi_r$ such that $t \in trans_r(x)$, we have $\nu'_{\mathcal{T}}(t) = v'_A(x)$; hence $s' \simeq_{mc} r'$. Hence, we have shown part 1.

We now consider part 2. For any $t \in T$, suppose that there exists $s' = (M', \nu'_{\mathcal{T}}) \in \overline{Q}_{\mathcal{T}}$ such that $s \xrightarrow{t} s'$. We can deduce that t is enabled from M; consequently there exists an edge of the form $Mc \xrightarrow{t}_{mc} Mc'$ in the MCG associated to the TPN and hence there exists a switch $e = (Mc, t, \phi, \lambda, \rho, Mc')$ in the associated MCTA. Furthermore, since we have $s \xrightarrow{t} s'$ for the TPN we can deduce that $\nu_{\mathcal{T}}(t) \geq \alpha(t)$. We consider the unique $x \in \chi_r$ such that $t \in trans(x)$. Since $s \simeq_{mc} r$, we have $v_A(x) \geq \alpha(t)$, and since the guard ϕ on the switch e is by construction $x \geq \alpha(t)$, we conclude that v_A satisfies ϕ. We can conclude that there exists $r' = (Mc', v'_A) \in Q_A$ such that $r \xrightarrow{t} r'$ and $Mc' = \langle M'_r, \chi'_r, trans'_r \rangle$ with, by construction of Mc', $M'_r = M_r + W^+(t) - W^-(t) = M + W^+(t) - W^-(t) = M'$.

Now we want to prove that $\forall t' \in T$ such that t' is enabled from M', for the unique $x' \in \chi'_r$ such that $t \in trans'_r(x')$, we have $v'_A(x') = \nu'_{\mathcal{T}}(t')$. We consider a transition $t' \in T$ such that t' is enabled from M'. Two cases are possible :

1. $\uparrow enabled(t', M, t) = False$. This means that t' is enabled from M, and we deduce that for the unique $x \in \chi_r$ such that $t \in trans_r(x)$, we have $\nu_{\mathcal{T}}(t) = v_A(x)$. By definition of $s \xrightarrow{t} s'$, we have $\nu_{\mathcal{T}}(t') = \nu'_{\mathcal{T}}(t')$. We denote x' the clock of χ'_r such that $t' \in trans'_r(x')$. By construction of the MCTA, since t' is not newly enabled, we deduce that $trans'_r(x') \subseteq trans_r(x)$. In fact, during the construction of the MCG, when a set $trans(x)$ that contains enabled

transitions which are not newly enabled, is built, no transition t is added to this set; only removal of transitions is performed, and here t' is not removed because the associated transition is not disabled by the firing of t. We deduce that the renaming function ρ of the switch is such that $\rho(x') = \rho(x)$. Hence we have, by construction of $r \xrightarrow{t} r'$, $v'_A(x') = v_A(\rho(x')) = v_A(x)$. Recalling that $v_A(x) = \nu_{\mathcal{T}}(t)$, we have $v'_A(x') = \nu'_{\mathcal{T}}(t')$.

2. $\uparrow enabled(t', M, t) = True$. We have by definition of $s \xrightarrow{t} s'$, $\nu'_{\mathcal{T}}(t') = 0$. By construction of the MCG, a new clock x' has been created for Mc' such that $t' \in trans'_r(x')$ and such that $x' \in \lambda$. Hence we have $v'_A(x') = \nu'_{\mathcal{T}}(t') = 0$.

We conclude that $s' \simeq_{mc} r'$. ∎

Lemma 3.2.3 *For all $(s, r) \in \overline{Q}_{\mathcal{T}} \times Q_A$, if $s \simeq_{mc} r$ then:*

1. *for all $\delta \in \mathbb{R}_{\geq 0}$, if $r \xrightarrow{\delta} r'$, then there exists $s \xrightarrow{\delta} s'$ such that $s' \simeq_{mc} r'$;*

2. *for all $t \in T$, if $r \xrightarrow{t} r'$, then there exists $s \xrightarrow{t} s'$ such that $s' \simeq_{mc} r'$.*

The proof of this lemma proceeds in the converse manner to that of Lemma 3.2.2. Lemma 3.2.2 and Lemma 3.2.3 can be combined to obtain Theorem 3.2.1.

If we consider a TPN $\mathcal{T} = \langle P, T, W^-, W^+, M^0, (\alpha, \beta)\rangle$ and its associated MCTA $A(\mathcal{T})$, because we have by construction $(M^0, \overline{0}) \simeq_{mc} (Mc_0, \overline{0})$, we conclude that a marking m is reachable from M^0 in $\mathcal{T}$ if and only if there exists a state of $A(\mathcal{T})$ whose associated marking is M. The timed bisimulation property also allows us to obtain the set of states of $\mathcal{T}$ which satisfy a TCTL property: the TCTL property can be verified on $A(\mathcal{T})$, and the resulting set of states of $\mathcal{T}$ satisfying the property can then be obtained using $\simeq_{mc}$.

An example.

Now we show an application of our procedure to the TPN of Figure 3.1. The corresponding MCTA is given in Figure 3.2. The structure (locations and arcs) of the MCTA is directly derived by the MCG, which furnishes also the following information:

- for every location, information regarding the corresponding marking of the considered (PN underlying the) TPN, as well as information about which clock is linked to the currently enabled transitions in the corresponding state of the original model.
- for every arc, the transition which fires in the TPN.

The MCTA construction step labels the locations with the invariants, while guards, clock resets and clock renaming functions are added to the arcs. Guards are written above the line labeling each arc, whereas resets and clock renaming are indicated below. Starting from the initial location MC_0, we have two newly enabled transitions, t_1 and t_2, to which an unique clock, named x, is assigned; the corresponding invariants and guards, indicated on the corresponding outgoing arcs, are indicated with respect to the timing intervals in the TPN under translation. When the outgoing arc labeled t_1 is taken from location MC_0 to location MC_1 (between time 4 and 5), the transition named t_2 is still enabled, so the clock x remains assigned to t_2, and must not be reset before entering MC_1. In location MC_1 the automaton cycles forever, taking the arc labeled t_2 every 1 time unit, and always resetting the clock x before entering the same location, because t_2 is always newly enabled after each firing. When the outgoing arc labeled t_2 is taken from location MC_0 to location MC_2 (after 1 time unit), the transition named t_1 is still enabled, so the clock x remains assigned to it (x is not reset before), while the just fired transition t_2 is newly enabled, and so is assigned to a new clock, y, which must be reset before entering in MC_2. In location MC_2 the automaton can cycle every 1 time unit, resetting the clock y on every cycle, because t_2 is always newly enabled after each firing. When the outgoing arc labeled t_1 is taken from location MC_2 to location MC_1(between 4 and 5 time units since it was enabled), the transition named t_2 is still enabled, but in MC_1 the transition t_2 is already assigned to a clock named x; this implies that the clock y must be renamed into x while taking the arc. Note that the guard on

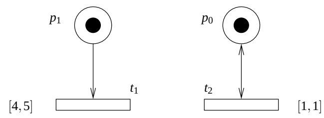
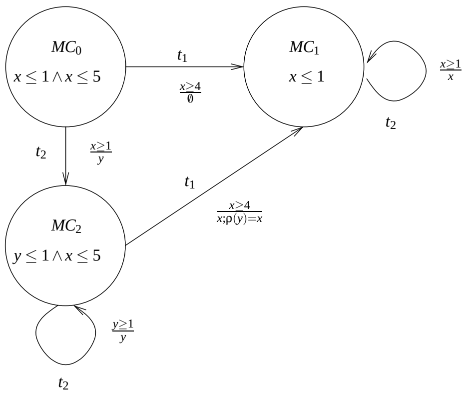


Figure 3.1: A TPN model $\mathcal{T}$



MC	M	$trans$
MC_0	p_0+p_1	$x \leftarrow \langle t_1, t_2 \rangle$
MC_1	p_0	$x \leftarrow \langle t_2 \rangle$
MC_2	p_0+p_1	$x \leftarrow \langle t_1 \rangle$ $y \leftarrow \langle t_2 \rangle$

Figure 3.2: The MCTA corresponding to TPN $\mathcal{T}$ in Figure 3.1

the arc between MC_0 and MC_1 is never true, due to the invariant associated with MC_0, but that MC_1 is reachable via MC_2.

3.2.2 Comparing the MCG, ESCG, and ZBMCG approaches

In this section we compare the ESCG, MCG, and ZBMCG approaches, taking into account the cardinality of locations and edges, as well as the number of needed clocks of the produced TA. We recall that, with respect to the MCG, the ESCG nodes are enriched by the firing domain constraints [48], while in the ZBMCG nodes the available information regards only the reached markings [31].

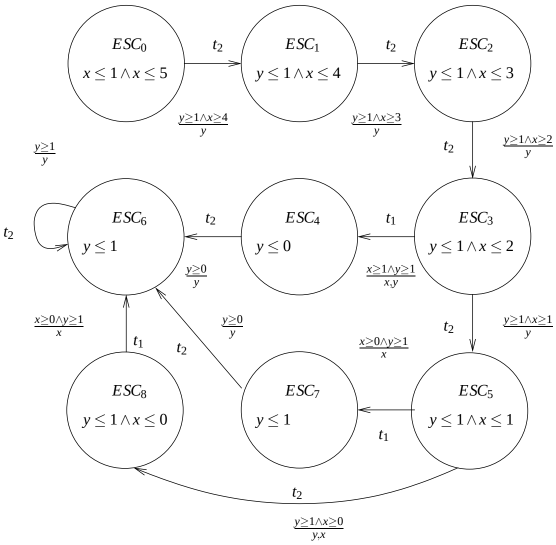
Remark The MCG and the ESCG approaches are incomparable with respect to the cardinality of locations generated.

We provide two examples to substantiate this remark. Let $|\text{MCG}|_{\mathcal{T}}$ and $|\text{ESCG}|_{\mathcal{T}}$ be the cardinality of locations of the MCG and ESCG respectively, of a TPN $\mathcal{T}$

and consider the TPN $\mathcal{T}$ of Figure 3.1. $|\mathrm{MCG}| = 3$ can be derived from the TA shown in Figure 3.2, while the ESCG construction for $\mathcal{T}$ leads to $|\mathrm{ESCG}| = 9$. The ESCG and correspondent TA are shown in Figure 3.3. The table of Figure 3.3 defines, for each extended class ESC, the net marking M, the association of transitions to clocks *trans*, and the firing domains of transitions D. It is clear that, in this net, the ESCG construction distinguishes more than the MCG one. This happens because, in the ESCG, to each reachable marking there may be a number of associated firing domains. Figure 3.4, instead, gives us an example of a TPN $\mathcal{T}$, for which $|\mathrm{MCG}|_{\mathcal{T}} \geq |\mathrm{ESCG}|_{\mathcal{T}}$, as shown in Figures 3.5 and 3.6. In this case, the MCG algorithm, being unable to identify unreachable paths, produces an higher number of locations, some of which are unreachable in the MCTA. In fact, the ESCG construction process, thanks to the firing domain computation, correctly cuts off the untakeable t_2 and t_3 transitions, and so the MC_2 and MC_3 locations are not reached, while this does not happen, as explained previously, during the MCG step. Observe that, thanks to the timed bisimulation result of Theorem 3.2.1, reachability on the timed automaton will reveal that two of the four locations of the TA built from MCG are not reachable.

Remark The ZBMCG approach results in no more locations and edges than the MCG and ESCG approaches.

The ZBMCG method generates only markings which are reachable in the TPN, whereas our MCG approach generates markings which are reachable in the underlying untimed PN. For this reason alone, it is easy to show an example in which the number of locations and edges produced by the ZBMCG method is less than or equal to the number of locations and edges produced by our MCG method. Now note that each location produced by the ZBMCG method corresponds to a set of locations produced by the ESCG method: the marking corresponding to the locations will be the same, but, in the case of the ESCG method, the locations are enriched with firing domains. A similar argument can be used for the edges. Taking again the TPN as in Figure 3.1, in Figure 3.7 we give the TA obtained by applying the ZBMCG technique.



ESC	M	D	$trans$
ESC_0	p_0+p_1	$4 \leq t_1 \leq 5$ $1 \leq t_2 \leq 1$ $3 \leq t_1 - t_2 \leq 4$	$x \leftarrow \langle t_1, t_2 \rangle$
ESC_1	p_0+p_1	$3 \leq t_1 \leq 4$ $1 \leq t_2 \leq 1$ $2 \leq t_1 - t_2 \leq 3$	$x \leftarrow \langle t_1 \rangle$ $y \leftarrow \langle t_2 \rangle$
ESC_2	p_0+p_1	$2 \leq t_1 \leq 3$ $1 \leq t_2 \leq 1$ $1 \leq t_1 - t_2 \leq 2$	$x \leftarrow \langle t_1 \rangle$ $y \leftarrow \langle t_2 \rangle$
ESC_3	p_0+p_1	$1 \leq t_1 \leq 2$ $1 \leq t_2 \leq 1$ $0 \leq t_1 - t_2 \leq 1$	$x \leftarrow \langle t_1 \rangle$ $y \leftarrow \langle t_2 \rangle$
ESC_4	p_0	$0 \leq t_2 \leq 0$ $-1 \leq t_1 - t_2 \leq -1$	$y \leftarrow \langle t_2 \rangle$
ESC_5	p_0+p_1	$0 \leq t_1 \leq 1$ $1 \leq t_2 \leq 1$ $-1 \leq t_1 - t_2 \leq 0$	$x \leftarrow \langle t_1 \rangle$ $y \leftarrow \langle t_2 \rangle$
ESC_6	p_0	$1 \leq t_2 \leq 1$	$y \leftarrow \langle t_2 \rangle$
ESC_7	p_0	$0 \leq t_2 \leq 1$	$y \leftarrow \langle t_2 \rangle$
ESC_8	p_0+p_1	$0 \leq t_1 \leq 0$ $1 \leq t_2 \leq 1$ $t_1 - t_2 = -1$	$x \leftarrow \langle t_1 \rangle$ $y \leftarrow \langle t_2 \rangle$

Figure 3.3: The SCTA corresponding to TPN $\mathcal{T}$ in Figure 3.1

Despite the fact that the ESCG and the ZBMCG can result in smaller TA in terms of locations and edges or clocks, we show in Section 3.4 that, when applied to a number of examples from the literature, the sizes of the TA obtained by the MCG approach are competitive with respect to those obtained by the ESCG and the ZBMCG approaches.

3.3 Improving the effectiveness of the MCG approach

In this section we present some modifications of the MCG algorithm, in order to improve the effectiveness and applicability of our proposed solution.

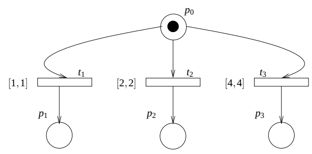
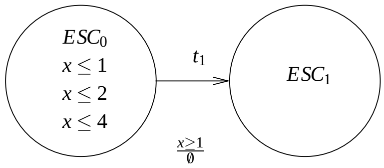


Figure 3.4: A TPN model $\mathcal{T}$, with $|\text{MCG}|_{\mathcal{T}} \geq |\text{ESCG}|_{\mathcal{T}}$

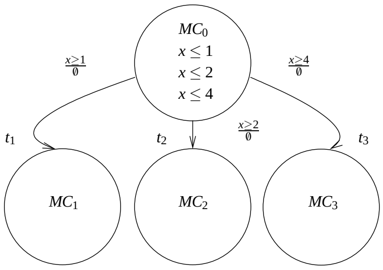


ESC	M	D	$trans$
ESC_0	p_0	$t_1 = 1$ $t_2 = 2$ $t_3 = 4$ $t_2 - t_1 = 1$ $t_3 - t_2 = 2$ $t_3 - t_1 = 3$	$x \leftarrow \langle t_1, t_2, t_3 \rangle$
ESC_1	p_1	$\emptyset$	$\emptyset$

Figure 3.5: The SCTA corresponding to TPN $\mathcal{T}$ in Figure 3.4

3.3.1 Reducing the number of unreachable locations

The first modification allows to cut off paths that could obviously be taken, such as the firing of t_1 in MC_0 of the example in Figures 3.1 and 3.2. As observed before in the TPN of Figure 3.1, when t_1 and t_2 are newly enabled only t_2 can fire. Cutting off the edge from MC_0 to MC_1 does not change $|MCG|$ in this case, but it does for the TPN of Figure 3.4, since it discards the possibility of firing of t_2 and t_3. The MCG computation algorithm can be changed after line 1.4, as in Algorithm 2, that introduces a simple check of the earliest and latest firing time of the newly enabled transitions to remove from consideration transitions that are not firable. Observe that this modification takes timing information into account, as ESCG and ZBMCG do, but with the difference that the check is on the single state and no history of the enabling time of transitions is kept. The TPN on the left part of Figure 3.8 is the most effective case of the modification of the algorithm: the original MCG is infinite (since the $\mathcal{R}_{un}$ is unbounded), but the modified algorithm stops since, as shown on the central and right part of Figure 3.8, the firing of t_1 in MC_1 is not considered.



MC	*M*	*trans*
MC_0	p_0	$x \leftarrow \langle t_1, t_2, t_3 \rangle$
MC_1	p_1	$\emptyset$
MC_2	p_2	$\emptyset$
MC_3	p_3	$\emptyset$

Figure 3.6: The MCTA corresponding to TPN $\mathcal{T}$ in Figure 3.4

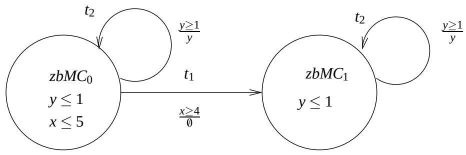
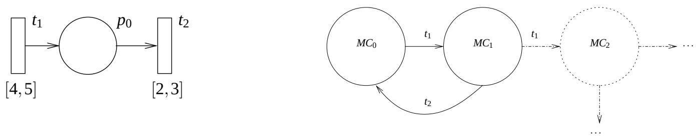


Figure 3.7: The ZBMTA corresponding to TPN $\mathcal{T}$ in Figure 3.1



MC	*M*	*trans*
MC_0	$\emptyset$	$x \leftarrow \langle t_1 \rangle$
MC_1	p_0	$x \leftarrow \langle t_1, t_2 \rangle$
MC_2	$2p_0$	$x \leftarrow \langle t_2 \rangle$ $y \leftarrow \langle t_1 \rangle$

Figure 3.8: A TPN model $\mathcal{T}$, for which the application of the local optimization is useful

3.3.2 Trading clocks for locations and speed

The second modification increases the number of clocks, but decreases the number of locations and the computation time. The variant to the MCTA generation

for *all transitions* t, t' *newly enabled for* M_C **do**
    if $\alpha(t) > \beta(t')$ **then**
        t will not fire in M_C;
    end
end

Algorithm 2: The local optimization to the MCG construction procedure proposed consists of the assignment of a unique clock for every enabled transition, and not a unique clock for every set of newly enabled transitions. In doing so, we aim to avoid the duplication of locations with the same marking, the same set of used clocks, but different (even *via* renaming) mapping functions *trans*. Basically, we do not need anymore the notion of clock similarity, an expensive condition to check, and we consider equivalent the marking classes that have the same marking and use the same set of clocks. We call MCTA^{clock} the automata obtained with such a procedure. Figure 3.7 gives an example showing the benefits provided by the MCTA^{clock} by an application to the TPN of Figure 3.1: assigning two different clocks, x and y, to the newly enabled transitions t_1 and t_2 in location MC_0 let us merge MC_0 and MC_2 into a unique location, decreasing from 3 to 2 the number of required locations. Notice that the MCTA^{clock} is the same as the ZBMTA of the TPN of Figure 3.1.

3.3.3 Dealing with unboundedness

Consider the TPN on the left part of Figure 3.9, illustrating a model provided by the ROMEO package, that depicts an example of a producers-consumers system. The set $\mathcal{R}_{un}$ of this net is unbounded, but the TPN itself has a bounded behavior because the consumers (top part of the net) are always faster the the producers, so that tokens never "pile-up" in place $P3$. Observe that in TPN models whose boundedness depends of time, even the smallest change in the time definition may cause non-termination of the ESCG and ZBMCG algorithms; on the other hand such models may be of interest in many application fields. The method we propose here is to artificially bound the net, using an initial, random guess for this bound, and then to check on the corresponding TA whether the bound is too low. We proceed as follows: compute the P-semiflows of the untimed PN; if all places are covered by at least one P-semiflow, then the net is bounded and the MCG algorithm terminates; else, for all places p_i not covered by a P-semiflow, build the complementary places $\bar{p_i}$, and set $M^0(\bar{p_i})$ to a guessed value (we term $\bar{P}$ the set of complementary places); then build the MCTA using Algorithm 1; finally,

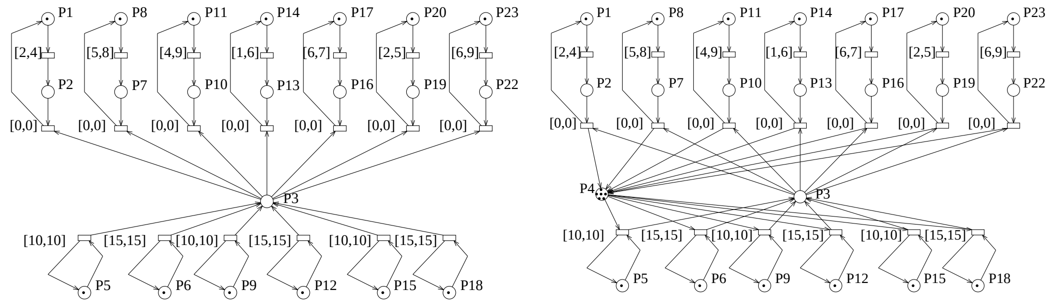


Figure 3.9: An unbounded TPN (left), and the same model after the bounding procedure (right)

check on the MCTA whether there is a reachable state of the TA of marking M, in which the complementary place is actually limiting the original timed behavior (formally, $\exists t$: $\forall p \in P, M(p) \geq W^{-}(p,t) \wedge \exists p \in \overline{P}, M(P) < W^{-}(p,t)$); if such a state exists, increase the initial guess for $M^0(\overline{p})$ and repeat. Note that, if the TPN is unbounded the algorithm does not stop. P-semiflow and complementary places are standard PN concepts, and we do not recall them here. We only show how the net on the left part of Figure 3.9 is modified into the net on the right part of the same Figure. P-semiflow analysis reveals that place $P3$ is unbounded and the complementary place P is inserted. Choosing $M^0(P4) = 6$ bounds also $P3$ to a maximum of 6 tokens. The check on the MCTA reveals that this was a good choice, and we can safely use the MCTA built from the net on the right part of Figure 3.9, rather than the TPN on the left part of the Figure (the underlying PN of which is unbounded), since they have the same behavior over reachable states.

3.4 The GREATSPN2TA tool

In this section we present the tool we have implemented for the computation of the MCTA (or MCTA^{clock}) of a given TPN, where the underlying PN is described with the tool GREATSPN [16]. We will first describe how the tool can be used, then we will discuss some experimental results and also a comparison with the tool ROMEO, which can compute the SCTA and the ZBMTA of a specified TPN.

3.4.1 Using GREATSPN2TA

In the following we describe briefly GREATSPN and KRONOS. Finally, we describe how to use our tool.

GREATSPN **and** KRONOS. GREATSPN [57] is a software package for the modeling, validation and performance evaluation of distributed systems using models based on stochastic Petri nets. GREATSPN offers a graphical interface which can be used to create and save PN models (in .net and .def files) to be subsequently analyzed. It is also possible to interact with GREATSPN in a command line manner, for example using the script showRG to obtain the reachability graph of a given PN. The version of GREATSPN we used is *GreatSPN2.0*. KRONOS [70] is a tool which allows the verification of real-time systems, described by means of TA. KRONOS accepts renaming of clocks, and it is possible to label each location with a set of identifiers; we use this latter fact to associate in the TA the description of the markings with the corresponding MC.

GREATSPN2TA **features** The tool we have developed allows the computation of the MCTA of a given TPN and its translation into the input format of KRONOS. The user has to define the TPN model, called net in the following, using GREATSPN. When the user calls the executable Net2ta_Kronos, the program showRG of GREATSPN is executed in order to compute the reachability graph of the untimed PN. The obtained reachability graph is stored in a user readable file, called net.rg_desc. The executable net2ta_kronos is invoked next: it takes the net description, the time constraints and the reachability graph, computes the MCTA, and outputs it in KRONOS input format.

3.4.2 Experiments and comparison with ROMEO

The software ROMEO [65] permits the state space computation of TPN, on-the-fly TCTL model-checking and the translation from TPN to TA with equivalent behavior. ROMEO incorporates two tools of interest in our context: GPN and MERCUTIO. The former exploits the SCTA computation for transforming a given TPN in the UPPAAL or KRONOS input format, while the latter uses the ZBMTA.

We ran MERCUTIO, GPN, and GREATSPN2TA (using also the variant GREATSPN2TA^{clock}, which implements the MCTA^{clock} variant), on a number of different models. Our experiments were executed on a 1.60 GHz Pentium 4 PC with 512 MB of RAM, running Linux. Table 3.1 lists, for every model, the number of locations, the number of clocks of the TA, and the elapsed time to compute the TA. We considered two classical PN model: the dining philosophers (with 4 philosophers, *Philo*4, as shown in Figure 3.11), the slotted ring with 4 devices (*SR*4, as shown in Figure 3.10), and three models taken from the ROMEO package: a producer-consumer with 6 producers and 7 consumers (*P*6*C*7, as described before), and a set of parallel sequences (*Oex*15, as shown in Figure 3.12), that we have also modified so that each sequence cycles (*Oex*15^{cycle}, as shown in Figure 3.13). For *Philo*4 and *Oex*15^{cycle} a number of different timings of the TPN were considered. The results, shown in Table 3.1, provide examples of the various trade-off that the three methods offer. Due to the different characteristics of the four algorithms, it makes sense to compare the algorithms by pairs: GREATSPN2TA with GREATSPN2TA^{clock}, GPN and MERCUTIO, GPN with GREATSPN2TA and MERCUTIO with GREATSPN2TA^{clock}.

GREATSPN2TA **and** GREATSPN2TA^{clock}. GREATSPN2TA always has a bigger number of locations and a smaller number of clocks than the GREATSPN2TA^{clock} variant: the smaller number of clock is nevertheless paid for in terms of execution time, especially in models in which, in each state, there is an high number of enabled transitions (indeed we stopped the execution of GREATSPN2TA on *P*6*C*7). The larger number of locations can be explained observing that in the MCTA of Figure 3.2, if we assign a clock to t_1 and one to t_2, then MC_0 and MC_2 collapse into a single state. As expected, execution times do not change when changing the timing of transitions.

GPN **and** MERCUTIO. As already observed, GPN optimizes clocks and MERCUTIO optimizes locations: there is not a definitive winner in terms of execution times, although they are all very sensitive to the timing of transitions (most notably in the *Philo*4 case).

GPN **and** GREATSPN2TA For the examples considered, the two tools generate the same number of clocks, but in the *P*6*C*7 case the MCTA computation explodes while computing clock similarity, due to the high number of transitions enabled in each state. In all other cases, the execution time is smaller for GREATSPN2TA.

MERCUTIO **and** GREATSPN2TA^{clock}. MERCUTIO statically assigns one clock per transition and leaves to the TA tool (UPPAAL or KRONOS) the job of minimizing the number of clocks, while GREATSPN2TA^{clock} assigns a different clock to each enabled transition: this explains the smaller number of clocks in the GREATSPN2TA^{clock} column. As expected, the number of locations is smaller in MERCUTIO (which is optimal in this respect), but the execution times can be much worse, especially when changing transition timings.

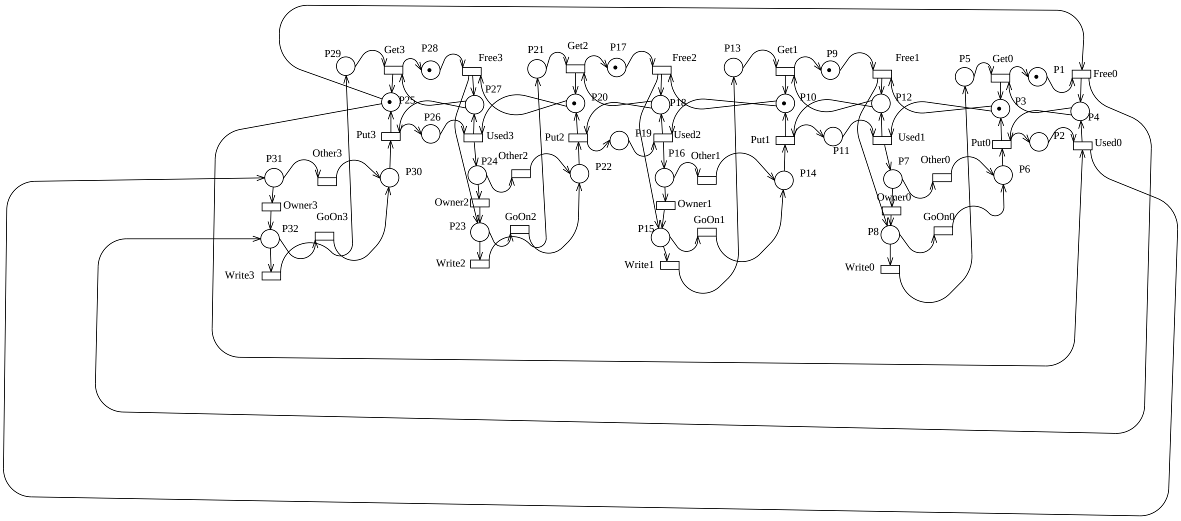
3.5 Conclusions and future works

In this chapter we have presented a method to translate a TPN to a TA by exploiting reachability of the underlying untimed PN of the TPN. This technique has a disadvantage that the untimed PN can be unbounded, even if the TPN is bounded; to address this issue, we have described an empirical method for bounding the PN using complementary places, and then checking if this bound is too restrictive. The experimental results show that the computation time used by our method is competitive for a number of classes of system, and the produced TA generally offer a good compromise between the number of locations and the number of clocks. In

Model	GPN	MERCUTIO	GREATSPN2TA	GREATSPN2TA^{clock}
*SR*4	22907 *loc*	5136 *loc*	7327 *loc*	5136 *loc*
	4 *clocks*	33 *clocks*	4 *clocks*	8 *clocks*
	4.30 *s*	3.86 *s*	2.63 *s*	2.08 *s*
*Philo*4	4406 *loc*	322 *loc*	1161 *loc*	322 *loc*
	6 *clocks*	17 *clocks*	6 *clocks*	8 *clocks*
$Timing_1$	1.50 *s*	0.16 *s*	0.11 *s*	0.07 *s*
$Timing_2$	6.7 *s*	6.2 *s*	0.11 *s*	0.07 *s*
$Timing_3$	$>$ 300 *s*	$>$ 300 *s*	0.11 *s*	0.07 *s*
$Timing_4$	$>$ 300 *s*	$>$ 300 *s*	0.11 *s*	0.07 *s*
$Timing_5$	$>$ 300 *s*	$>$ 300 *s*	0.11 *s*	0.07 *s*
$Timing_6$	6 *s*	0.2 *s*	0.11 *s*	0.07 *s*
*P*6*C*7	11490 *loc*	449 *loc*	*n.a.*	896 *loc*
	3 *clocks*	21 *clocks*	*n.a.*	13 *clocks*
	3.44 *s*	4.70 *s*	$>$ 300 *s*	1.24 *s*
*Oex*15	1048 *loc*	360 *loc*	625 *loc*	625 *loc*
	4 *clocks*	17 *clocks*	4 *clocks*	4 *clocks*
	0.36 *s*	0.63 *s*	0.12 *s*	0.11 *s*
$Oex15^{cycle}$	3510 *loc*	256 *loc*	369 *loc*	256 *loc*
	4 *clocks*	17 *clocks*	4 *clocks*	4 *clocks*
$Timing_1$	3.10 *s*	7.9 *s*	0.07 *s*	0.06 *s*
$Timing_2$	7.8 *s*	32.5 *s*	0.07 *s*	0.06 *s*
$Timing_3$	4.7 *s*	32.7 *s*	0.07 *s*	0.06 *s*
$Timing_4$	4.8 *s*	25.9 *s*	0.07 *s*	0.06 *s*

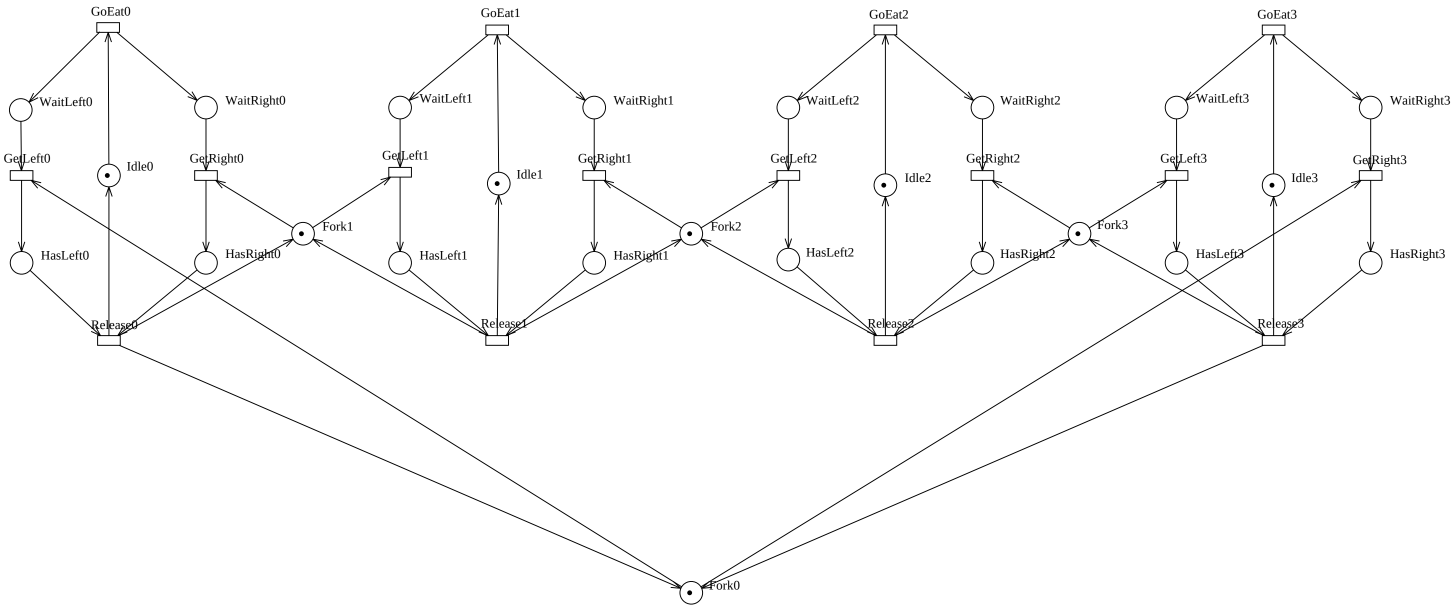
Table 3.1: Experiments results for GPN, MERCUTIO, GREATSPN2TA, and GREATSPN2TA^{clock}.

future work, we plan to address methods for obtaining information about bounds on the number of tokens in places of the TPN, which can then be used in our approach based on complementary places. We also intend to implement a translation to UPPAAL TA (which requires a translation of the MCTA, which has clock renaming, to an equivalent TA without renaming [10]), and to consider the use of clock reduction, as implemented in model-checking tools for TA, in the context of our technique. Finally, the exploitation of some notion of colors and of the use of some symbolic representation for the underlying reachability graph could be a promising approach to be investigated on.



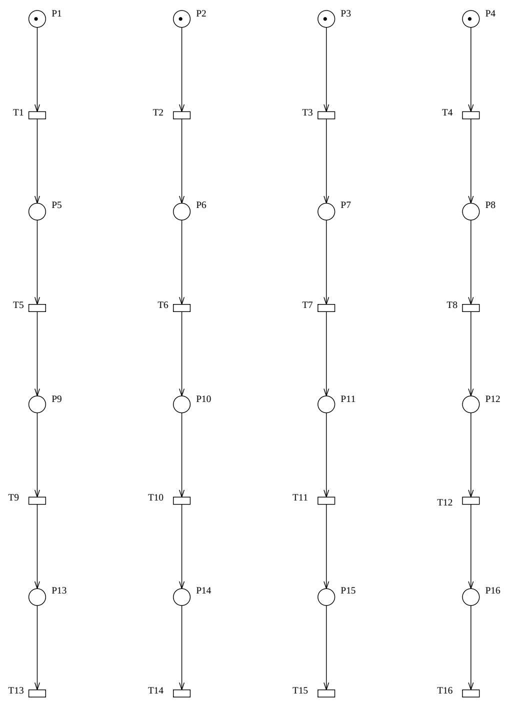
Transition	*Timing*
$Write_i$	$[0,2]$
$Owner_i$	$[1,2]$
$Other_i$	$[1,2]$
$GoOn_i$	$[0,1]$
Put_i	$[0,1]$
Get_i	$[0,1]$
$Free_i$	$[0,2]$

Figure 3.10: The slotted ring TPNmodel



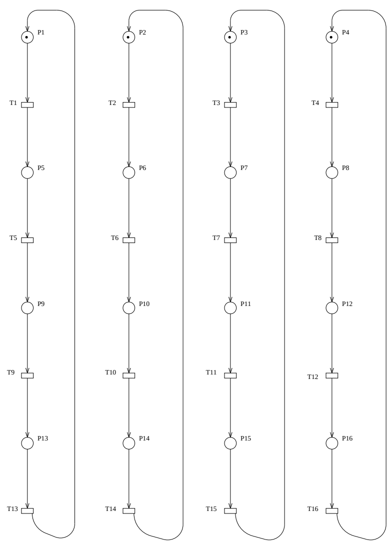
$Transition$	$Timing_1$	$Timing_2$	$Timing_3$	$Timing_4$	$Timing_5$	$Timing_6$
$GoEat_1$	$[0,3]$	$[10,10]$	$[100,100]$	$[1000,1000]$	$[1,1]$	$[100,inf]$
$GoEat_2$	$[0,3]$	$[1,1]$	$[1,1]$	$[1,1]$	$[1000,1000]$	$[1,1]$
$GoEat_3$	$[0,3]$	$[1,1]$	$[1,1]$	$[1,1]$	$[1000,1000]$	$[1,1]$
$GoEat_4$	$[0,3]$	$[1,1]$	$[1,1]$	$[1,1]$	$[1000,1000]$	$[1,1]$
$WaitLeft_i$	$[0,2]$	$[0,2]$	$[0,2]$	$[0,2]$	$[0,2]$	$[0,2]$
$WaitRight_i$	$[0,2]$	$[0,2]$	$[0,2]$	$[0,2]$	$[0,2]$	$[0,2]$
$Release_i$	$[1,2]$	$[1,2]$	$[1,2]$	$[1,2]$	$[1,2]$	$[1,2]$

Figure 3.11: The philosopher TPN model



Transitions	*Timing*
$T1, T2, T3, T4$	$[2, 10]$
$T5, T6, T7, T8$	$[1, 3]$
$T9, T10, T11, T12$	$[4, 6]$
$T13, T14, T15, T16$	$[0, 1]$

Figure 3.12: The TPN model of a four task parallel computation



Transitions	$Timing_1$	$Timing_2$	$Timing_3$	$Timing_4$
$T1, T2, T3, T4$	$[2, 10]$	$[1, 3]$	$[1, 5]$	$[1, 10]$
$T5, T6, T7, T8$	$[1, 3]$	$[4, 6]$	$[3, 7]$	$[3, 10]$
$T9, T10, T11, T12$	$[4, 6]$	$[7, 9]$	$[5, 9]$	$[5, 10]$
$T13, T14, T15, T16$	$[0, 1]$	$[10, 12]$	$[7, 11]$	$[7, 10]$

Figure 3.13: The modified TPN model of Figure 3.12, with different timings

Chapter 4

CSL model checking of GSPN and SWN models

4.1 Introduction

Systems which exhibit non-trivial probabilistic or stochastic behavior are modeled more appropriately using formalisms such as stochastic Petri nets [52] or stochastic process algebra [39], the underlying semantics of which take the form of Markov chains. Model-checking algorithms for continuous-time Markov chains (CTMCs) have been developed, where the property to be verified is described in terms of Continuous Stochastic Logic (CSL) [5, 6]. The logic CSL provides a formal way to describe potentially complex properties reasoning about both the functional behavior and the performance of a stochastic system. It includes a steady-state operator, which can refer to the probabilities of the system being in certain states in equilibrium, and a probabilistic operator, which can refer to the probability with which a certain (possibly timed) property is satisfied, such as "does the system reach an error state within 5 minutes with probability greater than 0.01?" The model-checking tools ETMCC [38], its successor MRMC [45], and PRISM [46, 62], have been used to analyze CSL properties of stochastic systems in application areas such as fault-tolerant systems, manufacturing systems and biological processes.

In this chapter, we focus on the use of CSL model-checking tools for the verification of systems described using *Generalized Stochastic Petri Nets* [1] (GSPN) and *Stochastic Well-formed Nets* [15] (SWN)[1]. SWN are the colored extension GSPN, and have two main advantages over them. Firstly, SWN allow us to define models that are parametric in the "structure" of the system: for example, a GSPN model of a system in which a pool of servers visit a set of stations in a given order is parametric in the number of servers, in the number of clients at each station, but not in the number of stations, whereas in an SWN model a color can be used to distinguish the stations, without the need of replicating station subnets. The second advantage of SWN is related to the solution process: if the color is used to represent a symmetric behavior of the system, this symmetric behavior can be exploited to build a "compact" state space, thus enlarging the size of systems that can be solved. Despite the appealing characteristics of SWN, and the availability of an associated tool (GREATSPN [7, 57] of the University of Torino and the University of Paris 6), SWN modelling is not widespread. One reason for this is the limited support available to validate an SWN model (for example, structural analysis, such as P- and T- invariant computation, is not available for SWN models).

In order to (partially) address this problem, model-checking techniques have been used previously to provide support for the validation of SWN models: in [27], an extension of GREATSPN that allows the use of the PROD tool [63] (a reachability graph analyzer defined for predicate-transition nets) to model check an SWN net against a number of temporal logic properties is presented. However, this approach considers only functional properties of the system, such as "can the system reach an error state", rather than performance indices, and hence is only partially adequate when considering analysis of stochastic formalisms such as SWN.

[1] Since a GSPN model may be considered a degenerate case of a SWN model, what will be illustrated for this latter formalism will hold also for the former one; however, when needed in the context, the two formalisms will be distinguished

In this chapter, we extend the work of [27], and our work on CSL model checking of stochastic Petri nets [25], by considering CSL model checking of GSPN and SWN.

Following our work in [25], we have not built a new CSL model checker, but we have provided a CSL model-checking facility for SWN models in GREATSPN by linking GREATSPN to MRMC and PRISM. These links take the form of programs which translate SWN from the format used in GREATSPN to the (different) input languages of PRISM and MRMC.

We have preferred to consider two "stand-alone" model checkers because they have been built and are maintained by researchers in the specific field of stochastic modelling and verification, and are constantly updated to reflect recent research results. Moreover, although neither tool is based on Petri nets, we considered that they would not be difficult to connect with the GREATSPN modules for SWN, given that there is already some reported experience in interfacing these tools with other tools (ETMCC, the precursor of MRMC, has been interfaced with the Petri net tool DaNAMiCS [55] and the process algebra tool TIPP [37], while PRISM has been interfaced with the PEPA process algebra [39]). We observe that this choice is based on re-use, and it has the significant advantage of being able to profit of all future developments of MRMC and PRISM. However, on the other hand, there is the drawback of not being able to exploit all the peculiarities and properties of nets in the model checking algorithm, as it would have been the case if an ad-hoc solution had been devised. Note that both of the CSL model-checking tools which we consider in this chapter, PRISM and MRMC, do not allow actions to happen in zero-time and with priority over exponential activities: therefore we limit our scope to nets without immediate transitions only.

Note that, as a by-product of the translation from GREATSPN models to PRISM models, we have implemented a method for unfolding SWN nets to GSPN nets within GREATSPN, which can be used in other contexts aside from CSL model checking.

Another contribution of this chapter is the consideration of the way in which meaningful CSL properties of GSPN and SWN may be specified. In particular, we concentrate on the way in which "atomic propositions", which are used in model checking to identify portions of the state space of interest (for example, error states or goal states), can be described. Note that, for SWN, this task is not always straightforward, as the model-checking tool user may need to refer to complex expressions including net places, transitions, and colored tokens, to identify the required part of the state space.

In this chapter we concentrate on SWN and on the GREATSPN tool, but there are other colored extensions of stochastic Petri nets with associated tools. The tool APNN has a built-in CSL model checker [11] which works on the colored GSPN class defined in APNN; however, we were unfortunately not able to use APNN extensively due to lack of documentation. A notion of colors based on replication is present in SAN nets in UltraSAN [23] and Möbius [22]. There are plans to add a CSL model checker to the latter.

The chapter is derived by our work presented in [13], and it is so organized: Section 4.2 discusses CSL model-checking of SWN, in particular with respect to the choice of the set of atomic propositions. Section 4.3 describes linking GREATSPN to PRISM and to MRMC with the help of the running example, as depicted in Figure 1.8 and illustrated in Section 1.2.2. Section 4.4 presents some study cases, useful for understanding the operative steps needed to model check GSPN/SWN by means of our presented approaches. Section 4.5 concludes the chapter.

4.2 CSL model checking of SWN

CSL model checking of SWN requires the following set of ingredients: a SWN model, a set of atomic propositions expressed in terms of net elements, a CSL model checker, and a way to interpret the results of model checking.

Let us consider the set of atomic propositions *AP*. With regard to place-related atomic propositions, which we henceforth call *place propositions*, the expressions of interest are of the form:

(Type M): $\sum_{\mathsf{p}\in P} w_p \cdot \mathrm{M}(\mathsf{p}) \sim K$

(Type Mcol): $\sum_{\mathsf{p}\in P, c\in CD(\mathsf{p})} w_{p,c} \cdot \mathrm{M}(\mathsf{p})[c] \sim K$

where w_p and $w_{p,c}$ are integers, $CD(\mathsf{p})$ is the color domain of place p (set of possible colors of tokens in p), and $\sim \in \{\leq, =, \geq\}$: observe that, since w_p and $w_{p,c}$ can be zero, the atomic propositions can refer also to an arbitrary set of places or to an arbitrary set of colors and places.

Examples of place propositions, using the net of Figure 1.8, are: $\mathrm{M}(\mathsf{loc}) = 1$ (there is one job at the central server), $\mathrm{M}(\mathsf{wait})[d2] \geq 2$ (there are at least two jobs waiting for the services of device $d2$).

With regard to transition-related atomic propositions, which we henceforth call *transition propositions*, the expressions of interest are of the form **(Type T)** the transition t is enabled, or **(Type Tcol)** the transition t is enabled for a given assignment to the variables of t. Examples of transition propositions, using the net of Figure 1.8, are: transition s_srv is enabled (a device has been assigned to a job), transition s_srv is enabled for variable x instantiated to color $d1$ (device $d1$ has been assigned to a job).

For what concerns the evaluation of atomic propositions we can again distinguish marking propositions from transition propositions. Propositions of type M can be defined for GSPN and SWN, and can be computed trivially using the RG for GSPN, and with some effort using the CRG or SRG for SWN (indeed a sum over all colors is necessary for the CRG case, while the cardinality of the dynamic subclasses is used for the SRG case). Propositions of type Mcol are defined instead only for SWN, and can be computed only for the CRG, since this is a type of proposition that may hold in some of the states corresponding to a symbolic marking, but not in all of them. In the running example, the proposition $\mathrm{M}(\mathsf{srv})[d_1] \geq 1$, which is of type Mcol, is valid only for two of the three states represented by the symbolic marking S_1 (more precisely, states C_{1b} and C_{1c}). In this case, the proposition distinguishes more than the net, and requires a modified SRG construction. We do not consider this possibility in this chapter.

Propositions of type T can be defined for GSPN and SWN, and can be computed trivially on the three types of reachability graphs, by simply checking the name of the transitions associated to the arcs out of a state. Propositions of type Tcol are defined instead only for SWN, and can be computed easily from the CRG, where this information is explicitly present, while for the SRG the considerations are similar to the Mcol case; again using the example, if we consider the Tcol proposition "s_srv enabled for an assignment to x of d_1," then not all colored states represented by a symbolic marking enable transition s_srv for the required variable assignment.

There is another type of proposition of interest when dealing with SWN, which we call *symbolic*: these are proposition that consider the color, but not the specific value of the color. Examples of such proposition on the SWN of Fig. 1.8 are: there are at least two tokens of the same color in place wait; transition s_srv is enabled for a value of variable x equal to the value of variable y (assuming a modified example in which the function on the arc from av to s_srv is y). Observe that the first (second) atomic proposition can be computed as the logical disjunction of atomic propositions of type Mcol (Tcol respectively).

Instead of defining two new types of properties that may be cumbersome to compute, we prefer to consider the use of "observation transitions": the SWN is modified so as to include a new transition for each such property, and the property is associated to a colored or symbolic marking only if the transition is enabled in that marking. In the first case we can add to the net a transition having an arc to and from wait, with the function $2\langle x \rangle$, and in the second case we have to split transition s_srv in two transitions with the same input and output function, but with a guard to distinguish whether $x = y$ or not. Observe that Tcol can be rephrased in terms of enabling of an observation transition.

Observation transitions have been introduced by the authors of SPOT, a model checker for colored (non-stochastic) nets [69], and are described in detail in [71]:

"observation" transitions are there defined as transitions with a modified semantics (they are enabled but they never fire), a change in the semantics that is not necessary in our case, if we exclude the use of the Next operator of CSL, because an exponential transition which does not change the marking does not alter the (infinitesimal generator of the) CTMC.

Running example: properties of interest. The verification of our running example (see Figure 1.8 in Section 1.2.2) can range from classical Petri net properties like liveness of transitions and marking invariants, to probabilistic properties that ensure that the service is provided according to certain quality criteria. The liveness of a transition t can be restated as the CSL formula:
$(\Psi_1) : \mathcal{S}_{\geq 1.0}(\mathcal{P}_{\geq 1.0}(\Diamond^{[0,\infty)}\mathsf{t}(\text{ is enabled}) \geq 1))$
Less trivial is the check of marking invariants. The fact that, in any state, there is only one device per type, can be checked as the logical conjunction over all $d \in D$ of following property:
$(\Psi_2) : \mathcal{S}_{\geq 1.0}(\mathrm{M}(\mathsf{av})[d] + \mathrm{M}(\mathsf{srv})[d] + \mathrm{M}(\mathsf{un\_av})[d] = 1)$
Note that $\mathrm{M}(\mathsf{av})[d] + \mathrm{M}(\mathsf{srv})[d] + \mathrm{M}(\mathsf{un\_av})[d] = 1$ is an atomic proposition of Mcol type and it is not simple to define the property in a symbolic manner using an observation transition.

If we are interested instead in proving that each device d will be available upon request, we can verify that the following formula is satisfied in all states:
$(\Psi_3) : \mathrm{M}(\mathsf{wait})[d] \geq 1 \Rightarrow \mathcal{P}_{\geq 1.0}(\Diamond^{[0,\infty)}\mathrm{M}(\mathsf{srv})[d] \geq 1)$

For what concerns probabilistic aspects, assume we are interested in identifying states that are "hot spots", in which the number of jobs waiting for a device exceeds a certain amount. Given a constant *hs*, the atomic proposition HS is valid in states in which the number of tokens in place wait exceeds *hs*, HS[d] is valid in those states in which the number of tokens of color d in place wait exceeds *hs*, while HSx is valid in those states in which there are at least *hs* jobs waiting for the same device (independently of the device identity). The first two propositions belong to the M and Mcol types, while the third requires the introduction of an observation transition t_HS having an arc to and from wait, with the function $hs * \langle x \rangle$.

In the following properties, hot_spot can be either HS, HS[d], or HSx.
(Φ_1) : $\mathcal{S}_{>0.7}(\mathsf{hot\_spot})$: in steady state the sum of the probabilities of states that are hot spots is greater than 0.7. The running example has a CTMC with a single strongly connected component, hence Φ_1 is either true in all states or false in all states.
(Φ_2) : $\mathcal{S}_{\leq 0.2}(\mathcal{P}_{\geq 0.9}(\Diamond^{[0,5]}\mathsf{hot\_spot}))$: this property is true for those states in which the probability of being, in equilibrium, in "bad" states which can reach a hot spot within 5 time units with probability 0.9 or greater, is at most 0.2.
(Φ_3) : $\mathcal{P}_{\geq 0.9}(\Diamond^{[0,5]}(\mathsf{hot\_spot} \wedge \mathcal{P}_{\geq 0.7}(\Diamond^{[0,3]}\neg\mathsf{hot\_spot}))$: this property is true for those states in which the probability of reaching "good hot spot" states within 5 time units is at least 0.9, where "good hot spot" states are hot spot states in which, with probability 0.7 or greater, the system exits from hot spot states within 3 time units.

In the following sections, we illustrate two different approaches to SWN model checking using existing tools: the first is the interface with PRISM, and is realized at the *net level*. The second is the interface with MRMC, and is realized at the *CTMC level*. The two approaches will be illustrated on our central server example.

4.3 Linking GREATSPN to PRISM and MRMC

4.3.1 From GREATSPN to PRISM

PRISM [62] is a probabilistic model-checking tool of the University of Birmingham. The PRISM input language is a state-based language in which a state of a system is described in terms of a valuation of a number of bounded variables declared by the user. The variables may be organized into a series of interacting modules. The dynamics of the system is represented by a set of guarded commands which define sets of state-to-state transitions of the CTMC model. Each command is of the form GUARD $\longrightarrow$ RATE : UPDATE, where GUARD specifies a logical condition on the system variables describing the states in which the command is enabled, RATE is the rate of the command, and UPDATE is a set of assignments that specify the new values of the variables in terms of old ones (where a prime is used to distinguish the new value from the old one), and thus describes the target states of the CTMC transitions defined by the command.

The generation of a PRISM model corresponding to an SPN has been presented in [25]: a single PRISM module is created with as many variables as there are places, and as many commands as there are transitions, where GUARD encodes the transition enabling conditions, and UPDATE encodes the state modification caused by the firing. The set *AP* of atomic propositions is implicitly defined in PRISM models, as the user is allowed to include in a CSL formula any logical condition on the values of the variables. In the implemented translation place names are mapped one-to-one to variable names, and therefore any logical expression on place markings is allowed and is realized trivially (*there is no need to translate type* M *and* Mcol *atomic propositions, whereas* T *and* Tcol *propositions have to be restated in terms of markings*).

Based on our experience with the translation of SPN models we examined two possible ways of connecting GREATSPN to PRISM for SWN: producing directly a PRISM module that is equivalent to the SWN, meaning that the CRG of the SWN and the state space of the PRISM module are isomorphic, and that the same CTMC (up to state numbering) is produced; or *unfolding* the SWN into an SPN, followed by the translation of the SPN into a PRISM module using the already-existing translation for SPN.

Let us consider the feasibility of the first option. Because variables in PRISM are simple, unstructured variables, it is not possible to associate a single variable to an SWN place; moreover, since transitions can fire for different assignment of colors to transition variables, it is not possible to associate a single PRISM command to an SWN transition. Furthermore, the enabling conditions are much more complicated to write. Basically, to translate an SWN into a PRISM module means that we have to address the same problems as defining an unfolding of the SWN into an SPN.

We have therefore taken the second option: the SWN is unfolded into an equivalent SPN, and then the translator to PRISM is applied. Unfortunately, although unfolding algorithms have been defined for Colored Nets (for example in [42]), there is no such detailed definition for SWN, and there is no implementation available for the complete SWN class. The algorithm that we implemented works as follows:

- for each colored place p, and for each distinct tuple of colors in the color domain of p, a neutral place *p_colortuple* is generated;
- for each colored transition t, and for each possible assignment γ of colors to the variables in the input and output colored functions of t, a neutral transition t_γ is created, if it does not violate the predicate associated to t;
- for each neutral transition t_γ, and for each colored place p (input or output place of t), the colored function associated with the arc from p to t (or viceversa) is evaluated on the variable assignment γ; the result of the evaluation is a multiset on the set of neutral places generated from p, and which can be used to assign a multiplicity to the arcs between t_γ and the neutral places.

As an example of the application of the algorithm, we consider as input the net of Figure 1.8 with only two devices: the SPN of Figure 1.9 is the algorithm's output. The uncolored place loc is translated in the uncolored place loc_, whereas the colored place wait of color domain $D = \{d_1, d_2\}$ is translated in two uncolored places wait_d1 and wait_d2.

The color domain cd of transitions in SWN is defined by a pair $\langle \mathit{transition\_parameters\_type}, \mathit{guard} \rangle$. For example, for transition tloc we have $cd(\mathsf{tloc}) = \langle \langle x \rangle \in D, \mathit{true} \rangle$. The set of possible assignments Γ for tloc is $\Gamma(\mathsf{tloc}) = \{x \leftarrow d_1, x \leftarrow d_2\}$. Transition tloc is therefore replaced by two new transitions: tloc_0, for binding of variable x with d_1, and tloc_1 for binding of x with d_2. The resulting output arcs connect transition tloc_0 with uncolored place wait_d1, for assignment γ_1, and transition tloc_1 with uncolored place wait_d2, for assignment γ_2; the multiplicity of each arc is 1.

Some care is necessary when translating arc expressions such as $[x \neq z]\langle x{+}!y, z\rangle + \langle w, Sc\rangle$ where the multiset returned by a tuple of basic functions is obtained by Cartesian product composition of the multisets returned by the tuple elements and where there are predicates associated to single components of the arc expression.

Unfolding in SWN is non-trivial to implement due to the structured form of the color domains, the complexity of the functions (which may have a variable number of terms), and the use of predicates associated to transitions and arcs. The current implementation included in the GREAT2PRISM translator treats the full class of SWN, except for server semantics different from "single server per color", and for the use of dynamic subclasses to define the initial marking.

Although efficiency was not the main objective of the unfolding, the tool was able to translate an SWN net with composite color domain of five classes, producing a SPN of about 100 places and 3500 transitions in a matter of minutes. For the examples in this chapter the time required to produce the unfolding was negligible. The resulting SPN model is then translated into a PRISM module using the GREAT2PRISM translator [25]. As for the SPN case, the CSL formulae are expressed using variable names, which, as explained before, encode the place names and the color.

Running example. We will now show the results of the CSL model-checking of the properties defined for the running example. We considered a variable number of jobs (N), and a variable number of devices ($|D|$ is either 2 or 3).

First the net has been unfolded and translated into a PRISM module using the program GREAT2PRISM. Following is a fragment of PRISM code obtained for the case in which N is equal to 4. Note that the colored place un_av is translated into two PRISM variables, `un_av_d1` and `un_av_d2`, one for each device. Instead, the neutral place loc corresponds to the single PRISM variable `loc_`.

```
const int N = 4;
module M
un_av_d2 : [0..1];
loc_ : [0..4] init 4;
un_av_d1 : [0..1];
wait_d2 : [0..4];
av_d2 : [0..1] init 1;
srv_d2 : [0..1];
wait_d1 : [0..4];
av_d1 : [0..1] init 1;
srv_d1 : [0..1];

[tloc_0] (loc_ > 0) & (wait_d1 < N)
    -> 1.000000 : (wait_d1' = wait_d1 +1) & (loc_' = loc_ -1);

[tloc_1] (loc_ > 0) & (wait_d2 < N)
    -> 1.000000 : (wait_d2' = wait_d2 +1) & (loc_' = loc_ -1);

[back_0] (un_av_d1 > 0) & (av_d1 < 1)
    -> 10.000000 : (av_d1' = av_d1 +1) & (un_av_d1' = un_av_d1 -1);

[back_1] (un_av_d2 > 0) & (av_d2 < 1)
    -> 10.000000 : (av_d2' = av_d2 +1) & (un_av_d2' = un_av_d2 -1);
```

Observe that an Mcol atomic proposition, such as $\mathrm{M}(p)[d] > k$, is translated as $\mathtt{p\_d} > k$, where p_d is the PRISM variable representing the place p with color d. Given an appropriate choice for the constant hs (we chose $hs = \lfloor \frac{N}{2} \rfloor$), the atomic propositions HS, HS[d] and HSx can then be defined in terms of PRISM variables and used as input to the tool.

Table 4.1 shows the results for the various instances of the model (with different values of N, different numbers of devices in the set D, and different choices for the atomic proposition hot_spot) for the CSL formulae Φ_1, Φ_2 and Φ_3. The probabilities illustrated in the table correspond to those computed for the outermost probabilistic or steady-state operators in the formulae, and were computed for the initial state of the system, as in Figure 1.8. The cases of Φ_1 and Φ_2 for HS[d1] are combined, as they result in the same probability for all values of $|D|$ and N. The results were obtained using the backwards Gauss-Seidel method, with a maximum number of iterations equal to 100000, and with an error of 10^{-6} We note that all of the other properties listed in Section 4.2 (Ψ_1, Ψ_2 and Ψ_3) are satisfied in all

Dimension			HS			HS[d1]		HSx		
$\|D\|$	N	$\|States\|$	Φ_1	Φ_2	Φ_3	Φ_1, Φ_2	Φ_3	Φ_1	Φ_2	Φ_3
2	4	106	0.6207	1.0	0.9461	0.2559	0.5121	0.5115	0.8336	0.8580
	16	1276	0.9064	0.9289	0.0695	0.3583	0.0053	0.7166	0.7494	0.0107
	32	4852	0.9826	0.9858	2.0745e-4	0.4175	4.4289e-7	0.8351	0.8495	8.8578e-7
	48	10732	0.9968	0.9973	6.5822e-8	0.4447	1.8819e-12	0.8894	0.8987	3.7638e-12
	64	18916	0.9994	0.9995	3.2673e-12	0.4590	8.4724e-19	0.9180	0.9248	1.6944e-18
3	4	584	0.5756	1.0	0.9929	0.1411	0.3800	0.2821	0.2821	0.6712
	16	22184	0.9298	0.9663	0.2231	0.1728	0.0057	0.3456	0.3456	0.0116
	32	16184	0.9885	0.9920	0.0035	0.1964	4.4399e-7	0.3928	0.3928	8.8798e-7
	48	529640	0.9980	0.9984	0.0053	0.2098	1.8819e-12	0.6294	0.6294	5.6458e-12
	64	123614	0.9996	0.9997	8.9402e-5	0.2177	8.4724e-19	0.6531	0.6531	2.5417e-18

Table 4.1: PRISM results (10^{-6} error) for the running example of Figure 1.8

states of the system.

4.3.2 From GreatSPN to MRMC

MRMC [45] is a probabilistic model-checking tool for Markov chains of the Universities of Twente and Aachen. When interfacing GreatSPN to MRMC, we consider verification of the CTMCs corresponding to the CRG or the SRG. To link GreatSPN to MRMC it is necessary to provide the two MRMC input files: a .tra file that contains an ASCII description of the CTMC rate matrix, and a .lab file that lists all possible atomic propositions and associates to each state the atomic propositions valid in that state. The link for SWN model checking has been realized as an upgraded version of our previous tool GreatSPN2ETMCC [25], which linked GreatSPN to ETMCC, and which had the limitation of working only for SPN and of being able to express only atomic propositions of type M. The resulting tool, named GreatSPN2MRMC, consists of two main modules, GMC2MRMC and APgenerator.

The operational flow to model check an SWN with GreatSPN2MRMC is the following. *Firstly,* GreatSPN is used to define a SWN. *Secondly,* the user creates a .ap text file that contains the atomic propositions of interest of the M, Mcol, T, Tcol types. *Thirdly,* the description of the net (.net and .def-file) is passed to the GMC2MRMC module, which uses offline solvers[2] for the generation of the net's CRG or SRG and associated CTMC. Two output files are generated: the .tra-file, in the MRMC syntax, and a .xlab-file, which is the base for the production of the .lab file required by MRMC. The .xlab-file contains triples of the form ($CTMC-state-id$, $corresponding-net-marking$, $enabled-transitions$) if the CRG is considered, or pairs ($CTMC-state-id$, $enabled-transitions$) if the SRG is used. *Finally*, the .xlab-file and the .ap-file are processed by the APGENERATOR module; if the SRG is used, only M and T propositions present in the .ap-file are evaluated and the labelled CTMC is produced; if CRG is used, also Mcol and Tcol propositions are considered.

Running example. We now discuss the approaches to the specification of atomic propositions in term of SWN elements for the cases of CRG and SRG. To do this, we describe some of the steps involved in the verification of the running example (with $N = 8$ and $|D| = 2$) against property Φ_1, using either the CRG or the SRG option.

CRG approach. If we pass the .net and .def-file to the GMC2MRMC module, then we obtain the following MRMC-ready-to-use .tra-file, consisting of a list of transitions, each described in terms of a source state, target state and rate, respectively:

```
STATES 352
TRANSITIONS 1206
1 2 1.000000
1 3 1.000000
2 4 10.000000
...
```

GMC2MRMC also produces an .xlab file, which encodes the marking of each state and the transition enabled in the state:

[2] These are stand-alone SWN solvers that are not yet part of the current GREATSPN distribution, and for which we would like to thank Marco Beccuti from the University of Piemonte Orientale.

```
1 av(1<d2>1<d1>) loc(8) tloc
2 av(1<d2>1<d1>)loc(7)wait(1<d1>) s_srv_1 ...
...
```

With the latter file, we are able to derive the hot_spot definitions. To achieve this, we create an .ap-file (the content of which is shown in Table 4.2) and call the APGENERATOR module, which scans the .xlab-file and the .ap file, and produces the final .lab file for MRMC. The .lab file, which associates atomic propositions to each state, takes the following form:

```
#DECLARATION
t_HS
#END
...
25 wait>=4 wait_d1>=4
...
34 wait>=4 wait_d2>=4
...
```

SRG approach. For the SRG case, only HS and HSx will be considered as hot_spot (since HS[d] cannot be checked against the SRG, as explained in Section 4.2). If we pass the .net and .def-file to the GMC2MRMC module (after having modified the net by the addition of observation transitions called t_HS and t_HSx, the meaning of which will be clear in the following), then we obtain the MRMC-ready-to-use .tra-file and the .xlab-file. The latter file contains *only* the enabled transitions for each state, and not the encoding of the markings (which is lost because we are working with the SRG). We consider the two hot_spot definitions HS and HSx as enabling conditions of two observation transitions called t_HS (self-loop transition on place wait, labelled as $< x1 > + < x2 > + < x3 > + < x4 >$) and t_HSx (self-loop transition on place wait, labelled as $4* < x >$). Therefore, the .ap file contains two lines, for t_HS and t_HSx, and will be passed, with the .xlab-file, to the APGENERATOR module, in order to produce the .lab file for MRMC.

We performed a set of model-checking tests for the CSL formulae described in Section 4.2, obtaining identical results as those obtained for PRISM for properties

Atomic proposition	**Content of .ap-file**	
hot_spot	CRG **approach**	SRG **approach**
HS	wait $>=4$	t_HS
HS[d]	wait_$d1 >= 4$	N/A
HSx	(wait_$d1 >= 4$)||(wait_$d2 >= 4$)	t_HSx

Table 4.2: Atomic proposition specification for the running example of Figure 1.8

Ψ_1, Ψ_2, Ψ_3 and Φ_3, and results which differ with an error of approximately 10^{-6} for the steady-state computations in properties Φ_1 and Φ_2. We note that the size of the CRG for each instance of the model was the same as the unfolded PRISM model, as shown in Table 4.1, and as expected. Instead, the number of states of the SRGs were approximately a half of the CRG/unfolded state spaces in the case of two devices (for example, when $N = 64$ the SRG contained 9507 states), and approximately one fifth of the CRG/unfolded state spaces in the case of three devices (for example, when $N = 48$ the SRG contained 90992 states).

4.4 Study cases

In this section we present three study cases, in order to illustrate in a deeper way the main differences of the two above described approaches, in particular with regards to:

- input and output formats;
- property specifications;
- expressiveness and computational capabilities;

4.4.1 The ad-hoc system

This study case was chosen because the underlying CTMC *has few states; this fact let us to show the full content of the involved input and output files of the two tools under investigation, in order to well understand the differences between*

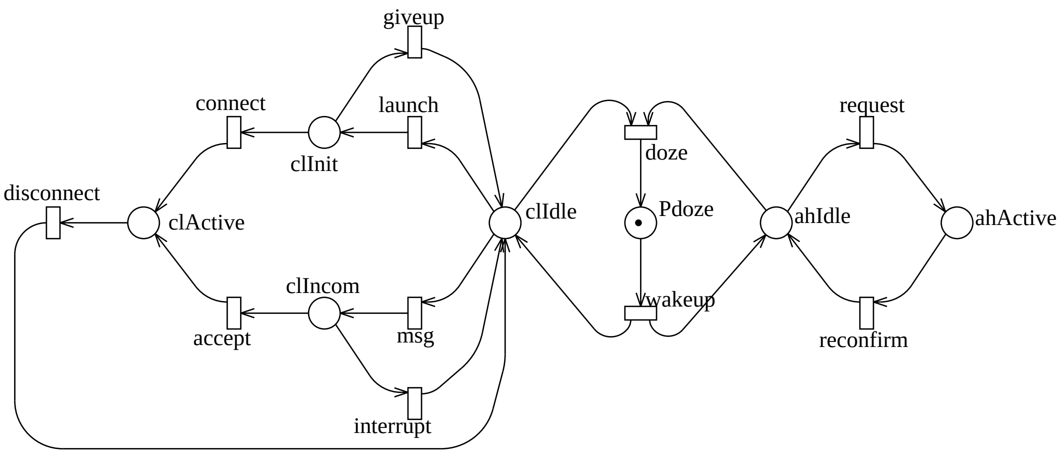


Figure 4.1: The SPN of a battery powered mobile station in an ad hoc network (from [35])

*working at the net level (*GREAT2PRISM *and* PRISM*) or at the* CTMC *level (*GREATSPN2MRMC *and* MRMC*) during the model and property specification and analysis.*

The example is a GSPN taken from [35], and that describes, in a simplified manner, the behavior of a battery powered mobile station in an ad-hoc network. Figure 4.1 shows the model: a station can be in a *doze* state (place *Pdoze*) in which it cannot treat concurrently ad-hoc traffic (right portion of the net) and ordinary calls (left portion). When it exits from the doze state (transition *wakeup*), it activates the right and left portion: ad-hoc traffic is modelled by a simple cycle, while ordinary traffic is split into outgoing calls (upper part of the net) and incoming calls (lower part).

The model is very simple (the CTMC has only 9 states), and for this we have checked the properties defined in [35]: observe that since in [35] the underlying logic is CSRL (CSL with rewards) we had to simplify the formulae by removing the bounds on rewards.

All properties was coded by using M *type propositions.*

Property (1) and (2) - Steady state probability of station in doze state and in call active state.

$$\mathcal{S}_{\geq 0}\ [Pdoze > 0] \quad \text{and} \quad \mathcal{S}_{\geq 0}\ [clActive > 0]$$

Property (3) - Reproducibility of initial marking

$$\mathcal{P}_{\geq 1}\ [true\ U\ Pdoze = 1])]$$

The property is satisfied by all states.

Property (4) - Reproducibility of initial marking with deadline

$$\mathcal{P}_{>0}\ [true\ U\ \leq 10\ Pdoze = 1]$$

Satisfied by all states (only in the initial marking the probability is 1.0).

Property (5) - Probability to receive an incoming call within 24 time units

$$\mathcal{P}_{=?}\ [true\ U\ \leq 24\ clIncom > 0]$$

Again, only the states in which there is at least a token in place *clIncom* have probability 1.0 (there are only two of them).

Property (6) - Probability of launching an outbound call within 24 time units, without using the phone except for ad hoc transfer beforehand

$$\mathcal{P}_{=?}\ [(clIdle > 0)|(Pdoze > 0)\ U\ \leq 24\ clInit > 0]$$

This property is satisfied by 5 states (whose that do not have a token in *clIncom* or *clActive*): of the five states only the two that have a token in *clInit* have probability one.

Now we will show how the above specified properties was checked, using the GREAT2PRISM/PRISM and GREATSPN2MRMC/MRMC approaches.

GREAT2PRISM/PRISM case.

In the following we show the content of the PRISM input .sm-file, describing the ad-hoc system model, and obtained by using the GREAT2PRISM executable on the net of Figure 4.1, specified in the .net and .def-files of the GREATSPN tool.

```
// adhoc

stochastic

// number of tokens
const int T;

// const doubles
const double wakeup     = 3.75;
const double launch     = 0.75;
const double giveup     = 60;
const double connect    = 360;
const double disconnect = 15;
const double msg      = 0.75;
const double interrupt = 60;
const double accept     = 180;
const double doze      = 12;
const double request    = 6;
const double reconfirm = 15;

module station

Pdoze  : [0..T] init T;
cidle  : [0..T];
clInit  : [0..T];
clInc  : [0..T];
clActive  : [0..T];
ahIdle  : [0..T];
ahActive  : [0..T];

[] (Pdoze>0) & (clIdle<T) & (ahIdle<T)  -> wakeup : (Pdoze'=Pdoze-1) & (clIdle'=clIdle+1) & (ahIdle'=ahIdle+1);

[] (clIdle>0) & (ahIdle>0) & (clInit<T) & (clInc<T) & (Pdoze<T)
-> launch : (clIdle'=clIdle-1) & (clInit'=clInit+1)
+ msg : (clIdle'=clIdle-1) & (clInc'=clInc+1)
+ request : (ahIdle'=ahIdle-1) & (ahActive'=ahActive+1)
+ doze : (clIdle'=clIdle-1) & (ahIdle'=ahIdle-1) & (Pdoze'=Pdoze+1);

[] (clIdle>0) & (ahIdle=0) & (clInit<T) & (clInc<T)
-> launch : (clIdle'=clIdle-1) & (clInit'=clInit+1)
+ msg : (clIdle'=clIdle-1) & (clInc'=clInc+1);

[] (clInit>0) & (clActive<T) & (clIdle<T)
-> connect : (clInit'=clInit-1) & (clActive'=clActive+1)
+ giveup : (clInit'=clInit-1) & (clIdle'=clIdle+1);

[] (clInc>0) & (clActive<T) & (clIdle<T)  -> accept : (clInc'=clInc-1) & (clActive'=clActive+1)
```

```
+ interrupt : (clInc'=clInc-1) & (clIdle'=clIdle+1);

[] (clActive>0) & (clIdle<T)   -> disconnect : (clActive'=clActive-1) & (clIdle'=clIdle+1);

[] (clIdle=0) & (ahIdle>0) & (ahActive<T) -> request : (ahIdle'=ahIdle-1) & (ahActive'=ahActive+1);

[] (ahActive>0) & (ahIdle<T)   -> reconfirm : (ahActive'=ahActive-1) & (ahIdle'=ahIdle+1);

endmodule
```

As may be noticed, after the declaration of constants (coding the rates at which the net's transitions fire) and variables (coding the markings of the net's place), a set of guards and commands specifies the net evolution; for example, the second guard-command pair, say us that when the values of *clIdle* and *ahIdle* are greater than 0 and the values of *clInit*, *clInc* and *Pdoze* are less than T, then one of the specified commands, rated by labels *launch*, *msg*, *request*, and *doze*, must be executed. If, for example, the first command is executed (at rate *launch*), then the values of *clInit* and *clIdle* will be incremented and decremented, respectively, by 1 (coding that, in the net under investigation, the firing of transition *launch* consumes one token in place *clIdle* and produces one token in place *clInit*.

In the following we give the content of the PRISM input .csl-file, which contains the first of the above described CSL properties to be verified.

```
S=? [ Pdoze>0 ]
```

Note the use of the ?, which let us to request the PRISM model checker for a precise value, instead of a specific comparison. We do not have the same capability using the MRMC model checker, as will be show later.

In the following we give the content of the output .out-file, after the run of the PRISM tool against the .sm and .csl files, just described.

```
PRISM
=====
Version: 2.0.beta3
Date: Mon May 10 17:07:13 MEST 2004
Command line: prism adhoc10-5.sm -const T=1 adhoc.csl
```

```
Parsing model file "adhoc10-5.sm"...
Parsing properties file "adhoc.csl"...

1 PCTL property:
(1) S=? [ Pdoze>0 ]

Building model (T=1)...
Computing reachable states...
Reachability: 5 iterations in 0.00 seconds (average 0.000000, setup 0.00)
Time for model construction: 0.131 seconds.
Type:        Stochastic (CTMC)
Modules:     station
Variables:   Pdoze clIdle clInit clInc clActive ahIdle ahActive
States:      9
Transitions: 24
Rate matrix: 119 nodes (9 terminal), 24 minterms, vars: 7r/7c

PROPERTY (1): ==========================================
Model checking: S=? [ Pdoze>0 ]
CSL Steady State:
b = 1 states
Computing (B)SCCs... SCCs: 1 BSCCs: 1
Computing steady state probabilities for BSCC 1
Building hybrid MTBDD matrix... [nodes=119] [2.8 KB]
Adding sparse bits... [levels=7, bits=1] [0.3 KB]
Creating vector for diagonals... [0.1 KB]
Allocating iteration vectors... [2 x 0.1 KB]
TOTAL: [3.3 KB]
Starting iterations...
Jacobi: 102 iterations in 0.00 seconds (average 0.000000, setup 0.00)
BSCC 1 Probability: 0.6780560062419575
All states are in a BSCC (so no reachability probabilities computed)
Time for model checking: 0.011 seconds.
RESULT: 0.6780560062419575
```

As may be noticed, the output of PRISM model checker is very clearly structured, and it also provides some indications about used algorithms and data structures, as well as memory and space consumptions for every phases of the verification task.

GREATSPN2MRMC/MRMC case.

When using the GREATSPN2MRMC tool for interfacing GREATSPN with MRMC, the .net and .def- files, specifying the ad-hoc system and created by

GREATSPN are given in input to the GMC2MRMC executable, which produces the underlying CTMC, which may be observed in the following MRMCinput .stc-file:

```
STATES 9
TRANSITIONS 24
1 2 3.750000
2 1 12.000000
2 3 6.000000
2 4 0.750000
2 5 0.750000
3 2 15.000000
3 6 0.750000
3 7 0.750000
4 2 60.000000
4 7 6.000000
4 8 180.000000
5 2 60.000000
5 6 6.000000
5 8 360.000000
6 3 60.000000
6 5 15.000000
6 9 360.000000
7 3 60.000000
7 4 15.000000
7 9 180.000000
8 2 15.000000
8 9 6.000000
9 3 15.000000
9 8 15.000000
```

After the two declarative lines at the beginning, each triplet in the above listed file indicates the source state, the target state and the rate at which the transition occurs.

During the previous step, another file, with .xlab extension, is produced. It labels every CTMC state with the corresponding marking in the original net, and also with the enabled transitions (not shown here). We list this file just below:

```
#DECLARATION
Pdoze(1)
clIdle(1)
```

```
ahActive(1)
clIncom(1)
clInit(1)
ahIdle(1)
clActive(1)
wakeup
launch
giveup
connect
disconnect
msg
interrupt
accept
doze
request
reconfig
#END

1 Pdoze(1) wakeup
2 clIdle(1) ahIdle(1) request launch msg doze
3 clIdle(1) ahActive(1) reconfig launch msg doze
4 clIncom(1) ahIdle(1) accept interrupt request doze
5 clInit(1) ahIdle(1) connect giveup request doze
6 clInit(1) ahActive(1) connect giveup reconfig
7 clIncom(1) ahActive(1) accept interrupt reconfig
8 clActive(1) ahIdle(1) disconnect request
9 clActive(1) ahActive(1) disconnect reconfig doze
```

The .xlab-file is useful because it is used by the APGENERATOR executable of the GREATSPN2MRMC tool, in order to produce a CTMC labeled with the atomic propositions of interest for the verification of a given set of CSL requirements. So, we launch the APGENERATOR binary, giving the .xlab file and the below listed .ap file to it.

```
Pdoze>0
```

Since we want to show only the verification process for the first of the above explained CSL properties of our ad-hoc system, we need to specify the only one atomic proposition in the above .ap-file.

In the following we give the content of the APGENERATOR output .lab-file.

```
#DECLARATION
Pdoze>0
#END

1 Pdoze>0
```

As we can observe, only in state 1 the atomic proposition of interest holds.

At this point we can pass the .stc, .ap and .lab-files to MRMC, within the requirement script file:

```
 $S(>=0 )[Pdoze>0]$
```

As may be seen, in MRMC we do not have the ? operator, but we must use the comparison operator and, subsequently, look in the output file, that we show below, for the real value.

```
RuntimeTask: Checking formula S(>=0 )[Pdoze>0]
FormulaCoding: S(>=0.0)[Pdoze>0] encoded to 1. Stored.
FormulaCoding: Pdoze>0 encoded to 2. Stored.

Verifier: CheckingAP Pdoze>0
Labelling: Property Pdoze>0 encoded to 2 satisfied by state:
1
Labelling: #success states = 1 (out of 9 states)
Verifier:  Time consumption: 0.0 seconds.

Verifier: CheckingSteadyState S(>=0.0)[Pdoze>0]
Steady State analysis: 1 bottom strongly connected component(s) detected.
Steady State analysis: Computing Probablitities for ergodic chain
SSGaussSeidelPred: Running with Accuracy = 1.0E-4, MaxLoopCount = 1000000
SSGaussSeidelPred: Initial solution:
0.1111111119389534
0.1111111119389534
0.1111111119389534
0.1111111119389534
0.1111111119389534
0.1111111119389534
0.1111111119389534
0.1111111119389534
0.1111111119389534
```

```
SSGaussSeidelPred: Positive diagonal elements:
3.75
19.5
16.5
246.0
426.0
435.0
255.0
21.0
30.0
SSGaussSeidelPred: Loops: 10
SSGaussSeidelPred: Result:
0.6780558854635154
0.21189246521358907
0.08475698618951026
6.621639538500187E-4
3.783794021871297E-4
1.5135176104656432E-4
2.648655818244424E-4
0.017027073175809627
0.0068108292586673514
Steady State analysis: SS-Probablitity for ergodic chain  = 0.6780558854635154
Steady-state probabilities:

 state: probability:
--------------------------------------------------
 1:  0.6780558854635154
 2:  0.6780558854635154
 3:  0.6780558854635154
 4:  0.6780558854635154
 5:  0.6780558854635154
 6:  0.6780558854635154
 7:  0.6780558854635154
 8:  0.6780558854635154
 9:  0.6780558854635154
--------------------------------------------------
Labelling: Property S(>=0.0)[Pdoze>0] encoded to 1 satisfied by state:
1
2
3
4
5
6
7
8
9
Labelling: #success states = 9 (out of 9 states)
```

```
Verifier:  Time consumption: 0.07 seconds.
RuntimeTask: Time consumption for formula S(>=0 )[Pdoze>0] : 0.07 seconds.


RuntimeTask: Complete time consumption: 0.14 seconds.
RuntimeTask: Verification terminated.
Output written to adhoc.stc.log
```

We can say that the output provided by MRMC is less intuitive and too much verbose (all verbosity parameters was, however, set to the minimum). In addition it is not structured very well and this represents a quite serious problem, when the produced CTMC has got a great number of states, without an automatic scanner which extract the results.

4.4.2 The multi-server polling system

This example is presented in order to exemplify how the CSL *logics may be used to specify some properties of interest regarding the probabilistic behavior of typical systems. Finally, a brief comparison between* GREATSPN2MRMC/MRMC *and* GREAT2PRISM/PRISM *will be provided.*

Consider a cyclic multiserver polling system, as for example in [1]-chapter 9. Polling systems comprise a set of stations and a number of servers shared among the stations and that move from station to station. The servers follow a given order and a precise policy determining when and for how long the server should serve a given station before moving to the next one. Figure 4.2 depicts an SPN model of a polling system consisting of $N = 4$ stations and S servers. In each station i there are K clients that execute the cycle composed by places Pa_i, Pq_i, and Ps_i. A client in Pa_i is doing some local work (transition $arrive_i$ which has single server policy), it then goes to a queue place (Pq_i) where it waits for a server. When a server arrives, one client in the queue is selected and receives service (place Ps_i and transition $serve_i$ which has infinite server policy). A server which has provided service to queue i "walks" to the next queue (queue $(i+1) \mod 4$ since we are assuming a circular ordering), represented by a token in place Pw_i. When the server arrives

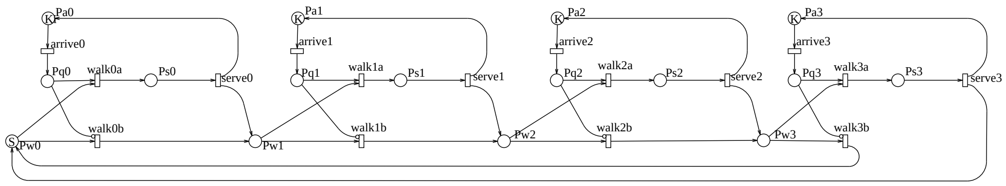


Figure 4.2: The SPN of a four-stations multiple server polling system (from [1]).

at station (transition *walk*$_i$*a*) it provides service if some client is waiting in the queue; if not it moves on to the next queue (transition *walk*$_i$*b*). Below we show the fragment of the PRISM code in which the variables and commands of one station are defined.

```
// variables: station1
Ps1 : [0..K];
Pw1 : [0..S];
Pa1 : [0..K] init K;
Pq1 : [0..K];

// commands: station1
// of transition walk1a
[] (Pq1>0) & (Pw1>0) -> 1 : (Pq1′=Pq1-1) & (Pw1′=Pw1-1) & (Ps1′=Ps1+1);
// of transition walk1b
[] (Pq1=0) & (Pw1>0) -> 1 : (Pw1′=Pw1-1) & (Pw2′=Pw2+1);
// of transition serve1
[] (Ps1>0) -> 1 : (Ps1′=Ps1-1) & (Pw2′=Pw2+1) & (Pa1′=Pa1+1);
// of transition arrive1
[] (Pa1>0) -> 1 : (Pa1′=Pa1-1) & (Pq1′=Pq1+1);
```

Our experiments have considered three models. Model A has one client in each station, a single server and all transitions of rate 1. Model B has ten clients in station 0, two clients in the remaining stations, and two servers. All rates in model B are set to 1, apart from the rate of arrival of clients in station 0, which is 0.25, and the rate of arrival for all other stations is set to 0.5. Model C has thirty clients in station 0, four clients in the remaining stations, and three servers. All rates in model C are set to 1, apart from the rate of arrival of clients in station 0, which is 0.4, and the rate of arrival for all other stations is set to 0.25. All transitions in each model have a single server policy. Models A, B and C have 96, 7902 and 360104 states respectively.

We have verified the following properties using PRISM and MRMC.

All the atomic propositions used are of M *type, being based on place propositions of uncolored models*.

Property (1) - Steady state probability of at least one client in queue 0:

$$\mathcal{S}_{=?}\left[Pq_0 > 0\right].$$

We are using here the PRISM notation $\mathcal{S}_{=?}$ to indicate that we do not want to check a value, but we ask the tool to compute the steady state probability for all states that verify $Pq_0 > 0$. In MRMC we have checked instead $\mathcal{S}_{>0}\left[Pq_0 > 0\right]$. As expected, this property is satisfied in all states in all models, because the models are ergodic.

Property (2) - Absence of starvation (clients waiting in a queue will be served):

$$\left(Pq_0 > 0 \Rightarrow \mathcal{P}_{\geq 1}\left[true\, U\; Pq_0 = 0\right]\right).$$

In all models, the property is true in all states reachable from the initial state (including those in which $Pq_0 = 0$, since the second operand of the implication is satisfied in all states of each model). This property that does not require the CTMC solution, and instead relies on reachability analysis of the model's underlying graph.

Property (3) - Probability of service within a deadline: since all transitions have infinite support and the CTMC is ergodic, then all states will have a non-null probability of service within a deadline, while only the states in which the service is being provided will have a 1 probability. The CSL formula that we have checked is:

$$\mathcal{P}_{=?}\left[\left(Pq_0 > 0 \wedge Ps_0 = 0\right) U^{[0,5]}\; Ps_0 > 0\right].$$

The output of PRISM and MRMC lists the probability of satisfying the until formula from each state.

Property (4) - Reproducibility of the initial marking: since we are working only with reachable states, we can check the simple property:

$$\mathcal{P}_{\geq 1}\ [\mathit{true}\ U\ \text{“init”}]$$

where “init” is a logical condition that fully characterizes the initial marking (and it is therefore different for the various systems that we have verified). This property is satisfied by all states in all models.

Property (5) - Reproducibility of the initial marking with a deadline:

$$\mathcal{P}_{=?}\ [\mathit{true}\ U^{[0,10]}\ \text{“init”}].$$

This property is similar to property (3): it is satisfied by all states and the same comments as above apply.

The tools produce the probability of reaching the “init” state from any of the states of the model. This probability is 1 from the “init” state itself and is instead very low for all other states: for model A, these probabilities for all states are less than 0.01, whereas, after changing the interval to [0,1000], the probabilities of all states are less than 0.25.

Property (6) - Circularity of a server: we wish to check whether a server will present itself more than once at station 0.

$$\mathcal{P}_{\geq 1}\ [G\ (Pw_0 = 1 \Rightarrow \mathcal{P}_{\geq 1}\ [X\ (Pw_0 = 0 \Rightarrow \mathcal{P}_{\geq 1}[\mathit{true}\ U\ Pw_0 = 1])])]$$

where $\mathcal{P}_{\geq 1}\ [G\Phi]$ (read “globally Φ with probability 1”) abbreviates the CSL formula $\neg\mathcal{P}_{\leq 0}\ [\mathit{true}\ U\ \neg\Phi]$. The property is satisfied by all states in all models.

Comment. We observed that the speed of the two tools MRMC and PRISM in obtaining results for our polling system models was comparable. Note that we chose the fully sparse verification engine of PRISM in order to be able to make this comparison; instead, PRISM also supports MTBDD and hybrid verification engines, which can facilitate verification of larger systems. We experimented with the Jacobi and Gauss-Seidel options, and generally found that verification using Gauss-Seidel was more efficient for our models (we terminated the experiments using Jacobi on model C after one hour). Finally, we observed that for models B and C (which both have thousands of states) methods for "filtering" the results of model checking were required, in order to output the probabilities for a small number of states of interest.

4.4.3 The workstation cluster system

This last example is important because provides some highlights on the different ways for specifying the atomic propositions occurring in a CSL *formula, when a user is working on* GSPN *or on a* SWN, *and when it chooses to work with place propositions or with transition propositions*.

Consider a workstation cluster, as described in [36], which is a computer network system decomposed into two sub-clusters consisting of N workstations connected by a central switch and linked by a backbone. Each of the components of the system can break, and can be repaired by the system's repair unit. The GSPN model of the workstation cluster system is shown in Figure 4.3, and is taken from [36], where a more complete description of the system may be found. In the table provided in Figure 4.3 the meaning of each places and transitions modeling a left workstation life cycle is given. For the other subnets analogous names are used, *mutatis mutandis*: *rw* refers to the right cluster, *ls* and *rs* refer to the left and right switches, respectively, *bb* refers to the backbone, and *riIdle* contains a token if the repair unit is idle. As in the PRISM model in [62], we consider an exponential transition also for the acquisition of the repair unit.

The SWN model of the workstation cluster is illustrated in Figure 4.4: the subnets for left and right cluster have been folded into one, as well as the subnets for left and right switch. To maintain the left and right information a color class C has been defined with two colors $\{l, r\}$.

For each model we check the following CSL properties (taken from [36]):

Property(Φ_1). $\mathcal{S}_{>0.7}(\mathsf{premium})$: in the long run, premium QOS will be delivered with probability at least 0.7.

Basic formula	**Meaning**
$LeftOperational_i$	$\sharp\, LeftWorkstationUP \geq i \wedge \sharp\, LeftSwitchUP > 0$
$RightOperational_i$	$\sharp\, RightWorkstationUP \geq i \wedge \sharp\, RightSwitchUP > 0$
$Conn$	$\sharp\, LeftSwitchUP > 0 \wedge \sharp\, RightSwitchUP > 0 \wedge \sharp\, BackboneUP > 0$
$Operational_i$	$\sharp\, LeftWorkstationUP + \sharp\, RightWorkstationUP \geq i \wedge Conn$
$Minimum$	$LeftOperational_k \vee RightOperational_k \vee Operational_k$
$Premium$	$LeftOperational_N \vee RightOperational_N \vee Operational_N$

Table 4.3: Basic formula abbreviations for the workstation cluster model in Figure 4.3

Property(Φ_2). $\mathcal{S}_{<0.05}(\neg\mathsf{minimum})$: in the long run, the chance that QOS is below its minimum level is less than 0.05.

Property(Φ_3). $\mathcal{P}_{\geq 1}(\mathtt{true}U\mathsf{premium})$: the system will always be able to offer premium QOS at some point in the future.

Property(Φ_4). $\mathcal{P}_{=?}(\mathtt{true}U^{[0,T]}\neg\mathsf{minimum})$: the probability that the QOS drops below the minimum quality level within T time units.

where minimum and premium are defined as in Table 4.3

Using GREAT2PRISM/PRISM approach

We will now show the use of the GREAT2PRISM translator on the workstation cluster example. We considered the SPN and equivalent SWN nets for the example with N=8, 16, 32, 64; each models differs only in number of workstation per cluster.

Model checking of properties (1)-(3) produces "yes" or "no" answers, whereas formula (4) produces probability values for the path formula $(\mathtt{true}U^{[0,T]}\neg\mathsf{minimum})$. The atomic propositions premium and minimum are expressed by PRISM formulae which characterize the quality of service provided by the system. Each formula is expressed in terms of variable names taken from the PRISM module obtained from the translation of the SPN and SWN nets.

SPN **case.** The following is a fragment of PRISM code obtained from the translation of SPN net in Figure 4.3. The fragment models the left workstation cluster.

```
const int N = 8;
module M
lwu : [0..8] init N;
lwd : [0..8];
lwINr : [0..1];
ruIdle : [0..1] init 1;
  [lwf] (lwu > 0)->0.002000:(lwu' = lwu -1) & (lwd' = lwd +1);
  [lwi] (lwd > 0) & (ruIdle > 0)->10.000000 :    (lwd' = lwd -1) &
                                          & (ruIdle' = ruIdle -1) &
                                          & (lwINr' = lwINr +1);
  [lwr] (lwINr > 0) -> 2.000000 : (lwINr' = lwINr -1) &
                              & (lwu' = lwu +1) &
                              & (ruIdle' = ruIdle +1);
```

Every place of the nets has been translated into a variable with the same name (no renaming occurs). An atomic proposition, such as $m[p] > 0$, is then translated as the equivalent PRISM formula P>0. In this way we achieve the maximum expressiveness for CSL formulae: i.e. every CSL formula on the SPN model can be translated in an equivalent PRISM formula.

Note that in this case all obtained atomic propositions are of M *type*.

The two expressions for premium and minimum are the following:

```
const int k =floor(0.75*N);
label "minimum" = (lwu>=k & lsu>0) | (rwu>=k & rsu>0) |
         ((lwu+rwu)>=k & lsu>0 & bbu>0 & rsu>0);
label "premium" = (lwu>=N & lsu>0) | (rwu>=N & rsu>0) |
         ((lwu+rwu)>=N & lsu>0 & bbu>0 & rsu>0);
```

where k is set to $3/4$ of the workstation number per cluster. The condition lwu $\geq k$ states for "the number of workstation in the left cluster is greater than three quarters of total workstations per cluster", whereas the condition lsu ≥ 0 states for "the switch in left cluster is up". Similarly the formula minimum indicates that there are at least three quarters of the total workstations in the left cluster and the left switch is up, or in the right cluster and the right switch is up, or there are at least three quarter of total workstation in either of the clusters and each switch and the backbone line is up.

Note that, while the size of the models obtained by the translator agreed with those of the models on the PRISM web-page, the size of the MTBDD representation of the transition matrix of the models produced by the translator generally exceeded those of the web-page models, to the point that the largest models were not solvable. The MTBDD representation of the transition matrix depends heavily on the ordering of the PRISM variables in the module description. A good ordering for the variables of the workstation cluster is obtained by declaring the variables `lwd`, `lwu`, `rwd` and `rwu` together, and last in the ordering, as suggested by the developers of PRISM [56], so that related MTBDD variables are ordered close together. The translator does not have any associated heuristic: for the purposes of this chapter we chose to manually change the variable ordering in the PRISM models obtained by our translator.

SWN **case.** To translate the SWN model the net has first been unfolded, and the unfolded model has been translated using GREAT2PRISM: the net resulting from the unfolding is the same as in Figure 4.3, but the name of places and transitions are different, since the unfolding appends the color to the name of the place. What follows is a fragment of PRISM code obtained from the unfolding:

```
const int N = 8;
module M
wu_l : [0..8] init N;
wd_l : [0..8];
wINr_l : [0..1];
ruIdle : [0..1] init 1;
  [wf_l] (wu_l > 0)->0.002000:(wu_l' = wu_l -1) & (wd_l' = wd_l +1);
  [wi_l] (wd_l > 0) & (ruIdle > 0)->10.000000 : (wd_l' = wd_l -1) &
                                            & (ruIdle' = ruIdle -1) &
                                            & (wINr_l' = wINr_l +1);
  [wr_l] (wINr_L > 0) -> 2.000000 : (wINr_l' = wINr_l -1) &
                              & (wu_l' = wu_l +1) &
                              & (ruIdle' = ruIdle +1);
```

In this case, places of color domain $C = \{l, r\}$ has been translated in two variables: for example, place *wu* has been translated in variable *wu_l*, which represents instances of a token of color *l*, and variable *wu_r*, which represents a token

	N							
	8		16		32		64	
	SPN	SWN	SPN	SWN	SPN	SWN	SPN	SWN
Φ_1	true	true	true	true	true	true	true	true
Φ_2	true	true	true	true	true	true	true	true
Φ_3	true	true	true	true	true	true	true	true
Φ_4 with $T = 2500$	0.00148460	0.00148460	0.00130158	0.00130158	0.00129237	0.00129237	0.00130603	0.00130603

Table 4.4: PRISM results (10^{-6} error) for the workstation cluster models of Figures 4.3 and 4.4

of color r. Place *ruIdle* is neutral, therefore has been translated as one variable with the same name as in SPN case.

For the SWN case, the two atomic propositions minimum and premium translate to:

```
label "minimum" = (wu_l>=k & su_l>0) | (wu_l>=k & su_l>0) |
          ((wu_l+wu_r)>=k & su_l>0 & bbu>0 & su_r>0);
label "premium" = (wu_l>=N & su_l>0) | (wu_l>=N & su_l>0) |
          ((wu_l+wu_r)>=N & su_l>0 & bbu>0 & su_r>0);
```

where each of minimum and premium have the same meaning as in the SPN case.

Note that even in this case, the atomic propositions are of M *type, and not of* Mcol *type, since we are working on the unfolded* SWN.

Table 4.4 shows the results for the various models: observe the perfect match of GSPN and SWN results that is actually not surprising since the underlying CTMC is the same, and the same ordering of variables has been used

Using GREATSPN2MRMC/MRMC approach

SPN **case.** Consider again the SPN model of the workstation cluster example. Here is a portion of the .xlab file for the $N = 8$ case: labels are listed first, one label for every possible numbers of tokens in a place in any reachable state, and then each state is associated with the labels valid in that state (in this case state 1 is the initial marking of the net).

```
#DECLARATION
```

QoS definition	Translation
Minimum	$(lwu >= 6$ && $lsu > 0)$ | | $(rwu >= 6$ && $rsu > 0)$ | | $(\ lwu + rwu >= 6$ && $(lsu > 0$ && $rsu > 0$ && $bbu > 0)\ )$
Premium	$(lwu >= 8$ && $lsu > 0)$ | | $(rwu >= 8$ && $rsu > 0)$ | | $(\ lwu + rwu >= 8$ && $(lsu > 0$ && $rsu > 0$ && $bbu > 0)\ )$

Table 4.5: Translation of Qos definitions, to be used with MRMC, using the GSPN net elements of Figure 4.3

```
lsu(1) lwu(8) rsu(1) bbd(1) ruidle(1) rwu(8) ......
...
#END
1 lsu(1) lwu(8) rsu(1) bbu(1) ruidle(1) rwu(8)
2 lsu(1) lwu(8) bbu(1) rsd(1) ruidle(1) rwu(8)
3  ......
...
```

Table 4.5 lists the definition of the MINIMUM and PREMIUM QoS for the GSPN model, in terms of expression on the markings, when $N = 8$ and $K = 6$ (three quarters of N). The expression is parsed by APGENERATOR to produce the .lab file, that is then given in input to MRMC, a small portion of which is shown in the following:

```
#DECLARATION
lwu>=6 lwu>=8 rwu>=6 rwu>=8 lsu>0 rsu>0
bbu>0 lwu+rwu>=6 lwu+rwu>=8
#END

1 bbu>0 lsu>0 lwu>=6 lwu>=8 lwu+rwu>=6 lwu+rwu>=8......
2 lsu>0 lwu>=6 lwu>=8 lwu+rwu>=6 lwu+rwu>=8 .....
3 .......
.......
```

As may be observed, the states are labeled with atomic propositions of M *type.*

The obtained results with MRMC will be discusses at the end of the next paragraph.

SWN **case.** *In this paragraph we provide an example that utilizes the observation transitions for specifying* Tcol *atomic propositions.*

QoS definition	Translation
Minimum	$(PART_operational_i)$ | | $(operational_i$ && $conn)$
Premium	$(PART_operational_n)$ | | $(operational_n$ && $conn)$

Table 4.6: Translation of Qos definitions, to be used with MRMC, using the SWN net elements of Figure 4.5

Type	N	$\|TRS\|$	Φ_1	Φ_2	Φ_3	Φ_4
GSPN	8	2772	yes	yes	yes	0.001484433
SWN	8	1413	yes	yes	yes	0.001484439
GSPN	16	10132	yes	yes	yes	0.001301666
SWN	16	5117	yes	yes	yes	0.001284814
GSPN	32	38676	yes	yes	yes	0.001293746
SWN	32	9885	yes	yes	yes	0.001289851
GSPN	64	151060	yes	yes	yes	0.001306365
SWN	64	38819	yes	yes	yes	0.001304252

Table 4.7: MRMC results (10^{-6} error) for the workstation cluster models of Figures 4.3 and 4.5

Figure 4.5 shows how we added the observation transitions to Figure 4.4, in order to manage the colored instances of the net elements for specifying the properties of interest.

Table 4.6 lists the definition of the MINIMUM and PREMIUM QoS for SWN in terms of observation transitions.

Table 4.7 shows the results for the GSPN and SWN models for various values of N.

4.5 Conclusions and future works

This chapter discusses how we have exploited two CSL model checkers, PRISM and MRMC, to add CSL model checking facilities for GSPN and SWN in the GREATSPN tool.

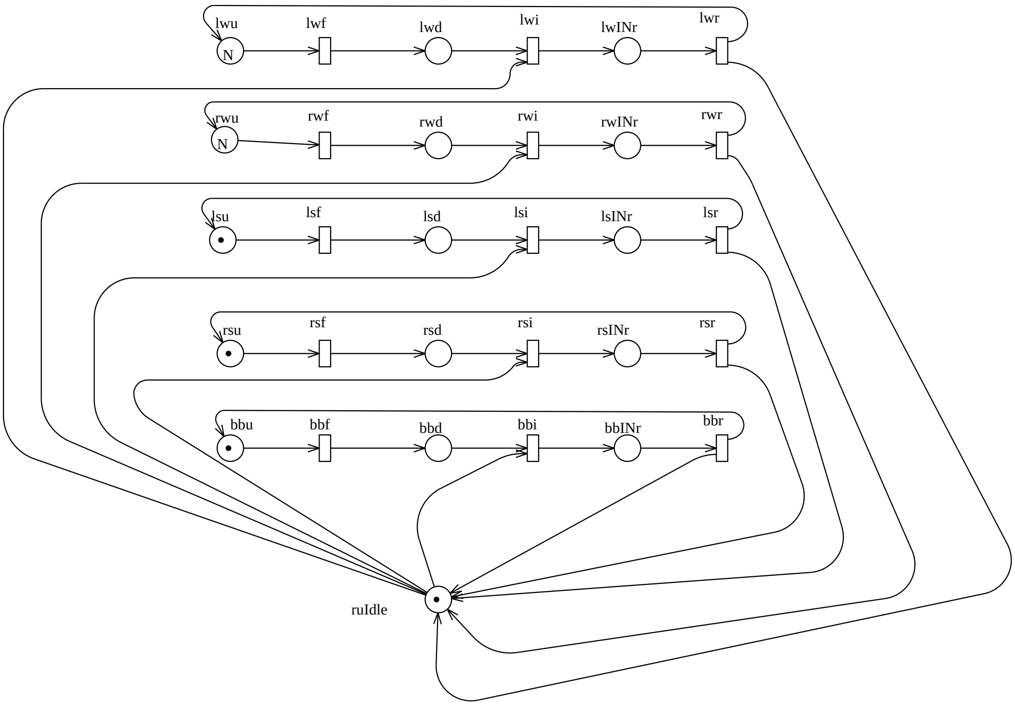
We allow checking of the unfolding of an SWN via PRISM permitting us to take advantage of the efficient MTBDD data structures used in PRISM. We can alternatively check the CRG or SRG of an SWN, in the case of SRG taking advantage of the symmetries of the SWN, using MRMC.

We note that model-checking methods for variants of CSL which refer to rewards have also been implemented in MRMC and PRISM, and can be used to verify an even wider variety of performance properties than those considered by CSL. The presence of rewards is orthogonal to our translation methods, and therefore our programs can be extended easily to accommodate rewards.

We intend to improve the links between GREATSPN and the two model-checking tools by reducing the level of expertise that a GREATSPN user requires to have of the tools. In particular, we aim to automate as much as possible the description of CSL properties, so that the user can express such properties at the net level. For example, in the context of the translation to PRISM models, the mapping of places to variables is partially lost due to unfolding, and therefore it may be advantageous to have a translator from CSL formulae of Mcol and Tcol type to formulae given in terms of PRISM variables. Another problem that we will address is the presentation of the model checking results to the GREATSPN user.

With regard to the translation to PRISM, the MTBDD representation of the transition matrix depends heavily on the ordering of the PRISM variables in the module description. A good ordering for the variables of the example is obtained by declaring together the variables which are closely related (as suggested by the developers of PRISM [56]). For example, in general, places which are connected to each other by transitions are listed together in the variable ordering. The translator does not have any associated heuristic: for the purposes of this chapter we chose to manually change the variable ordering in the PRISM models obtained by our translator. To avoid such manual intervention, we intend to exploit the structure of the Petri net model to define an efficient ordering of the variables used in the PRISM model's description.

Finally, recall that the reachability graph of a GSPN/SWN corresponds to a semi-Markov process, from which a CTMC can be obtained by eliminating vanishing states (in which no time elapses). As noted in [11], the elimination of vanishing states removes information about the dynamic behavior of the system which may be relevant for the evaluation of a CSL property. The problem is considered in [14], in which the CSL logic and its related model checking algorithms are modified to allow the use of immediate transitions in GSPN models. Therefore other future work includes extensions of our tools in this direction.



Net Element	Meaning
lwu	A left workstation is up
lwd	A left workstation is down
lwINr	A left workstation is in repair
lwf	A left workstation is failing
lwi	A left workstation is being inspected
lwu	A left workstation is being repaired

Figure 4.3: The SPN of a workstation cluster system (from [36]).

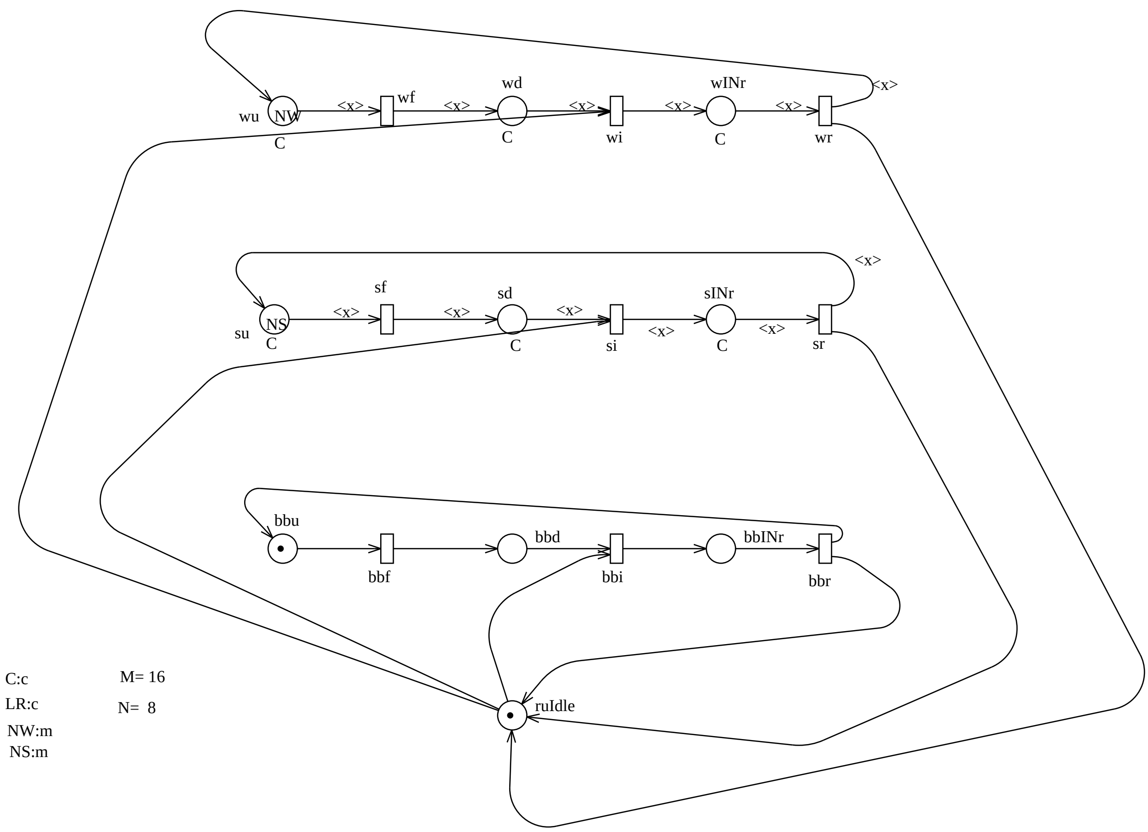


Figure 4.4: The SWN obtained from the SPN of Figure 4.3.

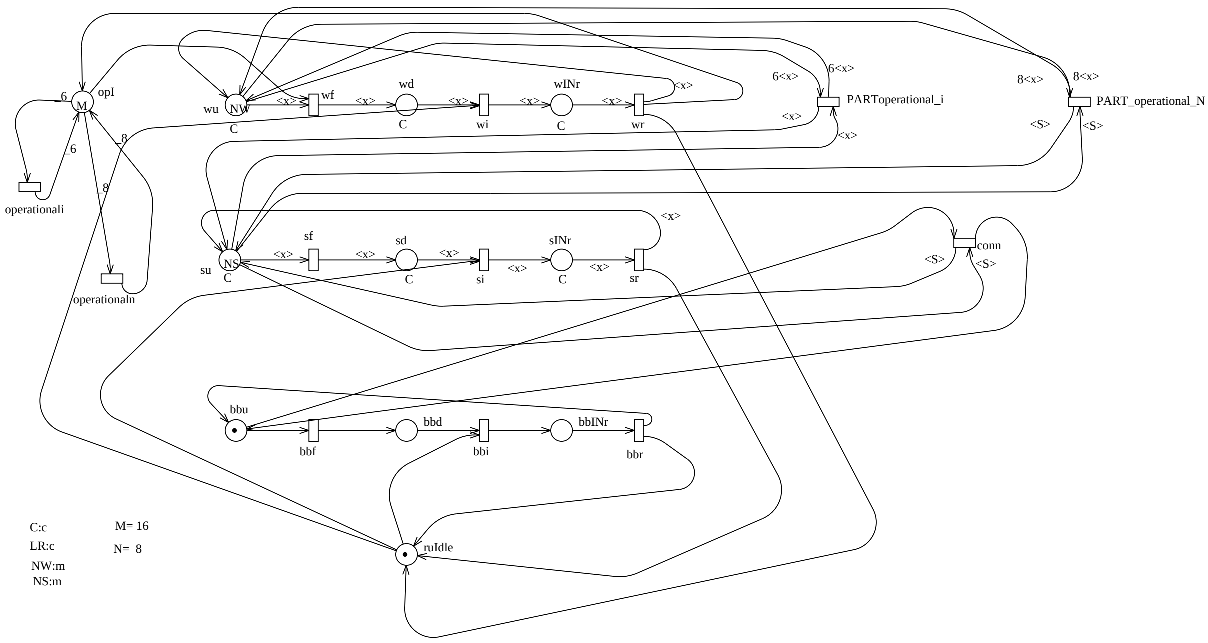


Figure 4.5: The SWN with observation transitions, obtained from the SWN of Figure 4.4.

Part IV

Conclusions

Chapter 5

Conclusions, open problems and future works

In this thesis we have presented our contributes regarding methods and tools for the model checking of PN, in particular TPN, GSPN and SWN. In the following we recall all contributes within their open problems and potential future developments.

Contribute 1.

We have presented a method to translate a TPN *to a* TA *by exploiting reachability analysis of the underlying untimed* PN *of the considered* TPN. *By using such approach, the obtained* TA *may be used to perform* TCTL *model checking using the* KRONOS *model checker. The proposed method was implemented, finally, to add* TCTL *model checking capabilities to our* GREATSPN *tool*[1].

Advanteges. The experimental results show that the computation time used by our method is competitive for a number of classes of system, and the produced

[1]This work is based on a translation defined in the Master Thesis of the student Arnaud Sangnier, of the "Universite' de Paris 6", prepared while he was a visiting student in our research group. A first preliminary translation was also implemented, but it was completely re-done in this thesis to efficiently compute the bisimulation. A similar translation was present also in my Laurea Thesis, presented at the end of 2002

TA generally offer a good compromise between the number of locations and the number of clocks.

Open problems and future works. The untimed PN can be unbounded, also if the TPN is bounded; even if, for addressing the just cited problem, we have described an empirical method for bounding the PN using complementary places, and then checking if this bound is too restrictive, we plan to address methods for obtaining information about bounds on the number of tokens in places of the TPN, which can then be used in our approach based on complementary places. We also intend to implement a translation to UPPAAL TA. Finally, the exploitation of some notion of colors and of the use of some symbolic representation for the underlying reachability graph could be a promising approach to be investigated on.

Contribute 2.

We have exploited two CSL *model checkers,* PRISM [2] *and* MRMC*, to add* CSL *model checking facilities for* GSPN *and* SWN *into the* GREATSPN *tool; this led to the implementation of two tools* GREAT2PRISM *and* GREATSPN2MRMC.

Advantages. With respect to SWN:

- we allow checking of the unfolding of an SWN via PRISM, permitting us to take advantage of the efficient MTBDD data structures used in PRISM;
- we can alternatively check the CRG or SRG of an SWN, in the case of SRG taking advantage of the symmetries of the SWN, using MRMC.

Open problems and future works. We intend to improve the links between GREATSPN and the two model-checking tools by reducing the level of expertise that a GREATSPN user requires to have of the tools. In particular, we aim to automate as much as possible the description of CSL properties, so that the user can express such properties at the net level.

[2]The link to PRISM was implemented by Davide Cerotti, a PhD student belonging to our research group

With regard to the translation to PRISM, the MTBDD representation of the transition matrix depends heavily on the ordering of the PRISM variables in the module description. We intend to exploit the structure of the Petri net model to define an efficient ordering of the variables used in the PRISM model's description.

Since the elimination of vanishing states during the reachability graph of a GSPN/SWN removes information about the dynamic behavior of the system which may be relevant for the evaluation of a CSL property, we want to extend the CSL and associated tools to allow the use of immediate transitions (as in [14]).

Finally, we will address the presentation of the model checking results to the GREATSPN user.

We want to recall, as a by-product of the above contribute, the followings.

Contribute 2.1.

We have implemented a method for unfolding SWN *to* GSPN *within* GREATSPN, *which can be used in other contexts aside from* CSL *model checking*[3].

Contribute 2.2.

We concentrate on the way in which meaningful CSL *properties of* GSPN *and* SWN *may be specified. In particular, we considered how "atomic propositions", which are used in model checking to identify portions of the state space of interest (for example, error states or goal states), can be described.*

[3]This contribute was given by Davide Cerotti, a PhD student belonging to our research group

Part V

Bibliography

Bibliography

[1] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. J. Wiley, 1995.

[2] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[3] R. Alur and D. Dill. Automata-theoretic verification of real-time systems. *Formal Methods for Real-Time Computing*, pages 55–82, 1996.

[4] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[5] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model-checking continuous time Markov chains. *ACM Transactions on Computational Logic*, 1(1):162–170, 2000.

[6] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model-checking algorithms for continuous-time Markov chains. *IEEE Transactions on Software Engineering*, 29(6):524–541, 2003.

[7] S. Bernardi, C. Bertoncello, S. Donatelli, G. Franceschinis, G. Gaeta, M. Gribaudo, and A. Horváth. GreatSPN in the new millenium. Technical report, In Tools of Aachen 2001, International MultiConference on Measurement, Modelling and Evaluation of Computer-Communication System, 2001. Research Report no. 760/2001, Universität Dortmund (Germany), September 2001, pages 17–23.

[8] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, Mar. 1991.

[9] B. Berthomieu and F. Vernadat. State class constructions for branching analysis of time Petri nets. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, volume 2619 of *Lecture Notes in Computer Science*, pages 442–457. Springer Verlag, 2003.

[10] P. Bouyer, C. Dufourd, E. Fleury, and A. Petit. Updatable timed automata. *Theoretical Computer Science*, 321(2-3):291–345, 2004.

[11] P. Buchholz, J.-P. Katoen, P. Kemper, and C. Tepper. Model-checking large structured Markov chains. *Journal of Logic and Algebraic Programming*, 56:69–96, 2003.

[12] F. Cassez and O. H. Roux. Structural translation from Time Petri Nets to Timed Automata – Model-Checking Time Petri Nets via Timed Automata. *Journal of Systems and Software*, 79(10):1456–1468, 2006.

[13] D. Cerotti, D. D'Aprile, S. Donatelli, and J. Sproston. Verifying Stochastic Well-formed Nets with CSL model- checking tools. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design (ACSD 2006)*. IEEE, 2006.

[14] D. Cerotti, S. Donatelli, A. Horvath, and J. Sproston. CSL model- checking for Generalized Stochastic Petri Nets. In *Proceedings of the 3rd International Conference on Quantitative Evaluation of Systems (QEST 2006)*. IEEE, 2006.

[15] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic Well-Formed coloured nets for symmetric modelling applications. *IEEE Transaction on Computers*, 42(11):1343–1360, 1993.

[16] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN1.7: GRaphical Editor and Analyzer for Timed and Stochastic Petri Nets. *Performance Evaluation*, 24, 1995. Special Issue on Performance Modelling Tools.

[17] E. M. Clarke and E. A. Emerson. In logic of programs: Workshop. volume 131. Springer Verlag, 1981.

[18] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[19] E. M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks. *ACM Transactions on Programming Languages and Systems*, 19(5):726–750, September 1997.

[20] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[21] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[22] T. Courtney, D. Daly, S. Derisavi, S. Gaonkar, M. Griffith, V. Lam, , and W. H. Sanders. The möbius modeling environment: Recent developments. In *Proceedings of the 1st International Conference on Quantitative Evaluation of Systems (QEST 2004)*, pages 328–329. IEEE Computer Society, 2002.

[23] J. A. Couvillion, R. Freire, R. Johnson, W. Douglas Obal II, M. A. Qureshi, M. Rai, W. H. Sanders, and J. E. Tvedt. Performability modeling with UltraSAN. *IEEE Software*, 8(5):69–80, 1991.

[24] D. D'Aprile, S. Donatelli, A. Sangnier, and J. Sproston. Translating timed automata into time petri net: An untimed approach. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS2007)*, 2007.

[25] D. D'Aprile, S. Donatelli, and J. Sproston. CSL model checking for the GreatSPN tool. In *Proceedings of the 19th International Symposium on Computer and Information Sciences (ISCIS'04)*, volume 3280 of *LNCS*, pages 543–552. Springer-Verlag, 2004.

[26] Department of Information Technology at Uppsala University, Sweden and Department of Computer Science at Aalborg University, Denmark. The UPPAAL tool. http://www.uppaal.com.

[27] S. Donatelli and L. Ferro. Validation of GSPN and SWN models through the PROD tool. In *Proceedings of the 12th International Conference on Modelling Tools and Techniques for Computer and Communication System Performance Evaluation*, volume 2324 of *LNCS*. Springer Verlag, 2002.

[28] E.Clarke, O. Grumberg, and D. Long. Verification Tools for Finite State Concurrent Systems. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency-Reflections and Perspectives*, volume 803, pages 124–175. Springer-Verlag, 1993.

[29] E. A. Emerson and C.-L. Lei. Modalities for model checking (extended abstract): branching time strikes back. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96. ACM Press, 1985.

[30] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. *Real-Time Systems*, 4(4):331–352, 1992.

[31] G. Gardey, O. H. Roux, and O. F. Roux. State space computation and analysis of time Petri nets. *Theory and Practice of Logic Programming (TPLP). Special Issue on Specification Analysis and Verification of Reactive Systems*, 6(3):301–320, 2006.

[32] C. Girault and R. Valk. *Petri Nets for Systems Engineering — A Guide to Modeling, Verification, and Applications.* Springer Verlag, 2003.

[33] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*. PhD thesis, University of Liege, Computer Science Department, PhD Thesis, 1994.

[34] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, 1994.

[35] B. Haverkort, L. Cloth, H. Hermanns, J.-P. Katoen, and C. Baier. Model checking performability properties. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'02)*, pages 103–112. IEEE Computer Society, 2002.

[36] B. Haverkort, H. Hermanns, and J.-P. Katoen. On the use of model checking techniques for dependability evaluation. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, pages 228–237. IEEE Computer Society, 2000.

[37] H. Hermanns, U. Herzog, U. Klehmet, V. Mertsiotakis, and M. Siegle. Compositional performance modelling with the TIPPtool. *Performance Evaluation*, 39(1–4):5–35, 2000.

[38] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle. A tool for model-checking Markov chains. *International Journal on Software Tools for Technology Transfer*, 4(2):153–172, 2003.

[39] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge Unievrsity Press, 1996.

[40] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[41] K. Jensen. High level petri nets. In Pagnoni, A. and Rozenberg, G., editors, *Informatik-Fachberichte 66: Application and Theory of Petri Nets — Selected Papers from the Third European Workshop on Application and Theory of Petri Nets, Varenna, Italy, September 27–30, 1982*, pages 166–180. Springer-Verlag, 1983.

[42] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Monographs in Theoretical Computer Science. Springer-Verlag, 1997.

[43] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33. IEEE Computer Society Press, 1990.

[44] R. M. Karp and R. E. Miller. Parallel program schemata. *Journ. Computer and System Sciences 3*, (2):147–195, May 1969.

[45] J.-P. Katoen, M. Khattri, and I. S. Zapreev. A Markov reward model checker. In *Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, pages 243–244. IEEE Computer Society Press, 2005.

[46] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 2.0: A tool for probabilistic model checking. In *Proc. 1st International Conference on Quantitative Evaluation of Systems (QEST'04)*, pages 322–323. IEEE Computer Society Press, 2004.

[47] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 97–107. ACM Press, 1985.

[48] D. Lime and O. H. Roux. Model checking of time Petri nets using the state class timed automaton. *Journal of Discrete Events Dynamic Systems - Theory and Applications (DEDS)*, 16(2):179–205, 2006.

[49] K. McMillan. *Symbolic model checking: An approach to the state explosion problem*. PhD thesis, Carnegie-Mellon University,Pittsburgh, PhD Thesis, 1992.

[50] M. Menasche and B. Berthomieu. Time petri nets for analyzing and verifying time dependent protocols. *Protocol Specification, Testing and Verification III*, pages 161–172, 1983.

[51] P. M. Merlin and D. J. Farber. Recoverability of communication protocols: Implications of a theoretical study. *IEEE Trans. Comm.*, 24(9):1036–1043, Sept. 1976.

[52] M. Molloy. Performance analysis using Stochastic Petri Nets. *IEEE Transaction on Computers*, 31(9):913–917, 1982.

[53] T. Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, pages 541–580, Apr. 1989.

[54] G. J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., 1979.

[55] D. L. G. of the University of Cape Town. DANAMiCS web page. http://www.cs.uct.ac.za/Research/DNA/DaNAMiCS.

[56] D. Parker. Personal communication, 2005.

[57] Performance Evaluation group of University of Torino. The GreatSPN tool. http://www.di.unito.it/∼greatspn.

[58] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Englewood Cliffs, New Jersey: Prentice Hall, Inc., 1981.

[59] C. A. Petri. *Kommunikation mit Automaten.* Bonn: Institut fr Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[60] M. Pezzé and M. Toung. Time petri nets: A primer introduction. tutorial., 1999.

[61] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[62] PRISM web site. http://www.cs.bham.ac.uk/ dxp/prism.

[63] PROD web site. http://www.tcs.hut.fi/Software/prod/.

[64] C. Ramchandani. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets.* PhD thesis, Cambridge, Mass.: MIT, Dept. Electrical Engineering, PhD Thesis, 1974.

[65] Real-Time Systems Team at IRCCyN, Nantes, CEDEX, France. The ROMEO tool. http://romeo.rts-software.org/.

[66] W. Reisig and G. Rozenberg. *Lectures on Petri Nets I: Basic Models.* Springer Verlag, 1998.

[67] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, 1985.

[68] A. P. Sistla and P. Godefroid. Symmetry and reduced symmetry in model checking. *ACM Trans. Program. Lang. Syst.*, 26(4):702–734, 2004.

[69] SPOT web site. http://spot.lip6.fr/.

[70] The KRONOS working group at VERIMAG, Grenoble, France. The KRONOS tool. http://www-verimag.imag.fr/TEMPORISE/kronos/.

[71] Y. Thierry-Mieg. *Techniques pour le Model-Checking de spécifications de Haut Niveau*. PhD thesis, Laboratoire d'Informatique de Paris 6, 2004.

[72] J. Toussaint, F. Simonot-Lion, and J.-P. Thomesse. Time constraints verification method based on time petri nets. In *Proc. 6th IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems, 29-31 October 1997, Tunis, Tunisia*, pages 262–267, 1997.