Model Checking of Time Petri Nets

Hanifa Boucheneb and Rachid Hadjidj
Department of Computer Engineering, École Polytechnique de Montréal
Canada

1. Introduction

To model time constraints of real time systems various time extensions are proposed, in the literature, for Petri nets. Among these extensions, time Petri nets model (TPN model) is considered to be a good compromise between modeling power and verification complexity. Time Petri nets are a simple yet powerful formalism useful to model and verify concurrent systems with time constraints (real time systems). In time Petri nets, a firing interval is associated with each transition specifying the minimum and maximum times it must be maintained enabled, before its firing. Its firing takes no time but may lead to another marking. Time Petri nets are then able to model time constraints even if the exact delays or durations of events are not known. So, time Petri nets are appropriate to specify time constraints of real time systems which are often specified by giving worst case boundaries. This chapter reviews the well known techniques, proposed in the literature, to model check real time systems described by means of time Petri nets.

Model checking are very attractive and automatic verification techniques of systems (Clarke et al., 1999). They are applied by representing the behavior of a system as a finite state transition system (state space), specifying properties of interest in a temporal logic ($\text{LTL}$, $\text{CTL}$, $\text{CTL}^*$, $\text{MITL}$, $\text{TCTL}$) and finally exploring the state space to determine whether they hold or not. To use model checking techniques with timed models, an extra effort is required to abstract their generally infinite state spaces. Abstraction techniques aim to construct by removing some irrelevant details, a finite contraction of the state space of the model, which preserves properties of interest. For best performances, the contraction should also be the smallest possible and computed with minor resources too (time and space). Several abstractions, which preserve different kinds of properties, have been proposed in the literature for time Petri nets. The preserved properties can be verified using standard model checking techniques on the abstractions (Alur & Dill, 1990); (Penczek & Polrola, 2004); (Tripakis & Yovine, 2001).

This chapter has 6 sections, including introduction and conclusion sections. Section 2 introduces the time Petri nets model and its semantics. Section 3 presents the syntaxes and semantics of temporal logics $\text{LTL}$, $\text{CTL}$, $\text{CTL}^*$ and $\text{TCTL}$. Section 4 is devoted to the TPN state space abstractions. In this section, abstractions proposed in the literature are presented, compared and discussed from state characterization, agglomeration criteria, preserved properties, size and computing time points of view. Section 5 considers a subclass of $\text{TCTL}$ temporal logics and proposes an efficient on-the-fly model checking.
2. Time Petri nets

2.1 Definition of the TPN
A TPN is a Petri net with time intervals attached to its transitions. Formally, a TPN is a tuple \( \mathcal{N} = (P, T, \text{Pre}, \text{Post}, m_0, Is) \) where

- \( P \) and \( T \) are finite sets of places and transitions such that \( P \cap T = \emptyset \),
- \( \text{Pre} \) and \( \text{Post} \) are the backward and the forward incidence functions: \( P \times T \rightarrow \mathbb{N} \),
- \( m_0 \) is the initial marking: \( P \rightarrow \mathbb{N} \),
- \( Is: T \rightarrow \mathbb{Q}^{+} \times (\mathbb{Q}^{+} \cup \{\infty\}) \) associates with each transition \( t \) an interval called the static firing interval of \( t \).

Let \( \text{Int} \) be an interval, \( \downarrow \text{Int} \) and \( \uparrow \text{Int} \) denote respectively its lower and upper bounds.

Let \( M \) be the set of all markings of a TPN, \( m \in M \) a marking.
A transition \( t \) is said to be enabled in \( m \), iff all tokens required for its firing are present in \( m \), i.e.:

\[ \forall p \in P, m(p) \geq \text{Pre}(p,t) \]

Two transitions \( t \) and \( t' \) of \( \text{En}(m) \) are in conflict for \( m \) iff \( (\exists p \in P, m(p) < \text{Pre}(p,t) + \text{Pre}(p,t')) \).

If \( m \) results from firing some transition \( t \) from another marking, \( \text{New}(m,t) \) denotes the set of all transitions newly enabled in \( m \), i.e.:

\[ \text{New}(m,t) = \{ t' \in \text{En}(m) | t' = t \lor (\exists p \in P, m(p)-\text{Post}(p,t) < \text{Pre}(p,t')) \} \]

Note that only one instance of each transition is supposed to be active at the same time. A transition which remains enabled after firing one of its instance is considered newly enabled.

2.2 The semantics of the TPN
There are two known definitions of the TPN state. The first one, based on clocks, associates with each transition \( t \) of the model a clock to measure the time elapsed since \( t \) became enabled most recently (Yoneda & Ryuba, 1998); (Boucheneb & Hadjidj, 2004). The TPN clock state is a pair \( s=(m,v) \), where \( m \) is a marking and \( v: \text{En}(m) \rightarrow \mathbb{R}^{+} \). The initial clock state of the TPN is \( s_0 = (m_0,v_0) \), where \( m_0 \) is the initial marking and \( v_0(t) = 0 \), for all \( t \) in \( \text{En}(m_0) \). The TPN state evolves either by time progression or by firing transitions. When a transition \( t \) becomes enabled, its clock is initialized to zero. The value of this clock increases synchronously with time until \( t \) is fired or disabled by the firing of another transition. \( t \) can fire, if the value of its clock is inside its static firing interval \( Is(t) \). It must be fired immediately, without any additional delay, when the clock reaches \( \uparrow Is(t) \). Its firing takes no time but may lead to another marking. Formally, let \( \theta \in \mathbb{R}^{+} \) be a nonnegative real number, \( t \) a transition of \( T, s=(m,v) \) and \( s'=(m',v') \) two clock states of a TPN.

We write \( s \xrightarrow{\theta} s' \) iff state \( s' \) is reachable from state \( s \) after a time progression of \( \theta \) time units, i.e.: \( m=m', \forall t \in \text{En}(m), v(t)+\theta \leq \uparrow Is(t) \) and \( \forall t \in \text{En}(m), v'(t)=v(t)+\theta \).

We write \( s \xrightarrow{t} s' \) iff state \( s' \) is immediately reachable from state \( s \) by firing transition \( t \), i.e.:

\[ \forall p \in P, \, m'(p)=m(p)-\text{Pre}(p,t)+\text{Post}(p,t), \, t \in \text{En}(m), \, v(t) \geq \downarrow Is(t) \quad \text{and} \quad \forall t' \in \text{En}(m'), \, v(t')= \quad \text{if} \ t' \in \text{New}(m',t) \quad \text{then} \quad 0 \quad \text{else} \quad v(t') \].

\(^1\) \( \mathbb{N} \) is the set of nonnegative integers.
\(^2\) \( \mathbb{Q}^{*} \) is the set of nonnegative rational numbers.
The second characterization, based on intervals, defines the TPN state as a marking and a function which associates with each enabled transition a firing interval (Berthomieu & Vernadat, 2003). The TPN interval state is a couple \( s=(m,I) \), where \( m \) is a marking and \( I: En(m) \rightarrow Q^+ \times (Q^+ \cup \{ \infty \}) \) is an interval function. The initial interval state of the TPN is \( s_0=(m_0, I_0) \), where \( m_0 \) is the initial marking and \( I_0(t)=Is(t) \), for all \( t \in En(m_0) \). When a transition \( t \) becomes enabled, its firing interval is set to its static firing interval \( Is(t) \). The lower and upper bounds of this interval decrease synchronously with time, until \( t \) is fired or disabled by another firing. \( t \) can fire, if the lower bound of its firing interval reaches \( 0 \), but must be fired, without any additional delay, if the upper bound of its firing interval reaches \( 0 \). Its firing takes no time but may lead to another marking. Formally, let \( \theta \in R^+ \) be a nonnegative real number, \( t \) a transition of \( T \), \( s=(m,I) \) and \( s=(m',I') \) two interval states of a TPN.

We write \( s \xrightarrow{\theta} s' \) iff state \( s' \) is reachable from state \( s \) after a time progression of \( \theta \) time units, i.e.: \( m=m' \) and \( \forall t \in En(m), \theta \in \mathcal{I}(t) \wedge \mathcal{I}'(t)=[\max(0, \mathcal{J}(t)-\theta), \mathcal{J}(t)-\theta] \). \( s' \) is also denoted \( s+\theta \).

We write \( s \xrightarrow{t} s' \) iff state \( s' \) is immediately reachable from state \( s \) by firing transition \( t \), i.e.: \( \forall p \in P, m'(p)=m(p)-\text{Pre}(p,t)+\text{Post}(p,t), t \in En(m), \mathcal{J}(t)=0 \), and \( \forall t' \in En(m'), I(t')=I(t) \) if \( t' \in \text{New}(m',t) \) then \( Is(t') \) else \( I(t') \).

The semantics of the TPN can be defined using either the clock state or interval state characterization. In both cases, the TPN state space is defined as a structure \((S, \rightarrow, s_0)\), where \( s_0 \) is the initial clock or interval state of the model, and \( S = \{ s \mid s_0 \xrightarrow{\ast} s \} \) is the set of reachable states (\( \xrightarrow{\ast} \) is the reflexive and transitive closure of the relation \( \rightarrow \) defined above).

Let \( s \) and \( s' \) be two TPN states, \( \theta \in R^+ \) and \( t \in T \). As a shorthand, we write \( s \xrightarrow{\theta} s' \) iff \( s \xrightarrow{\theta} s'' \) and \( s'' \xrightarrow{t} s' \) for some state \( s'' \). We write \( s \xrightarrow{t} s' \) iff \( \exists \theta \in R^+, s \xrightarrow{\theta} s' \).

An execution path in the TPN state space, starting from a state \( s \in S \), is a maximal sequence \( \rho = s^0 \xrightarrow{\theta_0} s'^1 \xrightarrow{\theta_1} \ldots \), such that \( s^0=s \). We denote by \( \pi(s) \) the set of all execution paths starting from state \( s \). \( \pi(s_0) \) is therefore the sets of all execution paths of the TPN. The total elapsed time during an execution path \( \rho \), denoted \( \text{time}(\rho) \), is the sum \( \sum_{i=0}^{\infty} \theta_i \).

An infinite execution path is diverging if \( \text{time}(\rho)=\infty \), otherwise it is said to be zeno. A TPN model is said to be non-zeno if all its execution paths are not zeno. Zenoness is a pathological situation which suggests that infinity of actions may take place in a finite amount of time.

The TPN state space defines the branching semantics of the TPN model, whereas \( \pi(s_0) \) defines its linear semantics. The graph of its execution paths, called concrete state space, is the structure \((\Sigma, \rightarrow, s_0)\) where \( s_0 \) is the initial state of the model, \( \Sigma = \{ s \mid s_0 \xrightarrow{\ast} s \} \) is the set of reachable concrete states of the TPN model, and \( \xrightarrow{\ast} \) is the reflexive and transitive closure of \( \rightarrow \).

### 3. Temporal logics for time Petri nets

Properties of timed systems are usually specified using temporal logics (Penczek & Polrola, 2004). We consider here CTL* (computation tree logic star) and a subclass of TCTL (timed
computation tree logic). Since our goal is to reason about temporal properties of time Petri nets, an atomic proposition is a proposition on a marking. Let $M$ be the set of reachable markings of a TPN model and $PV$ the set of propositions on $M$, i.e., $\{ \phi \mid \phi: M \rightarrow \{true, false\} \}$.

### 3.1 CTL$^*$ and its semantics

CTL$^*$ is a temporal logic which allows to specify both linear and branching properties. The syntax of CTL$^*$ is given by the following grammar:

$$\phi_s \equiv false | \neg \phi_s | \phi_s \land \phi_s | \forall \phi_p | \exists \phi_p$$

$$\phi_p \equiv \phi_s | \neg \phi_p | \phi_p \land \phi_p | X\phi_p | \phi_p U \phi_p$$

In the grammar, $\phi \in PV$ stands for an atomic proposition. $\phi_s$ and $\phi_p$ define respectively formulas that express properties of states and execution paths. $\forall$ ("for all paths") and $\exists$ ("there exists a path") are path quantifiers, whereas $U$ ("until") and $X$ ("next") are temporal operators (path operators). The sublogics CTL and LTL are defined as follows:

- In CTL formulas, every occurrence of a path operator is immediately preceded by a path quantifier: $\phi = false | \neg \phi | \phi \land \phi | \forall(\phi U \phi) | \exists(\phi U \phi) | \forall X \phi | \exists X \phi$

- In LTL, formulas are supposed to be in the form $\forall \phi_p$ where state subformulas of $\phi_p$ are propositions: $\phi = false | \neg \phi | \phi \land \phi | \forall \phi U \phi$ and $G \phi = \neg F \neg \phi$.

CTL$^*$ formulas are interpreted on states and execution paths of a model $M = (S, V)$, where $S=(S, \rightarrow, s_0)$ is a transition system and $V: S \rightarrow 2^{PV}$ is a valuation function which associates with each state the subset of atomic propositions it satisfies.

Let $s \in S$ be a state of $S$, $\pi(s)$ the set of all execution paths starting from $s$, and $\rho = s^0 \xrightarrow{\theta_{s^0}} s^1 \xrightarrow{\theta_{s^1}} s^2 \xrightarrow{\theta_{s^2}} s^3 \ldots$ an execution path with $\rho$ its suffix starting from $s^i$. The formal semantics of CTL$^*$ is given by the satisfaction relation $\models$ defined as follows (the expression $M, s \models \phi$ is read: "$s$ satisfies property $\phi$ in the model $M$"):

- $M, s \not\models false$,
- $M, s \models \phi$ iff $\phi \in V(s)$,
- $M, x \not\models \neg \phi$ iff $M, x \not\models \phi$ for $x \in \{s, \rho\}$,
- $M, x \models \phi \land \psi$ iff $M, x \models \phi$ and $M, x \models \psi$ for $x \in \{s, \rho\}$,
- $M, s \models \forall \phi$ iff $\forall \rho \in \pi(s)$, $M, \rho \models \phi$,
- $M, s \models \exists \phi$ iff $\exists \rho \in \pi(s)$, $M, \rho \models \phi$,
- $M, \rho \models \phi$ iff $M, s^0 \models \phi$, for a state formula $\phi$,
- $M, \rho \models X \phi$ iff $M, \rho^1 \models \phi$,
- $M, \rho \models \phi U \psi$ iff $\exists j \geq 0$, $M, \rho^j \models \psi$ and $\forall 0 \leq i < j$, $M, \rho^i \models \phi$.

We say that $M$ satisfies $\phi$, written $M \models \phi$, iff $M, s_0 \models \phi$. For instance $M, s_0 \models (\forall \phi U \psi)$, iff for any execution path starting from $s_0$, $\phi$ is true in $s_0$ and the following states, until a state that satisfies $\psi$ is reached.
To be able to specify explicitly time constraints of some important real-time properties such as, for example, the bounded response property, timed versions have been proposed for these logics (MITL, TCTL). Among these logics, we consider here a subclass of TCTL logic for an on-the-fly model checking.

### 3.2 TCTL and its semantics

TCTL is a timed extension of CTL (computation tree logic) where a time interval is associated with each temporal operator. The syntax of TCTL formulas is defined by the following grammar (in the grammar, $\phi \in PR$ and index $I$ is an interval of $Q^+ \times (Q^+ \cup \{\infty\})$):

$$\phi \equiv \text{false} \mid \phi \mid \neg \phi \mid \phi \land \psi \mid \forall (\phi U_{I} \psi) \mid \exists (\phi U_{I} \psi)$$

TCTL formulas are also interpreted on states and execution paths of a model $M = (S,V)$. To interpret a TCTL formula on an execution path, we introduce the notion of dense execution path. Let $s \in S$ be a state of $S$, $\pi(s)$ the set of all execution paths starting from $s$, and $\rho = s^0 \xrightarrow{a_0} s^1 \xrightarrow{a_1} \ldots$ an execution path of $s$. The dense path of the execution path $\rho$ is the mapping $\hat{\rho} : R^+ \rightarrow S$ defined by:

$$\delta_{\rho} = \sum_{r=0}^{i-1} \theta_j + \delta, i \geq 0 \text{ and } 0 \leq \delta \leq \theta_i.$$

The formal semantics of TCTL is given by the satisfaction relation $=\text{def}$ as follows:

- $M, s \not\models \phi$ if $\phi \notin V(s)$,
- $M, s \models \neg \phi$ if $M, s \not\models \phi$,
- $M, s \models \phi \land \psi$ if $M, s \models \phi$ and $M, s \models \psi$,
- $M, s \models \forall (\phi U_{I} \psi)$ if $\forall \rho \in \pi(s), \exists r \in I, M, \hat{\rho}(r) \models \psi$ and $\forall 0 \leq r' < r, M, \hat{\rho}(r') \not\models \phi$.
- $M, s \models \exists (\phi U_{I} \psi)$ if $\exists \rho \in \pi(s), \exists r \in I, M, \hat{\rho}(r) \models \psi$ and $\forall 0 \leq r' < r, M, \hat{\rho}(r') \not\models \phi$.

The TPN model is said to satisfy a TCTL formula $\phi$ iff $M, s_0 \models \phi$. When interval $I$ is omitted, its value is $[0, \infty]$ by default. Our timed temporal logic, we call $TCTL_{TPN}$, is defined as follows:

$$TCTL_{TPN} \equiv \forall (\phi m U_{I} \psi m) \mid \exists (\phi m U_{I} \psi m) \mid \phi m \Rightarrow \psi m \mid \psi m \Rightarrow \phi m \mid \forall G_{I} \phi m \mid \forall \exists_{I} \phi m \mid \forall F_{I} \phi m$$

$$\phi m \equiv \phi m \land \phi m \mid \phi m \lor \phi m \mid \neg \phi m \mid \phi \mid \text{false}$$

$\phi$ is a proposition on markings (i.e., $\phi \in PR$). $I$ is a time interval. $I_r$ is a time interval which starts from 0. Formula $\phi_1 \Rightarrow \phi_2$ is a shorthand for TCTL formula $\forall G(\phi_1 \Rightarrow \forall F_{I_r} \phi_2)$ which expresses a bounded response property.

Several efficient model checking techniques were developed in the literature for LTL, CTL, CTL*, MITL and TCTL, using timed Büchi automata, fix point techniques, or hesitant alternating automata (Penczek & Polrola, 2004); (Tripakis et al., 2005); (Tripakis et al., 2001); (Visser & Barringer, 2000). To apply these techniques to time Petri nets, we must construct a finite abstraction for its generally infinite state space which preserves properties of interest.
4. TPN state space abstractions

Abstraction techniques aim to construct by removing some irrelevant details, a finite contraction of the state space of the model, which preserves properties of interest (markings, linear or branching properties). The preserved properties are then verified on the contraction using the classical model checking techniques. The challenge is to construct, with less resources (time and space), a much coarser abstraction preserving properties of interest.

4.1 Abstract state space

An abstract state space of the TPN model is defined as a structure $AS=(A, \Rightarrow, \alpha_0)$ where (Boucheneb & Hadjidj, 2006):

- $A$ is a cover of $S$ or $\Sigma$. Each element $\alpha$ of $A$, called abstract state, is an agglomeration of some states sharing the same marking.
- $\alpha_0$ is the initial abstract state class of $AS$, such that $s_0 \in \alpha_0$, and
- $\Rightarrow \subseteq A \times T \times A$ is the successor relation that satisfies condition $EE$, i.e.:
  
  i. $\forall (\alpha, t, \alpha') \in \Rightarrow, \exists s \in \alpha, \exists s' \in \alpha', s \xrightarrow{t} s'$ and
  
  ii. $\forall (s, t, s') \in \Rightarrow, \forall \alpha, s.t. s \in \alpha, \exists \alpha' \in A, (s' \in \alpha' \land \alpha \Rightarrow \alpha')$

The first part of condition $EE$ prevents the connection of two abstract states with no connected states. The second one ensures that all sequences of transitions in the state space are represented in the abstraction.

Note that there are some differences between condition $EE$ presented here and those given in (Berthomieu & Vernadat, 2003) and (Penczek & Polrola, 2004):

- $EE$ in (Berthomieu & Vernadat, 2003):
  
  $$\forall (\alpha, t, \alpha') \in A \times T \times A, \quad [(\exists s \in \alpha, \exists s' \in \alpha', s \xrightarrow{t} s') \Rightarrow \alpha \Rightarrow \alpha']$$

- $EE$ in (Penczek & Polrola, 2004):
  
  $$\forall (\alpha, t, \alpha') \in A \times T \times A, \quad [(\exists s \in \alpha, \exists s' \in \alpha', s \xrightarrow{t} s') \Rightarrow \alpha \Rightarrow \alpha']$$

Theses conditions impose to connect each two abstract states $\alpha$ and $\alpha'$ whenever some state of the first one has a successor in the second one. However, TPN abstractions proposed in the literature do not obey this rule, while they are still valid. As an example, consider the TPN with its Strong State Class Graph SSCG\(^4\) shown in figure 1. Inequalities associated with each abstract state characterize the clock domains of all states agglomerated in the abstract state. The abstract state $\alpha_3$ has a state which has a successor by $t_0$ in $\alpha_5$, but no transition by $t_0$ exists from $\alpha_3$ to $\alpha_5$. In fact, the transition from $\alpha_3$ to $\alpha_1$ by $t_0$ ensures that some state in $\alpha_3$ has as successor by $t_0$ the unique state of $\alpha_1$. However, there is no transition from $\alpha_3$ to $\alpha_5$ by $t_0$, while the unique state of $\alpha_1$ belongs also to $\alpha_5$. This situation contradicts conditions $EE$ given in (Berthomieu & Vernadat 2003) and (Penczek & Polrola, 2004).

\(^3\) A cover of a set $X$ is a collection of sets whose union contains $X$.

\(^4\) The SSCG is a TPN abstraction proposed in (Berthomieu & Vernadat, 2003).
The relation \( \Rightarrow \) may satisfy other additional conditions such as (see figure 2):

**EA:** \( \forall (\alpha, t, \alpha') \in \Rightarrow, \forall s\in\alpha', \exists s\in\alpha, s\xrightarrow{t} s' \),

**AE:** \( \forall (\alpha, t, \alpha') \in \Rightarrow, \forall s\in\alpha, \exists s'\in\alpha', s\xrightarrow{t} s' \)

---

Fig. 1. A TPN model and its SSCG.
Fig. 2. Conditions $EE$, $EA$ and $AE$.

A state class space which satisfies condition $AE$ is called an atomic state class graph. An abstract state which satisfies condition $AE$ for each outgoing edge is said to be atomic. The theorem below establishes a relation between conditions $AE$, $EA$ and properties of the model preserved in the abstraction.

**Theorem** (Boucheneb & Hadjidj, 2006): Let $AS=(A, \rightarrow, \alpha_0)$ be an abstraction of a TPN. Then:

1. If $AS$ satisfies condition $EA$ and $\alpha_0 = \{s_0\}$ then $AS$ preserves LTL properties of the TPN,
2. If $AS$ satisfies condition $AE$ then it preserves CTL* properties of the TPN.

### 4.2 Abstract states

We can find, in the literature, several state space abstractions for the TPN model: the state class graph SCG (Berthomieu & Vernadat, 2003), the zone based graph ZBG (Gardey & Roux, 2003), the geometric region graph GRG (Yoneda & Ryuba, 1998), the strong state class graph SSG (Berthomieu & Vernadat, 2003) and the atomic state class graphs ASCGs (Boucheneb & Hadjidj, 2006); (Berthomieu & Vernadat, 2003); (Yoneda & Ryuba, 1998). These abstractions may differ mainly in the characterization of abstract states (interval states (Berthomieu & Vernadat, 2003), clock states (Boucheneb & Hadjidj, 2006) or firing dates (Yoneda & Ryuba, 1998), the agglomeration criteria of states, the kind of properties they preserve and their size.

In all these abstractions except the GRG, abstract states are defined as a couple $\alpha=\langle m,f \rangle$, where $m$ is a marking and $f$ is a conjunction of atomic constraints of the form $x-y@c$, $-x<y@c$ or $x<y@c$, where $c \in \mathbb{Q}\cup[-\infty,\infty]$, $< \in \{=, \leq, \geq, <, >\}$, and $x, y$ are time variables. Each transition enabled in $m$ is represented in $f$ by a time variable, with the same name, representing either its delay or its clock ($\text{Var}(f) = \text{En}(m)$). All time variables are either clocks (clock abstract states) or delays (interval abstract states). Time variables are clocks in the SSG, ZBG and ASCGs (clock state abstractions) but they are delays in the SCG (an interval state abstraction). Abstract states of the SCG, SSG, ZBG and ASCGs are respectively called state classes, strong state classes, state zones and atomic state classes.

Abstract states of the GRG are triples $(m,f,\eta)$, where $m$ is the marking obtained by firing from the initial marking $m_0$ the sequence of transitions $\eta$ and $f$ is a set of atomic constraints on firing dates of transitions in $m$ and their parents (transitions of $\eta$ which made transitions of $\text{En}(m)$ enabled). This definition needs more time variables and constraints. It is therefore less interesting than those used in other abstractions. In addition, the relation of equivalence used in GRG involves large graphs and may induce infinite abstractions for some time Petri
nets with unbounded firing intervals (Berthomieu & Vernadat, 2003). Two abstract states are equivalent if they have the same marking, their enabled transitions have the same parents, and these parents could be fired at the same dates. For all these reasons, we do not consider here the abstract state definition of the GRG.

Though the same domain may be expressed by different conjunctions of atomic constraints, equivalent formulas have a unique form, called canonical form. Canonical forms make operations needed to compute and compare abstract states more simple. Let \( f \) be a conjunction of atomic constraints. The canonical form of \( f \) is:

\[
f = \bigwedge_{x,y \in (\text{Var}(f) \cup \{o\})} x - y \prec_{f}^{x-y} \text{Sup}_{f}(x-y)
\]

where \( \text{Var}(f) \) is the set of time variables of \( f \), \( o \) represents the value zero, \( \text{Sup}_{f}(x-y) \) is the supremum of \( x-y \) in the domain of \( f \), \( \prec_{f}^{x-y} \) is either \( \leq \) or \( < \), depending respectively on whether \( x-y \) reaches its supremum in the domain of \( f \) or not. \( \text{Dom}(f) \) denotes the domain of \( f \). By convention, we suppose that \( \text{Is}(o)=[0,0] \).

The canonical form of \( f \) is usually represented by a DBM (Daws et al., 1996) of order \( |\text{Var}(f)|+1 \), defined by:

\[
\forall x, y \in (\text{Var}(f) \cup \{o\}), B_{xy} = (\text{Sup}_{f}(x-y), \prec_{f}^{x-y})
\]

An element of a DBM is called a bound. Operations like \( +, \gamma, <, \leq, \geq, = \), and min on bounds of DBMs are defined as usual: \( \forall (c_1, \prec_1), (c_2, \prec_2) \in B, \forall \prec \in \{\leq, <, =, \geq\}, \)

- \((c_1, \prec_1) < (c_2, \prec_2) \) iff \( (c_1 < c_2 \land (c_1 = c_2 \Rightarrow \prec_1 < \prec_2)) \); ("\( < \)" is less than operator "\( \leq \)."
- \((c_1, \prec_1) + (c_2, \prec_2) = (c_1 + c_2, \min(\prec_1, \prec_2))\);
- \((c_1, \prec_1) - (c_2, \prec_2) = (c_1 - c_2, \min(\prec_1, \prec_2))\);
- \(\neg(c_1, \prec_1) = (-c_1, \neg\prec_1)\)
- \(\min((c_1, \prec_1), (c_2, \prec_2)) = \text{if} (c_1, \prec_1) \leq (c_2, \prec_2) \text{ then } (c_1, \prec_1) \text{ else } (c_2, \prec_2)\)

The computation of canonical forms is based on the shortest path Floyd-Warshall’s algorithm and is considered as the most costly operation (cubic in the number of variables in \( f \)) (Berhmann et al., 2002). In (Boucheneb & Mullins, 2003) and (Boucheneb & Hadjidij, 2006), authors have shown how to compute, in \( O(n^2) \), for respectively the SCG and the SSCG, the canonical form of each successor abstract state, \( n \) being the number of variables in the abstract state. An abstract state is said in canonical form iff its formula is in canonical form.

The convexity of abstract states is an important criterion to maintain their computation simple. The simplicity of the method is particularly guaranteed by the usage of DBMs. This data structure adapts well to all computation aspects involved in constructing abstractions, but fails to efficiently represent non convex domains (DBMs are not closed under set-union).

To avoid this limitation, \textit{Clock Difference Diagrams} (CDDs) (Larsen et al., 1999) seems to be a better alternative. CDDs allow to represent in a very concise way the union of convex domains. They are also closed under set-union, intersection and complementation. However, due to the lack of a known simple computing canonical form, CDDs fail to compete with DBMs when it comes to computing successors and predecessors of abstract
states. A detailed description of CDDs can be found in (Berhmann et al., 2002) and (Larsen et al., 1999), where the authors use this data structure to represent computed state zones in the list PASSED of the reachability algorithm, implemented in the tool UPPAAL. Yet, they still use DBMs to compute successors of abstract states. Note that abstract states within the list PASSED are handled using only two basic operations which are well supported by CDDs (set-union and inclusion).

4.3 Abstractions preserving linear properties
An abstraction is said to preserve linear properties if it has exactly the same firing sequences as its concrete state space. In abstractions preserving linear properties, we distinguish, in general, three levels of abstraction (see figure 3). In the first level, states reachable by time progression may be either represented (ZBG) or abstracted (SCG, GRG, SSCG). In the second level, states reachable by the same firing sequence independently of their firing times are agglomerated in the same node. In the third level, the agglomerated states are then considered modulo some relation of equivalence (firing domain of the SCG (Berthomieu & Vernadat, 2003), the approximations of the ZBG (Gardey & Roux, 2003) and the SSCG (Berthomieu & Vernadat, 2003)). These abstractions, except the GRG, are finite for all bounded time Petri nets. Indeed, for some bounded TPNs with unbounded static firing intervals, the GRG may be infinite. However, in (Pradubsuwun et al., 2005), authors used the approximation of timed automata to ensure the convergence of the construction of the GRG for bounded TPNs with unbounded static firing intervals.

![Fig. 3. Different levels of abstraction](image)

4.3.1 Basic operations on abstract states
Let $\alpha$ be an abstract state and $t$ a transition of $T$. We define the basic operations on $\alpha$, used to construct abstractions preserving linear properties:

- $\text{succ}(\alpha, t) = \{s' | \exists s \in \alpha, s \xrightarrow{t} s'\}$ is the set of all states reachable from $\alpha$ by firing immediately transition $t$. 

www.intechopen.com
\[ \bar{\alpha} = \{ s' | \exists s \in \alpha, \exists \theta \in \mathbb{R}^+, s \xrightarrow{\theta} s' \} \]

contains \( \alpha \) and all states reachable from \( \alpha \) via some time progression.

Let us now show how to compute successor abstract states for clock abstract states and for interval abstract states.

Let \( \alpha = (m,f) \) be a clock abstract state in canonical form, and \( t \) a transition of \( T \):
\[ \text{suc}(\alpha,t) \not\emptyset \iff t \in \text{En}(m) \text{ and } f \land \downarrow \text{Is}(t) \leq t \] is consistent. This means that there is at least a state in \( \alpha \) such that \( t \) is firable from it (its clock reaches its static firing interval).

If \( \text{suc}(\alpha,t) \not\emptyset \) then \( \text{suc}(\alpha,t) = (m',f') \) is computed in four steps:
1. \( \forall p \in P, \quad m'(p) = m(p) - \text{Pre}(p,t) + \text{Post}(p,t) \).
2. Initialize \( f' \) with \( f \land \downarrow \text{Is}(t) \leq t \). This step eliminates from \( f \) states from which \( t \) is not immediately firable.
3. Put \( f' \) in canonical form and eliminate \( t \) and all transitions conflicting with \( t \) in \( m \).
4. Add constraints \( t' = 0 \text{ and put } f' \text{ in canonical form (clocks of newly enabled transitions are set to 0).} \)

Let \( \alpha = (m,f) \) be an interval abstract state in canonical form, and \( t \) a transition of \( T \):
\[ \text{suc}(\alpha,t) \not\emptyset \iff t \in \text{En}(m) \text{ and } \downarrow \text{Is}(t) \leq t \] is consistent. This means that there is at least a state in \( \alpha \) such that \( t \) is firable from it (its delays is equal to 0).

If \( \text{suc}(\alpha,t) \not\emptyset \) then \( \text{suc}(\alpha,t) = (m',f') \) is computed in four steps:
1. \( \forall p \in P, \quad m'(p) = m(p) - \text{Pre}(p,t) + \text{Post}(p,t) \).
2. Initialize \( f' \) with \( t = 0 \). This step eliminates from \( f \) states from which \( t \) is not immediately firable.
3. Put \( f' \) in canonical form and eliminate \( t \) and all transitions conflicting with \( t \) in \( m \).
4. Add constraints \( t' \) and put \( f' \) in canonical form. The firing interval of each newly enabled transition is set to its static firing interval.

\[ \bar{\alpha} = (m, f') \] is computed in three steps:
1. Initialize \( f' \) with \( f \);
2. Replace each constraint \( o - t \leq c \) with \( o - t \leq 0 \). Delays decrease with time until reaching 0 or their transitions are fired or disabled.
3. Put \( f' \) in canonical form.

\[ \text{4.3.2 Approximation of clock abstract states} \]
Let \( \alpha = (m,f) \) be a clock abstract state in canonical form and \( \text{En}_{=\infty}(m) = \{ t \mid t \in \text{En}(m) \land \uparrow \text{Is}(t) = \infty \} \) the set of unbounded transitions enabled in \( m \).
The SSCG approximation of \( \alpha \) denoted \( \text{approx}_{\text{SSCG}}(\alpha) \) produces a partition of \( \alpha \) \( \{(m, f) \mid (e \subseteq \text{En}_{\alpha}(m) \vee e = \emptyset) \wedge f_e \text{ is a consistent formula characterizing states of (m, f)} \} \), where \( f_e \) is a consistent formula characterizing states of \( (m, f) \) in which all transitions of \( e \) have not yet reached their minimal delays, while those of \( \text{En}_{\alpha}(m) - e \) have reached or over-passed their minimal delays. \( f_e \) is computed in three steps:

1. Initialize \( f_e \) with: 
   \[
   f_e = \bigwedge_{t \in e} (t \leq \text{Is}(t)) \wedge (t' \geq \text{Is}(t'));
   \]

2. Put \( f_e \) in canonical form and eliminate all variables in \( \text{En}_{\alpha}(m) - e \);

3. Add the constraint 
   \[
   \bigwedge_{t' \in \text{En}_{\alpha}(m) - e} (t' \leq \text{Is}(t') \leq t).
   \]

Steps (2) and (3) extend \( f_e \) with possibly non reachable states when replacing the domain of each variable \( t' \) in \( \text{En}_{\alpha}(m) - e \) by \( [\text{Is}(t'), \infty] \). Nevertheless, these states correspond all to the same interval state (Berthomieu & Vernadat, 2003). Therefore, this operation preserves linear properties of the abstract state \( \alpha \).

Let \( k \) be the greatest finite bound appearing in the static firing intervals of the considered TPN. The ZBG approximation of \( \alpha \) proposed in (Gardey & Roux, 2003), denoted \( \text{approx}_k(\alpha) \), is the abstract state \( (m, f') \) where \( f' \) is the canonical form of the formula computed from \( f \) as follows: For each \( t \in \text{En}_{\alpha}(m) \), \( x \in \text{En}(m) \cup \{o\} \),

1. Replace constraint \( x - t < c \) with \( x - t \leq k \) if \( c < k \);
2. Remove constraint \( t - x < c \) if \( k < c \).

This operation extends \( f \) with possibly non reachable states but the added states do not alter linear properties of the abstract state \( \alpha \) (Gardey & Roux, 2003). In (Boucheneb et al., 2006), authors proposed two other approximations for the ZBG, denoted respectively \( \text{approx}_{kx} \) and \( \text{approx}_{kx'} \) which lead to much compact graphs. They showed that \( \alpha \), \( \text{approx}_{kx}(\alpha) \) and \( \text{approx}_{kx'}(\alpha) \) have the same firing domain and then the same firing sequences.

\( \text{approx}_{kx}(\alpha) \) is the abstract state \( (m, f') \) where \( f' \) is the canonical form of the formula computed from \( f \) as follows: For each \( t \in \text{En}_{\alpha}(m) \), \( x \in \text{En}(m) \cup \{o\} \),

1. Replace constraint \( x - t < c \) with \( x - t \leq 0 \) if \( t \in \text{En}_{\alpha}(m) \);
2. Remove constraint \( t' - t < c' \) if \( t' \in \text{En}_{\alpha}(m) \) or the constraint \( o - t < c' \) is s.t. \( c' - \text{Is}(t') \geq c \).

\( \text{approx}_{kx'}(\alpha) \) has been integrated recently in the tool Romeo5 in replacement of the one proposed in (Gardey & Roux, 2003). This approximation is referred in the sequel as \( \text{approx}_{\text{ZBG}} \).

### 4.3.3 Construction of abstractions preserving linear properties

An abstraction preserving linear properties is generated progressively by computing the successors of the initial abstract states and those of each newly computed abstract state, until no more new abstract states are generated. All computed abstract states are considered
modular some relation of equivalence. In table 1, we give the formal definition of the SCG, ZBG and SSCG from which the construction algorithms can be derived.

<table>
<thead>
<tr>
<th>AS</th>
<th>SCG</th>
<th>ZBG</th>
<th>SSCG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial abstract state</td>
<td>$f_0 = \bigwedge_{i \in E_{in}(m)} \downarrow I_s(t) \leq t \leq I_s(t)$</td>
<td>$f_0 = \bigwedge_{i \in E_{in}(m)} t = 0$</td>
<td>$f_0 = \bigwedge_{i \in E_{in}(m)} t = 0$</td>
</tr>
<tr>
<td>$(\alpha, t, \alpha') \in \Rightarrow_{AS}$</td>
<td>$\text{succ}(\alpha, t) \neq \emptyset \land \alpha' = \text{succ}(\alpha, t)$</td>
<td>$\text{succ}(\alpha, t) \neq \emptyset \land \alpha' = \text{approx}_{\text{ZBG}}(\text{succ}(\alpha, t))$</td>
<td>$\text{succ}(\alpha, t) \neq \emptyset \land \alpha' = \text{approx}_{\text{SSCG}}(\text{succ}(\alpha, t))$</td>
</tr>
<tr>
<td>$A$</td>
<td>${\alpha \mid a_0 \Rightarrow_{\text{SCG}} a}$</td>
<td>${\alpha \mid \text{approx}<em>{\text{ZBG}}(a_0) \Rightarrow</em>{\text{ZBG}} a}$</td>
<td>${\alpha \mid a_0 \Rightarrow_{\text{SSCG}} a}$</td>
</tr>
</tbody>
</table>

Table 1. Definition of SCG, ZBG and SSCG.

4.3.4 Interval state abstractions versus clock state abstractions
Clock based abstractions are less interesting than the SCG when only linear properties are of interest. They are in general larger, and their computation takes more time. The origin of these differences stems from the relationship between the two characterizations of states which can be stated as follows: Let $(m, v)$ be a clock state. Its corresponding interval state is $(m, l)$ s.t. $\forall t \in E_{in}(m), I(t) = [\max(0, \Delta Is(t) - v(t)), \Delta Is(t) - v(t)]$. Note that for any real value $u \geq \Delta Is(t)$, if $\Delta Is(t) = \infty$, $\Delta Is(t) - u = \infty$ and $\max(0, \Delta Is(t) - u) = 0$. This means that many clock states may map to the same interval state. In such a case, all these states will obviously exhibit the same future behaviour. The same remark extends also to interval abstract states and clock abstract states. As an example, consider the model shown in figure 4.a. The repetitive firing of transition $I_0$ from the initial abstract state, generates 2 strong state classes $sc_1$ and $sc_2$ (figure 4.c) which map to the state class $c_1$ (figure 4.b). Moreover, the number of strong state classes which map to $c_1$ depends and increases with the value of $\Delta Is(t)$. For example, for $\Delta Is(t) = 9$, we obtain 5 strong state classes which correspond to the state class $c_1$.

Moreover, abstractions based on clocks do not enjoy naturally the finiteness property for bounded TPNs with unbounded intervals as it is the case for abstractions based on intervals. The finiteness is enforced using an approximation operation on clock abstract states, which may involve some overhead computation. Another point which contributes to generate coarser abstractions concerns states reachable by time progression. We obtain coarser abstractions when we add to each abstract state all states reachable from it by time progression (relaxing abstract states). Indeed, two different abstract states may have the same relaxed abstract state. As an example, the two SCG state classes $\alpha = (m, 2 \leq t \leq 3)$ and $\alpha_2 = (m, 1 \leq t \leq 3)$ are s.t. $\alpha_1 \neq \alpha_2$ and $\alpha_1 = \alpha_2 = (m, 0 \leq t \leq 3)$. To achieve more contractions, we define a relaxed version to the SCG, named relaxed state class graph (RSCG), as a structure $(A, \Rightarrow_{RSCG}, \alpha_0)$ where:

1. $\alpha_0 = (m_0, f_0)$ where $m_0$ is the initial marking and $f_0 = \bigwedge_{t \in E_{in}(m)} \downarrow I_s(t) \leq t \leq I_s(t)$.
2. $\forall (\alpha, \alpha', t, (\alpha, t, \alpha') \in \Rightarrow_{RSCG}$ iff $\text{succ}(\alpha, t) \neq \emptyset$ and $\alpha' = \text{succ}(\alpha, t)$.
3. $A = \{\alpha \mid \alpha_0 \Rightarrow_{RSCG} \alpha\}$.

However, abstractions based on intervals are not appropriate for constructing abstractions preserving branching properties (ASCGs). Indeed, this construction, based on splitting abstract states, is not possible on state classes (the union of intervals is irreversible) whereas it is possible on clock abstract states. Together, the mentioned remarks suggest that the
interval characterization of states is more appropriate to construct abstractions preserving linear properties but is not appropriate to construct abstractions preserving branching properties.

We have implemented and tested several abstractions. We report in table 2 sizes (nodes/edges) and computing times of the RSCG, SCG, SSCG and ZBG we obtained for the producer consumer model (figure 5) and the level crossing model (figure 6). The level crossing model $T(n)$ is obtained by putting in parallel one copy of the controller model, $n$ copies of the train model (with $m = n$) and one copy of the barrier model. Trains and the barrier are synchronized with the controller on transitions with the same names. The producer consumer model $P(n)$ is the parallel composition of $n-1$ copies of the model in figure 6.b with one copy of the model in figure 6.a while merging all places named $P_1$ in one single place. The obtained results confirm that the RSCG is in general smaller and faster to compute too.

![Fig. 4. Example showing the abstracting power of the interval state abstraction.](image)

![Fig. 5. The crossing level model](image)
4.4 Abstractions preserving branching properties

Abstractions preserving branching properties (CTL* properties) are built using a partition refinement technique in two steps (Paige & Tarjan, 1987). An abstraction, which does not necessarily preserve branching properties, is first built then refined in order to restore the condition $AE$ (the resulting graph is atomic).

4.4.1 Refinement process

Let $AS = (A, \Rightarrow, a_0)$ be an abstract state space of a TPN model, $\alpha = (m, f)$, $\alpha' = (m', f')$ two abstract states of $A$, $t$ a transition of $T$ s.t. $(\alpha, t, \alpha') \in \Rightarrow$ and $\text{pred}(\alpha', t, \alpha) = \{s \in \alpha \mid \exists s' \in \alpha', s \xrightarrow{t} s'\}$. To verify the atomicity of $\alpha$ for the edge $(\alpha, t, \alpha')$, it suffices to verify that $\alpha$ is equal or included

---

The computation has not completed after an hour of time, or aborted due to a lack of memory.
in $\text{pred}(\alpha', t, \alpha)$. In case $\alpha$ is not atomic, it is partitioned into a set of convex subclasses so as to isolate the predecessors of $\alpha'$ by $t$ in $\alpha$, from those which are not.

$\text{Pred}(\alpha', t, \alpha)=(m,f')$ is computed in five steps:

1. Initialize $f'$ to $f' = \bigwedge_{t' \in \text{New}(m', t)} t' = 0$,

2. Put $f'$ in canonical form and eliminate by substitution all transitions in $\text{New}(m', t)$,

3. Add constraints: $\forall t \in \text{Is}(t) \leq t, \bigwedge_{t' \in \text{En}(m')} t' \leq t'$ and $\theta \geq 0$,

4. Replace each variable $t$ by $t + \theta$, put $f''$ in canonical form then eliminate $\theta$,

5. Add all constraints of $f$ and put $f''$ in canonical form.

Knowing that the firing of transition $t$ sets the clock of each newly enabled transition to zero, step (1) extracts from $\alpha'$ the subset of states where the clocks of newly enabled transitions are equal to zero. Step (3) adds the firing constraints of transition $t$. Step (4) goes back in time (each clock is decreased by $\theta$ time units). Finally, step (5) adds all constraints of class $\alpha$.

Since the domain of the difference is not necessarily convex, we construct a partition of $\alpha' - \text{Pred}(\alpha', t, \alpha)$ such that all its parts are convex. Let $\alpha = (m, f)$ and $\alpha' = (m, f')$ be two abstract states such that $\alpha' \subseteq \alpha$. A partition of the complement of $\alpha'$ in $\alpha$, denoted $\text{Comp}(\alpha, \alpha')$, is computed as follows:

Algorithm $\text{Comp}(\alpha = (m, f), \alpha' = (m', f'))$

\[
\text{Part} := \emptyset;
\]
\[
X := f;
\]
For each atomic constraint $g$ of $f$

\[
\text{if } (X \land \neg g) \text{ is consistent then } \text{Part} := \text{Part} \cup \{(m, X \land \neg g)\};
\]
\[
X := (X \land f);
\]

Return Part;

The refinement proceeds according to the following algorithm: After its splitting, $\alpha$ is replaced by its partition. Each subclass inherits all connections of $\alpha$ in accordance with condition $\text{EE}$. The refinement step is repeated until condition $\text{AE}$ is established. The refinement process generates a finite graph iff the intermediate abstraction is finite (Berthomieu & Vernadat, 2003).

Algorithm $\text{Refine}(\text{AS})$

\[
\text{Repeat } \{ \text{For each } \alpha \in \text{A such that } \alpha \text{ is not atomic for some transition } \alpha \xrightarrow{t} \alpha' \}
\]
\[
\alpha'' := \text{Pred}(\alpha', t, \alpha);
\]
\[
\text{Part} := \text{Comp}(\alpha, \alpha'');
\]
\[
\text{Part} := \text{Part} \cup \{\alpha''\};
\]
\[
\text{Replace } \alpha \text{ by Part in AS};
\]
\[
\} \text{ while (AS is not atomic)}
\]
4.4.2 Intermediate abstractions
The intermediate abstractions used in (Yoneda & Ryuba, 1998) (GRG) and (Berthomieu & Vernadat, 2003) (SSCG) preserve linear properties. However, these abstractions are in general large graphs with a high degree of state redundancy (the same state may appear in several abstract states). Experimental results showed that this redundancy induces the refinement procedure to waste time and space computing redundant abstract states. For instance, if an abstract state is included into another one, refining both abstract states may result in identical atomic abstract states. If both abstract states are replaced by the most including one, no pertinent information will be lost while refinement steps get reduced. To reduce state redundancy in abstraction preserving linear properties, we proposed to group together abstract states whenever one of them includes all the others (Boucheneb & Hadjidj, 2006) or their union is convex (Boucheneb & Hadjidj, 2004). When a set of abstract states are grouped, they are replaced by a new abstract state representing their union. All transitions between these abstract states become loops for their union. Ingoing and outgoing transitions of the grouped abstract states become respectively ingoing and outgoing of their union. If one of the grouped abstract states contains the initial abstract state, their union becomes the initial abstract state. The contraction may be performed either during or at the end of the construction. With these abstractions, we obtain an important reduction in refinement times and memory usage, resulting in graphs closer in size to the optimal (see table 3). Despite the simplicity of the used models, they allowed to illustrate some interesting features related to the computation pattern followed by the refinement procedure, depending on which abstraction is refined (see figure 7). If an inclusion or convex-combination abstraction is used, the refinement follows a linear pattern (i.e., the size of the graph grows linearly in time during its construction). When an abstraction preserving linear properties is refined, the size of the computed graph starts first to grow up to a peek size then decreases until an atomic state class space is obtained. In certain cases, the peek size grows out of control, leading to a state explosion.

The inclusion test is performed as follows: Let \( \alpha=(m,f) \) and \( \alpha'=(m,f') \) be two abstract states sharing the same marking and \( B, B' \) their DBMs in canonical form. \( (m,f) \) is included in \( (m,f') \) iff:
\[
\forall x, y \in En(m) \cup \{o\}, B_{xy} \leq B'_{xy}.
\]

For the convex-combination, before explaining how to perform the grouping of abstract states, we first define what a convex-hull is. Let \( \alpha=(m,f), \alpha'=(m,f') \) be two abstract states sharing the same marking (see figure 8):

- The convex-hull of \( \alpha \) and \( \alpha' \), denoted \( \hat{\alpha}_{(\alpha,\alpha')} \), is the abstract state \( \alpha''=(m,f'') \) where:
  \[
  f'' = \bigwedge_{x,y \in En(m) \cup \{o\}} x - y \leq f_{xy}, \quad \text{Sup}_{f \cup f'}(x - y)
  \]
- Let \( \alpha''=(m,f'') \) be the convex-hull of \( \alpha \) and \( \alpha' \). \( \alpha''=(m,f'') \) is the canonical form of the union of \( \alpha \) and \( \alpha' \) iff \( (\text{Dom}(f'') - \text{Dom}(f)) \subseteq \text{Dom}(f) \).

The convex-combination test of two abstract states involves three operations: convex-hull, complement of a domain and a test of inclusion. Moreover, abstract states which may not combine two by two may combine three by three or more. Figure 9 illustrates some situations involving the convex combination of abstract states with two enabled transitions only. In case a), abstract states \( \alpha \) and \( \alpha' \) are combined into the abstract state \( \alpha'' \). Case b) shows two abstract states whose union is not convex and therefore cannot be grouped by convex combination. Case c) illustrates a situation where three abstract states \( \alpha, \alpha' \) and \( \alpha'' \) cannot combine when taken two by two, but combine well in \( \alpha'' \) if taken all together. Cases
$d)$ and $e)$ show other situations, where the grouping two by two is not possible, but becomes possible for other grouping.

Fig. 7. A TPN model and the refinement patterns of its SSCG and CSCG.

CSCG is a contraction by inclusion of the SSCG.

$^7$ CSCG is a contraction by inclusion of the SSCG.
Table 3. Refining SSCG, CSCG and CCSCG.

<table>
<thead>
<tr>
<th>TPN</th>
<th>Refining SSCG</th>
<th>Refining CSCG</th>
<th>Refining CCSCG</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>P(2)</td>
<td>2615/ 28263</td>
<td>2444 / 26358</td>
<td>2411 / 26138</td>
<td>2334 / 25046</td>
</tr>
<tr>
<td>cpu(s)</td>
<td>8.42</td>
<td>1.15</td>
<td>1.01</td>
<td>9.41</td>
</tr>
<tr>
<td>P(3)</td>
<td>?</td>
<td>31197 / 485960</td>
<td>30828 / 480987</td>
<td>28319 / 430875</td>
</tr>
<tr>
<td>cpu(s)</td>
<td></td>
<td>40.18</td>
<td>35.62</td>
<td>3887.30</td>
</tr>
<tr>
<td>P(4)</td>
<td>?</td>
<td>151384 / 2887295</td>
<td>151384 / 2887295</td>
<td>?</td>
</tr>
<tr>
<td>cpu(s)</td>
<td></td>
<td>358.06</td>
<td>358.06</td>
<td></td>
</tr>
<tr>
<td>T(2)</td>
<td>195 / 849</td>
<td>192 / 844</td>
<td>188 / 814</td>
<td>185 / 786</td>
</tr>
<tr>
<td></td>
<td>0.02</td>
<td>0.02</td>
<td>0.01</td>
<td>0.03</td>
</tr>
<tr>
<td>T(3)</td>
<td>6983 / 50044</td>
<td>6966 / 49802</td>
<td>6918 / 49025</td>
<td>6905 / 48749</td>
</tr>
<tr>
<td></td>
<td>5.00</td>
<td>2.11</td>
<td>1.49</td>
<td>60.88</td>
</tr>
<tr>
<td>T(4)</td>
<td>?</td>
<td>356940 / 3447624</td>
<td>356930 / 3447548</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>288.21</td>
<td>317.29</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 8. Convex-hull of two abstract states

Fig. 9. Grouping abstract states by convex-combination.
To achieve a high degree of contraction, we need to test all possible combinations of abstract states sharing the same marking and having states in common. But this operation is computationally very expensive. Experimental results have shown that performing the test on abstract states two by two, results in very satisfactory contractions, in relatively short computing times too. Furthermore, when two abstract states are such that one is included into the other, their convex combination is simply the most including abstract state. So, before performing the convex combination test, we check first for inclusion in $O(n^2)$, where $n$ is the number of transitions enabled in the shared marking of the two abstract states.

All CTL* model checking techniques can be applied directly on the atomic state class graphs to determine linear and branching properties of time Petri nets. All states within the same atomic abstract state have the same CTL* properties and are then considered as an indivisible unit.

5. Model checking timed properties of time Petri nets

To verify some timed properties, in (Toussaint, J. et al., 1997), authors used observers to express them in the form of TPNs and reduce them to reachability properties. However, properties on markings are quite difficult to express with observers. Other techniques define translation procedures from the TPN model into timed automata (Cassez & Roux, 2006); (Lime & Roux, 2003), in order to make use of available model checking techniques and tools (Penczek & Polrola, 2004); (Tripakis et al., 2005). Model checking is then performed on the resulting timed automata, with results interpreted back on the original TPN model. The translation into timed automata may be either structural (each transition is translated into a timed automata using the same pattern) (Cassez & Roux, 2006) or semantic (the state class graph of the TPN is first constructed and then translated into a timed automaton) (Lime & Roux, 2003). Such translations show that $\text{CTL}^\ast$, $\text{TCTL}$, $\text{LTL}$, $\text{MITL}$ model checking are decidable for bounded TPNs and that developed algorithms on timed automata may be extended to TPNs. Though effective, these techniques face the difficulty to interpret back and forth properties between the two models. In (Virbitskaite & Pokozy, 1999), authors proposed a method to model check TCTL properties of TPN. The method is based on the region graph method and is similar to the one proposed in (Alur & Dill, 1990) for timed automata. However, the region graph is known to be a theoretical method which is not applicable in practice because of its lack of efficiency.

To achieve the same goal, it is possible to adapt to the TPN, the method proposed in (Penczek & Polrola, 2004) and (Tripakis et al., 2005) for timed automata. The verification of a TCTL formula proceeds by adding a transition named $t_8$ to the TPN, translating the TCTL formula into some CTL formula, constructing an abstraction which preserves CTL properties of the completed TPN and then applying a CTL model checking technique. The transformation of TCTL formulas into CTL ones needs to extend CTL with atomic propositions of the form $t \in I$, and a particular next operator $X_{ts}$ defined by: for each formula $\psi$ and each state $s'$ of the TPN, $s'$ satisfies $X_{ts} \psi$ iff the state resulting by firing $t_s$ satisfies $\psi$.

---

8 This transition is used to deal with time constraints of the property to be verified. Its firing interval is $[0, \infty]$. 
For example, the formula $\varphi = \forall (\varphi_1 \cup I \varphi_2)$ is translated into the formula $\varphi' = X_{t_0} (\forall (\varphi_1' \cup (\varphi_2' \land t_s \in I)))$. The verification of $\varphi'$ is performed using the classical CTL model checking technique by constructing an abstraction which preserves $\varphi'$. However, this method needs to compute the whole abstraction of the model before it is analyzed and then runs up against the state explosion problem. To attenuate the state explosion problem, on-the-fly model checking methods may be a good alternative, as they allow to verify a property during the construction of an abstraction preserving linear properties. The construction of the graph is stopped as soon as the truth value of the property is obtained. On-the-fly methods have proven to be very effective to model-check a subclass of TCTL on zone graphs of timed automata. So, they can be straightforward adapted to clock based abstractions of time Petri nets. However, TPN abstractions based on intervals are in general smaller and faster to compute than TPN abstractions based on clocks. So, applying on-the-fly methods on TPN abstractions based on intervals should give better performances. In this sense, in (Hadjidj & Boucheneb, 2006), we proposed, using the state class method (SCG), a forward on-the-fly model checking technique for a subclass of TCTL properties. The verification proceeds by augmenting the TPN model under analysis with a special TPN, called Alarm shown in figure 10, to allow the capture of relevant time events (reaching, over passing a time interval). A forward on-the-fly exploration combined with an abstraction by inclusion is then applied on the resulting TPN. In the sequel, we give algorithms to model check TCTL\textsubscript{TPN} properties. Note that all following developments apply similarly to both the SCG and the RSCG. The SCG will be considered for explanations.

Let $\mathcal{N}$ be a TPN model and $\varphi = \forall (\varphi_1 \leadsto [0,b] \varphi_2$. Model checking $\varphi$ on $\mathcal{N}$ could be performed by analyzing each execution path of the TPN SCG, until the truth value of $\varphi$ is established. The SCG is progressively constructed, depth first, while looking for the satisfaction of property $\varphi_1$. If $\varphi_1$ is satisfied at an abstract state $\alpha$, $\varphi_2$ is looked for in each execution paths which starts from $\alpha$ (i.e., $\forall \rho \in \pi(\alpha)$). For each execution path $\rho \in \pi(\alpha)$, $\varphi_2$ is required to be satisfied at a state class $\alpha'$ such that the time separating $\alpha$ and $\alpha'$ is within the time interval $[0,b]$. If this is the case the verification of $\varphi$ is restarted again from $\alpha'$, and so forth, until all state classes are explored. Otherwise, the exploration is stopped, and $\varphi$ is declared invalid.

![Fig. 10. The Alarm TPN](https://www.intechopen.com)
Some attention is required when dealing with transitions $t_a$ and $t_b$. If transition $t_a$ can be fired at exactly the same time as another transition $t$, and $t$ is fired before $t_a$, $\phi$ might be declared wrongly false if the resulting state class satisfies $\phi_2$. A similar situation might arise for transition $t_b$ if it is fired before a transition $t$ which can be fired at exactly the same time. To deal with these two special situations, we assign a high firing priority to transition $t_a$, so that it is fired before any other transition which can be fired at exactly the same time. At the contrary, we assign a low firing priority to $t_b$ so that it is fired after any other transition which can be fired at exactly the same time. To cope with this priority concepts, we need to change the way we decide if a transition is firable or not, and the way the successor of a state class $\alpha=(m,f)$, by a transition $t$, is computed (i.e., operation $\text{succ}$). $\text{succ}_{AC}(\alpha,t)$ replaces $\text{succ}(\alpha,t)$ to check whether a transition is firable or not and compute successor state classes. What changes is the way the firing condition $fc$ is computed:

1. If $(t \neq t_a$ and $t_a \in \text{En}(m))$ then $fc = f \land t < t_a$.
2. If $(t=t_b$ and $t_b \in \text{En}(m))$ then $fc = f \land t = 0$ (\begin{math} \forall t' \in \text{En}(m)-\{t_b\} \end{math} \begin{math} t_b < t' \end{math})
3. If $(t=t_a$ or $(t_a \notin \text{En}(m)$ and $t \neq t_b))$ then $fc = f \land t = 0$.

In case $t_a$ is enabled while we want to fire a different transition $t$ (case 1), we need to make sure that $t$ is fired ahead of time of $t_a$. In case $t_b$ is enabled and is the one we want to fire (case 2), we need to make sure that $t_b$ is the only transition that can be fired. The remaining cases are handled exactly as before.

$\text{Succ}_{AC}(\alpha, t) \neq \emptyset$ iff $fc$ is consistent. If $\text{succ}_{AC}(\alpha, t) \neq \emptyset$ then $\text{succ}_{AC}(\alpha, t) = (m', f')$ is computed in four steps:

1. $\forall p \in P, m'(p)=m(p)-\text{Pre}(p, t)+\text{Post}(p, t)$.
2. Initialize $f'$ with $fc$. This step eliminates from $f$ states from which $t$ is not immediately firable.
3. Put $f'$ in canonical form and eliminate $t$ and all transitions conflicting with $t$ for $m$.
4. Add constraints $\forall t' \in \text{New}(m'; t) \downarrow Is(t') \leq t' \leq \uparrow Is(t')$ and put $f'$ in canonical form. The firing interval of each newly enabled transition is set to its static firing interval.
The verification of $\varphi$ proceeds as follows: During the generation of the SCG of $N\mid \text{Alarm}$, if $\phi_1$ is satisfied in a state class $\alpha=(m,f)$, transition $t_a$ is enabled in $\alpha$ to capture the event corresponding to the beginning of time interval $I_r$. $t_a$ is enabled by changing the marking $m$ in $\alpha$ such that place $P_a$ would contain one token, and replacing $f$ with $f \land t_a=a$. These two actions correspond to artificially putting a token in place $P_a$ of $\text{Alarm}$. Since $a=0$ and transition $t_a$ has the highest priority, it is fired before all others. When $t_a$ is fired (which means that time has come to start looking for $\phi_2$, $t_b$ gets enabled in the resulting state class $\alpha=(m,f)$ to capture the event corresponding to the end of interval $I_r$. If $t_b$ is fired during the exploration, $\varphi$ is declared invalid and the exploration stops. If before firing $t_b$, $\phi_2$ is satisfied in a state class $\alpha''=(m'',f'')$ transition $t_b$ is disabled in $\alpha''$ by changing the marking $m''$ such that place $P_b$ would contain zero tokens, and eliminating variable $t_b$ from $f''$. These two actions correspond to artificially removing the token in place $P_b$. After $\alpha''$ is modified, $\varphi$ is checked again starting from $\alpha''$. Note that in this technique, the fact of knowing a state class and the transition that led to it, is sufficient to know which action to take\footnote{For uniformity reasons, we assume a fictitious transition $t_\varepsilon$ as the transition which led to the initial state class.}. This means that there is no need to keep track of execution paths during the exploration, and hence, the exploration strategy of the SCG (depth first, breadth first,..) is irrelevant. This in turn solves the problem of dealing with cycles and infinite execution paths for bounded TPN models.

Let $\alpha=(m,f)$ be a state class and $t$ the transition that led to it. The different cases that might arise during the exploration are given in what follows:

1. The case where $t_a, t_b \not\in En(m)$ and $t \not\in \{t_a, t_b\}$ corresponds to a situation where we are looking for $\phi$. In case $\phi_1$ is satisfied in $\alpha$, we enable $t_a$ in $\alpha$,

2. The case where $t_b \in En(m)$ corresponds to a situation where we are looking for $\phi_2$. If $\phi_2$ is satisfied in $\alpha$ then we disable $t_b$ and get in a situation where we are looking for $\phi_1$ (i.e., (1)).

3. The case where $t=t_b$ corresponds to a situation where interval $I_r$ has expired while we are looking for $\phi_2$. In this case, we stop the exploration and declare $\varphi$ invalid.

Another problem may arise for zeno TPNs. Indeed, if the model is zeno and has a zeno execution path such that all its state classes satisfy $\phi_1$ but its time is less than $b$. In this case, $t_b$ will never get fired to signal the end of interval $I_r$, and the verification would conclude that the property is valid while it is not. To correct this problem, one solution consists in detecting zeno cycles during the verification, but not any zeno cycle. The zeno cycles of interest are only those which arise when transition $t_a$ or $t_b$ is enabled.

---

Algorithm modelCheck($\varphi$)

```plaintext
{  continue:=true; /*global variable */  valid:=true; /*global variable */  COMPUTED:= \emptyset;  \alpha_0:= (m_0,f_0);}
```

www.intechopen.com
\[ \alpha_0 := \text{checkStateClass}(\alpha_0, t) \]

\[ \text{WAIT} = \{ \alpha_0 \} \]

while (continue)
{
    remove \( \alpha = (m, f) \) from \( \text{WAIT} \);
    for (\( t \in \text{En}(m) \) s.t. \( \text{succ}_{AC}(\alpha, t) \neq \emptyset \)) provided continue
    {
        \( \alpha' := \text{succ}_{AC}(\alpha, t) \);
        If (\( \varphi \neq \exists (\phi_1 \cup \phi_2) \) and (\( t_a \in \text{En}(m) \) or \( t_b \in \text{En}(m) \)) and \( I_{ts}(t) = 0 \)) then Connect \( \alpha \) to \( \alpha' \);
        \( \alpha' := \text{checkStateClass}(\varphi, \alpha', t) \);
        if (continue \( \land \alpha' \neq \emptyset \land \exists \alpha_0 \in \text{COMPUTED} \) s.t. \( \alpha' \subseteq \alpha_0 \)) then
            for (\( \alpha_p \in \text{COMPUTED} \) s.t. \( \alpha_p \subseteq \alpha' \)) remove \( \alpha_p \) from \( \text{COMPUTED} \) and from \( \text{WAIT} \);
            add \( \alpha' \) to \( \text{COMPUTED} \) and to \( \text{WAIT} \);
        }
    }
}

If (\( \varphi \neq \exists (\phi_1 \cup \phi_2) \) and \( \text{COMPUTED} \) has a cycle s.t. \( t_a \) or \( t_b \) is enabled in all its state classes) then
valid := false;
Return valid;

The on-the-fly \( \text{TCTL}_{TPN} \) model checking of formula \( \varphi \) is based on the following exploration algorithm \( \text{modelCheck}(\varphi) \). This algorithm uses two lists: \( \text{WAIT} \) and \( \text{COMPUTED} \), to manage state classes, and calls a polymorphic satisfaction function \( \text{checkStateClass}_\varphi \) to check the validity of formula \( \varphi \). \( \text{COMPUTED} \) contains all computed state classes, while \( \text{WAIT} \) contains state classes of \( \text{COMPUTED} \) which are not yet explored. The algorithm generates state classes by firing transitions. The initial state class is supposed to result from the firing of a fictive transition \( t \). Each time a state class \( \alpha \) is generated as the result of firing a transition \( t \), \( \alpha \) and \( t \) are supplied to \( \text{checkStateClass}_\varphi \) to perform actions and take decisions. In general, \( \text{checkStateClass}_\varphi \) enables or disables transitions \( t_a \) and \( t_b \) in \( \alpha \). It also takes decisions, and record them in two global boolean variables continue and valid, to guide the exploration process. Finally, it returns either \( \alpha \) after modification or \( \emptyset \) in case \( \alpha \) needs to be no more explored (i.e., ignored). The exploration continues only if continue is true. valid is used to record the truth value of \( \varphi \). After \( \text{checkStateClass}_\varphi \) is called, the state class \( \alpha' \) it returns is inserted in the list \( \text{WAIT} \) only if it is not included in a previously computed state class. Otherwise, \( \alpha' \) is inserted in the list \( \text{WAIT} \), while all state classes of the list \( \text{COMPUTED} \) which are included into \( \alpha' \) are deleted from both \( \text{COMPUTED} \) and \( \text{WAIT} \). This strategy, used also in the tool UPPAAL (Behrmann et al., 2002), attenuates considerably the state explosion problem. So instead of exploring both \( \alpha \) and \( \alpha' \), exploring \( \alpha' \) is sufficient. Operation \( \text{checkStateClass}_\varphi \) takes as parameters: a state class, and the transition that led to it. Three different implementations of \( \text{checkStateClass}_\varphi \) are required for the three principal forms of \( \varphi \), i.e., \( \phi \leadsto_{Ir} \phi_2 \), \( \forall (\phi_1 \cup I) \phi_2 \) and \( \exists (\phi_1 \cup I) \phi_2 \), with \( I = [a, b] \) and \( I_r = [0, b] \) (bound \( b \) can be either finite or infinite). All of these implementations handle four mutually exclusive cases corresponding to four types of state classes that can be encountered on an execution path.

The first implementation corresponds to property \( \varphi \equiv \phi \leadsto_{Ir} \phi_2 \). The first case it handles corresponds to a state class not reached by the firing \( t_a \) nor \( t_b \), and neither of them is enabled
in it. The remaining cases correspond respectively to: a state class where transition \( t_b \) is enabled and a state class reached by the firing of transition \( t_b \).

Algorithm `checkStateClass, \( \varphi \rightarrow_I \varphi (\alpha=(m,f),t) \)`

\[
\begin{align*}
\text{if} \ (t_a t_b \not\in \text{En}(m) \land t \not\in \{t_a t_b\}) \text{ then} & \\
& \quad \text{if} (\varphi_1(m)) \text{ then enable } t_a \text{ in } \alpha \\
& \quad \text{if} (t_b \in \text{En}(m) \land \varphi_2(m)) \text{ then disable } t_b \text{ in } \alpha \\
& \quad \text{if} (t=t_b) \ {\{} \ valid=false; \ continue=false; \ {}\} \\
\text{Return } \alpha;
\end{align*}
\]

The second implementation corresponds to property \( \varphi=\forall (\varphi_1 U \varphi_2) \). In it first case, this implementation looks for the initial state class only. The remaining cases are similar to those of the first implementation, but different actions are taken for each one of them. Intuitively the verification of property \( \varphi=\forall (\varphi_1 U \varphi_2) \) checks if proposition \( \varphi_1 \) is true in the initial state class and all state classes following it, until \( t_a \) fires. From the moment \( t_a \) is fired, the verifier checks for the satisfaction of either \( \varphi_1 \) or \( \varphi_2 \) until \( \varphi_2 \) is true or \( t_b \) is fired. If \( \varphi_2 \) becomes true in a state class \( \alpha \), \( \alpha \) is no more explored. In case \( t_b \) is fired, the exploration is stopped and the property is declared invalid.

Algorithm `checkStateClass, \( \forall (\varphi_1 U \varphi_2) (\alpha=(m,f),t) \)`

\[
\begin{align*}
\text{if} \ (t=t_a) \text{ then} & \\
& \quad \text{if} (\varphi_1(m)) \text{ then enable } t_a \text{ in } \alpha \\
& \quad \text{else if} (\neg \varphi_2(m) \lor a>0) \text{ then} \ {\{} \ valid=false; \ continue=false; \ {}\} \\
& \quad \text{else} \ \text{valid=true; continue=false;} & \\
& \text{if} (t_a \in \text{En}(m) \land \neg \varphi_1(m)) \text{ then} \ {\{} \ valid=false; \ continue=false; \ {\}} & \\
& \text{if} (t_b \in \text{En}(m)) \text{ then} & \\
& \quad \text{if} (\neg \varphi_2(m)) \ {\{} \ valid=false; \ continue=false; \ {\}} & \\
& \quad \text{else} \ \text{Return } \emptyset; & \\
& \text{if} (t=t_b) \ \text{then} \ {\{} \ valid=false; \ continue=false; \ {\}} & \\
\text{Return } \alpha & \end{align*}
\]

The implementation of `checkStateClass, \( \exists (\varphi_1 U \varphi_2) \)` corresponds to property \( \varphi=\exists (\varphi_1 U \varphi_2) \). It handles four similar cases as the previous implementation, but different actions are taken. For instance, this implementation initializes variable valid to false as soon as the initial state class is entered, and stops the exploration of a state class \( \alpha \) if it does not comply with the semantics of \( \varphi \). It also aborts the exploration as soon as a satisfactory execution path is found.

To illustrate our verification approach, we consider the simple TPN model shown in figure 11, we call cyclic. The \( \text{TCTL}_{\text{TPN}} \) property we verify is \( \varphi=\varphi_1 \rightarrow_{[0,3]} \varphi_2 \) with proposition \( \varphi_1(m)= (m(P_0)=0) \) and proposition \( \varphi_2(m)= (m(P_1)=1) \). For simplicity reasons, we selected a cyclic TPN model with a single execution path, for which property \( \varphi \) is trivially valid.
The verification process of $\phi$ starts first by constructing the TPN model cyclic $|$ Alarm, such that $a=0$ and $b=3$, then runs according to the following steps:

1. Compute the initial state class of cyclic $|$ Alarm: $\alpha_0 = (P_0, 1 \leq t_0 \leq 2)$.
2. Check if $\phi_1$ is valid in $\alpha_0$: $\phi_1$ is not valid in $\alpha_0$.
3. Fire $t_2$ from $\alpha_0$ and put the result in $\alpha_1$: $\alpha_1 = (P_1, 2 \leq t_1 \leq 3)$.
4. Check if $\phi_1$ is valid in $\alpha_1$: $\phi_1$ is valid in $\alpha_1$.
5. Enable $t_a$ in $\alpha_1$: $\alpha_1$ becomes $(P_1 + P_a, 2 \leq t_1 \leq 3)$.
6. Fire $t_a$ from $\alpha_1$ and put result in $\alpha_2$: $\alpha_2 = (P_1 + P_b, 2 \leq t_1 \leq 3 \land t_b=0)$.
7. Check if $\phi_2$ is satisfied in $\alpha_2$: $\phi_2$ is not satisfied in $\alpha_2$.
8. Fire $t_1$ from $\alpha_2$ and put the result in $\alpha_3$: $\alpha_3 = (P_0 + P_b, 1 \leq t_0 \leq 2 \land 0 \leq t_b \leq 1)$.
9. Declare $\phi$ valid since $\alpha_3$ has already been explored ($\alpha_3=\alpha_0$).

We have implemented and tested this approach on the level classical model. The properties we considered are:

10. The gate is never open whenever a train is crossing: $\phi_1 = \forall G - (open \land \bigvee_{1 \leq i \leq n} on_i)$.
11. If a train approaches, the gate closes in less than 2 time units: $\phi_2 = coming \leadsto [0,2]$ closed.
12. The level crossing model is deadlock free: $\phi_3 = \forall G (En(m) \neq \emptyset)$.

Table 3 reports results obtained for model checking the selected properties using our approach, applied on the SCG. Each result is given in terms of the final size of the list COMPUTED and the total number of explored state classes, followed by the exploration time. The second column recalls the size and computing time of the ASCGs. All properties have been successfully tested valid.

<table>
<thead>
<tr>
<th>TPN</th>
<th>ASCG</th>
<th>$\phi_1$</th>
<th>$\phi_2$</th>
<th>$\phi_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(2) cpu(s)</td>
<td>188 / 814</td>
<td>38 / 116</td>
<td>41 / 91</td>
<td>38 / 116</td>
</tr>
<tr>
<td>0.01</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T(3) cpu(s)</td>
<td>6918 / 49025</td>
<td>173 / 790</td>
<td>182 / 646</td>
<td>173 / 790</td>
</tr>
<tr>
<td>1.49</td>
<td></td>
<td>0</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>T(4) cpu(s)</td>
<td>356930 / 3447548</td>
<td>1176 / 7162</td>
<td>1194 / 6073</td>
<td>1176 / 7162</td>
</tr>
<tr>
<td>317.29</td>
<td></td>
<td>0.12</td>
<td>0.1</td>
<td>0.12</td>
</tr>
<tr>
<td>T(5) cpu(s)</td>
<td>?</td>
<td>10973 / 81370</td>
<td>11008 / 71152</td>
<td>10973 / 81370</td>
</tr>
<tr>
<td>2.37</td>
<td></td>
<td>2.04</td>
<td>2.30</td>
<td>2.30</td>
</tr>
<tr>
<td>T(6) cpu(s)</td>
<td>?</td>
<td>128116 / 1103250</td>
<td>128184 / 986939</td>
<td>128116 / 1103250</td>
</tr>
<tr>
<td></td>
<td></td>
<td>110.81</td>
<td>100.92</td>
<td>111.8</td>
</tr>
</tbody>
</table>

Table 4. Comparison of ASCGs with our on-the-fly method

6. Conclusion

In this chapter, we presented and discussed model checking techniques of time Petri nets. We pointed out some strategies which allow to make model checking techniques more efficient. For model checking LTL properties, we proposed a contraction for the state class
graph (SCG), called RSCG, which is both smaller and faster to compute than other abstractions. For CTL* model checking, we showed that refining abstractions contracted by inclusion or convex-combination allow to improve significantly the refinement process. For all tested models, the refinement follows a linear pattern when an inclusion or convex-combination abstraction is used. When an abstraction preserving linear properties is refined, the size of the computed graph starts first to grow up to a peek size then decreases until an atomic state class space is obtained. Finally, to attenuate the state explosion problem of model checking techniques, we considered a subclass of TCTL and proposed an on-the-fly method for the RSCG and SCG. On-the-fly methods have proven to be very effective to model-check a subclass of TCTL of timed automata.

7. References


Although many other models of concurrent and distributed systems have been developed since the introduction in 1964 Petri nets are still an essential model for concurrent systems with respect to both the theory and the applications. The main attraction of Petri nets is the way in which the basic aspects of concurrent systems are captured both conceptually and mathematically. The intuitively appealing graphical notation makes Petri nets the model of choice in many applications. The natural way in which Petri nets allow one to formally capture many of the basic notions and issues of concurrent systems has contributed greatly to the development of a rich theory of concurrent systems based on Petri nets. This book brings together reputable researchers from all over the world in order to provide a comprehensive coverage of advanced and modern topics not yet reflected by other books. The book consists of 23 chapters written by 53 authors from 12 different countries.

How to reference
In order to correctly reference this scholarly work, feel free to copy and paste the following: