Theoretical Computer Science 410 (2009) 4241–4261

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

On-the-fly *TCTL* model checking for time Petri nets

Rachid Hadjidj, Hanifa Boucheneb *

Department of Computer Engineering, École Polytechnique de Montréal, P.O. Box 6079, Station Centre-ville, Montréal, Québec, Canada

ARTICLE INFO

Article history:
Received 10 February 2006
Received in revised form 27 May 2009
Accepted 10 June 2009
Communicated by V. Sassone

Keywords:
Time Petri nets
State space abstractions
State class method
TCTL model checking

ABSTRACT

In this paper, we show how to efficiently model check a subset of *TCTL* properties for the Time Petri Net model (TPN model), using the state class method. The verification proceeds by augmenting the TPN model under analysis with a special TPN, called *Alarm-clock*, to allow the capture of relevant time events. A forward on-the-fly exploration is then applied on the resulting TPN state class space to verify a timed property. A relaxation operation on state classes is also introduced to further improve performances. *Alarm-clock* is the same for all properties, whereas the exploration technique is not. Three exploration techniques are presented to cover most interesting TCTL properties. We prove the decidability of our verification technique for bounded TPN models and compare it with the reachability algorithm implemented in the tool UPPAAL [G. Behrmann, J. Bengtsson, A. David, K.G. Larsen, P. Pettersson, W. Yi, Uppaal implementation secrets, in: Proc. of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, 2002]. Finally, we give some experimental results to show the efficiency of our verification technique.

1. Introduction

Time Petri nets provide a formal framework to model and verify the correct functioning of real-time systems. Petri nets extended with timing dependencies are very numerous in the literature [1,18,23,29,31]. The best known timed extensions are *timed Petri nets* proposed by Ramchandani [31] and *time Petri nets* proposed by Merlin and Farber [29]. We focus in this paper on the *time Petri net* model [29], referred in the sequel as the TPN model, and propose a model checking approach to verify its timed properties.

Several abstractions of the TPN model state space have been proposed in the literature to verify its untimed temporal properties [9,13,14,12,20,22,27,33]. Most proposed approaches are based on the so called *state class method* [9], where a state class is a symbolic representation for some infinite set of states sharing the same marking. These approaches allow one to construct state class spaces that preserve reachability [12,27], linear properties [9], and branching properties [13,14,22,33], which are then verified using standard model-checking techniques [17]. To our knowledge, no known techniques based on state class method exists yet to verify timed properties for the TPN model with the same versatility reached for timed automata [4]. The use of observers [32], for instance, allows one to express some timed properties in the form of TPNs, but properties on markings are quite difficult to express with observers [16]. Other techniques define translation procedures from the TPN model into a semantically equivalent timed automata [4], in order to make use of available model checking tools [15,19,21,28,33]. Model checking is then performed on the resulting timed automaton, with results interpreted back on the original TPN model. Though effective, these techniques face the difficulty to interpret back and forth properties between the two models. In this paper, we propose a model checking approach to directly verify timed properties on the TPN model

* Corresponding author. Tel.: +1 514 340 4711.
E-mail addresses: rachid.hadjidj@polymtl.ca (R. Hadjidj), hanifa.boucheneb@polymtl.ca (H. Boucheneb).

0304-3975/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2009.06.019

using the state class method, and prove its decidability for all bounded[1] TPN models. Timed properties we consider are mostly a subset of the *TCTL* timed logic [3], we call $TCTL_{TPN}$, sufficient in general to verify most interesting timed properties (reachability, safety, bounded liveness, etc). For reasons of efficiency, the model checking approach we propose is based on an on-the-fly exploration technique [5,11,25], combined with an abstraction by inclusion[2] [14] to better confine the state explosion problem. For further improvement, we introduce a relaxation operation where state classes are extended with states reachable via time progressions. The relaxation allows one to compute a relaxed version of the SCG, we call RSCG, with similar properties, but smaller and faster to compute in general.

Given a TPN model and a $TCTL_{TPN}$ property, the TPN model is first augmented (put in parallel) with a special TPN model, we call *Alarm-clock*. The state class graph[3] [9] of the combined TPN is then explored, while looking for important time events to decide the truth value of the considered timed property. *Alarm-clock* has two transitions with special firing priorities. During the exploration, these transitions can be set to fire at precise moments (the beginning and the end of a time interval), which allows one to capture important time events. State classes construction is also adapted to cope with the introduced priority concepts. To show the effectiveness of our verification technique, we compare it with the approach implemented in the tool UPPAAL [5], and give some experimental results. On average, we were able to verify some properties 13.85 times faster than UPPAAL with 1.48 times lesser memory. For some properties and models, our approach was 3485 times faster, and used 120 times lesser memory.

The rest of the paper is organized as follows. Section 2 introduces the TPN model and its semantics. Section 3 presents the state class method and shows how to construct the *state class graph* (SCG) as an abstraction of the TPN state space. In Section 4, we propose a relaxed version of the SCG. Section 5 describes the timed temporal logic *TCTL*, then introduces $TCTL_{TPN}$. In Section 6, we develop our model checking technique and give an illustrative example. In Section 7, we compare our approach with the reachability algorithm implemented in the tool UPPAAL [5]. Section 8 reports some experimental results and comparisons with the tool UPPAAL.

2. Time Petri nets

Let $\mathbb{Q}^+$ and $\mathbb{R}^+$ be, respectively, the set of nonnegative rational numbers and the set of nonnegative real numbers. Let $\mathbb{Q}^+_{[\,]}$ be the set of non empty intervals of $\mathbb{R}^+$ which bounds are respectively in $\mathbb{Q}^+$ and $\mathbb{Q}^+ \cup \{\infty\}$. For an interval $I \in \mathbb{Q}^+_{[\,]}$, $\downarrow I$ and $\uparrow I$ denote respectively its lower and upper bounds.

Definition 1. Time Petri net (TPN)
A TPN is a tuple $(P, T, Pre, Post, m_0, Is)$ where:

- P is a finite set of places,
- T is a finite set of transitions, such as $P \cap T = \emptyset$,
- Pre and $Post$ are the backward and the forward incidence functions: $P \times T \to \mathbb{N}$, where $\mathbb{N}$ is the set of nonnegative integers,
- $m_0 : P \to \mathbb{N}$, is the initial marking,
- $Is : T \to \mathbb{Q}^+_{[\,]}$ associates with each transition t an interval $[\downarrow Is(t), \uparrow Is(t)]$ called its *static firing interval*. The bounds $tmin(t) = \downarrow Is(t)$ and $tmax(t) = \uparrow Is(t)$ are called the *minimal* and *maximal static firing delays* of t.

Informally, a TPN is a Petri net with time intervals attached to its transitions [29]. Let M be the set of all markings of the TPN model, $m \in M$ a marking, and $t \in T$ a transition. t is said to be *enabled* in m, iff all tokens required for its firing are present in m, i.e.: $\forall p \in P, m(p) \geq Pre(p, t)$. For clarity reasons, we consider here only T-safe TPNs (no multi-enabled transitions). We denote by $En(m)$ the set of all transitions enabled in m. If m results from firing transition t_f from another marking, $New(m, t_f)$ denotes the set of all transitions newly enabled in m, i.e.: $New(m, t_f) = \{t \in En(m) | \exists p, m(p) - Post(p, t_f) < Pre(p, t)\}$.

2.1. The TPN state

There are two known characterizations of the TPN state. The first one, based on clocks, associates with each transition t of the TPN model a *clock* to measure the time elapsed since t became enabled most recently. The TPN state is defined as a marking and a clock valuation function which associates with each enabled transition the value of its clock [13–15,20].

Definition 2. Clock state
The TPN *clock state* is a couple (m, V), where m is a marking and V is a clock valuation function, $V : En(m) \to \mathbb{R}^+$.

[1] A TPN is bounded if the marking of each one of its places is bounded. In other words, a TPN is bounded if it has a finite number of reachable markings.
[2] When a state class is explored, there is no need to explore another state class which is included in it.
[3] The verification applies similarly on both the SCG and RSCG.

For a clock state (m, V) and $t \in En(m)$, $V(t)$ is the value of the clock associated with transition t. The initial clock state is $s_0 = (m_0, V_0)$ where $V_0(t) = 0$, for all $t \in En(m_0)$. The TPN clock 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 $Is(t) = [tmin(t), tmax(t)]$. It must be fired immediately, without any additional delay, when the clock reaches $tmax(t)$. The firing of a transition takes no time, but may lead to another marking (required tokens disappear while produced ones appear).

The second characterization, based on intervals, defines the TPN state as a marking and a function which associates with each enabled transition the time interval in which the transition can fire [9].

Definition 3. Interval state
The TPN interval state is a couple (m, I), where m is a marking and I is an interval function, $I : En(m) \to \mathbb{Q}^{+}_{[\,]}$.

For a state $s = (m, I)$, and $t \in En(m)$, $I(t)$ is called the *firing interval* of t. I is also called the *firing domain* of s, since it can be interpreted as a set of tuples $\{i | i(t) \in I(t), \forall t \in En(m)\}$. The initial state of the TPN model is $s_0 = (m_0, I_0)$, where $I_0(t) = Is(t)$, for all $t \in En(m_0)$. The TPN state evolves either by time progression or by firing transitions. When a transition t becomes enabled, its firing interval is set to its static firing interval $Is(t)$. The 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. The firing of a transition takes no time.

We focus, in the remainder of this paper, on the later characterization, for its advantages over the first one when computing abstractions for the TPN model that preserve markings and linear properties. These advantages will be made clear in Section 3.3.

Let $s = (m, I)$ and $s' = (m', I')$ be two states of the TPN model. We write $s \xrightarrow{\theta} s'$, iff state s' is reachable from state s after a time progression of θ time units (s' is also denoted $s + \theta$), i.e.:

$$\begin{cases} \exists \theta \in \mathbb{R}^{+}, \bigwedge\limits_{t \in En(m)} \theta \leq \uparrow I(t), \\ m' = m, \\ \forall t' \in En(m'), \qquad I'(t') = [\max(\downarrow I(t') - \theta, 0), \uparrow I(t') - \theta]. \end{cases}$$

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

$$\begin{cases} t \in En(m), \\ \downarrow I(t) = 0, \\ \forall p \in P, \qquad m'(p) = m(p) - Pre(p, t) + Post(p, t), \\ \forall t' \in En(m') \begin{cases} I'(t') = Is(t') & \text{if } t' \in New(m', t), \\ I'(t') = I(t) & \text{otherwise}. \end{cases} \end{cases}$$

As a shorthand, we write $s \xrightarrow{\theta t} s'$ iff $s \xrightarrow{\theta} s''$ and $s'' \xrightarrow{t} s'$ for some state s'', $\theta \in \mathbb{R}^{+}$ and $t \in T$. We write $s \overset{t}{\rightsquigarrow} s'$ iff $s \xrightarrow{\theta t} s'$, and write $s \rightsquigarrow s'$ iff $\exists t \in T\ s.t.\ s \overset{t}{\rightsquigarrow} s'$.

2.2. The TPN state space

Definition 4. TPN state space
The *state space* of a *TPN* model is the structure $(\mathcal{S}, \to, s_0)$, where:

- $s_0 = (m_0, I_0)$ is the initial state of the TPN model,
- $\mathcal{S} = \{s | s_0 \xrightarrow{*} s\}$, where $\xrightarrow{*}$ is the reflexive and transitive closure of $\to$, is the set of reachable states of the TPN model.

An *execution path* in the TPN state space, starting from a state s, is a maximal sequence $\rho = s^0 \xrightarrow{\theta_0 t_0} s^1 \xrightarrow{\theta_1 t_1} s^2 \ldots\ldots$, such that $s^0 = s$. When no starting state is specified, the initial state of the TPN model is intended. The sequence $\theta_0\ t_0\ \theta_1\ t_1\ \ldots\ldots$ is called the trace of ρ.We denote by $\pi(s)$ (respectively $\tau(s)$) the set of all execution paths (respectively traces) starting from state s. $\pi(s_0)$ and $\tau(s)$ are therefore the sets of all execution paths and traces of the TPN. The TPN state space defines the *branching semantics* of the TPN model, whereas $\pi(s_0)$ defines its *linear semantics*. The total elapsed time during an execution path ρ, denoted $time(\rho)$, is the sum $\sum_{i \geq 0} \theta_i$. An infinite execution path is *diverging* if $time(\rho) = \infty$, 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 an infinity of actions may take place in a finite amount of time. We consider, in this paper, non zeno TPN models. Zenoness can be detected using the approach proposed in [15].

3. Abstraction of the TPN state space preserving linear properties

Because of time density,[4] a state in the TPN state space may have an infinity of successors. To finitely represent the state space of a TPN model while preserving linear properties, Berthomieu and Menasche proposed in [9] to abstract time, and group states in what is called *state classes*.

3.1. The TPN concrete state space

The abstraction of time consists in hiding time progressions in the TPN state space, while keeping only those states which are immediately reachable after firing transitions. This operation results in a graph called *concrete state space* [30].

Definition 5. TPN concrete state space
The *concrete state space* of the TPN model is the structure $(\Sigma, \rightsquigarrow, s_0)$ where:

- s_0 is the initial state of the TPN model,
- $\Sigma = \{s | s_0 \overset{*}{\rightsquigarrow} s\}$, where $\overset{*}{\rightsquigarrow}$ is the reflexive and transitive closure of $\rightsquigarrow$, is the set of reachable concrete states of the TPN model.

3.2. The state class method

A state class is a symbolic representation for some infinite set of concrete states sharing the same marking. All concrete states reachable from the initial state by the firing of the same sequence of transitions are agglomerated in the same state class. The resulting graph is called *state class graph* (SCG) [9].

Let $\omega = t_0, t_1, t_2, \ldots t_n$ be a sequence of transitions firable from the initial TPN state. The state class corresponding to ω (i.e., $\{s \in \mathcal{S} | \exists s_1, \ldots, s_n \in \mathcal{S}, s_0 \overset{t_0}{\rightsquigarrow} s_1 \overset{t_1}{\rightsquigarrow} s_2 \ldots s_n \overset{t_n}{\rightsquigarrow} s\}$) is represented by the pair (m, F), where m is the common marking of all states agglomerated in the state class, and F is a formula that characterizes the union of all firing domains of these states. In F, each transition enabled in m is represented with a variable having the same name.

The initial state class (m_0, F_0) coincides with the initial state of the TPN model (i.e., m_0 is the initial marking and $F_0 = (\bigwedge_{t \in En(m_0)} tmin(t) \leq t \leq tmax(t))$). Let $\alpha = (m, F)$ be a state class and t_f a transition. The class α has a successor by t_f, denoted $succ(\alpha, t_f)$, iff t_f is enabled in m and can be fired before any other transition enabled in α. If this is the case, t_f is said to be *firable* from α. Algorithm 1 shows how to perform such a test.

Algorithm 1: $isFirable(\alpha = (m, F), t_f)$

```
1 if t_f ∉ En(m) then Return false
2 if (F ∧ (⋀_{t∈En(m)−{t_f}} t_f ≤ t)) is consistent then
3     Return true
4 Return false
```

Step 1 of algorithm 1 checks if t_f is enabled in m. Step 2 computes and checks the consistency of the formula corresponding to the part of the firing domain of α where t_f can be fired before any other enabled transition. If t_f is firable from α, $\alpha' = succ(\alpha, t_f)$ is computed according to algorithm 2.

Algorithm 2: $succ(\alpha = (m, F), t_f)$

```
1 Let m'(p) = m(p) − Pre(p, t_f) + Post(p, t_f), ∀p ∈ P
2 Let F' = F ∧ (⋀_{t∈En(m)−{t_f}} t_f ≤ t)
3 Replace in F' each variable t ≠ t_f with t + t_f
4 Eliminate by substitution, in F', t_f and all variables associated with transitions conflicting with t_f for m
5 forall t ∈ New(m', t_f) do
6     Add to F' the constraint tmin(t) ≤ t ≤ tmax(t)
7 Return α' = (m', F')
```

Step 1 of algorithm 2 computes the marking after firing t_f. Step 2 selects in F' states from which t_f is firable. Steps 3 and 4 decrease each firing delay by t_f time units to coincide with the moment t_f is fired. Steps 5 and 6 add constraints corresponding to the newly enabled transitions.

[4] The domain of clocks is $\mathbb{R}^+$ (continuous domain).

From algorithm 2, it is easy to see that the formula F of a state class can be rewritten as a conjunction of atomic constraints of the form $t - t' \prec c$ (called also *triangular constraints*) or $t \prec c$ (called also *simple constraints*), where $c \in \mathbb{Q} \cup \{\infty, -\infty\}$, $\prec \in \{=, \leq, \geq\}$ and $t, t' \in T$. The domain of F is therefore convex and has a unique canonical form defined by: $\bigwedge_{(x,y)\in(En(m)\cup\{o\})^2} x - y \prec_F^{x-y} \mathrm{Sup}(x - y, F)$
where:

- o represents the value *zero*,
- $\mathrm{Sup}(x - y, F)$ 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.

The canonical form of state class (m, F) can be represented by the pair (m, D), where D is a DBM of order $|En(m) \cup \{o\}|$, defined by:
$\forall (x, y) \in (En(m) \cup \{o\})^2, \quad D_{x,y} = (\mathrm{Sup}(x - y, F), \prec_F^{x-y})$.
Each element of D is called a bound. Operations on bounds are defined as usual [5,6]. The computation of the canonical form is based on the shortest path *Floyd–Warshall*'s algorithm [5,6].

State classes are considered modulo an equivalence relation, such that two state classes are equivalent iff they have the same marking and their domains are equal (i.e., their formulas are equivalent). To compare two state classes, each one is translated into its canonical form. They are equal if they have identical canonical forms [9]. Formally the SCG definition can be stated as follows:

Definition 6. State class graph (SCG)
The *state class graph* of a TPN model is the structure $(\mathcal{C}, \rightarrow, \alpha_0)$, where:

- $\alpha_0 = (m_0, F_0)$ is the initial state class,
- $\alpha \overset{t}{\rightarrow} \alpha'$ iff $(isFirable(\alpha, t_f) \wedge \alpha' = succ(\alpha, t))$,
- $\mathcal{C} = \{\alpha | \alpha_0 \rightarrow^* \alpha\}$, where $\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$.

Algorithm 3 shows how to progressively construct the SCG. It starts from the initial state class and uses the list WAIT to store state classes which are not yet explored. In [8–10], the authors proved that the SCG is finite for all bounded TPN models, and also preserves markings and traces (i.e., linear properties) of the concrete state space.

Algorithm 3: $SCG(\mathcal{N} = (P, T, Pre, Post, m_0, Is))$

```
1  α0= (m0, F0)
2  C= {α0}
3  →= Ø
4  WAIT={α0}
5  while WAIT ≠ Ø do
6      get α = (m, F) from WAIT
7      forall t ∈ En(m) s.t. isFirable(α, t) do
8          α'=succ(α, t)
9          if (α' ∉ C) then
10             Add α' to C and to WAIT
11             Add (α, t, α') to →
12 Return (C, →, α0)
```

3.3. Other abstractions of the TPN model

Several other abstractions of the TPN state space exist also in the literature, but based on the clock characterization of state classes. Some examples are the *geometric region graph* [33], the *zone based graph* (ZBG) [20], and the *strong state class graph* (SSCG) [10]. All these abstractions are, however, less interesting than the SCG when only linear properties are of interest. They are, in general, much larger, and do not naturally enjoy the finiteness property for all bounded TPNs, as is the case for the SCG.[5] 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, I) such that: $\forall t \in En(m), I(t) = [\max(0, tmin(t) - V(t)), tmax(t) - V(t)]$. In words, the bounds of $I(t)$ are respectively the minimal and maximal remaining times before t can fire. Note that for any real value $v \geq tmin(t)$, if $tmax(t) = \infty$, $tmax(t) - v = \infty$ and

[5] For bounded TPN models with unbounded firing intervals, finiteness is enforced using an approximation operation on state classes. This operation known under the name *k-approximation* or *k-normalization*, is similar to the one used for timed automata to achieve the same objective [20].

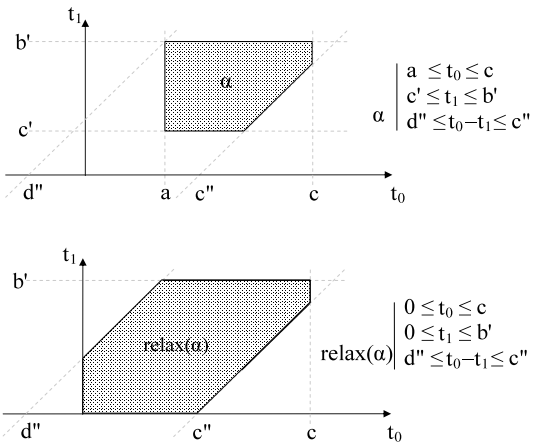

Fig. 1. The relaxation operation.

$\max(0, tmin(t) - v) = 0$. This means that for TPN models with unbounded firing intervals, infinitely many clock states may map to the same interval state. In such a case, all these states will obviously exhibit the same future behavior. In other words, they will be *bisimilar* to each other. The same remark also extends to state classes, where several state classes based on clocks (sometimes an infinity) may map to a single state class based on intervals. Note, also, that states in a clock state class can be distinguished one by one, whereas it is impossible with an interval state class. The reason is that the firing domain of an interval state class is the union of firing domains of all its states, and the union is known to be an irreversible operation. Together, the mentioned remarks suggest that the interval characterization of states has a more abstracting power than the clock characterization, and allows one to construct much more compact abstractions of the TPN state space. It is therefore a better choice in our context (see Table 1 in Section 8 for a comparison between the SCG and its equivalent computed using the clock characterization of states).

4. Relaxing state classes

We propose, in the following, to construct a relaxed version of the SCG, we call RSCG (Relaxed SCG), as an abstraction of the whole TPN state space, instead of just its concrete state space. The construction proceeds exactly as for the SCG, despite the fact that state classes are *relaxed* each time they are computed (including the initial state class). The *relaxation* of a state class consists in extending it with states reachable via time progressions. Knowing that delays decrease with time, the relaxation consists in replacing lower bounds of delay intervals with 0. Let $\alpha = (m, F)$ be a state class in canonical form. The relaxation of α, denoted by $relax(\alpha)$, is the class $\alpha' = (m, F')$ computed according to algorithm 4.

Algorithm 4: $relax(\alpha = (m, F))$

1 $F' = F$
2 **forall** *atomic constraint* f *of* F', *of the form* $o - t \prec_{F'}^{o-t} c$, *with* $t \in En(m)$ **do**
3 Replace f with $o - t \leq 0$ in F'
4 Return canonical form of $\alpha' = (m, F')$

In algorithm 4, steps 2 and 3 replace each atomic constraints of F of the form $o - t \prec_F^{o-t} c$ with $o - t \leq 0$, causing the lower bound of the firing domain of each transition to be retracted as close as possible to the bound zero. In fact, since F is in canonical form, in constraint $o - t \prec_F^{o-t} c$, $c = \mathrm{Sup}(o - t, F)$ with $c \leq 0$ and $o = 0$. This constraint can be written $-c \prec_F^{o-t} t$, and is extended to become $0 \leq t$. From the interval characterization perspective, this extension reflects a time progression, since firing delays decrease with time progression. Fig. 1 shows the relaxation of a state class with two enabled transitions.

Lemma 1. (i) *A state class and its relaxation (in canonical forms) have identical triangular constraints and upper bounds of transition delays.*
(ii) *A state class and its relaxation have the same successors, i.e.,* $succ(\alpha, t_f) = succ(relax(\alpha), t_f)$.

Proof. (i) Let $\alpha = (m, F)$ be a state class and $\alpha' = (m, F') = relax((m, F))$ its relaxed state class. F' is obtained from the canonical form of F by replacing constraints of the form $o - t \prec_{F'}^{o-t} \mathrm{Sup}(o - t, F), t \in En(m)$ with more larger constraints $o - t \leq 0$ and then putting the resulting formula in canonical form. It follows that:

- F' is consistent.

- $\forall t, t' \in En(m)$,
  $\mathrm{Sup}(t - t', F') = \mathrm{Sup}(t - t', F)$.
  $\mathrm{Sup}(t - o, F') = \mathrm{Sup}(t - o, F)$.
  $\mathrm{Sup}(o - t, F') \geq \mathrm{Sup}(o - t, F)$.

Then, (m, F) and its relaxed state class have identical triangular constraints and upper bounds of transition delays. Their lower bounds are more close to 0 in $relax((m, F))$ than in (m, F).

(ii) Note, first, that by $succ(relax(\alpha), t_f)$ we mean the direct successor of $relax(\alpha)$ before it is relaxed and not the one after applying the relaxation. The one after applying the relaxation (i.e., after letting some time to pass) would be $relax(succ(relax(\alpha), t_f))$, which is, in general, different from $succ(relax(\alpha), t_f)$.

Intuitively, since the computation of $succ(\alpha, t_f)$ already takes into account all states reachable from α by time progression (steps 3 and 4 in Algorithm 2), the fact of adding these states to α prior to computing the successor (i.e., using $relax(\alpha)$ instead of α) would not have any impact on the final result.

For a more formal proof, let us first give some definitions and notations. Let $\alpha = (m, F)$ be a state class and t_f a transition of $En(m)$ and $\alpha' = relax(\alpha) = (m, F')$.
A firing schedule of α is a function
$v : En(M) \cup \{o\} \to (\mathbb{R}^+ \cup \{\infty\})$ s.t. $v(o) = 0$ and

$$\bigwedge_{x,y \in En(m) \cup \{o\}} v(x) - v(y) \prec_F^{x-y} \mathrm{Sup}(x - y, F)).$$

Let v be a firing schedule of α, $d \in \mathbb{R}^+$, and $v + d$ the function over $En(M) \cup \{o\}$ defined by: $(v + d)(o) = 0$ and $\forall t \in En(m)$, $(v + d)(t) = v(t) + d$.
Let $FS(\alpha)$ be the set of firing schedules of α.
The proof of (ii) consists of showing that:

Claim 1- $FS(\alpha) \subseteq FS(\alpha')$.
Claim 2- For each firing schedule v' of α', there exists some firing schedule v of α and $d \geq 0$ such that $v = v' + d$.
Claim 3- t_f is firable from α iff t_f is firable from $relax(\alpha)$.

Claim 4- $succ(\alpha, t_f) = succ(relax(\alpha), t_f)$.
Proof of claim 1: is immediate since α' includes α.
Proof of claim 2: Let v' be a firing schedule of α'. Then:

$$\bigwedge_{t,t' \in En(m)} v'(t) - v'(t') \prec_F^{t-t'} \mathrm{Sup}(t - t', F) \ \wedge$$
$$\bigwedge_{t \in En(m)} v'(t) \prec_F^{t-o} \mathrm{Sup}(t - o, F) \ \wedge$$
$$\bigwedge_{t \in En(m)} - v'(t) \prec_F^{o-t} 0.$$

We have to show that there exists $d \geq 0$ such that:

$$\bigwedge_{t,t' \in En(m)} v'(t) + d - v'(t') - d \prec_F^{t-t'} \mathrm{Sup}(t - t', F)) \ \wedge$$
$$\bigwedge_{t \in En(m)} v'(t) + d \prec_F^{t-o} \mathrm{Sup}(t - o, F) \ \wedge$$
$$\bigwedge_{t \in En(m)} - v'(t) - d \prec_F^{o-t} \mathrm{Sup}(o - t, F).$$

Since $\forall d \in \mathbb{R}$, $v'(t) - v'(t') = v'(t) + d - v'(t') - d$, it suffices to show that: there exists $d \geq 0$ s.t.:

$$\bigwedge_{t \in En(m)} d \prec_F^{t-o} \mathrm{Sup}(t - o, F) - v'(t) \ \wedge$$
$$\bigwedge_{t \in En(m)} - d \prec_F^{o-t} \mathrm{Sup}(o - t, F) + v'(t).$$

Let x and y be respectively transitions of $En(m)$ s.t.

$$(\mathrm{Sup}(o - x, F) + v'(x), \prec_F^{o-x}) = \mathop{\mathrm{Min}}_{t \in En(m)} (\mathrm{Sup}(o - t, F) + v'(t), \prec_F^{o-t})$$
$$(\mathrm{Sup}(y - o, F) - v'(y), \prec_F^{y-o}) = \mathop{\mathrm{Min}}_{t \in En(m)} (\mathrm{Sup}(t - o, F) - v'(t), \prec_F^{t-o}).$$

Such a d exists iff:

$$-(\mathrm{Sup}(o-x,F)+v'(x),\prec_F^{o-x}) \leq (\mathrm{Sup}(y-o,F)-v'(y),\prec_F^{y-o}).$$

Or else

$$(v'(y)-v'(x),\leq) \leq (\mathrm{Sup}(y-o,F)+\mathrm{Sup}(o-x,F),\mathrm{Min}(\prec_F^{o-x},\prec_F^{y-o}).$$

The rest of the proof is straightforward from the fact that $v'(y)-v'(x) \prec_F^{y-x} \mathrm{Sup}(x-y,F)$ and

$$(\mathrm{Sup}(y-x,F),\prec_F^{y-x}) \leq ((\mathrm{Sup}(y-o,F)+(\mathrm{Sup}(o-x,F),\mathrm{Min}(\prec_F^{y-o}),\prec_F^{o-x})).$$

Proof of claim 3: t_f is firable from $\alpha=(m,F)$ iff $F \bigwedge_{t\in En(m)} t_f - t \leq 0$ is consistent.
Or else t_f is firable from $\alpha=(m,F)$ iff there exists a firing schedule v of α such that $\forall t \in En(m)$, $v(t_f) \leq v(t)$. Claims 1 and 2 allow us to state that such a firing schedule exists iff there exists a firing schedule $v+d$ of α' such that $d \in \mathbb{R}^+$ and $\forall t \in En(m)$, $(v+d)(t_f) \leq (v+d)(t)$.
Proof of claim 4: Consider steps 2 and 3 of algorithm 2 ($succ((m,F))$. Let $F2$ and $F3$ be, respectively, formulas obtained at steps 2 and 3. Firing schedules of $(m,F2)$ and $(m,F2)$ are respectively:

$$FS((m,F2)) = \{v \in FS((m,F)) \mid \forall t \in En(m),\ v(t_f) \leq v(t)\}$$
$$FS((m,F3)) = \{v - v(t_f) \mid v \in FS((m,F2))\}.$$

Let $F2'$ and $F3'$ be, respectively, formulas obtained at steps 2 and 3 for $relax((m,F))$. Firing schedules of $(m,F2')$ and $(m,F2')$ are respectively:

$$FS((m,F2')) = \{v' \in FS(relax((m,F))) \mid \forall t \in En(m),\ v'(t_f) \leq v'(t)\}$$
$$FS((m,F3')) = \{v' - v'(t_f) \mid v' \in FS((m,F2'))\} \quad \blacksquare$$

Claim 1 implies that $FS((m,F3)) \subseteq FS((m,F3'))$. Let $v' \in FS((m,F2'))$. Claim 2 states that there exists $v \in FS(relax(m,F))$ and $d \geq 0$ s.t. $v = v' + d$. Since $\forall t \in En(m)$, $v'(t_f) \leq v'(t)$, it follows that $\forall t \in En(m)$, $v(t_f) \leq v(t)$ and $v - v(t_f) = v' - v'(t_f)$. Therefore, $FS((m,F3)) = FS((m,F3'))$ and then $succ(\alpha,t_f) = succ(relax(\alpha),t_f)$. $\blacksquare$

The following theorem states some characteristics of the RSCG which make it more suitable than SCG for model checking linear properties of the TPN model.

Theorem 1. (i) *The RSCG preserves markings and traces of the complete TPN state space.*
(ii) *The RSCG is smaller or equal in size than the SCG.*
(iii) *The RSCG is finite for all bounded TPNs.*

Proof. (i) Recall that the construction of the RSCG proceeds exactly as for the SCG, except that state classes are *relaxed* each time they are computed, including the initial state class. From the fact that $isFirable((m,F),t_f)$ iff $isFirable(relax(m,F),t_f)$ and $succ(\alpha,t_f) = succ(relax(\alpha),t_f)$, we can conclude that state classes of the RSCG are relaxed state classes of the SCG. Since the relaxation of a state class consists in extending it with all states reachable by time progressions, it follows that the RSCG preserves markings and traces of the complete TPN state space.
(ii) Two different state classes may have identical relaxations. As an example, consider the two state classes $\alpha_1 = (m, 2 \leq t \leq 3)$ and $\alpha_2 = (m, 1 \leq t \leq 3)$. We have $relax(\alpha_1) = relax(\alpha_2) = (m, 0 \leq t \leq 3)$ while $\alpha_1 \neq \alpha_2$. So, because of the relaxation, the number of state classes of the RSCG gets reduced compared with the number of state classes of the SCG, as identical relaxed state classes get grouped in one class (see experimental results 8).
(iii) The proof is immediate from (ii) and the fact that the SCG is bounded for all bounded TPNs. $\blacksquare$

5. Our timed temporal Logic

We define a timed temporal logic for which we give algorithms to verify the satisfaction of its formulas in the context of the TPN model. The logic we consider is mostly a subset of the *TCTL* timed logic, for which atomic propositions are expressed on markings.

Let M be the set of reachable markings of a TPN model $\mathcal{N}$, and PR the set of propositions on M, i.e., $\{\wp | \wp : M \rightarrow \{true, false\}\}$. Before introducing our temporal logic we recall the syntax and semantics of *TCTL* logic in the context of the TPN model. The syntax of *TCTL* formulas is defined by the following grammar (in the grammar, $\wp \in PR$ and index I is an element of $\mathbb{Q}^+_{[\,]}$):

$$\varphi := \wp \mid \neg\varphi \mid \varphi \wedge \varphi \mid \forall(\varphi\ U_I\ \varphi) \mid \exists(\varphi\ U_I\ \varphi).$$

TCTL formulas are interpreted on states of a model $\mathcal{M} = (\mathcal{S}, \mathcal{V})$, where $\mathcal{S} = (S, \rightarrow, s_0)$ is the state space of the TPN model and $\mathcal{V} : S \rightarrow 2^{PR}$ is a valuation function such that: if $s = (m, I)$ is a TPN state, $\mathcal{V}(s) = \{\wp \in PR \mid \wp(m) = true\}$. To interpret a *TCTL* formula on an execution path, we introduce the notion of *dense execution path*. Let $s \in S$ be a TPN state and $\rho = s^0 \overset{\theta_1 t_1}{\rightarrow} s^1 \overset{\theta_2 t_2}{\rightarrow} s^2 \ldots$ an execution path such that $s^0 = s$ (i.e., $\rho \in \pi(s)$). The dense execution path corresponding to ρ is

the set defined by:
$\breve{\rho} = \{(s_{i\delta}, i, \delta) \in \mathcal{S} \times \mathbb{N} \times \mathbb{R}^+ \mid s_{i\delta} = s^i + \delta \wedge \delta \leq \theta_{i+1}\}$. Let $\prec \in \{<, \leq\}$ and $\succ \in \{>, \geq\}$ be relations over $\breve{\rho}$ defined by:
$\forall r = (s_{i\delta}, i, \delta), r' = (s_{i\delta'}, i', \delta') \in \breve{\rho}$,
- $r \prec r'$ iff $(i < i' \vee (i = i' \wedge \delta \prec \delta'))$.
- $r \succ r'$ iff $(i > i' \vee (i = i' \wedge \delta \succ \delta'))$.
Let $r = (s_{i\delta}, i, \delta) \in \breve{\rho}$, $\Theta_\rho(r) = (\sum_{j=0}^{i} \theta_j) + \delta$ (with $\theta_0 = 0$) and $\breve{\rho}(r) = s_{i\delta}$. The formal semantics of *TCTL* is given by the satisfaction relation $\models$ defined as follows:
- $\mathcal{M}, s \models p$ iff $p \in \mathcal{V}(s)$;
- $\mathcal{M}, s \models \neg p$ iff $p \notin \mathcal{V}(s)$;
- $\mathcal{M}, s \models \varphi \wedge \psi$ iff $\mathcal{M}, s \models \varphi$ and $\mathcal{M}, s \models \psi$;
- $\mathcal{M}, s \models \forall(\varphi U_I \psi)$ iff $\forall \rho \in \pi(s)$, s.t $time(\rho) = \infty, \exists r \in \breve{\rho}, \Theta_\rho(r) \in I, \mathcal{M}, \breve{\rho}(r) \models \psi$ and $\forall r' \in \breve{\rho}$ s.t. $r' < r, \mathcal{M}, \breve{\rho}(r') \models \varphi$;
- $\mathcal{M}, s \models \exists(\varphi U_I \psi)$ iff $\exists \rho \in \pi(s)$, s.t $time(\rho) = \infty, \exists r \in \breve{\rho}, \Theta_\rho(r) \in I, \mathcal{M}, \breve{\rho}(r) \models \psi$ and $\forall r' \in \breve{\rho}$ s.t. $r' < r, \mathcal{M}, \breve{\rho}(r') \models \varphi$.

The TPN model $\mathcal{N}$ is said to satisfy a *TCTL* formula ϕ iff $\mathcal{M}, s_0 \models \phi$. To ease *TCTL* formulas writing, some abbreviations are used: $\exists\Diamond_I\varphi = \exists(true U_I \varphi)$, $\forall\Diamond_I\varphi = \forall(true U_I \varphi)$, $\exists\Box_I\varphi = \neg\forall\Diamond_I\neg\varphi$, $\forall\Box_I\varphi = \neg\exists\Diamond_I\neg\varphi$. When interval I is omitted, its value is $[0, \infty[$ by default.

Our timed temporal logic, we call $TCTL_{TPN}$, is defined as follows:

$$TCTL_{TPN} ::= \exists(\wp_1 U_I \wp_2) \mid \forall(\wp_1 U_I \wp_2) \mid \wp_1 \mapsto_I \wp_2 \mid \exists\Diamond_I\wp_1 \mid \forall\Diamond_I\wp_1 \mid \exists\Box_I\wp_1 \mid \forall\Box_I\wp_1 \mid \wp_1 \rightsquigarrow_{I_r} \wp_2$$

$\wp_1$ and $\wp_2$ are propositions on markings (i.e., $\wp_1, \wp_2 \in PR$). Index I is an element of $\mathbb{Q}^+_{[}$. I_r is a time interval which starts from 0.

Formula $\wp_1 \rightsquigarrow_{I_r} \wp_2$ is a shorthand for the *TCTL* formula $\forall\Box(\wp_1 \Rightarrow \forall\Diamond_{I_r}\wp_2)$ which expresses a bounded response property.

Formula $\wp_1 \mapsto_I \wp_2$ expresses also a bounded response property, but with a slightly different semantics.
Intuitively, $\phi = \wp_1 \mapsto_I \wp_2$ holds at a state s iff for each execution path ρ starting from s, if $\wp_1$ is true for the first time on ρ at a state s', then $\wp_2$ should be true the first time at a state s'', reachable from s', within time interval I (starting from s'). Furthermore, ϕ must be recursively valid starting from s''. More precisely, this means that if $\wp_1$ holds the first time on ρ at state s', then:
1 - In case $\wp_2$ does not hold at state s' then $\wp_2$ will eventually hold the first time at a state s'', within time interval I (relatively to the time s' occurred), while ϕ holds also at state s''.
2 - In case $\wp_2$ holds at state s' then $\downarrow I$ must be equal to zero, and ϕ must hold at the first following state which does not satisfy both $\wp_1$ and $\wp_2$.
Formally, $\mathcal{M}, s \models \wp_1 \mapsto_I \wp_2$ iff $\forall \rho \in \pi(s)$, s.t $time(\rho) = \infty$, if $(\exists r' \in \breve{\rho}, (\mathcal{M}, \breve{\rho}(r') \models \wp_1)$ and $\forall r_1 \in \breve{\rho}$ s.t. $r_1 < r'$, $\mathcal{M}, \breve{\rho}(r_1) \not\models \wp_1$ then:
1 - $\mathcal{M}, \breve{\rho}(r') \not\models \wp_2 \Rightarrow \exists r'' \in \breve{\rho}, \mathcal{M}, \breve{\rho}(r'') \models \wp_2, \Theta_\rho(r'') - \Theta_\rho(r') \in I, \forall r_2 \in \breve{\rho}$ s.t. $r' < r_2 < r''$, $\mathcal{M}, \breve{\rho}(r_2) \not\models \wp_2$ and $(\mathcal{M}, \breve{\rho}(r'') \models \wp_1 \mapsto_I \wp_2)$.
2 - $\mathcal{M}, \breve{\rho}(r') \models \wp_2 \Rightarrow \downarrow I = 0$ and $\exists r'' \in \breve{\rho}$ s.t. $r'' \geq r'$, $(\mathcal{M}, \breve{\rho}(r'') \models \neg(\wp_1 \wedge \wp_2))$, $\forall r_2 \in \breve{\rho}$ s.t. $r' < r_2 < r''$, $(\mathcal{M}, \breve{\rho}(r_2) \models (\wp_1 \wedge \wp_2))$ and $(\mathcal{M}, \breve{\rho}(r'') \models \wp_1 \mapsto_I \wp_2)$.

In the sequel, $\wp_1 \mapsto_I \wp_2$ will be called the *bounded first response* property. One remark about this property is that it does not seem to have a simple TCTL equivalent[6] as is the case for the bounded response property. However, the next theorem states that, for intervals starting from 0, the bounded response and the bounded first response are equivalent.

Theorem 2. *$\mathcal{M}, s \models \wp_1 \mapsto_{I_r} \wp_2$ iff $\mathcal{M}, s \models \wp_1 \rightsquigarrow_{I_r} \wp_2$.*

Proof. Let b be the upper bound of I_r and $\rho \in \pi(s)$.
[⇒:] Let s' be the first state in ρ, starting from s where $\wp_1$ is true (if s' does not exist, the proof ends). Let s'' be the first state, in ρ starting from s' where $\wp_2$ is true. The semantics of $\wp_1 \mapsto_{I_r} \wp_2$ assures that the time θ which separates s' and s'' is such that $\theta \leq b$. From this, it is obvious that for each state s_i which is reached before s'' in ρ, if s_i satisfies $\wp_1$ then there exists a state s_j reachable, in ρ, from s_i within b time units s.t. s_j satisfies $\wp_2$ (in this case s_j would be s''). The recursive semantics of $\wp_1 \mapsto_{I_r} \wp_2$ assures also that the same property is also satisfied for the sub-path that starts from s''. This, in turn, assures that for any state s_i of ρ if s_i satisfies $\wp_1$ then there exists a state s_j reachable from s_i in ρ within b time units s.t. s_j satisfies $\wp_2$, which means that $\wp_1 \rightsquigarrow_{I_r} \wp_2$ is true for the path ρ.
[⇐:] Let s' be the first state, in ρ, starting from s where $\wp_1$ is true (if s' does not exist, the proof ends). Let s'' be the first state, in ρ, starting from s' where $\wp_2$ is true. From the semantics of $\wp_1 \rightsquigarrow_{I_r} \wp_2$ we can be sure that the time θ which separates s' and s'' is such that $\theta \leq b$. This property is also satisfied for the sub-path starting from s''. In other words $\wp_1 \mapsto_{I_r} \wp_2$ is true for the path ρ. ■

The following lemma asserts that two states with the same marking and identical traces have exactly the same $TCTL_{TPN}$ properties. Consequently, any TPN abstraction which preserves markings and traces, it also preserves $TCTL_{TPN}$ properties.

[6] In fact, it seems to have no equivalent at all. However, a full proof of this claim is necessary but is out of the scope of this paper.

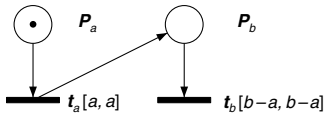

Fig. 2. The Alarm-clock TPN.

Lemma 2. *Let ϕ be a TCTL$_{TPN}$ formula, $s = (m, I)$ and $s' = (m', I')$ two states which have the same marking (i.e. $m = m'$) and identical traces (i.e. $\tau(s) = \tau(s')$).*
Then $\mathcal{M}, s \models \phi$ iff $\mathcal{M}, s \models \phi$.

Proof. The proof is immediate from the fact that two execution paths ρ of ρ' which have the same trace, satisfy the following relation: $\forall i \in \mathcal{N}$, states of $\breve{\rho}(i)$ and those of $\breve{\rho}'(i)$ have the same marking. ■

6. On-the-fly *TCTL$_{TPN}$* model checking

On-the-fly forward model checking of a TPN model $\mathcal{N}$ for a *TCTL$_{TPN}$* formula ϕ could be performed by progressively constructing an abstraction of the TPN state space that preserves its markings and traces (the SCG or the RSCG in our case), while checking the truth value of ϕ. The construction stops as soon as this truth value is established, which avoids computing the whole abstraction in most cases. This technique has proven to be very effective for model checking timed automata for a subset of *TCTL* formulas similar to *TCTL$_{TPN}$*, but without the bounded first response property [5]. Tools like UPPAAL [5] implement this technique and report better performances than when using standard model checking techniques [17].

First, we give an algorithm to model check the bounded first response property, then we show how to adapt this algorithm to model check the remaining *TCTL$_{TPN}$* properties. Note that all the following developments apply similarly to both the SCG and the RSCG. The SCG is considered for explanations.

6.1. Model checking the bounded first response property

Let $\mathcal{N}$ be a TPN model and $\phi = \wp_1 \mapsto_I \wp_2$ where $I = [a, b]$. Model checking ϕ on $\mathcal{N}$ could be performed by analyzing each execution path of $\mathcal{N}$'s SCG, until the truth value of ϕ is established. The SCG is progressively constructed, depth first, while looking for the satisfaction of property $\wp_1$. If $\wp_1$ is satisfied at a state class α, $\wp_2$ is looked for in each execution paths which starts from α (i.e., $\forall \rho \in \pi(\alpha)$). For each execution path $\rho \in \pi(\alpha)$, $\wp_2$ is required to be satisfied the first time at a state class α' such that the time separating α and α' is within the time interval I. If this is the case the verification of ϕ is restarted again from α', and so forth, until all state classes are explored. Otherwise, the exploration is stopped, and ϕ is declared invalid.

Two important issues need to be addressed in this technique: how to count time between the moments $\wp_1$ and $\wp_2$ are satisfied on a execution path, and how to deal with infinite paths resulting from cycles. To resolve these two issues, we propose to put the TPN model $\mathcal{N}$ in parallel with the TPN model of Fig. 2, we call *Alarm-clock*. The resulting TPN we denote $\mathcal{N}||Alarm$, is used instead of $\mathcal{N}$ to verify ϕ.

The verification of ϕ now proceeds as follows: During the generation of the SCG of $\mathcal{N}||Alarm$, if $\wp_1$ is satisfied in a state class $\alpha = (m, F)$, transition t_a is enabled in α to capture the event corresponding to the beginning of time interval I. t_a is enabled by changing the marking m in α such that place P_a would contain one token, and replacing F with $F \wedge t_a = a$. These two actions correspond to artificially putting a token in place P_a of *Alarm-clock*. The generation process continues while checking $\wp_2$. If $\wp_2$ is satisfied before t_a is fired, ϕ is declared invalid and the exploration stops. When t_a is fired (which means that time has come to start looking for $\wp_2$), t_b gets enabled in the resulting state class $\alpha' = (m', F')$ to capture the event corresponding to the end of interval I. If t_b is fired during the exploration, ϕ is declared invalid and the exploration stops. If before firing t_b, $\wp_2$ is satisfied in a state class $\alpha'' = (m'', F'')$, transition t_b is disabled in α'' by changing the marking m'' such that place P_b would contain zero tokens, and eliminating variable t_b from F''. These two actions corresponds to artificially removing the token in place P_b. After α'' is modified, ϕ is checked again starting from α''. 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.[7] 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 (see Section 6.3 for an illustrative example):

1- The case where $t_a, t_b \notin En(m)$ and $t \notin \{t_a, t_b\}$ corresponds to a situation where we are looking for $\wp_1$.
- In case $\wp_1$ is satisfied in α while $\wp_2$ is not, we enable t_a in α,
- In case $\wp_1$ and $\wp_2$ are both satisfied in α while $a \neq 0$, we stop the exploration and declare ϕ invalid.

[7] For uniformity reasons, we assume a fictitious transition t_ϵ as the transition which led to the initial state class.

2- The case where $t_a \in En(m)$ corresponds to a situation where $\wp_1$ has been satisfied before, and where we need to make sure that $\wp_2$ is not satisfied, unless $a = 0$. If $\wp_2$ is satisfied in α while $a > 0$, we stop the exploration and declare ϕ invalid.
3- The case where $t_b \in En(m)$ corresponds to a situation where we are looking for $\wp_2$. If $\wp_2$ is satisfied in α then we disable t_b and get in a situation where we are looking for $\wp_1$ (i.e., (1)).
4- The case where $t = t_b$ corresponds to a situation where interval I has expired while we are looking for $\wp_2$. In this case, we stop the exploration and declare ϕ invalid.

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, φ might be declared wrongly false if the resulting state class satisfies $\wp_2$. 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. As 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 (i.e., operation *isfirable*), and the way the successor of a state class $\alpha = (m, F)$, by a transition t, is computed (i.e., operation *succ*).

Algorithm 5: $isFirable_{AC}(\alpha = (m, F), t_f)$

1 **if** $t_f \notin En(m)$ **then** Return false
2 Let $F' = F \wedge (\bigwedge_{t \in En(m)-\{t_f\}} t_f \leq t)$
3 **if** $t_a \in En(m) \wedge t_f \neq t_a$ **then**
4 $\quad F' = F' \wedge t_f < t_a$
5 **else if** $t_b \in En(m) \wedge t_f = t_b$ **then**
6 $\quad F' = F \wedge (\bigwedge_{t \in En(m)-\{t_f\}} t_f < t)$
7 **if** F' *is consistent* **then** Return true
8 Return false

$isFirable_{AC}(\alpha, t_f)$ replaces $isFirable(\alpha, t_f)$ to check whether a transition is firable or not. What changes is the way formula F' is computed. In case t_a is enabled while we want to fire a different transition t_f (step 4), we need to make sure that t_f is fired ahead of time of t_a. In case t_b is enabled and is the one we want to fire (step 6), we need to make sure that t_b is the only transition that can be fired. The remaining cases are handled exactly as before.

$succ_{AC}(\alpha, t_f)$ replaces $succ(\alpha, t_f)$ for generating successor state classes during the exploration. What changes is also the way formula F' is computed.

Algorithm 6: $succ_{AC}(\alpha = (m, F), t_f)$

1 Let $m'(p) = m(p) - Pre(p, t_f) + Post(p, t_f), \forall p \in P$
2 **if** $t_f \notin En(m)$ **then** Return false
3 Let $F' = F \wedge (\bigwedge_{t \in En(m)-\{t_f\}} t_f \leq t)$
4 **if** $t_a \in En(m) \wedge t_f \neq t_a$ **then**
5 $\quad F' = F' \wedge t_f < t_a$
6 **else if** $t_b \in En(m) \wedge t_f = t_b$ **then**
7 $\quad F' = F \wedge (\bigwedge_{t \in En(m)-\{t_f\}} t_f < t)$
8 Replace in F' each variable t with $t + t_f$
9 Eliminate by substitution, in F', t_f and all variables associated with transitions conflicting with t_f for m
10 **forall** $t \in New(m', t_f)$ **do**
11 $\quad$ Add to F' the constraint $tmin(t) \leq t \leq tmax(t)$
12 Return (m', F')

6.2. Model checking algorithms

The on-the-fly $TCTL_{TPN}$ model checking of formula ϕ is based on the exploration algorithm 7.

The algorithm uses two lists: WAIT and COMPUTED, to manage state classes, and calls a polymorphic satisfaction function $checkStateClass_\phi$ to check the validity of formula ϕ. COMPUTED contains all computed state classes, while WAIT contains state classes of COMPUTED which are not yet explored. As a consequence WAIT is just a sublist of COMPUTED.[8] The algorithm

[8] From an implementation point of view, the list WAIT is a list of references to state classes in the list COMPUTED.

Algorithm 7: *modelChek*(ϕ)

```
1  Let continue=true                                              /* global variable */
2  Let valid=true                                                 /* global variable */
3  Let COMPUTED= ∅
4  Let α0= (m0, F0)
5  Let α'0=checkStateClassφ(α0, tε)
6  Let WAIT={α'0}
7  while continue do
8      remove α = (m, F) from WAIT
9      forall t ∈ En(m) s.t. isFirable_AC(α, t) provided continue do
10         α':=succ_AC(α, t)
11         α'':=checkStateClassφ(α', t)φ
12         if continue ∧ α'' ≠ ∅ ∧ ∄αp ∈ COMPUTED s.t. α'' ⊆ αp then
13             forall αp ∈ COMPUTED s.t. αp ⊆ α'' do
14                 remove αp from COMPUTED and from WAIT
15             Add α'' to COMPUTED and to WAIT
16 Return valid
```

generates state classes by firing transitions. The initial state class is supposed to result from the firing of a fictive transition t_ϵ. Each time a state class α is generated as the result of firing a transition t, α and t are supplied to *checkStateClass*$_\phi$ to perform actions and take decisions. In general, *checkStateClass*$_\phi$ enables or disables transitions t_a and t_b in α. It also takes decisions and records them in two global boolean variables *continue* and *valid*, to guide the exploration process. Finally, it returns either α after modification or $\emptyset$ in case α 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 ϕ. After *checkStateClass*$_\phi$ is called, the state class α' it returns is inserted in the list WAIT only if it is not included in a previously computed state class (i.e., $\nexists\alpha \in$ COMPUTED s.t. $\alpha' \subseteq \alpha$). Otherwise, α' is inserted in the list WAIT, while all state classes of the list COMPUTED which are included into α' are deleted from both COMPUTED and WAIT. This strategy, which is also used in the tool UPPAAL [5], attenuates considerably the state explosion problem. So instead of exploring both α and α', exploring α' is sufficient. Operation *checkStateClass*$_\phi$ takes as parameters: a state class, and the transition that led to it. Three different implementations of *checkStateClass*$_\phi$ are required for the three principal forms of ϕ, i.e., $\wp_1 \mapsto_I \wp_2$, $\forall(\wp_1 U_I \wp_2)$ and $\exists(\wp_1 U_I \wp_2)$, with $I = [a, b]$ (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 (algorithm 8) corresponds to property $\phi = \wp_1 \mapsto_I \wp_2$. Its steps match exactly those described in Section 6.1. 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_a is enabled, a state class where transition t_b is enabled, and a state class reached by the firing of transition t_b.

Algorithm 8: *checkStateClass*$_{\wp_1 \mapsto_I \wp_2}(\alpha = (m, F), t)$

```
1  if ta, tb ∉ En(m) ∧ t ∉ {ta, tb} then                          /* case 1 */
2      if ℘1(m) ∧ ¬℘2(m) then
3          enable ta in α;
4      if ℘1(m) ∧ ℘2(m) ∧ a > 0 then
5          valid=false; continue=false ;
6  if ta ∈ En(m) ∧ ℘2(m) then                                     /* case 2 */
7      valid=false; continue=false;
8  if tb ∈ En(m) ∧ ℘2(m) then                                     /* case 3 */
9      disable tb in α;
10 if t = tb then                                                 /* case 4 */
11     valid=false ; continue=false;
12 Return α;
```

The second implementation (algorithm 9) corresponds to property $\phi = \forall(\wp_1 U_I \wp_2)$. In the 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 $\phi = \forall(\wp_1 U_I \wp_2)$ checks if proposition $\wp_1$ is true in the initial state class (step 2) and all state classes following it (step 8), until t_a fires. From the moment t_a is fired, the verifier checks for the satisfaction of either $\wp_1$ or $\wp_2$ (steps 11 and 12), until $\wp_2$ is true or t_b is fired (case 4). If $\wp_2$ becomes true in

Algorithm 9: $checkStateClass_{\forall(\wp_1 U_l \wp_2)}(\alpha = (m, F), t)$

```
1  if t = t_ε then                                                    /* case 1 */
2      if ℘1(m) then
3          enable t_a in α
4      else
5          if ¬℘2(m) ∨ a > 0 then
6              valid=false; continue=false
7          else
8              valid=true; continue=false
9  if t_a ∈ En(m) ∧ ¬℘1(m) then                                        /* case 2 */
10     valid=false; continue=false
11 if t_b ∈ En(m) then                                                /* case 3 */
12     if ¬℘2(m) then
13         if ¬℘1(m) then valid=false; continue=false
14     else
15         Return ∅
16 if t = t_b then                                                    /* case 4 */
17     valid=false; continue=false
18 Return α
```

Algorithm 10: $checkStateClass_{\exists(\wp_1 U_l \wp_2)}(\alpha = (m, F), t)$

```
1  if t = t_ε then                                                    /* case 1 */
2      valid=false
3      if ℘1(m) then
4          enable t_a in α
5      else
6          if ¬℘2(m) ∨ a > 0 then
7              continue=false
8          else
9              valid=true; continue=false
10 if t_a ∈ En(m) then                                                /* case 2 */
11     if ¬℘1(m) then Return ∅
12 if t_b ∈ En(m) then                                                /* case 3 */
13     if ℘2(m) then
14         valid=true; continue=false
15     else
16         if ¬℘1(m) then Return ∅
17 if t = t_b then                                                    /* case 4 */
18     Return ∅
19 Return α
```

a state class α, α is no more explored (step 13). In case t_b is fired (case 4), the exploration is stopped and the property is declared invalid.

The last implementation of $checkStateClass_\phi$ (algorithm 10) corresponds to property $\phi = \exists(\wp_1 U_l \wp_2)$. 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 (step 2), and stops the exploration of a state class α if it does not comply with the semantics of ϕ (steps 10, 15 and 17). It also aborts the exploration as soon as a satisfactory execution path is found (steps 8 and 13).

The following theorem states the decidability of our model checking approach for all bounded TPNs.

Theorem 3. *$TCTL_{TPN}$ model checking is decidable for Bounded TPN models.*

Proof. If a TPN model $\mathcal{N}$ is bounded, it has a finite number of markings. The fact of putting $\mathcal{N}$ in parallel with $Alarm - clock$, which is itself bounded (it has only three different markings $\{(P_a = 0, P_b = 0), (P_a = 1, P_b = 0), (P_a = 0, P_b = 1)\}$) also results in a bounded TPN model. Since the number of possible state classes with the same marking is always finite [9], The

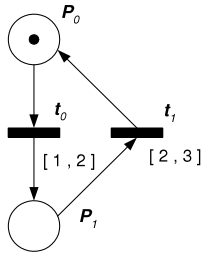

Fig. 3. *A cyclic* TPN model.

state class graph of $\mathcal{N}||Alarm$ is finite too. In other words, $TCTL_{TPN}$ model checking algorithm always terminates for bounded TPN models. ■

6.3. Illustrative example

To illustrate our verification approach, we consider the simple TPN model shown in Fig. 3, we call *cyclic*. The $TCTL_{TPN}$ property we verify is $\phi = \wp_1 \rightsquigarrow_{[0,3]} \wp_2$, with proposition $\wp_1(m) = (m(P_0) = 0)$ and proposition $\wp_2(m) = (m(P_1) = 1)$. For simplicity reasons, we selected a cyclic TPN model with a single execution path, for which property ϕ is trivially valid.

The verification process of ϕ starts first by constructing the TPN model *cyclic*||*Alarm*, such that $a = 0$ and $b = 3$, then runs according to the following steps (a marking is demoted only by its marked places):

(1) Compute the initial state class of *cyclic*||*Alarm* (step 4, algorithm 7),
result: $\alpha_0 = ((P_0 = 1), 1 \leq t_0 \leq 2)$.
(2) Check if $\wp_1$ and $\wp_2$ are valid in α_0, (case 1, algorithm 8)
result: $\wp_1$ and $\wp_2$ are not valid in α_0.
(3) Fire t_0 from α_0 and put the result in α_1 (step 10, algorithm 7),
result: $\alpha_1 = ((P_1 = 1), 2 \leq t_1 \leq 3)$.
(4) Check if $\wp_1$ and $\wp_2$ are valid in α_1 (case 1, algorithm 8)
result: $\wp_1$ is valid in α_1 but $\wp_2$ is not valid in α_1.
(5) Enable t_a in α_1 (step 3, algorithm 8),
result: α_1 becomes $((P_1 = 1, P_a = 1), 2 \leq t_1 \leq 3 \wedge t_a = 0)$.
(6) Fire t_a from α_1 and put result in α_2 (step 10, algorithm 7)
result: α_2 = $((P_1 = 1, P_b = 1), 2 \leq t_1 \leq 3 \wedge t_b = 3)$.
(7) Check if $\wp_2$ is satisfied in α_2 (step 8, algorithm 8),
result: $\wp_2$ is not satisfied in α_2.
(8) Fire t_1 from α_2 and put the result in α_3 (step 10, algorithm 7),
result: $\alpha_3 = ((P_0 = 1, P_b = 1), 1 \leq t_0 \leq 2 \wedge 0 \leq t_b \leq 1)$.
(9) Check if $\wp_2$ is satisfied in α_3 (step 8, algorithm 8),
result: $\wp_2$ is satisfied in α_3.
(10) Disables t_b in α_3 (step 9, algorithm 8),
result: α_3 becomes $((P_0 = 1), 1 \leq t_0 \leq 2)$.
(11) Declare ϕ valid since α_3 has already been explored $(\alpha_3 = \alpha_0)$ (step 16, algorithm 7).

7. Comparing with UPPAAL's reachability algorithm

UPPAAL [5] is a tool for modeling, simulation and verification of real time systems, based on timed automata [4]. The tool allows to verify systems modeled as a collection of processes (timed automata) communicating through channels and shared integer variables. Typical applications include real time controllers and communication protocols. The verification engine of UPPAAL is based on the classical reachability algorithm 11. The algorithm checks whether or not a state satisfying a given *state formula*[9] β is reachable from an initial configuration of the system to verify.

Properties that UPPAAL can check is a subset of CTL. The four temporal quantifiers E<>, A[], E[] and A<> are supported, which stand for *possibly* ($\exists\Diamond$), *always* ($\forall\Box$), *inevitably* ($\exists\Box$), and *potentially always* ($\forall\Diamond$). In addition, the operator $\beta_1 -- > \beta_2$ is supported, which stands for the *leads-to* property A[]($\beta_1 \Rightarrow$ A<> β_2).

The verification of timed properties is achieved by explicitly using clock variables in state formulas. While this may allow for great versatility, it may also oblige the user to modify the model and use extra clocks for a specific timed property. Using extra clocks also means extra burden on the verification, which is exponential in number of clocks. To verify properties that are not pure reachability, such as *bounded liveness* and *bounded response*, it is necessary to compute a *test automaton* for

[9] An atomic proposition which is a combination of control nodes and constraints on clocks and integer variables.

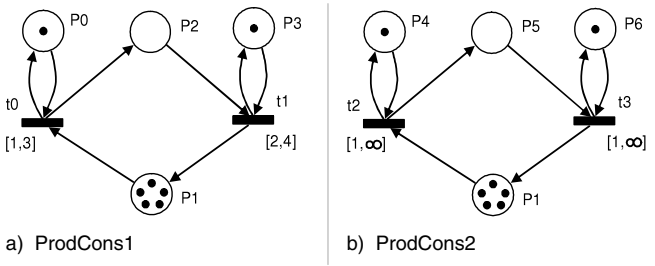

Fig. 4. The producer/consumer model.

the property to verify, and turns the verification problem to a reachability problem of the original automaton synchronized with the test automaton [2]. The truth value of the formula is established if a *fail state* of the test automaton is reached in the synchronized automaton, or if the abstract state space construction terminates.

Compared to the classical reachability algorithm 11, our exploration algorithm (i.e., Algorithm 7) shares lots of similarities, even if the verification strategy is different. This difference results mainly from the way states and state classes are characterized in both methods. UPPAAL uses the clock characterization of states. In our case, we use the interval characterization of states for two main reasons, even if the clock characterization is possible with TPNs [13–15,20]. First, the interval characterization is more abstracting than the clock characterization, and therefore, leads to smaller state class spaces to explore (see Table 2 for some experimental results). Second, the interval characterization requires no approximation to enforce finiteness. On the other hand, the clock characterization requires an approximation, called *k-approximation* or *k-normalization*, which is very sensitive to the magnitude of the biggest constant with which clocks are compared in both the formula and the model to verify [6]. In fact, the size of the state class space grows exponentially with the magnitude of this constant [4]. Table 7 in the next section effectively illustrates this situation.

Algorithm 11: UPPAAL reachability algorithm

```
1  PASSED:={}
2  WAITING:={(l_0, D_0))}
3  repeat
4      Get (l, D) from WAITING
5      if (l, D) ⊨ β then
6          Return "YES"
7      else if D ⊈ D' for all (l, D') ∈ PASSED then
8          Add (l, D) to PASSED
9          SUCC:={ (l_s, D_s): (l, D) ⇝ (l_s, D_s) ∧ D_s ≠ ∅ }
10         forall (l_s', D_s') in SUCC do
11             Put (l_s', D_s') to WAITING
12 until WAITING={}
13 return "NO"
```

8. Implementation results

We implemented our verification approaches in our experimental tool called RT-Studio. The tool, written in JAVA and C++, integrates several functionalities related to enumerative analysis of time Petri nets. It also includes a *CTL* model checker and a minimizer under bisimulation. All tests have been performed on a 3 Gigahertz Pentium-4 with two Gigabytes of RAM.

For our experimentations, we considered two classical examples: The *producer/consumer* model and the *level crossing model*. We tested several possible configurations of these models constructed by properly combining and synchronizing their TPN components shown in Figs. 4 and 5 respectively. The TPN model of n concurrent consumers and n producers sharing a single limited store, is obtained by the parallel composition of $n-1$ copies of the TPN in Fig. 4.b with one copy of the TPN in Fig. 4.a, while merging all places named P1 in one single place. We denote this model $P(n)$. The TPN model of n trains concurrently crossing the level is obtained by synchronously composing[10] the controller model with its parameter m[11] set to n, the barrier model, and n copies of the train model. The resulting model is denoted $T(n)$.

[10] Transitions with the same name are merged together.

[11] m is a number of tokens.

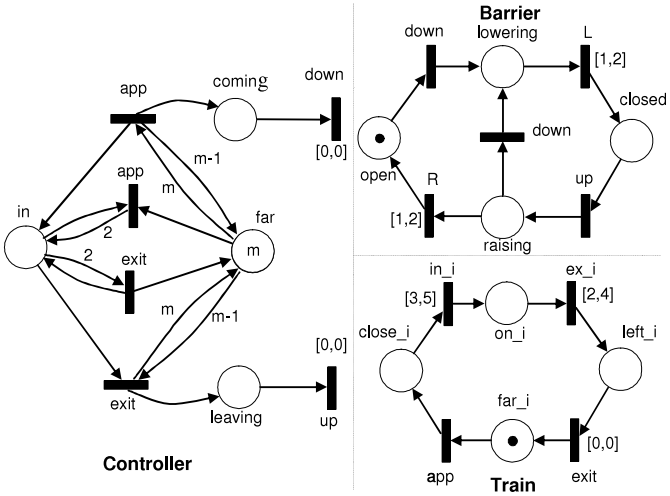

Fig. 5. The level crossing TPN model.

Our experimental results are presented in two subsections. In the first subsection, we give results to show the impact of some of our choices on performances. In the second subsection, we evaluate our verification technique on the level crossing example and compare our results with those of the tool UPPAAL 3.5.7.

8.1. Impact of some of our choices on performances

Results of this section are organized in three tables. In each table, each row reports results obtained for the model given in the first column. Results for a TPN model abstraction are given in terms of its size (nodes/arcs) and its computing time (in seconds). An *interrogations mark* indicates a situation where the computation has not completed after an hour, or the construction aborted for a memory shortage.

Table 1 shows the abstracting power of the interval characterization of states over the clock characterization (see Section 3.3 for more details). It compares the size of the SCG computed using the interval characterization of states, with its equivalent, computed using the clock characterization. In this last case we denote the resulting graph clock-SCG. This graph is computed according to the firing rule described in [14] for computing the CSCG,[12] but with state classes normalized using the *k-normalization* operation of timed automata implemented in the tool UPPAAL [7]. Let k be the higher finite constant appearing in the firing time intervals of the TPN. The k-normalization of a state class is computed based on its canonical form. It is to remove all upper bounds higher than k and replace all lower bounds, higher than k, by k. Column four of Table 1 gives, where appropriate, the ratio of the clock-SCG results over those of the SCG. Note that for the TPN $P(2)$, the SCG is 45.54 times smaller than the clock-SCG and 50.72 times faster to compute. The construction of $P(3)$ and $P(4)$ *clock − SCGs* have aborted because of a memory shortage.

Table 2 shows the impact on performances of both the state class relaxation introduced in Section 4, and the on-the-fly exploration technique we used to verify $TCTL_{TPN}$ properties. Column three and four give results for contracting the SCG and the RSCG by inclusion. These contractions correspond in a certain sense to the actual graphs explored by our model-checking approach during the verification of $TCTL_{TPN}$ properties. The resulting graphs, we denoted I-SCG and I-RSCG respectively, are computed similarly to the SCG and the RSCG, but with grouping, during the construction process, state classes in the most including ones. A state class (m, F) is grouped into another state class (m', F') iff $m = m'$ and $F \subseteq F'$.

To get a better idea about improvements, we report in Table 3 improvement ratios for results of the different contractions (RSCG, I-SCG and I-RSCG) over those of the SCG.

8.2. Evaluation of the TCTL verification technique

In this section, we report some results obtained for the verification of some $TCTL_{TPN}$ properties over the classical *level crossing* example. We also compare our results with those obtained using UPPAAL.

Fig. 6 shows a timed automata version of the level crossing example. The controller part of the timed automata is obtained by the parallel composition of three separate automata to maintain an equivalent semantics as the TPN version of the model.

[12] The CSCG is a version of the SCG computed using the clock characterization of states and contracted by inclusion (two state classes are merged together whenever one is included in the other).

Table 1
SCG compared to the clock-SCG.

TPN	SCG	clock-SCG	$\theta^{SCG}_{clock-SCG}$
P(2)	748 / 2460	3924 / 13222	5.344
cpu	0.04	0.18	4.5
P(3)	4604 / 21891	205129/1001659	45.54
cpu	0.40	20.29	50.72
P(4)	14086 / 83375	?	–
cpu	1.83		–
P(5)	31657 / 217423	?	–
cpu	5.67		–
T(1)	11 / 14	12 / 15	1.08
cpu	0	0.00	–
T(2)	123 / 218	146 / 264	1.20
cpu	0	0.01	–
T(3)	3101 / 7754	6174 / 16190	2.06
cpu	0.09	0.17	1.88
T(4)	134501 / 436896	664006/2317692	5.21
cpu	6.33	47.42	7.49
T(5)	?	?	–
cpu			–

Table 2
SCG, RSCG, I-SCG and I-RSCG of the tested models.

TPN	SCG	RSCG	I-SCG	I-RSCG
P(2)	748/2460	593/1922	41/137	30/94
cpu	0.04	0.01	0	0
P(3)	4604/21891	3240/15200	121/581	121/581
cpu	0.40	0.14	0.01	0.01
P(4)	14086/83375	9504/56038	275/1631	197/1051
cpu	1.83	0.62	0.03	0.01
P(5)	31657/217423	20877/145037	514/3604	367/2175
cpu	5.67	2.01	0.07	0.04
T(1)	11/14	11/13	10/13	10/13
cpu	0	0	0	0
T(2)	123/218	113/198	37/74	35/70
cpu	0	0	0	0
T(3)	3101/7754	2816/6941	172/494	166/476
cpu	0.09	0.07	0.01	0
T(4)	134501/436896	122289/391240	1175/4599	1151/4475
cpu	6.33	5.74	0.16	0.08
T(5)	?	?	10972/55682	10852/53573
cpu	$>$ 3600.00	$>$ 3600.00	2.04	1.81

Table 3
Improvement ratios of the RSCG, I-SCG and I-RSCG abstractions over those of the SCG.

TPN	RSCG	I-SCG	I-RSCG
P(2)	1.28	18.02	25.87
cpu	4.00	–	–
P(3)	1.44	37.74	37.74
cpu	2.86	40.00	40.00
P(4)	1.49	51.13	78.09
cpu	2.95	61.00	183.00
P(5)	1.50	60.49	97.99
cpu	2.82	81.00	141.75
T(1)	1.04	1.09	1.09
cpu	–	–	–
T(2)	1.10	3.07	3.25
cpu	–	–	
T(3)	1.11	16.30	16.91
cpu	1.29	9.00	–
T(4)	1.11	98.96	101.56
cpu	1.10	39.56	79.13
T(5)	–	–	–
cpu	–	$>$ 1764.70	$>$ 1988.95

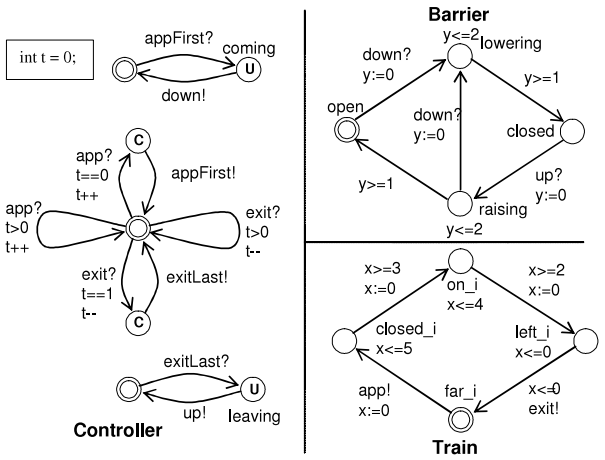

Fig. 6. The level crossing timed automata model.

The properties we considered are:

1- The gate is never open whenever a train is crossing:

$$\phi_1 = \forall\Box\neg\left(open \wedge \bigvee_{1\le i\le n} on\_i\right)$$

2- If a train approaches, the gate closes in less than 2 time units:

$$\phi_2 = coming \rightsquigarrow_{[0,2]} closed$$

3- The level crossing model is deadlock free:

$$\phi_3 = \forall\Box(\exists t \in En(m)).$$

The corresponding properties for UPPAAL are:

1- ϕ_1^u = A[]!($barrier$.open && ($train_1$.on_i||..||$train_n$.on_i)),
2- ϕ_2^u = $ctrlUp$.coming=>($barrier$.closed&&barrier.y<=2),
3- ϕ_3^u = A[] not deadlock.

Where $barrier$, $train_i$, $ctrlUp$ are, respectively, instances of **Barrier**, **Train** and the upper part of **Controller** in Fig. 6.

Note that we considered only properties which require an exhaustive exploration of the TPN SCG. To ease the writing of properties, we coincide atomic propositions with places. An atomic proposition is true iff the corresponding place is marked with at least one token.

Table 4 reports results obtained for model checking the selected properties using our approach, applied on both the SCG and the RSCG. Each result is given in terms of the number of *stored/explored* state classes (i.e., the final size of the list COMPUTED (PASSED for UPPAAL) and the total number of explored state classes), followed by the exploration time. Note that all selected properties have been successfully tested as valid.

Similarly to Table 4, Table 5 reports the results obtained using the tool UPPAAL for the same properties.

Table 6 compares the results of Table 5 to those of Table 4 (i.e, ratios of UPPAAL's results to those of RT-Studio). On average, properties were verified 13.85 times faster than UPPAAL, with 1.48 times lesser memory usage, using the SCG. Further improvements were made using the RSCG, with average ratios of 14.64 for time and 1.67 for space compared to UPPAAL.

Table 7 compares our results to those of UPPAAL for property $\phi(b) = coming \rightsquigarrow_{[0,b]} closed$, where b takes increasing values. Column four gives the ratios of UPPAAL's results to those of RT-Studio. From the obtained results, we can see that our approach is not sensitive to the magnitude of b, while it is for UPPAAL. The reason for this difference is related to the used characterization of states. In our case, if the property is already valid, the fact of increasing the values of parameters b has no impact on the verification process. For UPPAAL, states are characterized using clocks. This characterization requires an approximation operation to enforce finiteness (see Section 7). The size of the *symbolic state graph* (the equivalent of the SCG) to explore grows considerably with the magnitude of the biggest constant with which clocks are compared. As an example, the last row of Table 7 shows that for $b = 100$ our approach was 3485 times faster than UPPAAL, and used 120 times lesser memory. Note that UPPAAL uses a technique to attenuate this problem which can give rise to an unnecessary fragmentation of the symbolic state space [5,24]. The problem generally appears when the timed automata in a model use different time scales. As a result, model-checking requires more time and memory. UPPAAL uses an acceleration technique $<$ for a subset of

Table 4
Results for some $TCTL_{TPN}$ properties using our approach.

	TPN	ϕ_1	ϕ_2	ϕ_3
SCG	2 trains	38/116	42/91	38/116
	CPU (s)	0	0	0
	3 trains	173/790	182/646	173/790
	CPU (s)	0	0.01	0.01
	4 trains	1176/7162	1194/6073	1176/7162
	CPU (s)	0.12	0.10	0.12
	5 trains	10973/81370	11008/71152	10973/81370
	CPU (s)	2.37	2.04	2.30
	6 trains	128116/1103250	128184/986939	128116/1103250
	CPU (s)	110.81	100.92	111.18
RSCG	2 trains	36/110	40/87	36/110
	CPU (s)	0	0	0
	3 trains	167/706	176/583	167/706
	CPU (s)	0	0	0
	4 trains	1156/6122	1170/5241	1152/6122
	CPU (s)	0.12	0.08	0.12
	5 trains	10853/67660	10888/59727	10853/67660
	CPU (s)	2.22	1.90	2.24
	6 trains	127396/902658	127464/815003	127396/902658
	CPU (s)	100.97	90.02	100.06

Table 5
Results for some $TCTL_T PN$ properties using UPPAAL.

TPN	ϕ_1^u	ϕ_2^u	ϕ_3^u
2 trains	36/89	54/117	38/101
CPU (s)	0.01	0.02	0.02
3 trains	176/601	344/1306	194/826
CPU (s)	0.02	0.40	0.30
4 trains	1372/5859	3084/18243	1456/8947
CPU (s)	0.100	0.51	0.49
5 trains	14758/74047	35218/277612	14408/115182
CPU (s)	3.06	36.08	13.72
6 trains	192022/1106973	?	173332/1692129
CPU (s)	267.88		806.20

Table 6
Improvement ratios of our results compared to those of UPPAAL.

TPN		$\theta^{\phi_1^u}_{\phi_1}$	$\theta^{\phi_2^u}_{\phi_2}$	$\theta^{\phi_3^u}_{\phi_3}$
SCG	2 trains	0.81	1.29	0.90
	cpu	–	–	–
	3 trains	0.81	1.99	1.06
	cpu	–	40.00	30.00
	4 trains	0.87	2.93	1.25
	cpu	0.83	5.10	4.08
	5 trains	0.96	3.81	1.40
	cpu	1.29	17.69	5.97
	6 trains	1.05	–	1.51
	cpu	2.42	> 35.67	7.25
RSCG	2 trains	0.86	1.35	0.95
	cpu	–	–	–
	3 trains	0.89	2.17	1.17
	cpu	–	>40	>30
	4 trains	0.99	3.33	1.43
	cpu	0.83	6.38	4.08
	5 trains	1.13	4.43	1.65
	cpu	1.38	18.99	6.13
	6 trains	1.26	–	1.81
	cpu	2.65	39.99	8.06

timed automata, namely those that contain special cycles,[13] using a syntactical adjustment. In the case of the level crossing example we verified here, the acceleration technique does not apply, which explains the big difference in performances.

[13] The subset of cycles that can be accelerated may use only a single clock y in the invariants, guards and resets.

Table 7
Our results compared to those of UPPAAL for the property $\phi(b) = Coming \rightsquigarrow_{[0,b]} closed$, with increasing values for b.

TPN	Ours	UPPAAL	θ_{Ours}^{UPPAAL}
$\phi(2)$	1194/6073	3084/18243	2.93
cpu	0.10	0.51	5.10
$\phi(10)$	1194/6073	16301/65792	11.30
cpu	0.v 10	4.30	43.00
$\phi(20)$	1194/6073	32422/131420	22.55
cpu	0.10	13.60	136.00
$\phi(30)$	1194/6073	48950/197212	33.87
cpu	0.10	29.83	298.30
$\phi(50)$	1194/6073	84686/341231	58.61
cpu	0.10	88.43	884.30
$\phi(100)$	1194/6073	173658/695112	119.55
cpu	0.10	348.48	3484.80

9. Conclusion

In this paper, we considered the time Petri net model and proposed an efficient model checking approach to verify its timed properties. Our approach is based on the *state class* method [9], defined originally to verify untimed properties. To attenuate the state explosion problem, we use an on-the-fly exploration technique combined with an abstraction by inclusion. This technique has already been proven very effective to model check timed properties for timed automata [5,26]. We also relax state classes by extending them with states reachable via time progressions to further improve performances. The timed properties we considered are mainly a subset of *TCTL*, sufficient in general to verify most important timed properties. We have also proven the decidability of our verification technique for all bounded time Petri nets, and compared our results with those of the tool UPPAAL [5] to show its effectiveness.

Acknowledgment

Research supported by the NSERC grant num.238841-2001.

References

[1] P.A. Abdulla, A. Nyln, Timed Petri Nets and BQOs, in: Proc. of ICATPN'01, in: LNCS, vol. 2075, Springer-Verlag, 2001, pp. 53–70.
[2] L. Aceto, P. Bouyer, A. Burgueno, K.G. Larsen, The power of reachability testing for timed automata, Theoretical Computer Science 300 (1-3) (2003) 411–475.
[3] R. Alur, C. Courcoubetis, D. Dill, Model checking in dense real-time, Information and Computation 104 (1) (1993) 2–34.
[4] R. Alur, D. Dill, Automata for modelling real-time systems, in: Proc. Of ICALP'90, in: LNCS, 443, Springer-Verlag, 1990, pp. 322–335.
[5] G. Behrmann, J. Bengtsson, A. David, K. G. Larsen, P. Pettersson, W. Yi, Uppaal implementation secrets, in: Proc. of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, 2002.
[6] J. Bengtsson, W. Yi, On clock difference constraints and termination in reachability analysis in timed automata, in: Proc. of ICFEM'03, in: LNCS, vol. 2885, Springer-Verlag, 2003, pp. 491–503.
[7] J. Bengtsson, W. Yi, Timed automata: Semantics, algorithms and tools., in: Lectures on Concurrency and Petri Nets, 2003, pp. 87–124.
[8] B. Berthomieu, M. Diaz, Modeling and verification of time dependent systems using Time Petri Nets, IEEE Transactions on Software Engineering 17 (3) (1991) 259–273.
[9] B. Berthomieu, M. Menasche, An enumerative approach for analyzing Time Petri Nets, in: Proc. of the IFIP 9th World Computer Congress, in: Information Processing, vol. 9, IFIP, North Holland, 1983, pp. 41–46.
[10] B. Berthomieu, F. Vernadat, State class constructions for branching analysis of Time Petri Nets, in: Proc. of TACAS'03, in: LNCS, vol. 2619, Springer-Verlag, 2003, pp. 442–457.
[11] A. Bouajjani, S. Tripakis, S. Yovine, On-the-fly symbolic model checking for real-time systems, in: Proc. of RTSS'97, 1997, pp. 232–243.
[12] H. Boucheneb, G. Berthelot, Towards a simplified building of Time Petri Nets reachability graph, in: Proc. of the 5th Int. Workshop on Petri Nets and Performance Models, 1993, pp. 46–55.
[13] H. Boucheneb, R. Hadjidj, Towards optimal CTL* model checking of Time Petri Nets, in: Proc. of the International Workshop on Discrete Event Systems, WODES'04, Reims-France, 2004.
[14] H. Boucheneb, R. Hadjidj, CTL* model checking for time Petri nets, Theoretical Computer Science 353 (1-3) (2006) 208–227.
[15] F. Cassez, O.H. Roux, Structural translation from Time Petri Nets to timed automata, Electronic Notes in Theoretical Computer Science 128 (6) (2005) 145–160.
[16] F. Cassez, O.H. Roux, Structural translation from time Petri nets to timed automata, Journal of Systems and Software 29 (2006) 1456–1468.
[17] E.M. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, Cambridge, MA, 1999.
[18] J. Coolahan, N. Roussopoulos, Timing requirements for time-driven systems using augmented Petri nets, IEEE Transactions on Software Engineering SE 9 (5) (1983) 603–616.
[19] L.A. Corts, P. Eles, Z. Peng, Verification of real-time embedded systems using Petri net models and timed automata, in: Proc. of the 8th Int. Conf. on Real-Time Computing Systems and Applications, RTCSA'02, 2002, pp. 191–199.
[20] G. Gardey, O.H. Roux, O.F. Roux, Using zone graph method for computing the state space of a Time Petri Net, in: Proc. of FORMATS'03, in: LNCS, vol. 2791, Springer-Verlag, 2004.
[21] Z. Gu, K. Shin, Analysis of event-driven real-time systems with Time Petri Nets, in: Proc. of DIPES'02, in: IFIP, vol. 219, Kluwer, 2002, pp. 31–40.
[22] R. Hadjidj, H. Boucheneb, Much compact Time Petri Net state class spaces useful to restore CTL* properties, in: Proc of the Sixth International Conference on Application of Concurrency to System Design, ACSD'05, IEEE Computer Society Press, 2005.
[23] H.-M. Hanisch, Analysis of place/transition nets with timed arcs and its application to batch process control, in: Proc. of ICATPN'93, in: LNCS, vol. 691, Springer-Verlag, 1993, pp. 282–299.
[24] M. Hendriks, K.G. Larsen, Exact acceleration of real-time model checking, Electronic Notes in Theoretical Computer Science 65(6) (20) (2002) 435–459.

[25] O. Kupferman, T.A. Henzinger, M.Y. Vardi, A space-efficient on-the-fly algorithm for real-time model checking, in: Proc. of CONCUR'96, in: LNCS, vol. 1119, Springer-Verlag, 1996, pp. 514–529.
[26] K.G. Larsen, P. Pettersson, W. Yi, Model-checking for real-time systems, in: Proc. of Fundamentals of Computation Theory, in: LNCS, vol. 965, 1995, pp. 62–88.
[27] J. Lilius, Efficient state space search for Time Petri Nets, in: Proc. of MFCS Workshop on Concurrency, Brno'98, in: ENTCS, vol. 18, Elsevier Science Publishers, 1999.
[28] D. Lime, O.H. Roux, State class timed automaton of a time Petri net, in: Proc. of the 10th Int. Workshop on Petri Nets and Performance Models (PNPM'03), IEEE Comp. Soc. Press, 2003.
[29] P. Merlin, D.J. Farber, Recoverability of communication protocols - implication of a theoretical study, IEEE Transactions on Communications 24 (9) (1976) 1036–1043.
[30] W. Penczek, A. Polrola, Abstractions and partial order reductions for checking branching properties of Time Petri Nets, in: Proc. of ICATPN'01, in: LNCS, vol. 2075, Springer-Verlag, 2001, pp. 323–342.
[31] C. Ramchandani, Analysis of asynchronous concurrent systems by timed Petri nets, Massachusets Institute of Technology, 1974.
[32] J. Toussaint, F. Simonot-Lion, J.-P. Thomesse, Time constraint verifications methods based time Petri nets, Proc. of the 6th Workshop on Future Trends in Distributed Computing Systems, FTDCS 97, Tunis, Tunisia, 1997, pp. 262–267.
[33] T. Yoneda, H. Ryuba, CTL model checking of time Petri nets using geometric regions, Institute of Electronics Information and Communication Engineer (IEICE) Transactions on Information and Systems E81-D (3) (1998) 297–306.