Zenoness detection and timed model checking for real time systems

Rachid Hadjidj [1], Hanifa Boucheneb [1], Drifa Hadjidj [2]
[1] École Polytechnique de Montréal, Canada.
[2] Univesite de Boumerdes, Algeria.
E-mail: {rachid.hadjidj,hanifa.boucheneb}@polymtl.ca, dhadjidj@umbb.dz

Abstract

In this paper, we consider the time Petri net model (TPN model) and show how to detect zenoness using the state class method. Zenoness is a situation which suggests that an infinity of actions may take place in a finite amount of time. This behavior is often considered as pathological since it violates a fundamental requirement for timed systems for they cannot be infinitely fast. Models violating this property are called zeno. We state a necessary and sufficient condition for T-safe TPNs to be zeno and derive an algorithm to verify zenoness in the case of bounded TPN models. We also adapt a model checking approach to verify on-the-fly a subset of $TCTL$ properties while taking into account zeno behaviors.

Keywords: Formal methods, real time systems, time Petri nets, timed properties, on-the-fly $TCTL$ model checking, zeno models.

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, 11, 17, 22, 24]. The best known timed extensions are *timed Petri nets* proposed by Ramchandani [24] and *time Petri nets* proposed by Merlin and Farber [22]. We focus, in this paper, on the *time Petri net* model [22], referred in the sequel as the TPN model, for both zenoness detection and timed properties model-checking. Zenoness is a behavior which suggests that an infinity of actions may take place in a finite amount of time. This behavior is often considered as pathological since it violates a fundamental requirement for timed systems for they cannot be infinitely fast. Models violating this property are called zeno. To detect zenoness for bounded TPNs, the authors in [8], propose to translate the TPN model under study to an equivalent timed automata, then verify zenoness using the method developed in [18]. We proposed in this paper an approach to verify zenoness directly on the TPN model. We also state a necessary and sufficient condition for T-safe TPNs to be zeno and derive an algorithm to verify zenoness in the case of bounded TPNs. Our approach is based on the state class method [4] which allows to construct an abstraction for the TPN state space that preserving linear properties of the model. Several abstractions of the TPN model state space have been proposed in the literature to verify its untimed temporal properties [4, 6, 7, 5, 13, 15, 20, 26]. These approaches allow to construct state class spaces that preserve reachability [5, 20], linear properties [4], and branching properties [6, 7, 15, 26]. The verification is then performed using standard model-checking techniques [10].

In [16], we proposed an approach to verify on-the-fly timed properties for the TPN models using the state class method. The objective was to bridge the gap between timed automata [2] and time Petri nets. In fact, most proposed verification techniques of timed properties for TPNs define translation procedures from the TPN model into a semantically equivalent timed automata [2], in order to make use of available model checking tools [8, 12, 14, 21, 26]. 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. The use of observers [25], on the other hand, allows to express some timed properties in the form of TPNs, but properties on markings are quite difficult to express with observers [9]. With our verification approach, we were able to verify timed properties with the same versatility as for timed automata with tools like UPPAAL [3], but only non zeno models were considered. With the results we present in this paper for characterizing zenoness, we show how to adapt our timed model checking approach to take into account zeno behaviors.

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). In Section 4, we state a necessary and sufficient condition to detect zenoness and propose a detection algorithm using the SCG. In Section 5, we propose to contract the SCG by inclusion for better performances. In Section 6.2, we complete our model checking technique proposed in [16] to the case of zeno models.

2. TIME PETRI NETS

Let $\mathbb{Q}^+$ and $\mathbb{R}^+$ be respectively the set of positive rational numbers and the set of positive 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, with* $P \cap T = \emptyset$*,*
- Pre *and* $Post$ *are respectively the backward and forward incidence functions:* $P \times T \rightarrow \mathbb{N}$*, where* $\mathbb{N}$ *is the set of nonnegative integers,*
- $m_0 : P \rightarrow \mathbb{N}$*, is the initial marking,*
- $Is : T \rightarrow \mathbb{Q}^+_{[\,]}$ *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 [22]. A transition can fire iff it is enabled and the time elapsed since it has been enabled most recently is within its time interval. A transition t with $\downarrow Is(t) = 0$ will be called a *zero lower bound transition*. 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.: $\forall p \in P, m(p) \geq Pre(p,t)$. 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 newly enabled transitions in m, i.e.: $New(m, t_f) = \{t \in En(m) | \exists p, m(p) - Post(p, t_f) < Pre(p, t)\}$. For reasons of clarity, we consider in the sequel only T-safe TPNs (no multi-enabled transitions).

2.1. The TPN state

The TPN state is a couple (m, I), where m is a marking and I is an interval function, $I : En(m) \rightarrow \mathbb{Q}^+_{[\,]}$ [4]. For a state $s = (m, I)$, and $t \in En(m)$, $I(t)$ is called the *firing interval* of t, and is the interval of time where t can fire. 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 $\downarrow I(t)$ of its firing interval reaches 0, but must be fired, without any additional delay, if the upper bound $\uparrow I(t)$ of its firing interval reaches 0. The firing of a transition takes no time. 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.:

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

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

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

As a shorthand we write:

- $s \xrightarrow{\theta:t} s'$ iff $s \xrightarrow{\theta} s''$ and $s'' \xrightarrow{t} s'$ for some state s'',
- $s \overset{t}{\rightsquigarrow} s'$ iff $s \xrightarrow{\theta:t} s'$ for some $\theta \in \mathbb{R}^+$,
- $s \rightsquigarrow s'$ iff $s \overset{t}{\rightsquigarrow} s'$ for some $t \in T$.

2.2. The TPN state space

Definition 2 : ***TPN state space***
The state space of a TPN *model is the structure* $(\mathcal{S}, \rightarrow, s_0)$*, where:*

- $s_0 = (m_0, I_0)$ *is the initial state of the TPN model,*
- $s \rightarrow s'$ *iff either* $s \xrightarrow{\theta} s'$ *for some* $\theta \in \mathbb{R}^+$ *or* $s \xrightarrow{t} s'$ *for some* $t \in T$,
- $\mathcal{S} = \{s | s_0 \xrightarrow{*} s\}$*, where* $\xrightarrow{*}$ *is the reflexive and transitive closure of* $\rightarrow$*, 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 $\rho = s^0 \xrightarrow{\theta_0:t_0} s^1 \xrightarrow{\theta_1:t_1} s^2$....., such that $s^0 = s$. When no starting state is specified, the initial state s_0 of the TPN model is intended. The execution path suffix of ρ and starting from state s^i $(i \geq 0)$ is denoted $\rho(i)$. The sequence of transitions associated with ρ is the sequence of transitions (potentially infinite) $\omega_\rho = t_0, t_1, t_2, ...$ that occur on the execution path. We denote by $\pi(s)$ the set of all execution paths starting from state s. $\pi(s_0)$ is therefore the set of all execution paths 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* iff $time(\rho) = \infty$, otherwise it is said to be *zeno*. A TPN model is said to be *zeno* if at least one of its execution paths is zeno.

3. ABSTRACTION OF THE TPN STATE SPACE: THE STATE CLASS METHOD

Because of time density[1], a state in the TPN state space may have an infinity of successors. To finitely represent the state space of a TPN model, Berthomieu and Menasche proposed in [4] 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* [23].

Definition 3 : ***TPN concrete state space***
The concrete state space of the TPN model is the structure $(\Sigma, \rightsquigarrow, s_0)$ *where:*

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

- 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) [4]. Let $\omega = t_0, t_1, t_2, ...t_n$ be a sequence of transitions firable from the initial TPN state. The state class corresponding to ω (i.e., $\{s \in \mathcal{S} | \exists s_1, ..., s_n \in \mathcal{S}, s_0 \overset{t_0}{\rightsquigarrow} s_1 \overset{t_1}{\rightsquigarrow} s_2 ... s_n \overset{t_n}{\rightsquigarrow} s\}$) 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 of 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 $t_f \notin En(m)$ then
2     Return false
3 if $(F \wedge (\bigwedge_{t \in En(m) - \{t_f\}} t_f \leq t))$ is consistent then
4     Return true
5 Return false
```

Step 1 checks if t_f is enabled in m. Step 3 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. Note that if a zero lower bound transition is enabled in a state it is also immediately firable from that state with out any condition. 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 $m'(p) = m(p) - Pre(p, t_f) + Post(p, t_f), \forall p \in P$
2 Let $F' = F \wedge (\bigwedge_{t \in En(m) - \{t_f\}} t_f \leq t)$
3 Replace in $F'$ each variable $t \neq t_f$ 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 $t \in New(m', t_f)$ do
6     Add to $F'$ the constraint $tmin(t) \leq t \leq tmax(t)$
7 Return $\alpha' = (m', F')$
```

Step 1 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.

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: $\bigwedge_{(x,y) \in (En(m) \cup \{o\})^2} x - y \prec_F^{x-y} Sup(x - y, F)$ where: o represents the value *zero*, $Sup(x - y, F)$ 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. As an example, if $F = (0 \leq x < 20 \wedge 0 \leq y \leq 20 \wedge -10 \leq x - y \leq -10)$, its canonical form would be $(0 \leq x < 10 \;\wedge\; 0 \leq y \leq 20 \;\wedge\; -10 \leq x - y \leq -10)$.

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 formulae are equivalent). To compare two state classes, each one is translated into its canonical form. They are equal, if they have identical canonical forms [4]. Formally the SCG definition can be stated as follows:

Definition 4 : ***State Class Graph (SCG)***
The state class graph of a TPN model is the structure $(\mathcal{C}, \twoheadrightarrow, \alpha_0)$, where:

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

Similarly to the TPN state space, an *execution path* in the SCG starting from a state class α, is a maximal sequence $\zeta = \alpha^0 \overset{t_0}{\twoheadrightarrow} \alpha^1 \overset{t_1}{\twoheadrightarrow} \alpha^2 \overset{t_2}{\twoheadrightarrow}$ ....., where $\alpha^0 = \alpha$. We also denote by $\pi(\alpha)$ the set of all execution paths starting from α. 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 [4], the authors prove that the SCG is finite for all bounded TPN models, and also preserves reachability and execution paths (i.e., linear properties) [4]. In addition, each execution path in the state space is *inscribed*[2] within only one execution path of the SCG (The SCG is deterministic[3]).

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

```
1  $\alpha_0$= $(m_0, F_0)$
2  $\mathcal{C}$= $\{\alpha_0\}$
3  $\twoheadrightarrow$= $\emptyset$
4  WAIT=$\{\alpha_0\}$
5  while WAIT $\neq \emptyset$ do
6      get $\alpha = (m, F)$ from WAIT
7      forall $t \in En(m)$ s.t. $isFirable(\alpha, t)$ do
8          $\alpha'$=$succ(\alpha, t)$
9          if ($\alpha' \notin \mathcal{C}$) then
10             Add $\alpha'$ to $\mathcal{C}$ and to WAIT
11             Add $(\alpha, t, \alpha')$ to $\twoheadrightarrow$
12 Return $(\mathcal{C}, \twoheadrightarrow, \alpha_0)$
```

In terms of implementation, a state class (m, F) is generally represented with a pair (m, D), where D is the matrix representation of F, of order $|En(m) \cup \{o\}|$, defined by: $\forall (x, y) \in (En(m) \cup \{o\})^2$, $D_{xy} = (Sup(x - y, F), \prec_F^{x-y})$. D is known under the name *Difference Bound Matrix* (DBM) [19]. Its purpose is to render the implementation of operations on F simple. In fact, all operations performed on state classes to construct the *SCG* are very well defined for *DBMs*. However, they require each *DBM* to be put in its unique canonical form. The computation of this form is based on the shortest path Floyd-Warshall's algorithm, and is considered as the most costly operation on $DBMs$, with a time complexity of $O(n^3)$, where n is the DBM order.

4. DETECTING ZENONESS USING THE STATE CLASS METHOD

Many properties such as freedom from livelock and deadlock are in general required from correctly designed systems. One property specific to timed systems, called Non-Zenoness or time progress, suggests that a timed system cannot perform an infinite number of actions within a finite amount of time. In fact, such a behavior prevents time from converging. To detect zenoness

[2]A state space execution path $\rho = s_0 \overset{t_0}{\rightsquigarrow} s_1 \overset{t_1}{\rightsquigarrow} s_2$..... is inscribed in the SCG execution path $\zeta = \alpha_0 \overset{t_0}{\twoheadrightarrow} \alpha_1 \overset{t_1}{\twoheadrightarrow} \alpha_2$..... iff $s_i \in \alpha_i, \forall i \geq 0$, with $s = (m_s, I) \in \alpha = (m_\alpha, F)$ iff $m_s = m_\alpha \wedge I \subseteq F$.
[3]No state class in the SCG exists with two outgoing arcs labelled with the same transition.

for bounded TPNs, the authors in [8], proposed to translate the TPN model under study to an equivalent timed automata, then verify zenoness using the method developed in [18] . We propose, in this section, an approach to verify zenoness directly on the TPN model using the state class method. Let $\omega = t_0, t_1, t_2, ...$ be a sequence of transitions (potentially infinite). ω is called a *zero lower bound sequence* iff all its transitions are zero lower bound transitions. The following lemma states a necessary and sufficient condition for a TPN model to be zeno.

Lemma 1 : *A TPN is zeno iff $\exists \rho \in \pi(s_0), i \geq 0\ s.t.\ \rho(i)$ is associated with an infinite zero lower bound sequence.*

Proof In words, the lemma states that a TPN is zeno iff it can reach a marking from which it can perform an infinite sequence of zero lower bound transitions.
Note first that a finite execution path cannot be zeno since time diverges after the last transition is fired. If an infinite execution path $\rho = s_0 \overset{\theta_0:t_0}{\to} s_1 \overset{\theta_1:t_1}{\to} s_2....$ has no suffix associated with a zero lower bound sequence, then $\forall i \geq 0$, $\exists j \geq i\ s.t. \downarrow Is(t_j) > 0$. With the fact that the TPN model has a finite number of transitions, there must exist a transition t with $\downarrow Is(t) > 0$ which is fired an infinity of times. Each time t is fired, $\downarrow Is(t)$ time units (at least) would have elapsed since it was enabled the last time. Adding the fact that the TPN model is T-safe, it is easy to conclude that ρ is a diverging execution path. Now if $\exists \rho \in \pi(s_0), i \geq 0\ s.t.\ \rho(i)$ is associated with a zero lower bound sequence, then ρ is of the form $\rho = s_0 \overset{\theta_0:t_0}{\to} s_1 \overset{\theta_1:t_1}{\to} s_2....s_i \overset{\theta_i:t_i}{\to} s_{i+1}....$, with $\rho(i) = s_i \overset{\theta_i:t_i}{\to} s_{i+1} \overset{\theta_{i+1}:t_{i+1}}{\to} s_{i+2}.....$ Since $\forall j \geq i, \downarrow Is(j) = 0$ then the firing sequence ρ' obtained from ρ by replacing θ_j $(\forall j \geq 0)$ with the value zero is also an execution path of the TPN model. In fact this replacement is possible because if a zero lower bound transition is enabled in a state it is also immediately firable from that state. The result is a zeno execution path. ■

The next theorem characterizes zenoness in the case of bounded TPNs useful to derive a detection algorithm.

Theorem 1 : *A bounded TPN is zeno iff its state class graph has a cycle where all transitions are zero lower bound transitions (*a zero lower bound cycle*).*

Proof Note first that since the TPN is bounded its SCG is finite [4]. In the case the SCG has a zero lower bound cycle, there must exist an execution path that has a suffix inscribed inside this cycle, i.e., circles infinitely in the cycle. The sequence of transitions associated with this suffix is a zero lower bound sequence, and lemma 1 assures that the TPN is zeno. In case the SCG has no zero lower bound cycle, either the SCG has no cycle at all, or each cycle contains at least one non zero lower bound transition. If the SCG has no cycle, all its execution paths must be finite, and hence not zeno. In the other case, any infinite execution path must go through one or more of the cycles of the SCG an infinity of times, which assures that this execution path cannot have a suffix associated with a zero lower bound sequence, and lemma 1 assures that the TPN is not zeno in this case. ■

From theorem 1, an algorithm to check the zenoness of a bounded TPN models is straightforward. It consists only in constructing the SCG of the TPN model and detecting if it contains a zero lower bound cycle or not. If such a cycle exists, the TPN is zeno otherwise it is not (see algorithm 4).

5. CONTRACTING THE SCG FOR BETTER PERFORMANCE: THE I-SCG

In this section, we propose a contraction of the SCG by inclusion, we call I-SCG (Inclusion contracted SCG), and use it as an alternative in algorithm 4 to detect zenoness. The objective it to improve performances. The contraction by inclusion of the SCG consists in agglomerating its state classes into the most including ones. The inclusion test is performed as follows: Let *(m,F)* and *(m,F')* be two state classes in canonical forms sharing the same marking, and let *(m,D)* and *(m,D')*

Algorithm 4: $isZeno(\mathcal{N} = (P, T, Pre, Post, m_0, Is))$

```
Construct the SCG of $\mathcal{N}$
Remove all arcs of the SCG corresponding to non zero lower bound transitions
Repeatedly remove all nodes with no ingoing arcs until no arcs can be removed
if the resulting graph is empty then
    return false
else
    return true
```

be their representations using DBMs. (m, F) is included in (m, F') iff: $\forall(x, y) \in (En(m) \cup \{o\})^2$, $(D(x, y) \leq D'(x, y))$. Note that if $(v, \prec)$ and $(v', \prec')$ are elements of D and D', $(v, \prec) \leq (v', \prec')$ iff ($v < v'$ or $v = v' \wedge \prec \leq \prec'$), with "$<$" less than "$\leq$". The resulting graph will be denoted I-SCG (Inclusion contracted SCG). The I-SCG could be obtained by first constructing the SCG then repeatedly agglomerating its state classes by inclusion, until no more agglomerations are possible. It is however more interesting to perform this operation during the construction itself. By doing so, most state classes are not computed, which saves time and space. Algorithm 5 shows how to progressively compute the I-SCG.

Algorithm 5: I-SCG$(\mathcal{N} = (P, T, Pre, Post, m_0, Is))$

```
$\alpha_0$= $(m_0, F_0)$
$\mathcal{C}$= $\{\alpha_0\}$
$\twoheadrightarrow$= $\emptyset$
WAIT=$\{\alpha_0\}$
while WAIT $\neq \emptyset$ do
    get $\alpha = (m, F)$ from WAIT
    forall $t \in En(m)$ s.t. $isFirable(\alpha, t)$ do
        $\alpha'$=$succ_{AC}(\alpha, t)$
        if $\exists\alpha" \in \mathcal{C}$ s.t. $\alpha' \subseteq \alpha"$ then
            add $(\alpha, t, \alpha")$ to $\twoheadrightarrow$

        else
            if $\exists\alpha" \in \mathcal{C}$ s.t. $\alpha" \subseteq \alpha'$ then
                forall $\alpha" \in \mathcal{C}$ s.t. $\alpha" \subseteq \alpha'$ do
                    remove $\alpha"$ from $\mathcal{C}$ and from WAIT
                    replace $\alpha"$ by $\alpha'$ in all transitions of $\twoheadrightarrow$
            add $\alpha'$ to $\mathcal{C}$ and to WAIT
            add $(\alpha, t, \alpha')$ to $\twoheadrightarrow$
return $(\mathcal{C}, \twoheadrightarrow, \alpha_0)$
```

Note that the way the I-SCG is constructed, it must be a deterministic graph like the SCG. This is because whenever $\alpha_1 \overset{t}{\twoheadrightarrow} \alpha'_1$ and $\alpha_2 \overset{t}{\twoheadrightarrow} \alpha'_2$ with $\alpha_1 \subseteq \alpha_2$ then $\alpha'_1 \subseteq \alpha'_2$. So when α_1 is merged in α_2, α'_1 is also merged in α'_2 which prevents non determinism in the I-SCG.

To illustrate the difference in size and computing time between the SCG and the I-SCG, we consider TPNs obtained as the parallel composition of the simple model shown in figure 1. Table 1 reports the results in terms of the size of the obtained graphs (nodes/arcs) and the computing time. The parallel composition of *n* copies of the simple model is denoted S(n) in the table. The last column in the table compares the results obtained for the SCG and I-SCG, and shows the big difference in their size and computing time. The following theorem states some interesting properties preserved in the I-SCG which makes it a better alternative to verify reachability for the TPN model than the SCG.

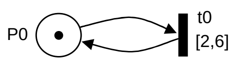

FIGURE 1: A simple TPN model

TPN	SCG	I-SCG	Ratio (SCG/I-SCG)

S(3)
cpu(s)
	79/210
	0
		6/21
		0
			11
			-

S(4)
cpu(s)
	837/3032
	0.01
		24/132
		0
			25
			-

S(5)
cpu(s)
	10951/50570
	0.40
		120/850
		0.05
			63
			8

TABLE 1: A comparison between the SCG and the I-SCG

Theorem 2 :
(i) The I-SCG is finite iff the TPN model is bounded.
(ii) The I-SCG preserves reachability properties of the TPN model.

Proof (i) Since the SCG is finite for all bounded TPN models, the I-SCG, which is its contraction, is also finite.
(ii) In an agglomeration step where a state class α is agglomerated in a state class α', α must be included in α'. Any marking reachable from α is also reachable from α'. Hence, reachability properties are preserved. ■

Another property which allows to use the I-SCG instead of the SCG in algorithm 4 to detect zenoness is stated in the next theorem.

Theorem 3 : *The SCG contains a zero lower bound cycle iff the I-SCG contains a zero lower bound cycle.*

Proof Note first that the deterministic aspect of both the SCG and I-SCG assures that any execution path in the state space of the TPN model is inscribed in exactly one execution path in either the SCG or the I-SCG. So if the SCG contains a zero lower bound cycle, it must contain a zeno execution path, which in turn should be inscribed in an execution path in the I-SCG. So we conclude that the I-SCG, which is finite, must contain a zero lower bound cycle. Now if the I-SCG contains a zero lower bound cycle, a zeno execution path with a suffix inscribed in this cycle must exist in the state space of the TPN model. This is because, if we take a state from a state class of the cycle, it must be reachable from the initial state of the TPN model. Knowing that each zero lower bound transition in the cycle is enabled and firable from its outgoing state class in the cycle, and its firing leads to the marking of its ingoing state class in the cycle, a zeno execution path that cycles around these markings must exist in the state space of the TPN model. This in turn assures that a zero lower bound cycle exists in the SCG. ■

6. ADAPTING TCTL MODEL-CHECKING TO ZENO MODELS

In [16], we proposed an approach to verify a subclass of the TCTL timed logic for the TPN model, but only non zeno models were considered. In the following, we repeat this verification approach in some details, then show how to adapt it to the case of zeno models using the results presented in sections 4 and 5.

6.1. The timed temporal Logic we consider

We define a timed temporal logic for which we give algorithms to verify the satisfaction of its formulae 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 $TCTL$ formulae is defined by the following grammar (in the grammar, $\wp \in PR$ and index I is an element of $\mathbb{Q}^{+}_{[\,]}$):

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

$TCTL$ formulae 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, $\mathcal{V}(s) = \{\wp \in PR | \wp(m) = true\}$. 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 $\rho = s^0 \overset{\theta_0 : t_0}{\rightarrow} s^1 \overset{\theta_1 : t_1}{\rightarrow} s^2$..... an execution path such that $s^0 = s$ (i.e., $\rho \in \pi(s)$). The dense execution path corresponding to ρ is the mapping $\breve{\rho} : \mathbb{R}^+ \rightarrow S$ defined by: $\breve{\rho}(r) = s^i + \delta$ such that $r = \sum_{j=0}^{i-1} \theta_j + \delta$, $i \geq 0$ and $0 \leq \delta \ < \theta_i$. 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 $time(\rho) = \infty$, $\exists r \in I, \mathcal{M}, \breve{\rho}(r) \models \psi$ and $\forall r' < r, \mathcal{M}, \breve{\rho}(r') \models \varphi$,
-$\mathcal{M}, s \models \exists(\varphi U_I \psi)$ iff $\exists \rho \in \pi(s)$, s.t $time(\rho) = \infty$, $\exists r \in I, \mathcal{M}, \breve{\rho}(r) \models \psi$ and $\forall r' < r, \mathcal{M}, \breve{\rho}(r') \models \varphi$.

The TPN model $\mathcal{N}$ is said to satisfy a $TCTL$ formula ϕ iff $\mathcal{M}, s_0 \ \models \ \phi$. To ease $TCTL$ formulae writing, some abbreviations are used: $\exists\Diamond_I\varphi = \exists(trueU_I\varphi)$, $\forall\Diamond_I\varphi = \forall(trueU_I\varphi)$, $\exists\Box_I\varphi = \neg\forall\Diamond_I\neg\varphi$, $\forall\Box_I\varphi = \neg\exists\Diamond_I\neg\varphi$. When interval I is omitted, its value is $[0, \infty[$ by default. Our timed temporal logic, we call $TCTL_{TPN}$, is defined as follows:

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

$\wp_1$ 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 $TCTL$ 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 ($\exists r_1 \geq 0$, $\breve{\rho}(r_1) \models \wp_1$ and $\forall r < r_1$, $\breve{\rho}(r) \not\models \wp_1$) then:
1 - $(\mathcal{M}, \breve{\rho}(r_1) \not\models \wp_2) \Rightarrow (\exists r_2, r_2 - r_1 \in I$, $\breve{\rho}(r_2) \models \wp_2$, $\forall r$, $r_1 < r < r_2$, $\breve{\rho}(r) \not\models \wp_2$ and $\mathcal{M}, \breve{\rho}(r_2) \models \wp_1 \mapsto_I \wp_2)$.
2 - $(\mathcal{M}, \breve{\rho}(r_1) \models \wp_2) \Rightarrow \downarrow I = 0$ and $\exists r_2 > r_1$ s.t. $\mathcal{M}, \breve{\rho}(r_2) \models \neg(\wp_1 \wedge \wp_2)$ and $\forall r, r_1 \leq r < r_2$, $\mathcal{M}, \breve{\rho}(r) \models (\wp_1 \wedge \wp_2)$ and $\mathcal{M}, \breve{\rho}(r_2) \models \wp_1 \mapsto_I \wp_2)$.

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[4] as it is the case for the

[4]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.

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 4 : *[16]* $\mathcal{M}, s \models \wp_1 \mapsto_{I_r} \wp_2$ *iff* $\mathcal{M}, s \models \wp_1 \leadsto_{I_r} \wp_2$.

6.2. On-the-fly $TCTL_{TPN}$ model checking

First we give an algorithm to model check the bounded first response property, then show how to adapt this algorithm to model check remaining $TCTL_{TPN}$ properties.

6.2.1. 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 figure 2, we call *Alarm-clock*. The resulting TPN we denote $\mathcal{N}||Alarm$, will be used instead of $\mathcal{N}$ to verify ϕ.

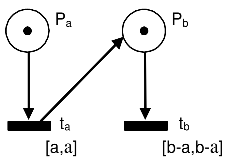

FIGURE 2: The Alarm-clock TPN

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[5]. 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:

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

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.

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. 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 (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 6: $isFirable_{AC}(\alpha = (m, F), t_f)$

```
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     $F' = F' \wedge t_f < t_a$
5 else if $t_b \in En(m) \wedge t_f = t_b$ then
6     $F' = F \wedge (\bigwedge_{t \in En(m)-\{t_f\}} t_f < t)$
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.

6.2.2. *Model checking algorithms*

The on-the-fly $TCTL_{TPN}$ model checking of formula ϕ is based on the exploration algorithm 8. 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[6]. The algorithm 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

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

Algorithm 7: $succ_{AC}(\alpha = (m, F), t_f)$

```
Let $m'(p) = m(p) - Pre(p, t_f) + Post(p, t_f), \forall p \in P$
if $t_f \notin En(m)$ then Return false
Let $F' = F \wedge (\bigwedge_{t \in En(m)-\{t_f\}} t_f \leq t)$
if $t_a \in En(m) \wedge t_f \neq t_a$ then
    $F' = F' \wedge t_f < t_a$
else if $t_b \in En(m) \wedge t_f = t_b$ then
    $F' = F \wedge (\bigwedge_{t \in En(m)-\{t_f\}} t_f < t)$
Replace in $F'$ each variable $t$ with $t + t_f$
Eliminate by substitution, in $F'$, $t_f$ and all variables associated with transitions conflicting with $t_f$ for $m$
forall $t \in New(m', t_f)$ do
    Add to $F'$ the constraint $tmin(t) \leq t \leq tmax(t)$
Return $(m', F')$
```

Algorithm 8: $modelCheck(\phi)$

```
Let continue=true                                              /* global variable */
Let valid=true                                                 /* global variable */
Let COMPUTED= $\emptyset$
Let $\alpha_0$= $(m_0, F_0)$
Let $\alpha'_0$=$checkStateClass_\phi(\alpha_0, t_\epsilon)$
Let WAIT=$\{\alpha'_0\}$
while continue do
    remove $\alpha = (m, F)$ from WAIT
    forall $t \in En(m)$ s.t. $isFirable_{AC}(\alpha, t)$ provided continue do
        $\alpha'$:=$succ_{AC}(\alpha, t)$
        $\alpha''$:=$checkStateClass_\phi(\alpha', t)_\phi$
        if continue $\wedge \alpha'' \neq \emptyset \wedge \nexists \alpha_p \in$ COMPUTED s.t. $\alpha'' \subseteq \alpha_p$ then
            forall $\alpha_p \in$ COMPUTED s.t. $\alpha_p \subseteq \alpha''$ do
                remove $\alpha_p$ from COMPUTED and from WAIT
            Add $\alpha''$ to COMPUTED and to WAIT
Return valid
```

takes decisions, and record 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 [3], 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 9) corresponds to property $\phi = \wp_1 \mapsto_I \wp_2$. Its steps match exactly those described in section 6.2.1. 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.

The second implementation corresponds to property $\phi = \forall(\wp_1 U_I \wp_2)$ [16]. 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

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

```
if $t_a, t_b \notin En(m) \wedge t \notin \{t_a, t_b\}$ then                /* case 1 */
    if $\wp_1(m) \wedge \neg\wp_2(m)$ then
        enable $t_a$ in $\alpha$;
    if $\wp_1(m) \wedge \wp_2(m) \wedge a > 0$ then
        valid=false; continue=false ;
if $t_a \in En(m) \wedge \wp_2(m)$ then                                      /* case 2 */
    valid=false; continue=false;
if $t_b \in En(m) \wedge \wp_2(m)$ then                                      /* case 3 */
    disable $t_b$ in $\alpha$;
if $t = t_b$ then                                                            /* case 4 */
    valid=false ; continue=false;
Return $\alpha$;
```

verification of property $\phi = \forall(\wp_1 U_I \wp_2)$ checks if proposition $\wp_1$ 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 $\wp_1$ or $\wp_2$, until $\wp_2$ is true or t_b is fired. If $\wp_2$ becomes true in a state class α, α is no more explored. In case t_b is fired, the exploration is stopped and the property is declared invalid. The last implementation of $checkStateClass_\phi$ corresponds to property $\phi = \exists(\wp_1 U_I \wp_2)$ [16]. 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 α if it does not comply with the semantics of ϕ. It also aborts the exploration as soon as a satisfactory execution path is found. The following theorem states the decidability of our model checking approach for all bounded TPNs.

Theorem 5 : *[16] $TCTL_{TPN}$ model checking is decidable for Bounded TPN models.*

6.3. Adapting the verification to the case of zeno TPNs

In the case of a zeno TPN model, the model checking approach described above may yield false results in some situations. The problem may arise only for formulae of the form $\forall(\wp_1 U_I \wp_2)$, $\wp_1 \mapsto_I \wp_2$ and those which are derived from them. For these formulae, the model checking approach start by assuming that the formula is true[7] (step 2, algorithm 8) at the beginning of the verification process and the property is declared invalid if a situation is encountered where a counter example is found. A counter example is found if transition t_b gets fired, which means that the end of interval I has occurred and property $\wp_2$ was not found to be true yet (steps 11, algorithm 9). Now consider that the TPN model is zeno, with only one zeno execution path ρ_z, and we are verifying property $\forall(\wp_1 U_I \wp_2)$ with $I = [a, b]$. Suppose also that this property is true on all execution paths except on ρ_z. If $\wp_1$ is true in all states of ρ_z and $time(\rho_z) < b$, transition t_b will never get fired to signal the end of interval I, and the verification would conclude that the property is valid while it is not. Note that a similar scenario may happen for formula $\wp_1 \mapsto_I \wp_2$. To correct this problem, our 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 we are looking for property $\wp_2$, i.e., when transition t_b is enabled. To do so, we modify the exploration algorithm 8 to keep track of zero lower bound transitions when they are fired from state classes where transition t_b is enabled. So each time a zero lower bound transition is fired from a state class α where t_b is enabled, we connect α with the resulting state class with an arc. At the end of the verification, if the property is found to be valid, we check the connection between computed state classes to see if a cycle is present. If it is case the property is declared invalid, otherwise it is valid. Algorithm 10 is an adapted version of algorithm 8 that take into account zeno TPNs.

[7]This is not true in the case $\exists(\wp_1 U_I \wp_2)$ where the formula is assumed to be false at the beginning of the verification by resetting the value of the variable $valid$ to false.

Algorithm 10: $modelCheck - with - zeno - detection(\phi)$

```
Let continue=true                                      /* global variable */
Let valid=true                                         /* global variable */
Let COMPUTED= $\emptyset$
Let $\alpha_0$= $(m_0, F_0)$
Let $\alpha'_0$=$checkStateClass_\phi(\alpha_0, t_\epsilon)$
Let WAIT={$\alpha'_0$}
while continue do
    remove $\alpha = (m, F)$ from WAIT
    forall $t \in En(m)$ s.t. $isFirable_{AC}(\alpha, t)$ provided continue do
        $\alpha'$:=$succ_{AC}(\alpha, t)$
        if $\phi \neq \exists(\wp_1 U_I \wp_2)$ and $t_b \in En(m) \wedge \downarrow Is(t) = 0$ then
            connect $\alpha$ to $\alpha'$
        $\alpha''$:=$checkStateClass_\phi(\alpha', t)_\phi$
        if continue $\wedge\ \alpha'' \neq \emptyset \wedge \nexists \alpha_p \in$ COMPUTED s.t. $\alpha'' \subseteq \alpha_p$ then
            forall $\alpha_p \in$ COMPUTED s.t. $\alpha_p \subseteq \alpha''$ do
                remove $\alpha_p$ from COMPUTED and from WAIT while substituting $\alpha_p$ by $\alpha''$ for all arcs of $\alpha_p$
            Add $\alpha''$ to COMPUTED and to WAIT
if $\phi \neq \exists(\wp_1 U_I \wp_2)$ and valid then
    if COMPUTED has a cycles then
        valid=false
Return valid
```

7. CONCLUSION

In this paper, we considered the time Petri net model and proposed an efficient approach to detect zeno behaviors using the state class method. Zenoness is in general considered as a pathological behavior for timed systems since it suggests that an infinity of actions may take place in a finite amount of time. The approach we proposed is based on the fact that a TPN is zeno iff it can reach a marking from which zero lower bound transitions are fired to infinity. As a consequence, a bounded TPN is zeno iff its state class graph has a cycle where all transitions are zero lower bound transitions (*a zero lower bound cycle*). An algorithm to check the zenoness of a bounded TPN model is straightforward from these observations. It consists in constructing the SCG of the TPN model and detecting if it contains a zero lower bound cycle or not. If such a cycle exists, the TPN is zeno, otherwise it is not. To improve performance we proposed to contract the SCG in what we called I-SCG (Inclusion contracted SCG) and use it as an alternative to the SCG in the zenoness detection algorithm. The I-SCG is much smaller than the SCG, and much faster to compute. It also preserves reachability of the TPN model which makes it a better alternative than the SCG to verify this kind of properties. We also adapted the model checking approach, we proposed in [16] to verify a subset of $TCTL$ properties, to take into account zeno behaviors. As a future work we expect to extend the characterization of zenoness to the case of TPN with multi-enabled transitions.

REFERENCES

[1] P. A. Abdulla and A. Nyln. Timed Petri Nets and BQOs. In *Proc. of ICATPN'01*, volume 2075 of LNCS, pages 53–70. Springer-Verlag, 2001.

[2] R. Alur and D. Dill. Automata for modelling real-time systems. In *Proc. Of ICALP'90*, volume 443 of LNCS, pages 322–335. Springer-Verlag, 1990.

[3] G. Behrmann, J. Bengtsson, A. David, K. G. Larsen, P. Pettersson, and W. Yi. Uppaal implementation secrets. In *Proc. of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, 2002.

[4] B. Berthomieu and M. Menasche. An enumerative approach for analyzing Time Petri Nets. In *Proc. of the IFIP 9th World Computer Congress*, volume 9 of Information Processing,

pages 41–46. IFIP, North Holland, September 1983.

[5] H. Boucheneb and 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*, pages 46–55, October 1993.

[6] H. Boucheneb and 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, May 2004.

[7] H. Boucheneb and R. Hadjidj. CTL* model checking for time Petri nets. *Theoretical Computer Science*, 353(1-3)(1-3):208–227, 2006.

[8] F. Cassez and O. H. Roux. Structural translation from Time Petri Nets to timed automata. *In Electronic Notes in Theoretical Computer Science*, 128(6)(145-160), 2005.

[9] F. Cassez and O. H. Roux. Structural translation from time petri nets to timed automata. *Journal of Systems and Software*, 29:1456–1468, 2006.

[10] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[11] J. Coolahan and N. Roussopoulos. Timing requirements for time-driven systems using augmented Petri nets. *IEEE Trans. on Software Eng*, SE-9(5):603–616, 1983.

[12] L. A. Corts, P. Eles, and 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)*, pages 191–199, March 2002.

[13] G. Gardey, O. H. Roux, and O. F. Roux. Using zone graph method for computing the state space of a Time Petri Net. In *Proc. of FORMATS'03*, volume 2791 of LNCS. Springer-Verlag, 2004.

[14] Z. Gu and K. Shin. Analysis of event-driven real-time systems with Time Petri Nets. In *Proc. of DIPES'02*, volume 219 of IFIP, pages 31–40. Kluwer, 2002.

[15] R. Hadjidj and H. Boucheneb. Much compact Time Petri Net state class spaces useful to restore CTL* properties. In *Proc. of of the Fifth International Conference on Application of Concurrency to System Design (ACSD'05)*. IEEE Computer Society Press, 2005.

[16] R. Hadjidj and H. Boucheneb. On-the-fly tctl model checking for time Petri nets using the state class method. In *Proc of the Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*, pages 111–120. IEEE Computer Society Press, 2006.

[17] H-M. Hanisch. Analysis of place/transition nets with timed arcs and its application to batch process control. In *Proc. of ICATPN'93*, volume 691 of LNCS, pages 282–299. Springer-Verlag, 1993.

[18] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2)(2):193–224, 1994.

[19] K. G. Larsen, C. Weise, W. Yi, and J. Pearson. Clock difference diagrams. *Nordic Journal of Computing*, 26(3), 1999.

[20] J. Lilius. Efficient state space search for Time Petri Nets. In *Proc. of MFCS Workshop on Concurrency, Brno'98*, volume 18 of ENTCS. Elsevier Science Publishers, 1999.

[21] D. Lime and 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, September 2003.

[22] P. Merlin and D. J. Farber. Recoverability of communication protocols - implication of a theoretical study. *IEEE Trans. on Communications*, 24(9):1036–1043, 1976.

[23] W. Penczek and A. Polrola. Abstractions and partial order reductions for checking branching properties of Time Petri Nets. In *Proc. of ICATPN'01*, volume 2075 of LNCS, pages 323–342. Springer-Verlag, 2001.

[24] C. Ramchandani. Analysis of asynchronous concurrent systems by timed Petri nets. Technical report, February 1974.

[25] J. Toussaint, F. Simonot-Lion, and J.-P. Thomesse. Time constraint verifications methods based time Petri nets. In *Proc. of the 6th Workshop on Future Trends in Distributed Computing Systems (FTDCS97)*, pages 262–267. Tunis, Tunisia, 1997.

[26] T. Yoneda and H. Ryuba. CTL model checking of time Petri nets using geometric regions. *IEICE Trans. Inf. and Syst., 3:1-10, 1998*.