Available online at www.sciencedirect.com

SCIENCE DIRECT®

Theoretical Computer Science 353 (2006) 208–227

Theoretical Computer Science

www.elsevier.com/locate/tcs

CTL^* model checking for time Petri nets[☆]

Hanifa Boucheneb*, Rachid Hadjidj

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

Received 15 April 2004; received in revised form 2 November 2005; accepted 2 November 2005

Communicated by R. Glabbeek

Abstract

This paper aims at applying the CTL^{*}[1] model checking method to the time Petri net (TPN) model. We show here how to contract its generally infinite state space into a graph that captures all its CTL^* properties. This graph, called atomic state class graph (ASCG), is finite if and only if, the model is bounded.[2] Our approach is based on a partition refinement technique, similarly to what is proposed in [Berthomieu, Vernadat, State class constructions for branching analysis of time Petri nets, Lecture Notes in Computer Science, vol. 2619, 2003; Yoneda, Ryuba, CTL model checking of time Petri nets using geometric regions, IEICE Trans. Inf. Syst. E99-D(3) (1998)]. In such a technique, an intermediate abstraction (contraction) of the TPN state space is first built, then refined until CTL^* properties are restored. Our approach improves the construction of the ASCG in two ways. The first way deals with speeding up the refinement process by using a much more compact intermediate contraction of the TPN state space than those used in [Berthomieu, Vernadat, State class constructions for branching analysis of time Petri nets, Lecture Notes in Computer Science, vol. 2619, 2003; Yoneda, Ryuba, CTL model checking of time Petri nets using geometric regions, IEICE Trans. Inf. Syst. E99-D(3) (1998)]. The second way deals with computing each ASCG node in $O(n^2)$ instead of $O(n^3)$, n being the number of transitions enabled at the node. Experimental results have shown that our improvements have a good impact on performances.
© 2005 Elsevier B.V. All rights reserved.

Keywords: Time Petri nets; State class spaces; Strong state class graph; Atomic state class graph; CTL^* properties; Model checking

1. Introduction

The need for including the time parameter in system analysis is obvious since they are real time in nature. Several models integrating the time parameter in different ways have been developed. Among these models, we find timed automata (TA) [1] and various time Petri net (TPN) models [9,11,15,18,19].

The integration of time in models increases their modelling power, but tremendously complicates their analysis. Indeed, because of time density, state spaces of timed models are in general infinite and thus not useful for enumerative analysis such as model checking. If enumerative techniques are to be used, the infinite state space must be contracted into a finite representation (a graph) which preserves all properties of interest.

[☆] Research supported by the NSERC Grant no. 238841-2001.
* Corresponding author. Departmento de Informaticae, Facultad de Ciencias, Ingenieria de Sistemas, E-50009 Zaragosa, Spain.
E-mail addresses: hanifa.boucheneb@polymtl.ca (H. Boucheneb), rachid.hadjidj@polymtl.ca (R. Hadjidj).
[1] Computation Tree Logic.
[2] The number of its reachable markings is finite.

0304-3975/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2005.11.002

Several analysis approaches, based on state space contractions, have been developed for TA [10,12,13] and some TPN models as well [3–9,11,16,17,20,21]. Resulting graphs are principally characterized by their sizes, the condition of their finiteness and the kind of properties they preserve (*LTL* [3], *CTL*, *CTL**, ...). Preserved properties can be verified by exploring these graphs. In this context, the model checking method is the most attractive verification technique.

This paper aims at applying the *CTL** model checking method to the TPN model.

Berthomieu and Menasche proposed 20 years ago (1982) a contraction of the TPN state space that preserves the *LTL* properties of the model [4], but not necessarily its *CTL** ones. Yoneda and Ryuba developed in [21], another contraction, called atomic state class graph (ASCG), which preserves *CTL** properties. Their approach has however the disadvantage of being limited to TPN models with bounded firing intervals. For TPN models with unbounded firing intervals, their approach may yield infinite ASCGs, even if the model is bounded. Recently, in [5], Berthomieu and Vernadat improved the approach of Yoneda and Ryuba and extended its application to all bounded TPN models. Their improvements allow also to generate smaller ASCGs in much shorter times. Both approaches proposed in [5,21] use a *partition refinement* technique, where an intermediate graph, representing a contraction of the TPN state space, is first built then refined until an ASCG is derived.

Both intermediate state class graphs used in [5,21] are computed such that they preserve linear properties of the TPN model. As we will show, this feature is not a requirement of the refinement technique. Besides, experimental results have shown that state class graphs that preserve linear properties are in general closer in size to their corresponding ASCGs, and even larger for some TPN models. For instance, the strong state class graph (SSCG) construction faces a sever state explosion problem for some TPN models with unbounded firing intervals. For these models, SSCGs may be several times larger than their corresponding ASCGs. This constitutes a major obstacle towards their construction and their refinement too.

In this paper, we propose to improve the construction of the ASCG in two ways. The first way deals with speeding up the refinement process by using a much more compact intermediate contraction of the TPN state space than what is proposed in [5,21]. The second way deals with computing each ASCG node in $O(n^2)$ instead of $O(n^3)$, n being the number of transitions enabled at the node.

Our experimental results have shown a significant impact of our improvements on the construction of ASCGs. For all tested TPN models, we recorded in general an important reduction in computing times and memory usage. For some tested models, computing times have been reduced by factors greater than 12, while memory usage has been reduced by factors greater than five. Furthermore, the improvements seem to increase as the model increases in size. [4] In this way, we have been able to compute some ASCGs which failed to compute without the proposed improvements, either in reasonable times or due to lack of memory.

Section 2 of this paper is devoted to some definitions related to the TPN model and its different state space contractions. In Section 3, we present the construction of our intermediate structure used in the refinement process. In Section 4, we propose an implementation for this construction approach that reduces the computing time complexity. Section 5 is devoted to the ASCG construction. Finally, Section 6 presents some experimental results. Our results are compared to those obtained in [5].

2. Time Petri nets

2.1. Definition and behavior

A TPN is a Petri net completed with time intervals attached to its transitions [4]. Formally, a TPN is a tuple $(P, T, Pre, Post, M_0, I_s)$ where:

- P is a finite set of places,
- T is a finite set of transitions ($P \cap T = \emptyset$),
- *Pre* and *Post* are the backward and the forward incidence functions: $P \times T \longrightarrow \mathbb{N}$, where $\mathbb{N}$ is the set of non-negative integers,
- M_0 is the initial marking, $M_0 : P \longrightarrow \mathbb{N}$,

[3] Linear-time Temporal Logic.
[4] More precisely, the size of its ASCG.

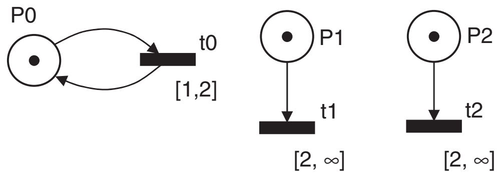


Fig. 1. A TPN model with unbounded static firing intervals.

- $I_s : T \to \mathbb{Q}^+ \times (\mathbb{Q}^+ \cup \{\infty\})$, $\mathbb{Q}^+$ is the set of non-negative rational numbers. Function I_s associates with each transition t an interval $[t\min(t), t\max(t)]$ called the static firing interval of t. $t\min(t)$ and $t\max(t)$ are, respectively, the minimal and maximal firing delays of the transition.

Let M be a marking and t a transition. t is enabled for M if and only if, all tokens required to fire t are present in M, i.e.: $\forall p \in P, M(p) \geqslant Pre(p, t)$.

We denote by $En(M)$ the set of all transitions enabled for the marking M.

As an example, Fig. 1 is the graphic representation of a small TPN defined by [5]:

- $P = \{P_0, P_1, P_2\}$,
- $T = \{t_0, t_1, t_2\}$,
- $Pre = (P_0, t_0) + (P_1, t_1) + (P_2, t_2)$,
- $Post = (P_0, t_0)$,
- $M_0 = P_0 + P_1 + P_2$,
- $I_s = \{(t_0, [1, 2]), (t_1, [2, \infty]), (t_2, [2, \infty])\}$.

There are mainly two known characterizations of the TPN state. The first characterization, called *interval state* [4], defines the state of the TPN model as a pair (M, Id) combining a marking M and a delay function Id. The function Id associates a firing interval with each enabled transition in M. When a transition t becomes enabled, its firing interval is initialized to its static firing interval $I_s(t)$. The bounds of the interval decrease synchronously with time, until t is fired or disabled by another firing. t can occur, if the lower bound of its interval reaches 0, but must be fired, without any additional delay, if the upper bound of its interval reaches 0. In this characterization, the initial state of the TPN model is the couple (M_0, Id_0) where M_0 is the initial marking and Id_0 is such that $Id_0(t) = I_s(t), \forall t \in En(M_0)$.

The second characterization of the TPN state, called *clock state* [16,17,21], defines the state of the model as a pair (M, V) combining a marking M and a clock valuation function V. In this characterization, a *clock* is associated with each transition to measure the elapsed time since its enabling. The function V associates with each enabled transition in M the value of its clock. When a transition t becomes enabled, its clock is initialized to 0. The clock increases with time until t is fired or disabled by another firing. t can occur if the value of its clock is within the static firing interval $I_s(t)$. It must be fired immediately, without any additional delay, when its clock reaches $t\max(t)$. In this characterization, the initial state of the TPN model is the couple (M_0, V_0) where M_0 is the initial marking and V_0 is such that $V_0(t) = 0, \forall t \in En(M_0)$.

The two characterizations of the TPN state given above are very closely related. If (M, V) is a *clock state*, its corresponding *interval state* is (M, Id), where: $\forall t \in En(M), Id(t) = [\mathrm{Max}(0, t\min(t) - V(t)), t\max(t) - V(t)]$. Note that, if $t\max(t) = \infty$, then $t\max(t) - V(t) = \infty$ independently of $V(t)$. In this case, all clock values of t greater or equal to $t\min(t)$ map to the same interval $[0, \infty]$. Consequently, several clock states may map to the same interval state, in which case they are obviously *bisimilar*.

Despite their relatedness, the two state characterizations still show some major differences. As an example, the characterization based on intervals is not appropriate for constructing ASCGs (the reasons will be made clear in Section 2.3). For this reason, our approach and those proposed in [5,21] are all based on the clock characterization of states. Note that, in [5], the authors do not speak explicitly about clock states. They characterize the TPN state in terms of intervals, but implicitly use the clock state characterization to construct both the intermediate abstraction, called SSCG and the ASCG.

[5] *Pre*, *Post* and M_0 are multi-sets represented by their formal sums.

Initially, the model is in its initial state *(clock state/interval state)*. The state evolves either by time progressions (clocks increase/delays decrease) or by firing transitions. The firing of a transition is supposed to take no time but leads to another marking (required tokens disappear while the produced ones appear). It follows that because of time density, the TPN model has in general an infinite number of reachable states (i.e.: infinite state space), even if it is bounded. Its analysis by enumerative techniques must pass by some state space contractions.

2.2. Concrete state spaces

The first abstraction of the TPN state space consists in hiding states reachable by time progression. We obtain a graph, called *concrete state space*, where only states reachable by firing transitions are represented [16,17]. This graph is defined by a tuple $(\Sigma, \longrightarrow, \sigma_0)$ where:

- $\sigma_0 \in \Sigma$ is the initial state ($\sigma_0 = (M_0, V_0)$),
- Σ is the set of states reachable from σ_0 by firing sequences of transitions,
- $\longrightarrow \subseteq (\Sigma \times T \times \Sigma)$ is the transition relation defined as follows:
  $((M, V), t_f, (M', V')) \in \longrightarrow$ if and only if, the transition t_f may occur from the state (M, V) after some delay $dh \geqslant 0$ and its firing leads to the state (M', V'), i.e.:
  - $t_f \in En(M)$,
  - $\exists dh \in \mathbb{R}$,[6]
    $(0 \leqslant dh) \wedge (t\min(t_f) - V(t_f) \leqslant dh) \wedge \bigwedge_{t \in En(M)} dh \leqslant (t\max(t) - V(t))$,
  - $\forall p \in P, M'(p) = M(p) - Pre(p, t_f) + Post(p, t_f)$,
  - $\forall t' \in En(M')$, $V'(t') = 0$, if t' is newly enabled (by transition t_f), $V'(t') = V(t') + dh$, if not.

We also denote by $(\sigma \longrightarrow_{t_f} \sigma')$ the condition $(\sigma, t_f, \sigma') \in \longrightarrow$.

The *concrete state space* of a TPN model is generally infinite and not suitable for an enumerative analysis. Hence, further contractions are needed.

2.3. State class spaces

A *state class space* of the TPN model is a graph representing a finite abstraction of its generally infinite *concrete state space* [16,17]. The nodes of this graph, called *state classes*, are agglomerations of concrete states. All concrete states agglomerated in one node must share the same marking.

Let $(\Sigma, \longrightarrow, \sigma_0)$ be the *concrete state space* of a TPN model. Formally, a *state class space* is defined as a structure $AS = (A, \Longrightarrow, \alpha_0)$ where:

- A is a cover of Σ.[7] Each element of A, called *state class*, is an agglomeration of concrete states sharing the same marking.
- α_0 is the initial state class of AS, such that $\sigma_0 \in \alpha_0$, and
- $\Longrightarrow \subseteq A \times T \times A$ is the successor relation that satisfies condition *EE*, i.e.:
  - $\forall(\alpha, t_f, \alpha') \in A \times T \times A, (\alpha \Longrightarrow_{t_f} \alpha') \Rightarrow (\exists \sigma \in \alpha, \exists \sigma' \in \alpha', \sigma \longrightarrow_{t_f} \sigma')$,
  - $\forall(\sigma, t_f, \sigma') \in (\Sigma \times T \times \Sigma), (\sigma \longrightarrow_{t_f} \sigma') \Rightarrow (\forall \alpha \in A \text{ s.t. } \sigma \in \alpha, \exists \alpha' \in A, (\sigma' \in \alpha' \wedge \alpha \Longrightarrow_{t_f} \alpha'))$.

The first part of condition *EE* prevents the connection of two state classes with no connected states. The second one ensures that all sequences of transitions in the *concrete state space* are represented within the *state class space*.

The relation $\Longrightarrow$ may satisfy other additional conditions such as

EA:

$$\forall(\alpha, t_f, \alpha') \in A \times T \times A, (\alpha \Longrightarrow_{t_f} \alpha') \Rightarrow (\forall \sigma' \in \alpha', \exists \sigma \in \alpha, \sigma \longrightarrow_{t_f} \sigma').$$

AE:

$$\forall(\alpha, t_f, \alpha') \in A \times T \times A, (\alpha \Longrightarrow_{t_f} \alpha') \Rightarrow (\forall \sigma \in \alpha, \exists \sigma' \in \alpha', \sigma \longrightarrow_{t_f} \sigma').$$

[6] $\mathbb{R}$ being the set of real numbers.

[7] A cover of Σ is a collection of sets whose union contains Σ.

Theorem 1 establishes a relation between conditions *AE*, *EA* and properties of the model preserved in the *state class space*.

Theorem 1. *Let* $AS = (A, \Longrightarrow, \alpha_0)$ *be a* state class space *of a TPN model. The following relations hold*:

(i) *If* (AS *satisfies condition EA and* $\alpha_0 = \{\sigma_0\}$) *then AS preserves LTL properties of the TPN model*,
(ii) *If AS satisfies condition AE then it preserves CTL* properties of the TPN model.*

Proof. (i) It suffices to show that *AS* and its concrete state space have the same sequences of transitions. Condition *EE* (the second part) ensures that any sequence of transitions in the concrete state space is also a sequence of transitions in *AS*. What remains to show is that any sequence of transitions within *AS* exists within the concrete state space. For this, let $\alpha_0 \Longrightarrow_{t_0} \alpha_1 \ldots \alpha_{n-1} \Longrightarrow_{t_{n-1}} \alpha_n$ be a path in *AS*. Since we have $\alpha_{n-1} \Longrightarrow_{t_{n-1}} \alpha_n$, condition *EA* assures that $\forall \sigma^n \in \alpha_n, \exists \sigma^{n-1} \in \alpha_{n-1}$ such that $\sigma^{n-1} \longrightarrow_{t_{n-1}} \sigma^n$. By going backward in the similar way, we can show that $\exists \sigma^0 \in \alpha_0, \exists \sigma^1 \in \alpha_1, \ldots, \exists \sigma^{n-1} \in \alpha_{n-1}$ such that $\sigma^0 \longrightarrow_{t_0} \sigma^1 \ldots \sigma^{n-1} \longrightarrow_{t_{n-1}} \sigma^n$. Since $\alpha_0 = \{\sigma_0\}$ then $\sigma^0 = \sigma_0$, which guarantees that the sequence starts from the initial state.

(ii) Knowing that, in absence of silent transitions [8] [5], bisimilar states satisfy identical *CTL** properties, it suffices to show that if *AS* satisfies *AE*, it is bisimilar to the TPN concrete state space. For this, we need to find a bisimulation containing (α_0, σ_0).

Let $\mathcal{B}$ be the binary relation defined by

$$\forall(\alpha, \sigma) \in (A \times \Sigma), (\alpha, \sigma) \in \mathcal{B} \quad \text{iff } \sigma \in \alpha.$$

The following three properties hold for $\mathcal{B}$:

(1) $(\alpha_0, \sigma_0) \in \mathcal{B}$;
(2) $\forall(\alpha, \sigma) \in \mathcal{B}, (\alpha \Longrightarrow_{t_f} \alpha') \Rightarrow \exists \sigma', (\sigma \longrightarrow_{t_f} \sigma') \wedge (\alpha', \sigma') \in \mathcal{B}$;
(3) $\forall(\alpha, \sigma) \in \mathcal{B}, (\sigma \longrightarrow_{t_f} \sigma') \Rightarrow \exists \alpha', (\alpha \Longrightarrow_{t_f} \alpha') \wedge (\alpha', \sigma') \in \mathcal{B}$.

Property (1) is obvious from the definition of α_0. Property (2) is a direct consequence of condition *AE*. Finally, property (3) is true because *AS* satisfies condition *EE*. From these properties, we conclude that $\mathcal{B}$ is a bisimulation, and therefore, *AS* is bisimilar to the *TPN* concrete state space. □

Note that there are some differences between condition *EE* presented here and those given in [5,17].
EE in [5]:

$$\forall(\alpha, t_f, \alpha') \in A \times T \times A, (\exists \sigma \in \alpha, \exists \sigma' \in \alpha', \sigma \longrightarrow_{t_f} \sigma') \Leftrightarrow (\alpha \Longrightarrow_{t_f} \alpha').$$

EE in [17]:

$$\forall(\alpha, t_f, \alpha') \in A \times T \times A, (\exists \sigma \in \alpha, \exists \sigma' \in \alpha', \sigma \longrightarrow_{t_f} \sigma') \Rightarrow (\alpha \Longrightarrow_{t_f} \alpha').$$

Conditions *EE* given in [5,17] impose to connect each two classes α and α' whenever some state of the first one has a successor in the second one. However, most contractions [4,5,21] proposed in the literature do not obey this rule, while they are still valid. As an example, consider the TPN model with its SSCG [9] shown in Fig. 2. In the figure, the inequalities associated with each state class constrain the values of clocks associated with transitions enabled at the class. [10] Note that the unique state of class C_0 (i.e., the initial state of the TPN model) belongs also to class C_1.

[8] Internal activities of a system not visible for an external observer.
[9] The TPN strong state class graph is a state class space proposed by Berthomieu and Vernadat [5].
[10] In other words, these inequalities characterize clock domains of all states agglomerated in the class.

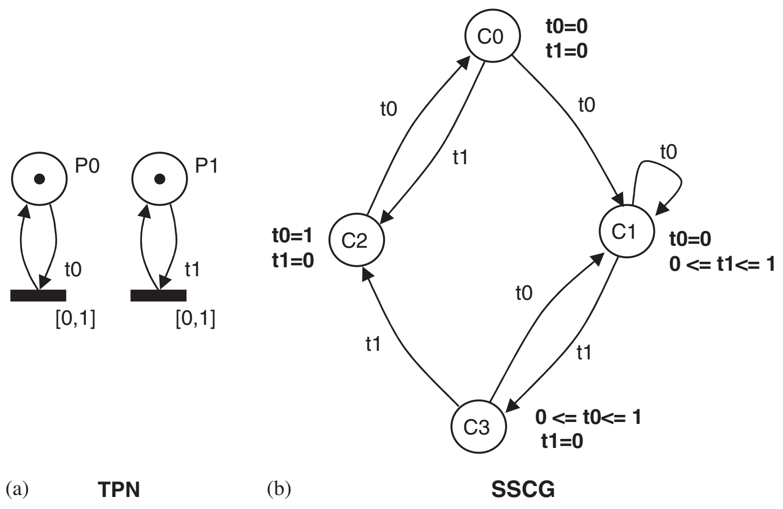


Fig. 2. A TPN model and its SSCG.

This state has itself as a successor by t_0. C_0 is connected by t_0 to C_1 but C_1 is not connected to C_0 by t_0, which contradicts condition *EE* given in [5] and the one given in [17].

Berthomieu and Menasche used the interval characterization of the TPN state and proposed 20 years ago to agglomerate, into one state class, all states reachable by firing the same sequence of transitions [4]. Each state class is characterized by the common marking of its states and the union of their firing intervals. The resulting graph, called linear state class graph (LSCG), preserves *LTL* properties of the model but not necessarily its *CTL** ones [5,21]. Moreover, LSCGs are not suitable to be refined into ASCGs. The refinement procedure is mainly based on splitting state classes, which is not possible with the characterization of state classes used for LSCGs. Indeed, interval states agglomerated into an LSCG state class cannot be identified one by one. The way the agglomeration is characterized makes it an irreversible operation.

Yoneda and Ryuba developed in [21] an approach to generate a *state class space* that preserves *CTL** properties of the TPN model. Their approach consists in two steps. The first step builds an intermediate contraction of the TPN state space as a *state class space* satisfying condition *EA*. The second step refines this state class space, using a partition refinement technique, until both conditions *EA* and *AE* are satisfied. The resulting graph, called ASCG, preserves the *CTL** properties of the TPN model, but the approach has some disadvantages. The state class definition used in this approach is somewhat complicated. [11] Furthermore, since condition *AE* is sufficient for a state class space to preserve *CTL** properties, enforcing condition *EA* complicates the computations and yields in general bigger ASCGs. The approach is also limited to TPN models with bounded firing intervals, which reduces its application extent.

Recently, Berthomieu and Vernadat brought in [5] some improvements to the characterization of *state classes* and to the ASCG construction. Their ASCGs are in general smaller than those obtained with Yoneda–Ryuba's approach and much faster to compute too. Moreover, Berthomieu–Vernadat's approach produces finite ASCGs for all bounded TPN models, including those with unbounded firing intervals.

3. Compact state class graph of the TPN model

Berthomieu–Vernadat in [5] and Yoneda–Ryuba in [21] use a partition refinement technique to construct an ASCG, and both use as an intermediate abstraction a *state class space* satisfying condition *EA*. Enforcing condition *EA* is however not necessary. It also leads in general to very large state class spaces with a high degree of state redundancy. [12] Experimental results have shown that this redundancy induces the refinement process to waste time and space generating

[11] A state class is characterized by a marking, a set of constraints and the firing sequence which led to that state class.

[12] One state may appear in several state classes.

redundant classes which need to be eliminated. To attenuate these problems, we propose to generate a compact state class graph (CSCG) with no regard to condition *EA*.

For reasons of clarity, we will consider only *T-safe* TPNs (no multi-enabled transitions). The case of multienabledness can be treated in the same way as in [3].

Let $(\Sigma, \longrightarrow, \sigma_0)$ be a concrete state space of a TPN model. The CSCG is a state class space where each state class is defined as a pair (M, F) combining a marking M and a formula F. F characterizes the clock domains of all concrete states agglomerated in the state class. In F, the clock of each enabled transition for M is represented by a variable with the same name. By abuse of language, F is also called *the domain of the class*.

The initial state class of the CSCG is the pair (M_0, F_0) where:

- M_0 is the initial marking, and
- $F_0 = (\bigwedge_{t \in En(M_0)} t = 0)$.

State classes are computed progressively by repeatedly applying the following *firing rule* starting from the initial state class.

3.1. Firing rule of transitions from state classes

Let $\alpha = (M, F)$ be a state class and t_f a transition.

- t_f can occur from α if and only if, there exists at least one state in α from which t_f can fire, i.e.:
  - t_f is enabled for the marking M, and
  - the following formula is consistent:
    $F \wedge (dh \geqslant 0) \wedge (t\min(t_f) \leqslant t_f + dh) \wedge (\bigwedge_{t \in En(M)} (t + dh \leqslant t\max(t)))$,
- If t_f can occur from α, its firing leads to the *class* $\alpha' = (M', F')$ such that:
  - $\forall p \in P, M'(p) = M(p) - Pre(p, t_f) + Post(p, t_f)$,
  - the clock domain F' is computed in four steps:
    1. Initialize F' with the formula $F \wedge (dh \geqslant 0)$, and replace each variable t by $(t - dh)$ (this substitution increases clocks of all enabled transitions by exactly dh time units),
    2. Add the constraints: $(t\min(t_f) \leqslant t_f)$ and $(\bigwedge_{t \in En(M)} t \leqslant t\max(t))$,
    3. Eliminate by substitution t_f, dh and all variables associated with transitions conflicting with t_f for the marking M. A transition t of $En(M)$ conflicts with t_f for M iff: $(\exists p \in P, M(p) < Pre(p, t) + Pre(p, t_f))$,
    4. For each transition t newly enabled in M', add the constraint: $t = 0$.

During the construction of the CSCG, each newly computed class is compared with the previously computed ones. All state classes, with the same marking, having domains such as one is included into the other are grouped into one node. If the initial state class is combined with another one, the node obtained becomes initial.

The idea of performing such an agglomeration is however not new. Similar techniques have been successfully used for TA [10], under the name *inclusion abstraction*, to check for reachability properties.

The objective behind computing a CSCG is to use it as an intermediate structure for constructing an ASCG. The intermediate structure used in [5] for the same objective is generated with the same firing rule given above, but state classes are agglomerated whenever they are equal to each other. The result is a state class space, called SSCG which preserves the *LTL* properties of the TPN model. Compared to the SSCG, the CSCG is just its abstraction by inclusion which satisfies condition *EE* but does not necessarily preserve *LTL* properties. Nevertheless, it is still appropriate to be refined into an ASCG with a good impact on performances (see Section 6 for experimental results).

3.2. Relaxing state classes with unbounded firing intervals

The CSCG construction approach presented in the previous subsection may generate infinite graphs for bounded TPN models with unbounded static intervals. To complete this approach for the case of bounded TPN models with unbounded static intervals, we use the relaxation operation proposed by Berthomieu and Vernadat in [5], to resolve the same problem.

Let (M, F) be a *state class* such as some of its enabled transitions have unbounded static firing intervals. We denote by $En_{<\infty}(M)$ and $En_{=\infty}(M)$ the following sets:

- $En_{<\infty}(M) = \{t | t \in En(M) \wedge t\max(t) < \infty\}$,
- $En_{=\infty}(M) = \{t | t \in En(M) \wedge t\max(t) = \infty\}$.

The relaxation of (M, F) consists in replacing it by the set of classes: $\{(M, F_e) | (e = \emptyset \vee e \subseteq En_{=\infty}(M))\}$, F_e is a consistent formula that characterizes states of the class (M, F) where all transitions of e have not yet reached their minimal delays, while those of $(En_{=\infty}(M) - e)$ have either reached, or passed over their minimal delays.

F_e is computed in three steps:

1. Initialize F_e with: $F \wedge (\bigwedge_{t \in e} t < t\min(t)) \wedge$
   $(\bigwedge_{t' \in En_{=\infty}(M)-e} t' \geqslant t\min(t'))$,
2. Eliminate all variables of $(En_{=\infty}(M) - e)$,
3. Add the constraint:
   $(\bigwedge_{t' \in En_{=\infty}(M)-e} t\min(t') \leqslant t' \leqslant \infty)$.

The last operation, called *class relaxation*, replaces the domain of each transition t' which has reached its minimal delay, with the domain $[t\min(t'), \infty]$. This operation may add some extra clock states to the relaxed state class, but does not alter the behavior of the class. The explanation could be well understood if we revert to the interval characterization of states. From this perspective, the relaxation does not add any new interval state to the relaxed class. [13] All added clock states are bisimilar to some states already present in the class before its relaxation.

3.3. Illustrative example

Consider the TPN model shown in Fig. 1. Its SSCG [14] shown in Fig. 3 consists of 15 nodes and 30 arcs. The successor state classes of the initial class, by transition t_0, are computed in two steps. The first step computes the successor of *class*0 by t_0, using the firing rule given in Section 3.1. The result is the state class: $(P0+P1+P3, 1 \leqslant t_1 \leqslant 2 \wedge t_2 = t_1 \wedge t_0 = 0)$.

The second step relaxes the resulting state class by replacing it with two classes, one for $e = \{t_1, t_2\}$, the other one for $e = \emptyset$ (formulas corresponding to $e = \{t_1\}$ and $e = \{t_2\}$ are not consistent). The subclasses are, respectively:

- $class1 = (P0 + P1 + P3, 1 \leqslant t_1 < 2 \wedge t_2 = t_1 \wedge t_0 = 0)$,
- $class2 = (P0 + P1 + P3, 2 \leqslant t_1 \leqslant \infty \wedge 2 \leqslant t_2 \leqslant \infty \wedge t_0 = 0)$.

For the remaining state classes, no relaxation is required, but some of them are included in the others. For example, state classes named *class*4, *class*9, *class*12 are all included in state class *class*3. In the CSCG (see Fig. 4), all these classes are grouped in the state class *class*3 (class $C3$ in Fig. 4). The same thing applies to state classes *class*6, *class*10, *class*13 which are grouped in *class*5 (class $C5$ in Fig. 4), and state classes *class*8, *class*11, *class*14 which are grouped in *class*7 (class $C7$ in Fig. 4). The resulting CSCG shown in see Fig. 4 consists only of 6 nodes and 15 arcs.

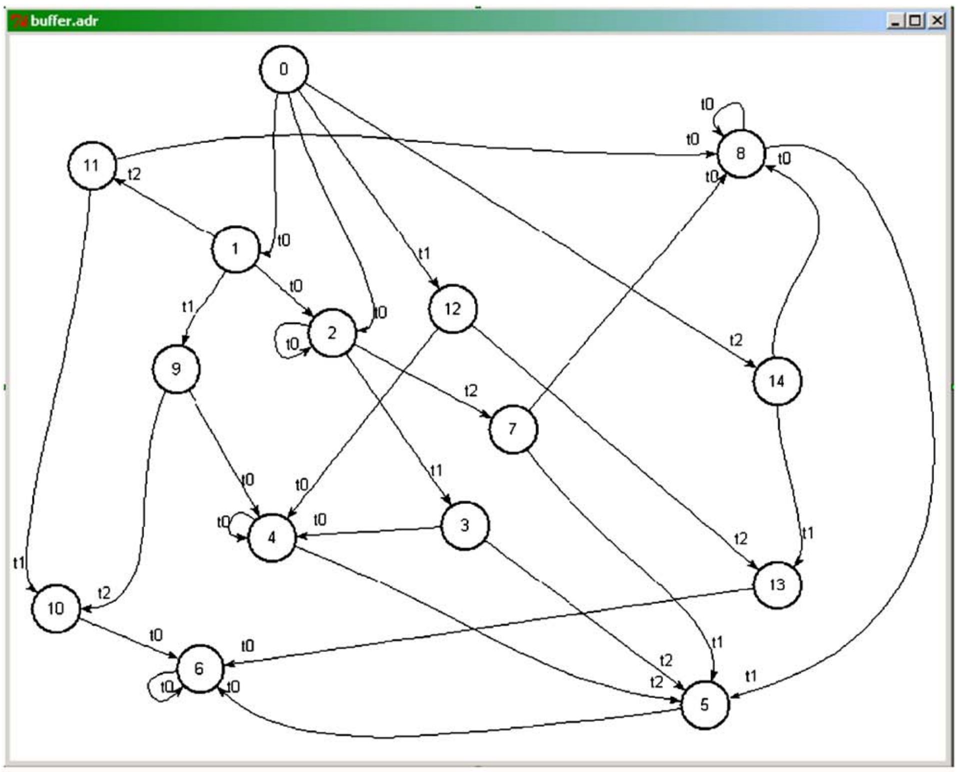
Note that the grouping of state classes by inclusion is not performed at the end of the SSCG construction but during the construction itself.

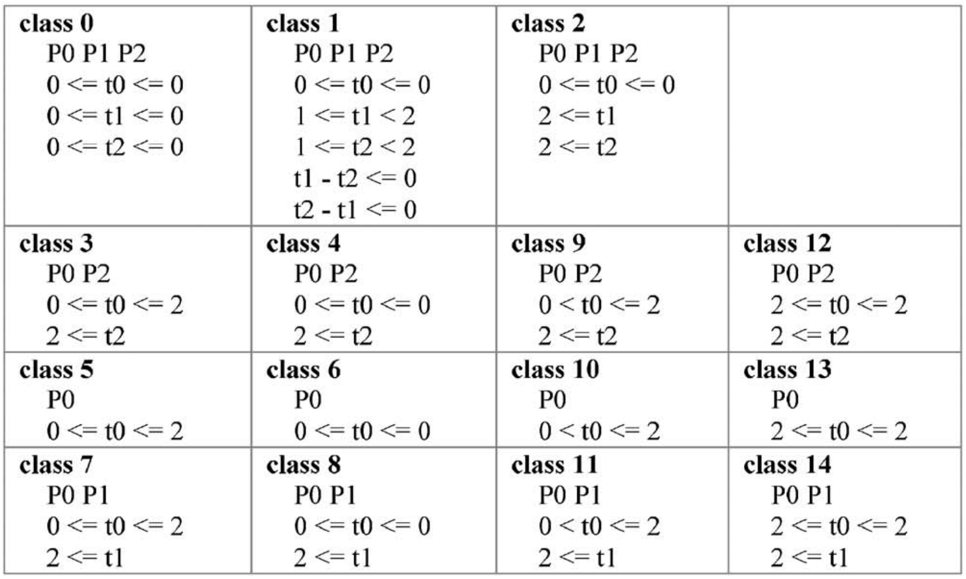
4. Reducing the computing complexity of state classes

The construction of the CSCG requires computing state classes using the firing rule, checking them for inclusion, and relaxing them whenever required. The efficiency of the construction relies heavily on the way state classes are implemented. From the firing rule, one can see that the formula F of each state class (M, F) is a conjunction of atomic constraints of the form $(t - t' \prec c)$, $(-t \prec c)$ or $(t \prec c)$, where $c \in \mathbb{Q} \cup \{\infty, -\infty\}$, $\prec \in \{<, =, \leqslant, >, \geqslant\}$ and t, t' are transitions. The conjunction of this type of inequations is known to define a convex domain. A well known data structure that fits perfectly with this framework is the difference bound matrix (DBM) data structure [2,22]. Furthermore, all operations performed on state classes when constructing CSCGs are well defined for DBMs. These operations are made

[13] See Section 2 for the relation between the clock and the interval characterization of states.

[14] The SSCG is computed using the tool Tina-2.2.6 which implements the approach of Berthomieu and Vernadat [5].





class 0 P0 P1 P2 0 <= t0 <= 0 0 <= t1 <= 0 0 <= t2 <= 0	**class 1** P0 P1 P2 0 <= t0 <= 0 1 <= t1 < 2 1 <= t2 < 2 t1 - t2 <= 0 t2 - t1 <= 0	**class 2** P0 P1 P2 0 <= t0 <= 0 2 <= t1 2 <= t2	
class 3 P0 P2 0 <= t0 <= 2 2 <= t2	**class 4** P0 P2 0 <= t0 <= 0 2 <= t2	**class 9** P0 P2 0 < t0 <= 2 2 <= t2	**class 12** P0 P2 2 <= t0 <= 2 2 <= t2
class 5 P0 0 <= t0 <= 2	**class 6** P0 0 <= t0 <= 0	**class 10** P0 0 < t0 <= 2	**class 13** P0 2 <= t0 <= 2
class 7 P0 P1 0 <= t0 <= 2 2 <= t1	**class 8** P0 P1 0 <= t0 <= 0 2 <= t1	**class 11** P0 P1 0 < t0 <= 2 2 <= t1	**class 14** P0 P1 2 <= t0 <= 2 2 <= t1

Fig. 3. SSCG of the model in Fig. 1 built with TINA tool.

simple by putting each DBM 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. Its time complexity is $O(n^3)$, where n is the DBM order.

In our implementation for building CSCGs, state classes are represented using DBMs with canonical forms computed in $O(n^2)$ instead of $O(n^3)$.

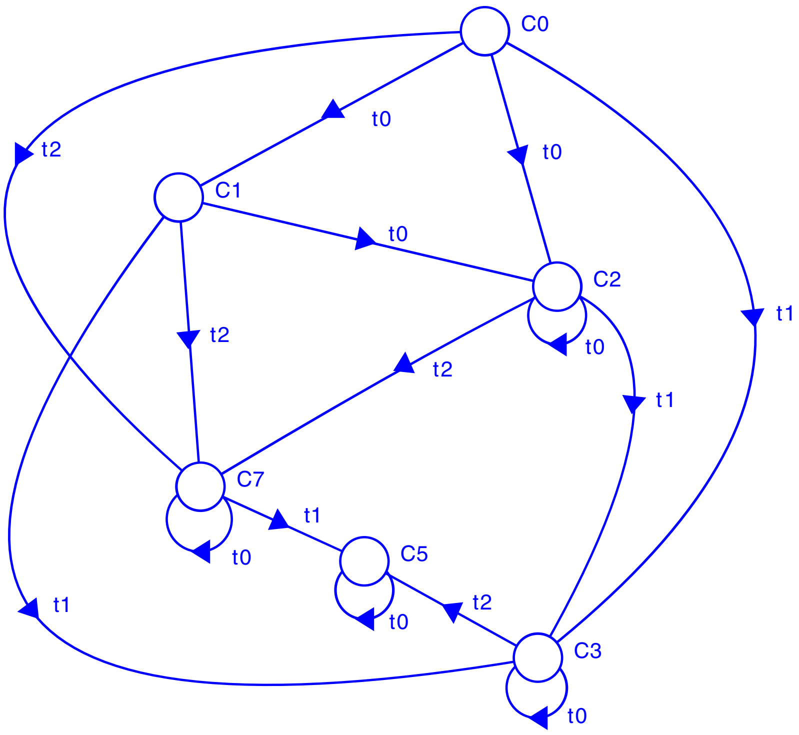


Fig. 4. CSCG of the model in Fig. 1 built with our tool.

Let (M, F) be a state class. Its canonical form is the pair (M, B) where B is the DBM of F in its canonical form. The order of B is $|En(M) \cup \{o\}|$, where o represents the value zero.

Usually, a DBM is represented as a matrix where each element is a couple composed of a constant (a bound) and a comparison operator ($\leqslant$ or $<$). For clarity, we separate, in what follows, bounds from operators into two distinct matrices H and S, respectively. Using this convention, the DBM of F is the couple (H, S) defined by:
$\forall(x, y) \in (En(M) \cup \{o\})^2$,

- $H(x, y) = Sup(x - y, F)$, where $Sup(x - y, F)$ is the supremum of $x - y$ in the domain of F,
- $S(x, y)$ is either $\leqslant$ or $<$, depending respectively on whether $x - y$ reaches its supremum in the domain of F or not.

The canonical form of the initial state class is $\alpha_0 = (M_0, (H_0, S_0))$, where M_0 is the initial marking of the model, H_0 is the null matrix and all elements of S_0 are set to $\leqslant$.

We establish in Propositions 1 and 2 an implementation of the *firing rule* that directly computes the canonical form of each reachable state class in $O(n^2)$ (n is the number of transitions enabled for the class). In the same way, we show in Proposition 3 how to relax state classes more efficiently.

Note that in the remaining, we will focus only on computations involving the matrix H. The matrix S is computed as for DBMs [2,22].

Proposition 1. *Let (M, F) be a state class, $(M, (H, S))$ its canonical form and t_f a transition.*
t_f is firable from $(M, (H, S))$ if and only if:

- $t_f \in En(M)$, *and*
- $t\min(t_f) \leqslant \text{Min}_{t \in En(M)}(t\max(t) + H(t_f, t))$.
Intuitively, this means that the clock t_f has to reach $t\min(t_f)$, before any other enabled transition t reaches $t\max(t)$.

Proof (*Sketch of proof*). The proof is based a graph representation of DBMs called *constraint graphs* [2]. A constraint graph of a DBM associated with a formula F is a weighted directed graph, where nodes represent variables of F, with one extra node, denoted o, representing the value 0. An arc connecting node x to node y with weight $(\prec, c)$, where $\prec$ is either $<$ or $\leqslant$, represents the constraint: $y - x \prec c$. An arc connecting node x to node y with weight c represents the constraint: $y - x \leqslant c$.

Let F be a set of constraints representable by a *constraint graph*.

- F has, at least, one solution (one tuple of values that satisfies, at once, all constraints in F) if and only if, the *constraint graph* of F has no negative cycle,
- If the *constraint graph* of F has no negative cycle, the weight of the shortest path, in the graph, going from a node y to a node x, is equal to $Sup(x - y, F)$.

The formula of each reachable state class is consistent and can be represented by a *constraint graph*. Consequently, its *constraint graph* has no negative cycle.

Let $\alpha = (M, F)$ be a reachable state class, (H, S) the DBM (in its canonical form) of F and G its *constraint graph*. Using the definition of H and the above results, the weight of the shortest path, in G, from a node y to a node x is equal to $H(x, y)$.

From the firing rule given in Section 3.1, we deduce that transition t_f is firable from α if and only if, t_f is enabled for the marking M and the following formula is consistent: $F \wedge (\bigwedge_{t \in En(M)} t - t_f \leqslant t\max(t) - t\min(t_f))$.

In other words, the *constraint graph* of F completed with the set of arcs $\{(t_f, t, t\max(t) - t\min(t_f)) | t \in En(M)\}$ has no negative cycle.

Since before adding these arcs, the graph did not contain any negative cycle, the completed graph will have no negative cycle if and only if, all cycles going through one of the added arcs are not negative (see Fig. 5b, added arcs are dotted).

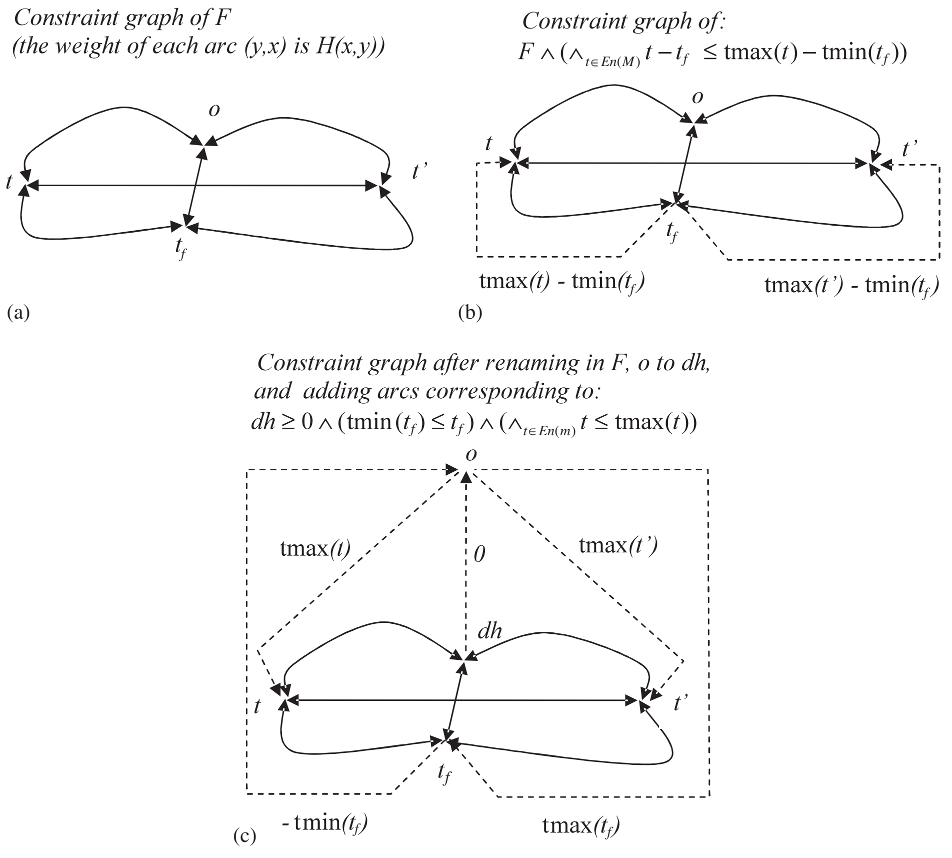

Constraint graph of F
(the weight of each arc (y,x) is H(x,y))
o
t
t'
t_f
(a)
Constraint graph of:
$F \wedge (\wedge_{t \in En(M)} t - t_f \leq \text{tmax}(t) - \text{tmin}(t_f))$
o
t
t'
t_f
tmax(t) - tmin(t_f)
tmax(t') - tmin(t_f)
(b)
Constraint graph after renaming in F, o to dh,
and adding arcs corresponding to:
$dh \geq 0 \wedge (\text{tmin}(t_f) \leq t_f) \wedge (\wedge_{t \in En(m)} t \leq \text{tmax}(t))$
o
tmax(t)
0
tmax(t')
dh
t
t'
t_f
- tmin(t_f)
tmax(t_f)
(c)

Fig. 5. Added arcs in the constraint graph of F.

The weight of the shortest cycle going through an added arc $(t_f, t, t\max(t) - t\min(t_f))$ is $t\max(t) - t\min(t_f) + H(t_f, t)$. This weight must be nonnegative, i.e.: $\forall t \in En(M), t\max(t) - t\min(t_f) + H(t_f, t) \geqslant 0$. □

Proposition 2. *Let (M, F) be a state class, $(M, (H, S))$ its canonical form and t_f a transition.*
If t_f is firable from $(M, (H, S))$, its firing leads to the state class $(M', (H', S'))$ computed as follows:

- $\forall p \in P, M'(p) = M(p) - Pre(p, t_f) + Post(p, t_f)$;
- $\forall t \in En(M')$,
  - $H'(t, t) = 0$,
  - *If t is newly enabled*: $H'(t, o) = H'(o, t) = 0$,
  - *If t is not newly enabled*: $H'(t, o) = \text{Min}_{t' \in En(M)}(t\max(t') + H(t, t'))$,
    and $H'(o, t) = \text{Min}(H(o, t), H(t_f, t) - t\min(t_f))$,
- $\forall (t, t') \in (En(M'))^2, t \neq t'$,
  - *If t and t' are newly enabled*: $H'(t, t') = H'(t', t) = 0$,
  - *If t is newly enabled and t' is not*: $H'(t, t') = H'(o, t')$ *and* $H'(t', t) = H'(t', o)$,
  - *If t and t' are not newly enabled*: $H'(t, t') = \text{Min}(H(t, t'), H'(t, o) + H'(o, t'))$.

Proof (*Sketch of proof*). Suppose that the transition t_f is firable from the state class $\alpha = (M, (H, S))$. From the firing rule given in Section 3.1, we deduce that the matrix H' of the class reachable from α by firing t_f, can be computed using the *constraint graph* corresponding to H as follows:

1. Rename the node o to dh, then add a new node o and the arc $(dh, o, 0)$. This operation corresponds to the step 1 of the firing rule given in Section 3.1.
2. Add the arc $(t_f, o, -t\min(t_f))$ and all arcs in the set $\{(o, t', t\max(t')) | t' \in En(M)\}$. This corresponds to the constraints: $(t\min(t_f) \leqslant t_f)$ and $((\bigwedge_{t \in En(M)} t \leqslant t\max(t)))$,
3. Rename the node t_f and all nodes associated with transitions conflicting with t_f for M, to avoid having different nodes with the same name,
4. For each transition x newly enabled in M', add a new node x and both arcs $(x, o, 0)$ and $(o, x, 0)$. This corresponds to the constraint: $x = 0$.

For each couple (x, y) of $(En(M') \cup \{o\})^2$, $H'(x, y)$ is the weight of the shortest path from node y to node x, in the completed *constraint graph* (see Fig. 5c, added arcs are dotted):

- If t is not newly enabled, the shortest path from node o to node t is the shortest path among those going through an arc $(o, t', t\max(t'))$, $t' \in En(M)$. Otherwise, its value is 0,
- If t is not newly enabled, the shortest path from node t to node o is the shortest path among those going through $(dh, o, 0)$ or $(t_f, o, -t\min(t_f))$. Otherwise, its value is 0. Note that dh corresponds to the node o in the constraint graph of F,
- If t and t' are not newly enabled, the shortest path from node t' to node t is the shortest path among those going through node o and those which do not pass through node o. Note that all added arcs are either ingoing or outgoing arcs of o,
- If t and t' are newly enabled, $t - t' = 0$,
- If t is newly enabled and t' is not, $t - t' = -t'$ and $t' - t = t'$. □

We have shown, in Propositions 1 and 2, how to reduce the computing complexity of each state class. The complexity of testing the consistency is reduced from $O(n^3)$ to $O(n)$ while computing complexity of the canonical form passes from $O(n^3)$ to $O(n^2)$.

To handle TPN models with unbounded firing intervals, some computed state classes have to be split and relaxed. Proposition 3 shows how to compute in $O(n^2)$ the canonical form of each resulting class.

Proposition 3. *Let (M, F) be a state class with $En_{=\infty}(M) \neq \emptyset$, $(M, (H, S))$ its canonical form and e a subset of $En_{=\infty}(M)$ (e can be empty).*

- *The class $\alpha_e = (M, F_e)$, corresponding to the subset e, is not empty if and only if:*
  - $\forall t \in e, t\min(t) + H(o, t) > 0$,
  - $\forall t \in En_{=\infty}(M) - e, H(t, o) - t\min(t) \geqslant 0$,
  - $\forall t \in e, \forall t' \in En_{=\infty}(M) - e, t\min(t) + H(t', t) - t\min(t') > 0$
- *If α_e is not empty, its canonical form $(M, (H_e, S_e))$ is computed using the canonical form of (M, F) as follows:*
  - $\forall t \in En(M), H_e(t, o) = \text{Min}(H(t, o), \text{Min}_{t' \in e}(t\min(t') + H(t, t')))$,
  - $\forall t \in En(M), H_e(o, t) = \text{Min}(H(o, t), \text{Min}_{t' \in En_{=\infty}(M)-e}(H(t', t) - t\min(t')))$,
  - $\forall (t, t') \in En(M)^2, H_e(t, t') = \text{Min}(H(t, t'), H_e(t, o) + H_e(o, t'))$,
  - $\forall t \in En_{=\infty}(M) - e, H_e(t, o) = \infty$ *and* $H_e(o, t) = -t\min(t)$,
  - $\forall t \in En_{=\infty}(M) - e, \forall t' \in En(M) - \{t\}, H_e(t, t') = \infty$ *and* $H_e(t', t) = H_e(t', o) - t\min(t)$.

Proof (*Sketch of proof*). The proof is also based on *constraint graphs* (see Proposition 1). The *constraint graph* of F_e can be obtained from the *constraint graph* of F by adding the arcs:
$\{(o, t, t\min(t)) | t \in e\}$ and
$\{(t', o, -t\min(t')) | t' \in En_{<\infty}(M) - e\}$.
These arcs correspond, respectively, to the constraints:
$(\bigwedge_{t \in e} t < t\min(t))$ and $(\bigwedge_{t' \in En_{=\infty}(M)-e} t' \geqslant t\min(t'))$.

F_e is consistent if and only if, the obtained graph has no negative cycle, in which case, the shortest path from a node x to a node y is the upper bound of the distance $(y - x)$, i.e.: $H_e(y, x)$.

The relaxation operation replaces the domain of each transition t, belonging to $(En_{=\infty}(M) - e)$, by the interval $[t\min(t), \infty]$. □

5. Atomic state class graph of the TPN model

Let $(\Sigma, \longrightarrow, \sigma_0)$ be the *concrete state space* of a TPN model and $AS = (A, \Longrightarrow, \alpha_0)$ one of its *state class spaces*. AS is said to be atomic if and only if, it satisfies condition AE, i.e.: $\forall(\alpha, t_f, \alpha') \in \Longrightarrow, (\forall \sigma \in \alpha, \exists \sigma' \in \alpha', (\sigma, t_f, \sigma') \in \longrightarrow)$.

In other terms, AS is atomic if and only if:

$$\forall(\alpha, t_f, \alpha') \in \Longrightarrow, (\alpha \subseteq Pred(\alpha', t_f)).$$

$Pred(\alpha', t_f)$ is the set of all states which may lead by firing the transition t_f to some states in α'. To verify the atomicity of class α for the transition (α, t_f, α'), it suffices to verify that α is equal or included in $Pred(\alpha', t_f)$. In case α is not atomic, it is partitioned into a set of convex subclasses so as to isolate the predecessors of α' by t_f in α, from those which are not. This *refinement operation* is repeated until all state classes are atomic.

A CSCG is a state class space which is not necessarily atomic. Nevertheless, the characterization of its state classes allows it to be refined into an atomic state class space. Starting from a finite CSCG, its *non-atomic* state classes are repeatedly split, until an ASCG is obtained.

5.1. Splitting non-atomic state classes

Let AS be a state class space and (α, t_f, α') a transition for which $\alpha = (M, F)$ is not atomic. Algorithm 1 shows how to split α according to $Pred(\alpha', t_f)$, represented by $\alpha'' = (M'', F'')$ in the algorithm, with $M'' = M$.

The *splitting* algorithm is used only if the test $\alpha \subseteq Pred(\alpha', t_f)$ fails. It returns either a partition of α or an empty set. A partition is returned if $Pred(\alpha', t_f) \cap \alpha \neq \emptyset$. The last subclass added to the partition is exactly the predecessors of α' by t_f in α. Each subclass of the partition inherits all connections of α. In case $(\alpha, t, \alpha) \in \Longrightarrow$, each subclass of the partition is connected, by t, to all subclasses of the partition, including itself. If α and $Pred(\alpha', t_f)$ do not share any state, the splitting function returns an empty set, and the transition (α, t_f, α') is removed.

Algorithm 1.

```
Partition **Function** Split (Class $\alpha$, Class $\alpha''$)
{
        Formula $X := F$; // $F$ is the formula of the class $\alpha$
        Partition $Part := \emptyset$;
        **for** each constraint $f$ of $F''$ **do**
        {
                **if** $(X \wedge f)$ is not consistent **then**
                // there is no predecessor of $\alpha'$ by $t_f$ in $\alpha$
                    **return** $\emptyset$;
                **if** $(X \wedge \neg f)$ is consistent **then**
                    $Part := Part \cup \{(M, X \wedge \neg f)\}$;
                $X := (X \wedge f)$;
        }
        // At this point, $(M, X)$ is equal to $Pred(\alpha', t_f)$
        $Part := Part \cup \{(M, X)\}$;
        **return** $Part$;
}
```

Similarly to what is done in Propositions 1 and 2, the consistency test of $(X \wedge f)$ and $(X \wedge \neg f)$, in Algorithm 1, is performed in $O(n)$, where n is the number of variables in X. Their canonical forms are computed in $O(n^2)$ too. The next proposition shows how to compute $Pred(\alpha', t_f)$.

Proposition 4. *Let $New(M', t_f)$ be the set of all transitions newly enabled in the marking M' of α', i.e.: $New(M', t_f) = \{t \in En(M') | \exists p \in P, M'(p) - Post(p, t_f) < Pre(p, t)\}$.*
The set $Pred(\alpha', t_f)$ of all states that lead by firing t_f to some states in $\alpha' = (M', F')$ is computed in six steps: Let M'' and F'' be the marking and the formula of $Pred(\alpha', t_f)$.

1. $\forall p \in P, M''(p) = M'(p) + Pre(p, t_f) - Post(p, t_f)$,
2. *Initialize F'' to $(F' \wedge \bigwedge_{t \in New(M', t_f)} t = 0)$,*
3. *Eliminate by substitution all transitions in $New(M', t_f)$,*
4. *Add constraints*: $(t\min(t_f) \leqslant t_f)$ *and* $(\bigwedge_{t \in En(M'')} 0 \leqslant t \leqslant t\max(t))$,
5. *Replace each variable t by $t - dh$ and add the constraints $dh \leqslant 0$, and $(\bigwedge_{t \in En(M'')} 0 \leqslant t)$,*
6. *Eliminate dh by substitution.*

Proof (*Sketch of proof*). Since the firing of transition t_f sets the clock of each newly enabled transition to 0, in step two, we extract from α' the subset of states where the clock of all newly enabled transitions are equal to 0. In step four, we add the firing constraints of transition t_f so as to only keep states that might fire t_f. Finally, in the fifth step, we go back in time (each variable is decreased by dh time units). $\square$

The canonical form of $Pred(\alpha', t_f)$ is computed in $O(n^2)$, using a technique similar to what is presented in Proposition 2.

5.2. Building the atomic state class graph

Algorithm 2, generates the ASCG from the CSCG using a partition refinement technique. It repeatedly scans the transitions (the arcs) of the graph, checking for the satisfaction of condition *AE*. The *algorithm* stops when all transitions satisfy this condition. During the process, if some transition does not satisfy condition *AE*, the *splitting* operation (Algorithm 1) is performed.

Algorithm 2.

```
Graph Function BuildingAtomicGraph (Graph $G$)
{
      Graph $G' := G$;
      Repeat
      {
        for each transition $(\alpha, t_f, \alpha')$ of $G'$ do
        {
            Class $\alpha'' := Pred(\alpha', t_f)$;
            // let F$''$ be the formula of $\alpha''$
            if $\neg(\alpha \subseteq \alpha'')$ then
            {
                  Partition Part :=Split $(\alpha, F'')$;
                  if $(Part = \emptyset)$ then
                     Eliminate transition $(\alpha, t_f, \alpha')$ from $G'$
                  else
                  {
                     Replace $(\alpha, Part, G')^{(*)}$;
                     Eliminate $\alpha$ from $G'$
                  }
            }
        }
      } until all classes become atomic;
      return $G'$;
}
```

$^{(*)}$The action *Replace*$(\alpha, Part, G')$ replaces, in G', the node α by all subclasses of the partition *Part*.

Algorithm 2 terminates because it operates on a finite CSCG where each state class has a finite number of possible splitting [3]. Note that the algorithm does not specify any strategy concerning the order in which transitions are considered during the refinement process. Experimental results have shown however that this order is relevant. In our implementation, we retained a simple strategy which seems to offer a good compromise between computing times and resulting graph sizes. In this strategy, all computed arcs are stored in a queue and treated in *first in first out* order.

6. Implementation results

We implemented our approach for building ASCGs in our experimental tool called *Real Time Studio*. The tool, written in *JAVA* and $C++$, integrates several functionalities related to enumerative analysis of the TPN model, including a CTL model checker and a minimizer under bisimulation [15] [17].

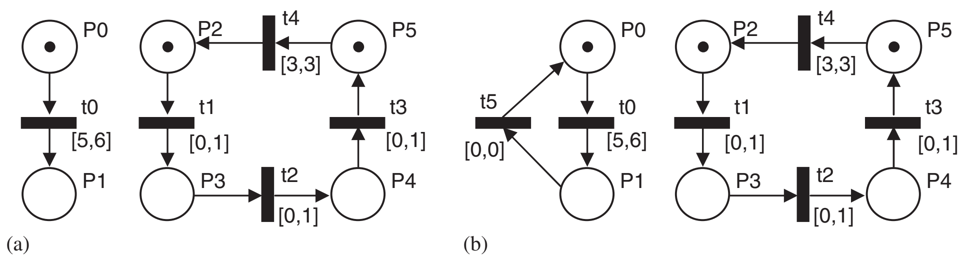
All results reported in this section have been obtained on a *three gigahertz Pentium-4* with *two gigabytes of RAM*.

We tested our approach on many TPN models, all of which have shown an important gain in performances when using our improvements. We report here the results obtained for some models considered by Berthomieu and Vernadat in [5] (Figs. 6 and 8), and Yoneda and Ryuba in [21] (Fig. 6).

Fig. 8 shows the TPN model components of the classical level crossing example: the train, the controller and the barrier models. The level crossing model is obtained by putting in parallel one copy of the controller model, *n* copies of the train model, and one copy of the barrier model. The model components are synchronized on transitions with the same names.

We also report results obtained for the TPN models in Fig. 7 which illustrate special cases where SSCGs are larger than their corresponding ASCGs. This situation generally happens when a TPN model has some transitions with unbounded

[15] Bisimilar *classes* must also share the same marking.



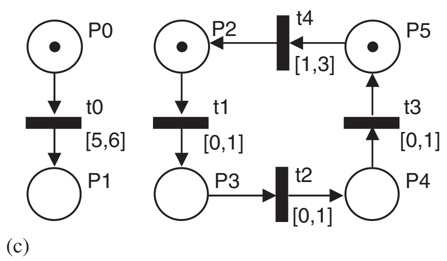


Fig. 6. Some TPN models used in our experiments.

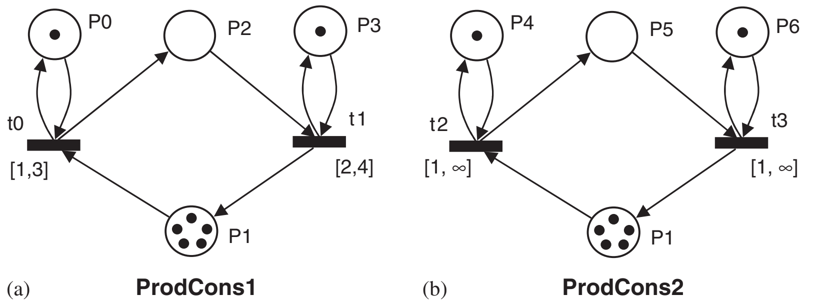


Fig. 7. Producer consumer model.

firing intervals, where lower bounds are greater than zero. The reason could be well understood since, in this special case, the relaxation operation induces a big fragmentation of state classes.

Note that we only compare our results to those of Berthomieu and Vernadat obtained using the tool *Tina-2.6.6*, which implements their approach. The results of Yoneda and Ryuba for the models given in Fig. 6 could be found in [5] where they are compared with those of Berthomieu and Vernadat.

Table 1 reports in each row the results obtained for the SSCG and its corresponding CSCG of the model given in the first column. The results are given in terms of graph size (nodes/arcs) followed by its computing time. The denotation $M_1 \parallel M_2$ is used in the first column to indicate the parallel composition of the TPN models M_1 with M_2 for the TPN models in Fig. 6. For the producer consumer models in Fig. 7, the parallel composition of $n-1$ copies of the model in Fig. 7b with one copy of the model in Fig. 7a while merging all places named P_1 in one single place, is denoted $Prod(n)$. For the level crossing Model in Fig. 8 with n trains crossing the level, we use the denotation $Trains(n)$. The results given in column two are those obtained for computing SSCGs using the tool *Tina*.[16] An *interrogations mark* indicates a situation where the computation has not completed after an hour or the execution has aborted due to a lack of memory. The last column gives the ratio of the results obtained for SSCGs to those obtained for CSCGs. It also allows to see the big gap in performances between the two constructions.

[16] SSCGs obtained using our tool are identical in sizes.

Table 1
SSCGs and CSCGs of some tested TPN models

TPNs	SSCG	CSCG	Ratio
Fig. 1	15/30	6/15	2.14
Time (ms)	0	0	–
Fig. 6a	21/29	12/17	1.72
Time (ms)	0	0	–
Fig. 6b	60/93	15/25	3.83
Time (ms)	10	0	> 10.00
Fig. 6c	39/63	12/17	3.52
Time (ms)	0	0	–
Fig. 6a‖6b	6410/15 759	509/1395	11.67
Time (ms)	391	10	39.10
Fig. 6a‖6c	3725/10 821	191/534	20.06
Time (ms)	250	0	> 250.00
Fig. 6b‖6c	9555/26 002	483/1707	16.24
Time (ms)	691	10	69.10
Fig. 6‖6b	200 087/547 822	1360/5048	116.71
Time (ms)	11 045	460	24.01
Fig. 7 Prod(2)	7963/42 566	165/896	47.62
Time (ms)	1833	10	183.30
Fig. 7 Prod(3)	122 191/1 111 887	1184/11 721	95.63
Time (ms)	106 994	510	209.79
Fig. 7 Prod(4)	659 377/7 987 583	4726/67 954	118.97
Time (ms)	551 610	8342	66.12
Fig. 7 Prod(5)	?	13 643/249 187	–
Time (ms)		57 502	–
Fig. 8 Trains(1)	11/14	10/13	1.09
Time (ms)	0	0	–
Fig. 8 Trains(2)	141/254	41/82	3.21
Time (ms)	10	0	> 10
Fig. 8 Trains(3)	5051/13 019	244/714	18.86
Time (ms)	500	10	50
Fig. 8 Trains(4)	351 271/1 193 376	1807/7091	173.59
Time (ms)	19 450	165	117.88

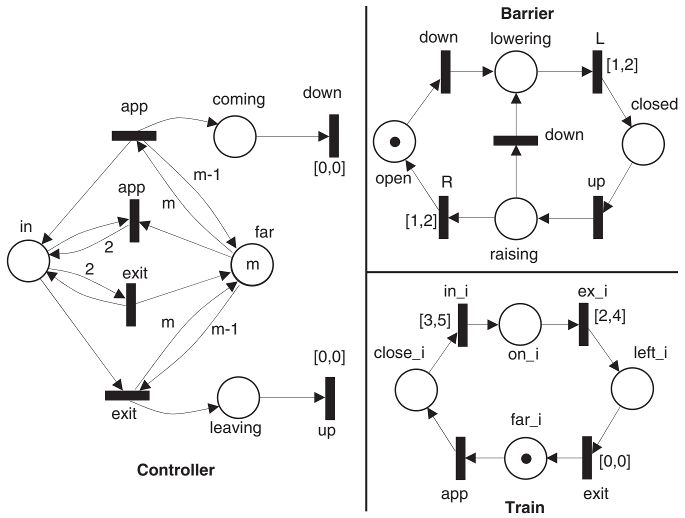


Fig. 8. The level crossing models.

Table 2
ASCGs of some tested models

TPNs	Tina	Ours	Optimal
Fig. 1	15/30	6/15	4/8
Time (ms)	0	0	0
Fig. 6a	36/61	27/49	26/47
Time (ms)	0	0	0
Fig. 6b	80/204	80/204	80/204
Time (ms)	0	0	0
Fig. 6c	61/162	46/135	46/135
Time (ms)	0	0	0
Fig. 6a‖6b	9349/36 709	8969/36 377	8599/34 591
Time (ms)	13 610	2784	48 169
Fig. 6a‖6c	4558/21 702	4195/20 531	3898/19 035
Time (ms)	4446	1091	19 035
Fig. 6b‖6c	9852/45 045	9630/44 786	9366/43 143
Time (ms)	16 904	3404	80 126
Fig. 6b‖6b	79 840/378 432	79 840/378 432	?
Time (ms)	912 910	70 911	
Fig. 7 Prod(2)	2615/27 348	2444/26 358	2334/25 046
Time (ms)	8422	1151	9414
Fig. 7 Prod(3)	?	31 197/485 960	28 319/430 875
Time (ms)		40 167	3 887 309
Fig. 7 Prod(4)	?	151 384/2 887 295	?
Time (ms)		358 060	
Fig. 7 Prod(5)	?	472 940/10 407 836	?
Time (ms)		1 993 170	
Fig. 8 (1 train)	12/16	11/15	11/15
Time (ms)	0	0	0
Fig. 8 (2 trains)	196/859	192/844	185/786
Time (ms)	40	10	30
Fig. 8 (3 trains)	6981/49 997	6966/49 802	6905/48 749
Time (ms)	9740	2113	60 878
Fig. 8 (4 trains)	?	356 930/3 447 548	?
Time (ms)		317 286	

Note that, during the construction of the CSCG, checking for inclusion has been performed sequentially. This operation could be improved significantly by using an appropriate way to order state classes.

Table 2 reports the results for ASCGs obtained using the tool *Tina* (column two) and using our approach (column three). Column four gives the optimal ASCG size obtained by minimization under bisimulation [17]. Times reported in this last column are minimization times only.
Note that we successfully tested the bisimilarity of all ASCGs we obtained with or without abstraction by inclusion.

An interesting feature which helps to explain the obtained results is illustrated in Fig. 9. [17] This feature relates to the computation pattern followed by the refinement procedure, depending on which state class space is refined (the SSCG or the CSCG). The feature has been noticed in all tested models, and appears to be independent of the implementation strategy of the refinement process. [18]

Among the reported models, Fig. 6b‖6b is where the feature is most apparent. As illustrated in Fig. 9, the refinement of the CSCG into an ASCG follows an almost linear like pattern, [19] whereas the refinement of the SSCG results in a graph which size starts first to grow up to a *peek size*, then decreases until the ASCG is obtained. We observed in general that the *peek size* increases as the ratio of the ASCG size to the CSGG size increases. In certain cases, the peek

[17] The results reported in Fig. 9 have been obtained using our implementation.

[18] We tested several implementation strategies of the refinement procedure with no major change in the observed refinement pattern.

[19] The size of the graph grows linearly in time during its construction.

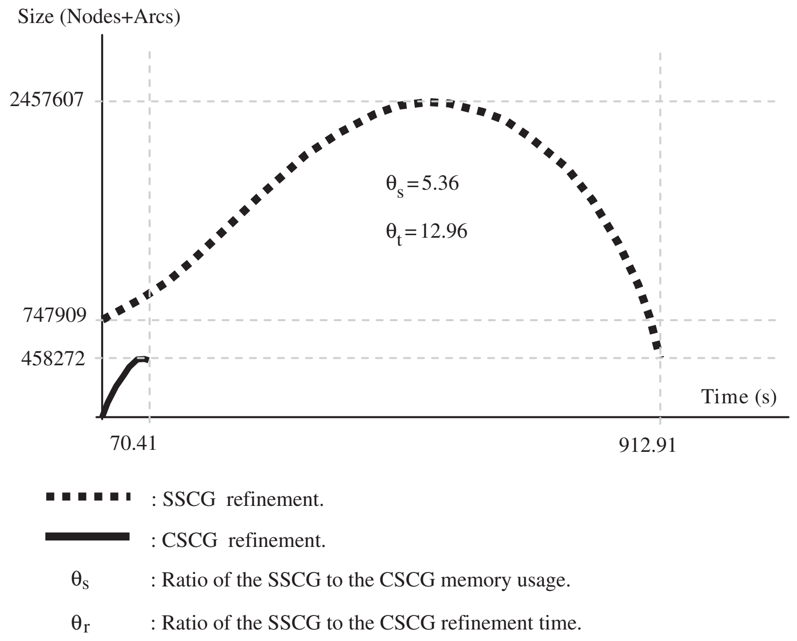


Fig. 9. Refinement patterns of the SSCG and the CSCG of the model Fig. 6b‖6b.

size grows out of control, leading to a state explosion. This is the main reason why some ASCGs have been computed successfully using CSCGs, but failed to compute, in reasonable times or due to a lack of memory, using SSCGs (ex: Fig. 8 (4 *trains*)).

Note also that for some models (ex: Fig. 6b‖6b, Fig. 7 *Prod*(n)) the SSCG may be much larger than the ASCG itself. This remark suggests that the state explosion problem may arise during the construction of the SSCG itself, before even starting its refinement. In this sense, using CSCGs helps to attenuate the state explosion problem.

7. Conclusion

This paper deals with the application of the *CTL** model checking technique to the *time Petri net* (TPN) model. Because of time density, this model has in general an infinite state space. To apply *CTL** model checking to the TPN model, its state space has to be contracted into a finite graph which preserves its *CTL** properties. In [5,21], the authors propose two different construction approaches of such a graph, called atomic state class graph (ASCG). Both approaches use a *partition refinement technique*, where an intermediate contraction of the TPN state space is first built then refined until an ASCG is produced.

To improve the ASCG construction, we propose to use an intermediate structure that is much more compact and much faster to compute than what is proposed in [5,12]. This structure, called contracted state class graph (CSCG), is a contraction of the intermediate structure proposed in [5], where states redundancy is reduced to a very low level. We also propose to compute each CSCG and ASCG node in $O(n^2)$ instead $O(n^3)$, n being the number of transitions enabled at the node.

These improvements have shown a significant impact on the ASCG construction procedure, in terms of computing times and memory usage. They allowed to compute some ASCGs, which failed to compute on the same computer configuration without their application.

In future works we want to investigate the possibility to contract further the CSCG and study its impact on the refinement procedure in more details.

References

[1] R. Alur, D. Dill, Automata for modeling real-time systems, 17éme ICALP, Lecture Notes in Computer Science, vol. 443, Springer, Berlin, 1990.

[2] G. Behrmann, J. Bengtsson, A. David, K.G. Larsen, P. Pettersson, W. Yi, UPPAAL implementation secrets, in: Proc. Seventh Internat. Symp. Formal Techniques in Real-Time and Fault-Tolerant Systems, 2002.

[3] B. Berthomieu, La méthode des classes d'états pour l'analyse des réseaux temporels—mise en oeuvre, extension à la multi-sensibilisation, in: Proc. Modélisation des Systèmes Réactifs, Toulouse, France, October 2001.

[4] B. Berthomieu, M. Menasche, An enumerative approach for analyzing time Petri nets, in: IFIP Congress, September 1983.

[5] B. Berthomieu, F. Vernadat, State class constructions for branching analysis of time Petri nets, Lecture Notes in Computer Science, vol. 2619, 2003.

[6] H. Boucheneb, G. Berthelot, Contraction of the ITCPN state space, Electronic Notes in Theoretical Computer Science, vol. 65, issue 6, June 2002.

[7] H. Boucheneb, J. Mullins, Analyse de réseaux de Petri temporels, Calculs des classes en $O(n^2)$ et des temps de chemin en $O(m \times n)$, Technique et Science Informatiques 22 (4) (2003).

[8] G. Bucci, E. Vicario, Compositional validation of time-critical systems using communicating time Petri nets, IEEE Trans. Software Eng. 21 (12) (1995).

[9] S. Christensen, L.M. Kristensen, T. Mailand, Condensed state spaces for timed Petri nets, 22nd Internat. Conf. Application and Theory of Petri nets and 2nd Internat. Conf. Application of Concurrency to System Design, Newcastle Upon Tyne, June 2001.

[10] C. Daws, S. Tripakis, Model checking of real-time reachability properties using abstractions, in: Tools and Algorithms for the Construction and Analysis of Systems, TACAS'98, Lecture Notes in Computer Science, vol. 1384, Lisbon (Portugal), 1998.

[11] M. Diaz, P. Senac, Time stream Petri nets a model for timed multimedia information, 15th Internat. Conf. Application and Theory of Petri Nets, Lecture Notes in Computer Science, vol. 815, Springer, Zaragoza (Spain), June 1994.

[12] T.A. Henzinger, P.-H. Ho, H. Wong-Toi, HyTech: a model checker for hybrid systems, Software Tools for Technology Transfer 1 (1997) 110–122.

[13] P.-A. Hsiung, F. Wang, A state-graph manipulator tool for real-time system specification and verification, in: Proc. 5th Internat. Conf. Real-Time Computing Systems and Applications, RTCSA'98, October 1998.

[15] P. Merlin, D.J. Farber, Recoverability of communication protocols, IEEE Trans. Comm. 24 (1976).

[16] W. Penczek, A. Polrola, Abstraction and partial order reductions for checking branching properties of time Petri nets, in: Proc. ICATPN, Lecture Notes in Computer Science, vol. 2075, 2001, pp. 323–342.

[17] W. Penczek, A. Polrola, Specification and model checking of temporal properties in time Petri nets and timed automata, in: Proc. ICATPN'04, 2004.

[18] J. Sifakis, Use of Petri nets for performance evaluation, in: H. Beilner, E. Gelenbe (Eds.), Measuring Modeling and Evaluating Computer Systems, North-Holland, Amsterdam, 1977.

[19] W.M.P. van der Aalst, Interval timed coloured Petri nets and their analysis, 14th Internat. Conf. Application and Theory of Petri Nets, Chicago, 1993.

[20] E. Vicaro, Static analysis and dynamic steering of time-dependent systems, IEEE Trans. Software Eng. 2 (8) (2001).

[21] T. Yoneda, H. Ryuba, CTL model checking of time Petri nets using geometric regions, IEICE Trans. Inf. System E99-D (3) (1998).

[22] S. Yovine, Méthodes et outils pour la vérification symbolique de systèmes temporisés, Thèse de Doctorat, Institut Nationale Polytechnique de Grenoble, France, May 1993.