A State Class Construction for Computing the Intersection of Time Petri Nets Languages

Éric Lubat, Silvano Dal Zilio, Didier Le Botlan, Yannick Pencolé, Audine Subias

▶ To cite this version:

Éric Lubat, Silvano Dal Zilio, Didier Le Botlan, Yannick Pencolé, Audine Subias. A State Class Construction for Computing the Intersection of Time Petri Nets Languages. 17th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS), Aug 2019, Amsterdam, Netherlands. 10.1007/978-3-030-29662-9_5 . hal-02263832

HAL Id: hal-02263832

https://hal.laas.fr/hal-02263832

Submitted on 5 Aug 2019

A State Class Construction for Computing the Intersection of Time Petri Nets Languages

Éric Lubat[1], Silvano Dal Zilio[1],
Didier Le Botlan[1], Yannick Pencolé[1], and Audine Subias[1]

LAAS-CNRS, Université de Toulouse, CNRS, INSA, Toulouse, France
name.surname@laas.fr

Abstract. We propose a new method for computing the language intersection of two Time Petri nets (TPN); that is the sequence of labels in timed traces common to the execution of two TPN. Our approach is based on a new product construction between nets and relies on the State Class construction, a widely used method for checking the behaviour of TPN. We prove that this new construct does not add additional expressive power, and yet that it can leads to very concise representation of the result. We have implemented our approach in a new tool, called Twina. We report on some experimental results obtained with this tool and show how to apply our approach on two interesting problems: first, to define an equivalent of the twin-plant diagnosability methods for TPN; then as a way to check timed properties without interfering with a system.

Keywords: Time Petri nets · Model Checking · State Classes · Realtime Systems Modeling and Verification.

1 Introduction

Formal languages, and the problem of efficiently checking intersection between languages, play an important role in formal verification. For instance, automata-theoretic approaches to model-checking often boils down to a language emptiness problem; that is finding whether there is a trace, in a system, that is also "in the negation of a property" [25]. Similarly, in the study of Discrete Event Systems [28], basic control-theoretic properties are often expressed in terms of language properties and language composition. We consider examples of these two problems at the end of this paper.

In this context, there is a large body of research where systems are expressed using Petri nets (PN). Indeed, PN are well-suited for modelling notions such as concurrency or causality in a very compact way; and they can be used for verification by building a Labeled Transition System out of them. Just as important, PN come equipped with a structural construct for *synchronous composition*, that coincides with language intersection when the set of labels of the nets are equal. Unfortunately, the situation is not as simple when we consider extensions of Petri nets that deal with time.

E. André and M. Stoelinga (Eds.): FORMATS 2019, LNCS 11750, pp. 1–17, 2019.
https://doi.org/10.1007/978-3-030-29662-9_5.

In this paper, we propose a new method for computing the language intersection of two *Time Petri nets* (TPN) [26,10]. This problem is quite complex and is hindered by two main problems. First, the state space associated with a TPN is typically infinite when we work with a dense time model; that is when time delays can be arbitrarily small. Therefore we need to work with an abstraction of their transition system. Second, there is no natural way to define the (structural) composition of two transitions that have non-trivial time constraints (meaning different from the interval $[0, \infty[$). These problems limit the possibility for compositional reasoning on TPN.

A solution to the first problem was proposed by Berthomieu and Menasche in [9], where they define a state space abstraction based on *state classes*. This approach is used in several model-checking tools, such as Romeo [22] and Tina [12] for instance. In the following, we propose a simple solution to overcome the second problem. Our approach is based on an extension of TPN with a dedicated product operator, called *Product TPN*, that can be viewed as an adaptation of Arnold-Nivat synchronization product [3] to the case of TPN. We show that it is possible to extend the state class construction to this new extension, which gives an efficient method for computing the intersection of two TPN when the nets are bounded.

Verification of Time Petri Nets. In the following, we consider TPN where transitions may have observable labels. In this context, an *execution* is the timed-event word obtained by recording the transitions that have been fired together with the delays between them. Our goal is to provide a method for symbolically computing the set of executions that are common to two labeled TPN. Without time, it is well-known that we can compute the language of a net from its marking graph. This gives a Labeled Transition System (LTS); an automaton that is finite as soon as the net is bounded. Likewise, we can compute the (language) intersection of two timed nets by computing the LTS of their *synchronous composition*, denoted $N_1 \| N_2$ thereafter. Actually, like in the untimed case, we are more interested by the *synchronous product* of two languages, rather than by their intersection.

The situation is quite different when we take time into account. Indeed, we may have fewer traces with a TPN than with the corresponding, "untimed" net (the one where timing constraints are deleted). This is because timing constraints may prevent a transition from firing, but never enable it. One solution to recover a finite abstraction of the state space is to use the *State Class Graph* (SCG) construction. Actually, SCG is an umbrella term for a family of different abstractions, each tailored to a different class of properties, or to a different extension of TPN. The first such construction, called *Linear State Class Graph* (LSCG) [9], is based on *firing domains*, that is the delays before a transition can fire. The LSCG preserves the set of reachable markings of a net as well as its language; which is exactly what is needed in our case. This is also the construction that we use in Sect. 3.

In the following we also mention the *Strong SCG* construction (SSCG) [11], based on *clock domains*, that is the duration for which a transition has been enabled. The SSCG preserves more information than the linear one. For example, we can infer from clocks when two transitions are enabled "at the same time", meaning we can handle priorities. The added expressiveness of the strong construction comes at a cost; the SSCG (for a given net) has always more classes than the corresponding LSCG, sometimes by a very large amount. (We give some examples of this in Sect. 6.) This is why we prefer to use the LSCG when possible.

Related Works and Review of Existing Methods. A motivation for our work is that we cannot rely on a synchronous product of TPN. Indeed, a major limitation with TPN is that there are no sensible way to define the composition of "non-trivial" transitions, and therefore no sensible way to define the synchronous composition of "non-composable" TPN; we say that *a transition is trivial* when it is associated to the time interval $[0, \infty[$ and that a net is *composable* when all its observable transitions are trivial. (We illustrate the problem at the end of Sect. 2). Likewise we cannot rely on the product of their SCG either. Indeed, the product of two SCG provides an over-approximation of the expected result, since it cannot trace time dependencies between events from different nets.

The situation is not the same with other "timed models". A notable example is *Timed Automata* [2], an extension of finite automata with variables, also called clocks, whose values progress synchronously as time elapses. Timed Automata (TA) can use boolean conditions on clocks to guard transitions and as local invariants on states. It is also possible to reset a clock when "firing" a transition. The classical product operation on finite automata can be trivially extended to TA: we only need to use the conjunction of guards, invariants and resets where needed. This provides a straightforward method for computing the (language) intersection of two TA, and also a trivial proof that the class of languages accepted by a TA are closed under intersection. Another related work is based on the definition of *Timed Regular Expressions* [4], that provides a timed analogue of Kleene Theorem for TA.

These results seem to promote Timed Automata as an algebraic model of choice for reasoning about timed words, and many works have studied the relation between TPN and TA. (On another note, we can remark that even a slight change in semantics may complicate the product construction; see for instance the case with signal-event languages [16].) For instance, Cassez and Roux [19] propose a structural encoding of TPN into TA that preserves the semantics in the sense of timed bisimulation, and therefore that preserves timed language acceptance. This encoding generates one automata and one clock for every transition in the TPN and can be extended in order to accommodate strict timing constraints; that is static time intervals that have a finite, open bound. Later, Bérard et al. [7,15] showed that TPN and TA are indeed equivalent with respect to language acceptance, but that TA are strictly more expressive in terms of weak timed bisimulation ($\approx$). These results are based on semantic encodings from TPN into TA and from TA into TPN that can be chained together to build an encoding from a TPN to an equivalent composable one. A similar result is also found in [27], which provides a structural encoding from a TPN, N, into a composable TPN that is of size linear with respect to N. But none of these encodings handle timing constraints that are bounded and right-open.

One of the main difference between TA and TPN is that, with TA, we can loose the ability to fire a transition just by waiting long enough (until some guards become false). The same behaviour can be observed with TPN when we add a notion of priorities. In particular, Berthomieu et al. [10,11] prove that (bounded) TPN with priorities are very close to TA, in the sense of $\approx$. They also define an extension of TPN [27] with *inhibitor arcs* between transitions (similar to priorities) and a dual notion of *permission arcs*. In this extension, called IPTPN, a net can always be transformed into a composable one. (We show an example of this construction in Sect. 5).

All these results can be used to define three different methods for computing the intersection of TPN. A first method is to use the structural translation from TPN to TA given in [19] and then to use the product construction on TA. This encoding is at the heart of the tool Romeo [22] and has been used to build a TCTL model-checker for TPN (which, incidentally, relies on the "product" of a net with observers for the formulas). Unfortunately, to the best of our knowledge, it is not possible to analyse the product of two nets with Romeo and therefore we have not been able to experiment with this method. Moreover, this approach is closer in spirit to the SSCG construction.

A second method is to use the (combination of) encodings defined in [15] to replace a TPN with an equivalent, composable one. Unfortunately, this construction relies on a semantic encoding that requires the computation of the entire symbolic state space of the net, and is only applicable on net that have closed timing constraints; meaning that we cannot use constraints of the form $[l, h[$ for example. While this method is not usable in practice, it could be used to prove expressiveness results. For example, it gives a proof that the set of TPN with closed timing constraints is closed under intersection; something we silently admitted until now.

A third method also relies on generating composable nets as a preprocessing step. In this case, the idea is to use the IPTPN of [27]. Like in the first method, the main drawback of this approach is that we need to use the strong SCG construction, which means that we could compute much more classes than with a method based on the LSCG. We describe the experimental results obtain with this method in Sect. 6.

Outline of the Paper and Contributions. In the next section we define the semantics of TPN and provide the technical background necessary for our work. Section 3 contains the semantics of Product TPN, while our two main results are given in Sect. 4 and 5, where we show that it is possible to extend the State Class Graph construction to the case of Product TPN and that this extension does not add additional expressiveness power. By construction, our method can be applied even when the TPN are not bounded and without any restrictions on the timing constraints.

We have implemented our approach in a new tool, called Twina [21] Before concluding, we report (Sect. 6) on some experimental results obtained with this tool. We also show some practical applications for our approach on two problems: first, to define an equivalent of the twin-plant diagnosability methods for TPN; then as a way to check timed properties without interfering with a system.

2 Time Petri Nets and other Technical Background

A *Time Petri Net* (TPN) is a net where each transition, t, is decorated with a (static) time interval $\mathbf{I}_s(t)$ that constrains the time at which it can fire. A transition is enabled when there are enough tokens in its input places. Once enabled, transition t can fire if it stays enabled for a duration θ that is in the interval $\mathbf{I}_s(t)$. In this case, t is said *time enabled.*

A TPN is a tuple $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, \mathbf{I}_s \rangle$ in which: $\langle P, T, \mathbf{Pre}, \mathbf{Post} \rangle$ is a net (with P and T the set of places and transitions); $\mathbf{Pre}$, $\mathbf{Post} : T \rightarrow P \rightarrow \mathbb{N}$ are the precondition and postcondition functions; $m_0 : P \rightarrow \mathbb{N}$ is the initial marking; and $\mathbf{I}_s : T \rightarrow \mathbb{I}$ is the *static interval function.* We use $\mathbb{I}$ for the set of all possible time intervals. To simplify our presentation, we only consider the case of closed intervals of the form $[l, h]$ or $[l, +\infty[$, but our results can be extended to the general case. TPN can be *k-safe*, which means the net has at most $k + 1$ reachable markings. We say that a TPN is *safe* when it is 1-safe.

We consider that transitions can be tagged using a countable set of labels, $\Sigma = \{a, b, \dots\}$. We also distinguish the special constant ϵ (not in Σ) for internal, silent transitions. In the following, we use a global labeling function $\mathcal{L}$ that associates a unique label in $\Sigma \cup \{\epsilon\}$ to every transition[1]. The alphabet of a net is the collection of labels (in Σ) associated to its transitions.

A Semantics for TPN Based on Firing Domains. A *marking* m of a net $\langle P, T, \mathbf{Pre}, \mathbf{Post} \rangle$ is defined as a function $m : P \rightarrow \mathbb{N}$ from places to natural numbers. A transition t in T is *enabled* at m if and only if $m \mathrel{\dot{\geqslant}} \mathbf{Pre}(t)$ (we use the pointwise comparison between functions) and $\mathcal{E}(m)$ denotes the set of transitions enabled at m.

A *state* of a TPN is a pair $s = (m, \varphi)$ in which m is a marking, and $\varphi : T \rightarrow \mathbb{I}$ is a mapping from transitions to time intervals, also called *firing domains.* Intuitively, if t is enabled at m, then $\varphi(t)$ contains the dates at which t can possibly fire in the future. For instance, when t is newly enabled, it is associated to its static time interval $\varphi(t) = \mathbf{I}_s(t)$. Likewise, a transition t can fire immediately only when 0 is in $\varphi(t)$ and it cannot remain enabled for more than its timespan, *i.e.* the maximal value in $\varphi(t)$.

For a given delay θ in $\mathbb{Q}_{\geq 0}$ and ι in $\mathbb{I}$, we denote $\iota - \theta$ the time interval ι shifted (to the left) by θ:, e.g. $[l, h] - \theta = [\max(0, l - \theta), \max(0, h - \theta)]$. By

[1] We may assume that there is a countable set of all possible transitions (identifiers) and that different nets have distinct transitions.

extension, we use $\varphi \mathbin{\dot{-}} \theta$ for the partial function that associates the transition t to the value $\varphi(t) - \theta$. This operation is useful to model the effect of time passage on the enabled transitions of a net.

The following definitions are quite standard, see for instance [7,10]. The semantics of a TPN is a (labeled) Kripke structure $\langle S, S_0, \rightarrow \rangle$ with only two possible kinds of actions: either $s \xrightarrow{a} s'$, meaning that the transition $t \in T$ is fired from s with $\mathcal{L}(t) = a$; or $s \xrightarrow{\theta} s'$, with $\theta \in \mathbb{Q}_{\geq 0}$, meaning that time θ elapses from s. A transition t can fire from the state (m, φ) if t is enabled at m and firable instantly. When we fire a transition t from state (m, φ), a transition k (with $k \neq t$) is said to be *persistent* if k is also enabled in the marking $m - \mathbf{Pre}(t)$, that is if $m - \mathbf{Pre}(t) \mathrel{\dot{\geqslant}} \mathbf{Pre}(k)$. The other transitions enabled after firing t are called *newly enabled.*

Definition 1 (Semantics). *The semantics of a TPN N, with N the net $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, \mathbf{I}_s \rangle$, is the Timed Transition System (TTS) $[\![N]\!] = \langle S, s_0, \rightarrow \rangle$ where S is the smallest set containing s_0 and closed by $\rightarrow$, where:*

- *$s_0 = (m_0, \varphi_0)$ is the initial state, with m_0 the initial marking and $\varphi_0(t) = \mathbf{I}_s(t)$ for every t in $\mathcal{E}(m_0)$;*
- *the state transition relation $\rightarrow \subseteq S \times (\Sigma \cup \{\epsilon\} \cup \mathbb{Q}_{\geq 0}) \times S$ is the relation such that for all state (m, φ) in S:*
  (i) if t is enabled at m, $\mathcal{L}(t) = a$ and $0 \in \varphi(t)$ then $(m, \varphi) \xrightarrow{a} (m', \varphi')$ where $m' = m - \mathbf{Pre}(t) + \mathbf{Post}(t)$ and φ' is a firing function such that $\varphi'(k) = \varphi(k)$ for any persistent transition and $\varphi'(k) = \mathbf{I}_s(k)$ elsewhere.
  (ii) if $\theta \mathrel{\dot{\leqslant}} \varphi$
  $\forall k \in Enabled(m), \theta \leq max\varphi(k)$ then $(m, \varphi) \xrightarrow{\theta} (m, \varphi \mathbin{\dot{-}} \theta)$.

Transitions in the case (i) above are called *discrete transitions*; those labelled with delays (case (ii)) are the *continuous*, or time elapsing, transitions. Like with nets, we say that the alphabet of a TTS is the set of labels, in Σ, associated to discrete actions. Using labels, we can define the product of two TTS by extending the classical definition for the product of finite automata.

Definition 2 (Product of TTS). *Assume $S_1 = \langle S_1, s_1^0, \rightarrow_1 \rangle$ and $S_2 = \langle S_2, s_1^0, \rightarrow_2 \rangle$ are two TTS with respective alphabets Σ_1 and Σ_2. The product of S_1 by S_2 is the TTS $S_1 \| S_2 = \langle (S_1 \times S_2), (s_1^0, s_2^0), \rightarrow \rangle$ such that $\rightarrow$ is the smallest relation obeying the following rules:*

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1' \qquad \alpha \in (\Sigma_1 \setminus \Sigma_2) \cup \{\epsilon\}}{(s_1, s_2) \xrightarrow{\alpha} (s_1', s_2)} \qquad \frac{s_2 \xrightarrow{\alpha}_2 s_2' \qquad \alpha \in (\Sigma_2 \setminus \Sigma_1) \cup \{\epsilon\}}{(s_1, s_2) \xrightarrow{\alpha} (s_1, s_2')}$$

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1' \qquad s_2 \xrightarrow{\alpha}_2 s_2' \qquad \alpha \neq \epsilon}{(s_1, s_2) \xrightarrow{\alpha} (s_1', s_2')}$$

Executions, Traces and Equivalences. An *execution* of a net N is a sequence in its semantics, $[\![N]\!]$, that starts from the initial state. It is a time-event word over the alphabet containing both labels (in $\Sigma \cup \{\epsilon\}$) and delays. Continuous transitions can always be grouped together, meaning that when $(m, \varphi) \xrightarrow{\theta} (m, \varphi')$ and $(m, \varphi') \xrightarrow{\theta'} (m, \varphi'')$ then necessarily $(m, \varphi) \xrightarrow{\theta+\theta'} (m, \varphi'')$ (and the firing domain φ' is uniquely defined from φ and θ). Based on this observation, we can always consider executions of the form $\sigma \overset{\text{def}}{=} \theta_0\, a_0\ \theta_1\, a_1\ \ldots$ where each discrete transition is preceded by a single time delay. By contrast, a *trace* is the untimed word obtained from an execution when we keep only the discrete actions. Then the language of a TPN is the set of all its (finite) traces.

By definition, the language of a TPN is prefix-closed; and it is regular when the net is bounded [9]. It is also the case [27] that the "intersection" of two nets N_1 and N_2—the traces obtained from (pairs of) executions common to the two nets—are exactly the traces in the TTS product $[\![N_1]\!] \parallel [\![N_2]\!]$. Our goal, in the next section, is to define a product operation, $N_1 \times N_2$, that is a *congruence*, meaning that $[\![N_1 \times N_2]\!]$ should be equivalent to $[\![N_1]\!] \parallel [\![N_2]\!]$.

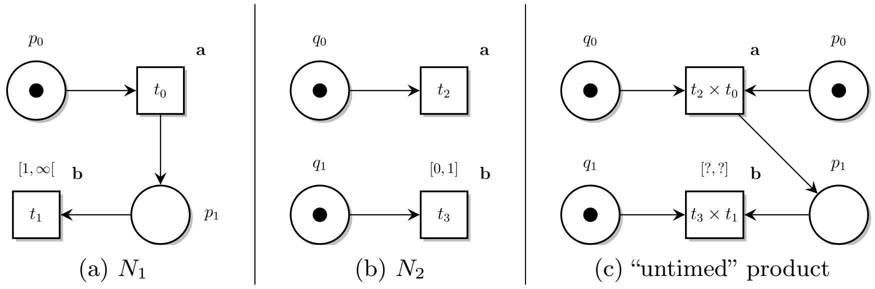
Language equivalence would be too coarse in this context. In this paper, we will instead prefer (a weak version of) timed bisimulation, which rely on a weak version of the transition relation $s \overset{\alpha}{\Rightarrow} s'$ (with α an action in $\Sigma \cup \{\epsilon\} \cup \mathbb{Q}_{\geq 0}$ and θ a delay in $\mathbb{Q}_{\geq 0}$) defined from the following set of rules:

$$\frac{}{s \overset{\epsilon}{\Rightarrow} s} \qquad \frac{s \overset{\epsilon}{\Rightarrow} s' \qquad s' \overset{\alpha}{\rightarrow} s'' \qquad s'' \overset{\epsilon}{\Rightarrow} s'''}{s \overset{\alpha}{\Rightarrow} s'''} \qquad \frac{s \overset{\theta}{\Rightarrow} s' \qquad s' \overset{\theta'}{\Rightarrow} s''}{s \xRightarrow{\theta+\theta'} s''}$$

Definition 3 (Behavioural Equivalence). *Assume $G_1 = \langle S_1, s_1^0, \rightarrow_1 \rangle$ and $G_2 = \langle S_2, s_2^0, \rightarrow_2 \rangle$ are two TTS. A binary relation $\mathcal{R}$ over $S_1 \times S_2$ is a weak timed bisimulation if and only if $s_1^0\ \mathcal{R}\ s_2^0$ and for all actions α and pair of states $(s_1, s_2) \in \mathcal{R}$ we have: (1) if $s_1 \overset{\alpha}{\Rightarrow} s_1'$ then there exists s_2' such that $s_2 \overset{\alpha}{\Rightarrow} s_2'$ and $s_1'\ \mathcal{R}\ s_2'$; and conversely (2) if $s_2 \overset{\alpha}{\Rightarrow} s_2'$ then there exists s_1' such that $s_1 \overset{\alpha}{\Rightarrow} s_1'$ and $s_1'\ \mathcal{R}\ s_2'$. In this case we say that G_1 and G_2 are timed bisimilar, denoted $G_1 \approx G_2$, and we use $\approx$ for the union of all timed bisimulations $\mathcal{R}$.*

Timed bisimulation is preserved by product [27], meaning that for all TTS G, G_1 and G_2 we have $G_1 \approx G_2$ implies $(G \| G_1) \approx (G \| G_2)$. In the following we say that two nets are bisimilar, denoted $N_1 \approx N_2$, when $[\![N_1]\!] \approx [\![N_2]\!]$.

Example. We give two examples of TPN with alphabet $\{a, b\}$ in Fig. 1. Executions for the net N_1 (left) include time-event words of the form $\theta_0\, a\, \theta_1\, b$ (and their prefix) provided that $\theta_1 \geq 1$. Executions for the net N_2 (middle) include time-event words of the form $\theta_2\, a\, \theta_3\, b$ and $\theta_3\, b\, \theta_2\, a$ (and their prefix) provided that $\theta_3 \leq 1$. If we consider executions that are in the product of both nets, we find all executions of the form $\theta_0\, a$, with the constraint $\theta_0 \leq 1$. We also have one execution of the form $\theta_0\, a\, \theta_1\, b$ provided that $\theta_0 + \theta_1 \leq 1$ and $\theta_1 \geq 1$. This corresponds to the case where event a fires exactly at date 0; any other case eventually leading to a *time deadlock* (a situation where time cannot progress). In the same figure (right), we display the "untimed" synchronous product $N_1 \| N_2$. It is clear that there are no possible choice of time constraint for transition $t_3 \times t_1$ that could lead to a net bisimilar to $[\![N_1]\!] \parallel [\![N_2]\!]$. This is a simple example of the "non-composability" of Time Petri nets.



(a) N_1 (b) N_2 (c) "untimed" product

Fig. 1: Two examples of TPN and their (failed, untimed) product.

3 Product TPN and their Semantics

We propose an extension of TPN with a *synchronous product* operation between TPN, $\times$, in the style of Arnold-Nivat synchronization of processes [3]. Our goal is to obtain a congruent composition operator, in the sense that $\llbracket N_1 \times N_2 \rrbracket \approx \llbracket N_1 \rrbracket \,\|\, \llbracket N_2 \rrbracket$. A *product TPN*, or PTPN, is a TPN $\langle P, T, \mathbf{Pre}, \mathbf{Post}, \mathbf{I}_s \rangle$ augmented with two projections, $\#_1$ and $\#_2$, such that the following properties hold:

- there are two sets $\#_1 P$ and $\#_2 P$ that partition the set of places P.
- there are two sets $\#_1 T$ and $\#_2 T$ that partition the set of transitions T.
- all the pre- and post-conditions of a transition in $\#_i T$ are places in $\#_i P$: if $t \in \#_i T$ and $\mathbf{Pre}(t)(p) > 0$ or $\mathbf{Post}(t)(p) > 0$ then $p \in \#_i P$.

Basically, this means that a PTPN N is the superposition of two distinct, non-interconnected components, that we call $\#_1 N$ and $\#_2 N$ for short.

Definition 4 (Product of TPN). *The product $N_1 \times N_2$ of two disjoint TPN N_1 and N_2 (such that $P_1 \cap P_2 = T_1 \cap T_2 = \emptyset$) is the PTPN obtained from the juxtaposition, preserving labels, of N_1 and N_2 with the two trivial projections $\#_i P = P_i$ and $\#_i T = T_i$ for all $i \in 1..2$.*

With our notations, a PTPN N is equivalent to the composition $(\#_1 N) \times (\#_2 N)$. In the following, we use the notation $\#_i m$ to denote the restriction of a marking m to the places in $\#_i P$ and similarly with $\#_i \varphi$ and the transitions in $\#_i T$. By convenience, $\#_i(m, \varphi)$ denotes the state $(\#_i m, \#_i \varphi)$ and we use $\#_i \Sigma$ for the alphabet of net $\#_i N$.

To ease the presentation, we limit the composition to only two components (instead of a sequence) and we do not define the equivalent of "synchronization vectors". As a result, we do not define the product over PTPN. This could be added, at the cost of more burdensome notations, but it is not needed in our applications (Sect. 6). This is also why we have the same limitations in our implementation [21].

Labels are not necessarily partitioned, so the same label can be shared between the two components of a product. We denote $\Sigma_{1,2}$ the set $(\#_1 \Sigma \cap \#_2 \Sigma)$

of labels occurring on "both sides" of a PTPN. We should also need the notation $\varSigma_1$ for the set $(\#_1 \varSigma \setminus \#_2 \varSigma) \cup \{\epsilon\}$ of labels that can occur in $\#_1$ concurrently with $\#_2$ (and similarly for $\varSigma_2$). The semantics for PTPN relies largely on the semantics of TPN but makes a particular use of labels.

Definition 5. *The semantics of a PTPN $\langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, \mathbf{I}_s \rangle$, with projections $\#_1$ and $\#_2$, is the TTS $[\![N]\!]_\times = \langle S, s_0, \longmapsto \rangle$ such that $s_0 = (m_0, \varphi_0)$ is the same initial state than in the TPN semantics $[\![N]\!]$, and $\longmapsto$ is the transition relation with actions in $\varSigma \cup \{\epsilon\} \cup \mathbb{Q}_{\geq 0}$ such that:*

$$\frac{\begin{array}{c}\alpha \in \mathbb{Q}_{\geq 0} \\ s \xrightarrow{\alpha} s' \in [\![N]\!]\end{array}}{s \overset{\alpha}{\longmapsto} s'} \qquad \frac{\begin{array}{c}t \in T \quad \mathcal{L}(t) = \alpha \notin \varSigma_{1,2} \\ s \xrightarrow{\alpha} s' \in [\![N]\!]\end{array}}{s \overset{\alpha}{\longmapsto} s'} \qquad \frac{\begin{array}{c}a = \mathcal{L}(t_1) = \mathcal{L}(t_2) \quad t_i \in \#_i T \\ \#_i s \xrightarrow{a} \#_i s' \in [\![\#_i N]\!] \quad i \in 1..2\end{array}}{s \overset{a}{\longmapsto} s'}$$

The only new case is for pairs of transitions, t_1 and t_2 , from different components but with the same label: $\mathcal{L}(t_1) = \mathcal{L}(t_2) = a$. This is the equivalent of a synchronization. Indeed the premisses entail that both t_1 and t_2 can fire immediately, and the effect is to fire both of them simultaneously. As a side effect, our choice of semantics entails that a transition on a "shared label" (in $\varSigma_{1,2}$) is blocked until we find a matching transition, with the same label, on the opposite component. This may introduce a new kind of time deadlock that has no direct equivalent in a TPN: when a transition has to fire urgently (hence time cannot progress) while there are no matching transition that is time-enabled.

It is the case that the reachable states, in $[\![N]\!]_\times$, are a subset of the states in $[\![N]\!]$. This is because we may forbid a synchronization on a shared label, but never create new opportunities to fire a transition. We also have a more precise result concerning the semantics of a PTPN and the product of its components.

Theorem 1. *The TTS $[\![N]\!]_\times$ is isomorph to the product $[\![\#_1 N]\!] \parallel [\![\#_2 N]\!]$.*

Proof. By induction on the shortest path from the initial state, s_0, to a reachable state s in $[\![N]\!]_\times$ and then a case analysis on the possible transitions from s. □

4 Construction of the State Class Graph for PTPN

We give a brief overview of the LSCG construction for a PTPN $N = \langle P, T, \mathbf{Pre}, \mathbf{Post}, m_0, \mathbf{I}_s \rangle$. In the following, we use the notation α_t^s and β_t^s for the left and right endpoints of interval $\mathbf{I}_s(t)$. For the sake of simplicity, we only consider inequalities that are non-strict (our definitions can be extended to the more general case) and assume that $\beta - \alpha = \infty$ when β is infinite.

A *state class* C is a pair (m, D), where m is a marking and D is a *domain*; a (finite) system of linear inequalities on the firing dates of transitions enabled at m. We will use variable x_i in D to represent the possible firing time of transition t_i. In the Linear SCG construction [8,9], we build an inductive set of classes C_σ, where $\sigma \in T^*$ is a sequence of discrete transitions firable from the initial state. Intuitively, the class $C_\sigma = (m, D)$ collects all the states reachable from the initial state by firing schedules of support sequence σ. For example, the initial class C_ϵ is (m_0, D_0) where D_0 is the domain defined by the static time constraints in φ_0, that is: $\alpha_i^s \leq x_i \leq \beta_i^s$ for all t_i in $\mathcal{E}(m_0)$.

The efficiency of the SCG construction relies on several factors: (1) First, we can restrict to domains D that are *difference systems*, that is a sequence of constraints of the form $\alpha_i \leq x_i \leq \beta_i$ and $x_i - x_j \leq \gamma_{i,j}$, where each variable in $(x_i)_{t_i \in \mathcal{E}(m_0)}$ corresponds to an enabled transition (and $i \neq j$). (2) Next, we can always put domains in *closure form*, meaning that each bounds α, β and γ are the tightest preserving the solution set of D. Hence we can encode D using a simple vector of values. This data structure, called *Difference Bound Matrix* (DBM), is unique to all the domains that have equal solution set. Hence testing class equivalence is decidable and efficient. (3) Finally, if $C_\sigma = (m, D)$ is defined and t is enabled at m, we can incrementally compute the coefficients of the DBM D', the domain obtained after firing t from C_σ, from the coefficients of D.

We only consider the new case where we simultaneously fire a pair of transitions (t_i, t_j) from a class (m, D). We assume that the resulting marking is m'. First, we need to check that both transitions can eventually fire. This is the case only if the condition $\gamma_{t,k} \geq 0$ is true for all $t \in \{i, j\}$ and k enabled at m (with $k \neq t$). In this case, the resulting domain D' can be obtained by following a short number of steps, namely:

1. add the constraints $x_i = x_j$ and $x_i \leq x_k$ to D, for all $k \notin \{i, j\}$ (since t_i, t_j must fire at the same date and before any other enabled transition);
2. introduce new variables x'_k for all transitions enabled in m', that will become the variables in D', and add the constraint $x'_k = x_k - x_i$ if t_k is persistent or $\alpha_k^s \leq x'_k \leq \beta_k^s$ if t_k is newly enabled;
3. eliminate all the variables from D relative to transitions in conflict with t_i, t_j and put the resulting system in normal form.

Except for step 1 above, with the constraint that $x_i = x_j$, this is exactly the procedure described in [8] for plain TPN. When both transitions (t_i, t_j) can fire, it is possible to completely eliminate all occurrences of the "unprimed" variables x_k in D' and the result is a DBM. Which is exactly what is needed in our case.

We can draw two useful observations from this result. First, we can follow the same procedure with any number of equality constraints, and still wind up with a DBM. Therefore it would be possible to fire more than two transitions simultaneously. Second, we have an indirect proof that forcing the synchronization of transitions is strictly less constraining than using priorities (because it is not possible to use the LSCG construction with priorities), something that was not obvious initially.

5 Expressiveness Results

It is not obvious that PTPN add any expressive power compared to TPN. On the one hand, the semantics of a PTPN N is quite close to the semantics of its components. In particular, $[\![N]\!]_\times = [\![N]\!]$ when there are no shared labels

($\varSigma_{1,2} = \emptyset$). Moreover, in a PTPN like in a TPN, it is not possible to lose the ability of firing a transition just by waiting long enough; a behaviour that distinguishes TPN from TA, or from TPN with priorities for instance. On the other hand, PTPN introduces new kind of time deadlocks which are affected by time delays (see our example at the end of Sect. 2). Next, we prove that the two models are equally expressive (up-to $\approx$) when all timing constraints are either infinite or closed on the right (in which case we say the net is *right-closed*).

Theorem 2. *Given a safe, right-closed PTPN N, we can build a safe, composable TPN N', whose size is linear with respect to N, such that $[\![N]\!]_\times \approx [\![N']\!]$.*

For the sake of brevity, we only sketch the proof. We rely on two auxiliary properties and on an encoding from TPN into *composable* net; meaning an equivalent net where all timing constraints have been "moved" to silent transitions. We find such result in [27, Def. 9], which provides a construction to build a composable net $\mathscr{T}_1(N)$ from every safe and right-closed TPN N. Our restrictions on N in Th. 2 come from this construction, as is our result on the size of N'.

Our first auxiliary property, (L1), compare the product of composable TPN with their synchronous product, namely: if N_1 and N_2 are composable TPN then $[\![N_1 \times N_2]\!]_\times \approx [\![N_1 \,\|N_2]\!]$. Property (L1) derives directly from the construction of the product $N_1 \,\|N_2$ of composable TPN. Indeed, with composable nets, the fusion of transitions sharing a common label are unaffected by continuous transitions. Hence they have the same behaviour in $N_1 \times N_2$ than in $N_1\|N_2$. (And this is the only place where the semantics of the two nets may diverge.)

Next, we use an equivalent of the congruence property for PTPN, (L2): given two pairs of TPN (N_1, N_2) and (M_1, M_2) such $N_1 \approx N_2$ and $M_1 \approx M_2$ we have that $[\![N_1 \times M_1]\!]_\times \approx [\![N_2 \times M_2]\!]_\times$. Property (L2) can be proved by defining a "candidate relation", $\mathcal{R}$, which contains the pair (s_0, s'_0) of initial states of $N_1 \times M_1$ and $N_2 \times M_2$; then proving that $\mathcal{R}$ is a weak timed bisimulation. A suitable choice for $\mathcal{R}$ is to take the smallest relation such that $(s_1 \uplus s'_1)\ \mathcal{R}\ (s_2 \uplus s'_2)$ whenever $s_1 \approx s_2$ and $s'_1 \approx s'_2$. Then the proof follows by simple case analysis.

Finally, we use construction $\mathscr{T}_1$ (above) to build composable TPN from the nets $\#_1 N$ and $\#_2 N$ and to define $N' \overset{\text{def}}{=} \mathscr{T}_1(\#_1 N)\|\mathscr{T}_1(\#_2 N)$. By property of $\mathscr{T}_1$ we have $\#_i N \approx \mathscr{T}_1(\#_i N)$ for all $i \in 1..2$. Hence by (L2) and (L1) we have $[\![N]\!]_\times = [\![\#_1 N \times \#_2 N]\!]_\times \approx [\![\mathscr{T}_1(\#_1 N) \times \mathscr{T}_1(\#_2 N)]\!]_\times \approx [\![\mathscr{T}_1(\#_1 N) \parallel \mathscr{T}_1(\#_2 N)]\!]$. The property follows by transitivity of $\approx$.

Our proof gives a constructive method to build a net N' with (at most) four extra transitions and places, compared to N, for each non-trivial labeled transition. We can use the SCG of N' to compute the language of N (and to compute the intersection of two nets when we choose $N = N_1 \times N_2$). Unfortunately this approach does not scale well. For example, the composition of the two nets given in Fig. 1 has 16 classes with this method instead of only 3 with our approach (and the intermediary TPN has 11 places and 7 transitions). Likewise, for the simple example in Fig. 4 we have a net with 25 places, 211 transitions and 1 389 classes instead of simply 3 classes with PTPN.

Another limitation of this approach are the restrictions imposed on the timing constraints of N. Indeed, to the best of our knowledge, there are no equivalent

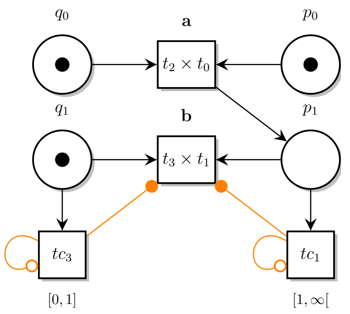


Fig. 2: Example of IPTPN

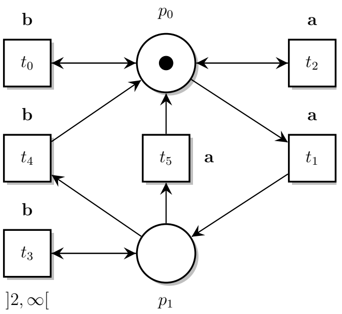


Fig. 3: TPN for the delay property

of construction $\mathscr{T}_1$ in the case of "right-open" transitions.

Composable Time Petri nets using IPTPN. Berthomieu et al. [27] define an extension of TPN with "inhibition and permission" that provides another method for building composable nets. With this extension, it is always possible to build a composable IPTPN from a TPN. For example, Fig. 2 displays the IPTPN corresponding to the "product" of the two nets in Fig. 1. In this construction, we create a silent, extra-transition tc_i for every non-trivial observable transition t_i. These transitions cannot fire (they self-inhibit themselves with an —○ arc) but "record the timing constraints" of the transition they are associated with. Then a permission arc (—●) is used to transfer these constraints on the (product of) labeled transitions.

Tina provides a SCG construction for IPTPN but, like with the addition of priorities, it is necessary to use the strong construction in this case. We use the encoding into IPTPN we just sketched above in our experiments.

6 Experimental Results and Possible Applications

We have implemented the state class construction for PTPN in a tool called Twina [21] that can generate the LSCG of both "plain" and product TPN. The tool and models mentioned here are available online at https://projects.laas.fr/twina/, with instructions on how to reproduce our results.

Performances Compared with IPTPN We compare the results obtained with PTPN and an encoding into IPTPN, which appears to be the best alternative among the three methods mentioned in Sect. 1. By default, Twina uses option -W, that computes the Linear SCG of a net. We also provide option -I to compute the LSCG for the product of two nets using the construction defined in Sect. 4. We use the same syntax for nets in Twina than in Tina [12]. In particular, our method can be used with nets that are not 1-safe and without any restriction on the timing constraints (so we accept right-open transitions). We also allow read- and inhibitor-arcs with the same semantics than in Tina.

		Twina (LSCG)		IPTPN (SSCG)		
Model	Exp.	States	Trans.	States	Trans.	Ratio
jdeds	plain	26	42	28	45	8%
jdeds	twin	544	1 144	706	1 432	30%
jdeds	obs	57	103	64	115	12%
train3	plain	3 101	7 762	5 051	13 027	63%
train3	twin	1 453 393	5 415 838	4 018 109	15 702 687	176%
train3	obs	6 202	16 614	10 102	27 801	63%
train4	plain	10 319	27 153	16 841	45 717	63%
train4	twin	20 954 198	79 768 434	57 567 538	229 935 082	175%
train4	obs	20 638	58 367	33 682	98 015	63%
plant	plain	2 696 558	7 359 339	4 628 698	12 870 710	72%
plant	twin	1 300	3 183	1 633	3 996	26%
plant	obs	5 715 293	15 639 336	9 790 043	27 215 355	71%
wodes	plain	2 554	6 080	5 363	13 047	110%
wodes	twin	55 402	155 586	151 352	426 928	173%
wodes	obs	5 767	13 506	14 663	34 508	154%
wodes232	plain	20 388	88 122	32 382	140 969	59%
wodes232	twin	39 588 981	304 246 211	339 165 870	2 552 685 724	757%
wodes232	obs	106 043	434 712	226 269	888 042	113%

Table 1: Comparing the PTPN and IPTPN methods

We compare the size of the LSCG with the results obtained using IPTPN and Tina in Table 1. The results are reported with the sizes of the SCG in number of classes and edges; we also give the ratio of classes saved between the LSCG and the SSCG. So a 100% ratio means twice as much states in the strong SCG.

We use different models for our benchmarks: *jdeds* is an example taken from [23] extended with time; *train* is a modified version of the train controller example in [13] with an additional transition that corresponds to a fault in the gate; *plant* is the model of a complex automated manufacturing system from [32]; *wodes* is the WODES diagnosis benchmark of Giua (see e.g. [18]) with added timed constraints. For each model, we give the result of three experiments: *plain* where we compute the SCG of the net, alone; *twin* where we compute the intersection between the TPN and a copy of itself with some transitions removed; and *obs* where we compute the intersection of the net with a copy of the TPN in Fig. 3. We explain the relevance of the last two constructions just afterwards.

We see that, in some of our examples, there is a large difference between the size of the LSCG and the size of the SSCG for the same example. This was one of our main reason for developing a specific tool. This is important since, on the extreme case, we can have a quadratic blow-up in the number of classes when analysing a twin product. (This is almost the case in example jdeds.) We also observe that, on model plant-twin, the size of the intersection may be much smaller than the size of one of the component alone; 1300 classes compared to 2 million. This is to be expected, since the intersection may have only one class. Nonethe-

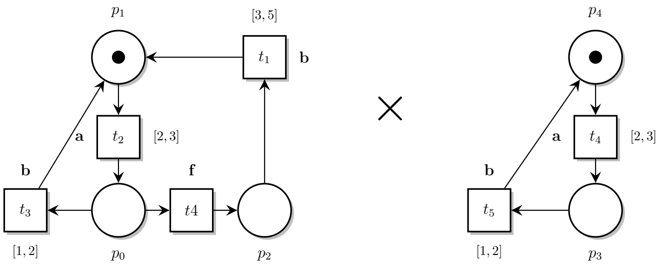


Fig. 4: Product of a TPN N_f (left) and its "twin" N_o (right) for the fault f.

less this emphasizes the need to have methods that can build the intersection on the fly, without computing a symbolic representation for each component first.

Diagnosability and the Twin Plant Method. One possible application of Twina—and our initial motivation for this work—is to check *fault diagnosability* [29] in systems modelled as TPN [5,17]. In this context, a system is described as a TPN with a distinguished unobservable event f that models a fault. (Any transition labelled with f is faulty.) Fault f is diagnosable if it is always possible to detect when a faulty transition has fired, in a finite amount of time, just by looking at the observable flow of events [31]. Under the assumptions that the system does not generate Zeno executions, and that any possible execution is not infinitely unobservable, one way to check diagnosability is to look for infinite *critical pairs* [20]. A critical pair consists of a couple of infinite executions of the TPN, one faulty the other one not, that have equal timed observations. Then fault f is diagnosable if no such pair exists. By using Twina, we aim at checking diagnosability by adapting the *twin-plant* method [24] to TPN. The idea is to make two copies of the same system, one with the fault, N_f, and the other without it, N_o, and to relabel all unobservable events to avoid collisions. Then checking for the existence of an infinite critical pair amounts to finding an infinite execution with f in the product $N_o \times N_f$.

We give an example of this construct in Fig. 4 where a and b are both observable. In system N_f, fault f is not diagnosable if we do not consider time, as we always observe b after an a in both faulty and non-faulty executions. Now considering the observation of time, f is diagnosable as the date of b is always discriminant. In the intersection of N_f and N_o, every execution where transition t_4 fires leads to a time deadlock. Indeed, in this case, we must wait at least 3 to fire transitions t_1 and at most 2 to fire t_5 (and both have label b).

The twin-plant construction is quite useful and we provide an option to directly build a twin TPN in our tool (option -twin). This is the construction we use in our experiments for Table 1. In this case, we can generate a LTS for the twin plant and check that every fault eventually leads to a deadlock in the product, meaning that the system is diagnosable. For instance using a LTL model-checker and a property such as $(\Diamond f) \Rightarrow (\Diamond \text{dead})$ [23]. We also provide a dedicated algorithm (option -diag) to check this property on-the-fly. When the system is not diagnosable, it allows us to find a counter-example before exploring the whole behaviour of the twin-plant.

Observer-based verification. Another application of our product construction is model checking TPN, in much the same way some "observer-based" verification techniques rely on the product of a system with an observer [1,30]. The idea is to express a property as the language of an observer, O, then check the property on the system N by looking at the behaviour of $N \times O$. A major advantage of this approach is that there is no risk to disrupt the system under observation, which is not always easy to prove with other methods.

We give an example of observer in Fig. 3. In this net, sequences of events a and b may occur in any order and at any date. On the other hand, the only way to fire t_3 is to "find" two successive occurrences of a and b with a delay (strictly) bigger than 2. Hence we can check if such behaviour is possible in a system, N, by checking whether t_3 can fire in $N \times O$. This is the problem we consider in the *obs* experiments of Table 1. We only consider one small example here. Nonetheless, the same approach could be used to check more complex timed properties. This will be the subject of future works.

7 Conclusion

We propose an extension of TPN with a product operation in the style of Arnold-Nivat. The semantics of our extension is quite straightforward. What is more surprising is that it is possible to adapt the LSCG construction to this case—which means that we do not need the equivalent of clocks or priorities—and that this extension does not add any expressive power. This is a rather promising result, complexity-wise, since it means that we can hope to adapt the same optimization techniques than with "plain" TPN, such as specific symmetry reduction techniques for instance [14].

We have several opportunities for extending our work. Obviously we can easily extend our product to a sequence of nets and add a notion of "synchronization vectors". This could lead to a more compositional framework for TPN, in the style of the BIP language [6]. Another promising application of our approach would be to extend classical results from the theory of supervisory control to the context of TPN. We already mentioned a possible application for diagnosability (which was the initial motivation for our work). A next step could be to study the "quotient" of two TPN language—the dual of the product—which can be used to reason about the controlability of a system and that is at the basis of many compositional verification methods, such as Assume-Guarantee for example.

Acknowledgments. The authors are grateful to Thomas Hujsa and Pierre-Emmanuel Hladik for their valuable comments. We also want to thank Bernard

Berthomieu, without whom none of this would have been possible; our work is a tribute to the versatility and the enduring qualities of the state class construction that he pioneered more than 30 years ago.

References

1. Abid, N., Dal Zilio, S., Le Botlan, D.: A formal framework to specify and verify real-time properties on critical systems. International Journal of Critical Computer-Based Systems (IJCCBS) **5**(1/2) (2014). `https://doi.org/10.1504/IJCCBS.2014.059593`
2. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126**(2) (1994). `https://doi.org/10.1016/0304-3975(94)90010-8`
3. Arnold, A.: Nivat's processes and their synchronization. Theor. Comput. Sci. **281**(1-2) (2002). `https://doi.org/10.1016/S0304-3975(02)00006-3`
4. Asarin, E., Caspi, P., Maler, O.: Timed regular expressions. Journal of the ACM **49**(2) (2002). `https://doi.org/10.1145/506147.506151`
5. Basile, F., Cabasino, M.P., Seatzu, C.: Diagnosability analysis of labeled time Petri net systems. IEEE Transactions on Automatic Control **62**(3) (2017). `https://doi.org/10.1109/TAC.2016.2588736`
6. Basu, A., Bozga, M., Sifakis, J.: Modeling Heterogeneous Real-Time Components in BIP. In: Software Engineering and Formal Methods (SEFM). IEEE (2006). `https://doi.org/10.1109/SEFM.2006.27`
7. Bérard, B., Cassez, F., Haddad, S., Lime, D., Roux, O.H.: Comparison of the expressiveness of timed automata and time Petri nets. In: Formal Modeling and Analysis of Timed Systems (FORMATS). LNCS, vol. 3829. Springer (2005)
8. Berthomieu, B., Diaz, M.: Modeling and verification of time dependent systems using time Petri nets. IEEE Trans. on Software Engineering **17**(3) (1991). `https://doi.org/10.1109/32.75415`
9. Berthomieu, B., Menasche, M.: An enumerative approach for analyzing time Petri nets. In: Proceedings IFIP (1983)
10. Berthomieu, B., Peres, F., Vernadat, F.: Bridging the gap between timed automata and bounded time Petri nets. In: Formal Modeling and Analysis of Timed Systems (FORMATS). LNCS, vol. 4202. Springer (2006). `https://doi.org/10.1007/11867340_7`
11. Berthomieu, B., Peres, F., Vernadat, F.: Model checking bounded prioritized time Petri nets. In: Automated Technology for Verification and Analysis (ATVA). LNCS, vol. 4762. Springer (2007). `https://doi.org/10.1007/978-3-540-75596-8_37`
12. Berthomieu, B., Ribet, P.O., Vernadat, F.: The tool TINA–construction of abstract state spaces for Petri nets and time Petri nets. International Journal of Production Research **42**(14) (2004)
13. Berthomieu, B., Vernadat, F.: State class constructions for branching analysis of Time Petri Nets. In: TACAS. LNCS, vol. 2619. Springer (2003). `https://doi.org/10.1007/3-540-36577-X_33`
14. Bourdil, P.A., Berthomieu, B., Dal Zilio, S., Vernadat, F.: Symmetry reduction for time Petri net state classes. Science of Computer Programming **132**(2) (2016). `https://doi.org/10.1016/j.scico.2016.08.008`
15. Bérard, B., Cassez, F., Haddad, S., Lime, D., Roux, O.H.: When are timed automata weakly timed bisimilar to time Petri nets? Theoretical Computer Science **403**(2-3) (2008). `https://doi.org/10.1016/j.tcs.2008.03.030`

16. Bérard, B., Gastin, P., Petit, A.: Intersection of Regular Signal-Event (Timed) Languages. In: Formal Modeling and Analysis of Timed Systems (FORMATS). LNCS, Springer (2006). https://doi.org/10.1007/11867340_5
17. Cabasino, M.P., Giua, A., Lafortune, S., Seatzu, C.: A new approach for diagnosability analysis of petri nets using verifier nets. IEEE Trans. Automat. Contr. **57**(12) (2012). https://doi.org/10.1109/TAC.2012.2200372
18. Cabasino, M.P., Giua, A., Seatzu, C.: Discrete event diagnosis using Petri nets. In: ICINCO-ICSO (2009)
19. Cassez, F., Roux, O.H.: Structural translation from time Petri nets to timed automata. Journal of Systems and Software **79**(10) (2006). https://doi.org/10.1016/j.jss.2005.12.021
20. Cimatti, A., Pecheur, C., Cavada, R.: Formal verification of diagnosability via symbolic model checking. In: IJCAI (2003)
21. Dal Zilio, S.: TWINA: A realtime model-checker for analyzing Twin-TPN. https://projects.laas.fr/twina/ (2019)
22. Gardey, G., Lime, D., Magnin, M., Roux, O.H.: Romeo: a tool for analyzing time petri nets. In: Computer Aided Verification (CAV). Springer (2005). https://doi.org/10.1007/11513988_41
23. Gougam, H.E., Pencolé, Y., Subias, A.: Diagnosability analysis of patterns on bounded labeled prioritized Petri nets. Discrete Event Dynamic Systems **27**(1) (2017). https://doi.org/10.1007/s10626-016-0234-5
24. Jiang, S., Huang, Z., Chandra, V., Kumar, R.: A polynomial algorithm for testing diagnosability of discrete-event systems. IEEE Transactions on Automatic Control **46**(8) (2001). https://doi.org/10.1109/9.940942
25. Kupferman, O., Vardi, M.Y., Wolper, P.: An automata-theoretic approach to branching-time model checking. Journal of the ACM (JACM) **47**(2) (2000). https://doi.org/10.1145/333979.333987
26. Merlin, P.M.: A study of the recoverability of computing systems. Ph.D. thesis, Department of Information and Computer Science, University of California (1974)
27. Peres, F., Berthomieu, B., Vernadat, F.: On the composition of time Petri nets. Discrete Event Dynamic Systems **21**(3) (2011). https://doi.org/10.1007/s10626-011-0102-2
28. Ramadge, P.J., Wonham, W.M.: The control of discrete event systems. Proceedings of the IEEE **77**(1) (1989)
29. Sampath, M., Sengupta, R., Lafortune, S., Sinnamohideen, K., Teneketzis, D.: Diagnosability of discrete-event systems. IEEE Transactions on automatic control **40**(9) (1995)
30. Toussaint, J., Simonot-Lion, F., Thomesse, J.P.: Time constraint verification methods based on time Petri nets. In: Workshop on Future Trends of Distributed Computing Systems. IEEE (1997). https://doi.org/10.1109/FTDCS.1997.644736
31. Tripakis, S.: Fault diagnosis for timed automata. In: Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT) (2002). https://doi.org/10.1007/3-540-45739-9_14
32. Wang, X., Mahulea, C., Silva, M.: Diagnosis of time Petri nets using fault diagnosis graph. IEEE Transactions on Automatic Control **60**(9) (2015). https://doi.org/10.1007/978-3-642-15297-9_12