Electronic Notes in Theoretical Computer Science 18 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume18.html 21 pages

Efficient State Space Search for Time Petri Nets

Johan Lilius

Åbo Akademi University
Department of Computer Science and
Turku Centre for Computer Science
FIN-20520 Åbo
Email: Johan.Lilius@abo.fi

Abstract

Partial-order reduction methods provide a number of well studied methods that have been succesfully applied to the *state-space explosion problem* that arises when analysing state based models of concurrent and reactive systems.

The techniques have mainly been studied within the context of *untimed* systems, in the context of *real-time* systems little progress has been made. The main problem seems to be the global nature of time, that makes all clocks in the system dependent on each other. Typically this is manifested in the semantics of real-time models in which the ordering of events is implicitely stored in the state of the system.

In this work we propose a new semantics for time Petri nets that, does not store the firing order into the timing constraints. This allows us to lift the notion of independence from untimed Petri nets, and it becomes possible to directly apply the theory of partial-order reductions to time Petri nets. We show that our semantics is finite and that it preserves reachable markings.

As a second contribution we propose to exploit the branching prefix originally introduced by McMillan in the state-space search of time Petri nets. To make sure that the state-space search algorithm terminates, one needs a list of states that have already been seen during the search. This list is searched each time a new state is generated. We propose to only store states after *cutoff events*. The cutoff events in a branching prefix describe events after which the system will return to a state in which it has previously been. Finally we also show how to extract *persistent sets* from the branching prefix.

1 Introduction

The design of concurrent and reactive systems is difficult. Therefore several formal approaches have been developed to the help the engineer in the automatic verification of his designs. Many such approaches are based on the use of state-spaces. In these approaches one models the system in some mathematically founded language and the mechanically calculates all the possible states of the system. Such methods however suffer from the *state-space explosion problem*, an instance of "combinatorial explosion". To alleviate this problem several techniques have been developed, among them reduction techniques that exploit the independence of events (cf. [8] for a good overview). The intuition behind these techniques is that if two events can be executed in any order such that we always end up in the same state then they are independent (cf. Fig. 1(a), aka. the "diamond property"). If these independent events do not affect the property we are interested in, then it does not matter in which order we execute them. The reduction is obtained by throwing one of these interleavings away, ie. executing a subset of the enabled events, called a *persistent set*. Methods based on the independence of events are often called *partial-order reduction methods*.

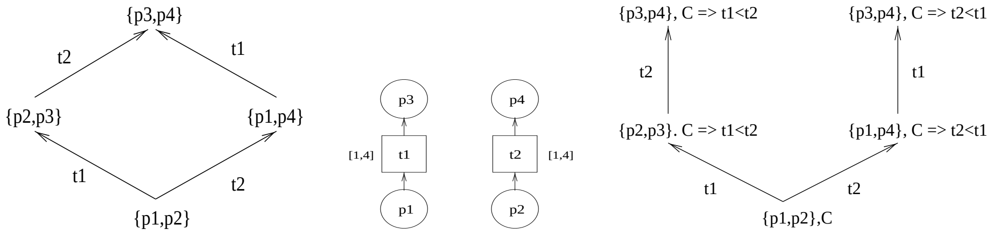
Partial-order reduction methods have been successfully applied to untimed systems, but for real-time systems less progress has been made. The main problem seems to be the global nature of time, that makes all clocks in the system dependent on each other. In particular the extension of partial-order methods to time Petri nets is hampered by two difficulties:

(i) The standard semantics for time Petri nets implicitly stores the firing order of transitions in the timing constraints. This means that the state space of the time Petri net will form a tree (cf. Fig. 1(c), which is the state space of the net in Fig. 1(b)). The correctness of partial-order methods is based on the *diamond property:* if two actions are independent the order of their execution does not matter, so we can pick on order. If the state space is a tree, this will not be the case anymore (cf. Fig. 1)

(ii) The firing time of *synchronisation transitions*, ie. transitions with more than one immediate predecessor requires the calculation of the firing times of these predecessors. For this we need absolute constraints which lead to an infinite state space.

In this work we propose a semantics for time Petri nets that allows us to lift the notion of independence from untimed Petri nets. Thus it becomes possible to directly apply the theory of partial-order reductions to time Petri nets. We show that the semantics is finite and that it preserves reachable markings.

As a second contribution we propose to exploit the branching prefix originally introduced by McMillan [11] in the state-space search of time Petri nets. The state-space is typically traversed with a standard depth-first traversal algorithm, that includes a list of states that have already been seen during the traversal (cf. Fig. 4). The test is needed so that we can guarantee termination. It is in this test that the state-space explosion causes its problem. To find out whether we have already seen a state we have to search the set of all previously seen states. In a timed system this set will be much larger,



(a) The "diamond" property (b) Time Petri net (c) A reachability tree

Fig. 1. Semantics of timed systems

because the systems typically exhibit the same untimed state several times with different timing constraints. As a consequence also the same persistent set will be recalculated several times during the analysis. To alleviate these two problems we propose two novel solutions:

- We use the branching prefix to calculate a set of *looping points*, ie. transitions after which the untimed system will return to a state in which it has previously been. In this way we only need to search the set of looping points.
- We also show how to extract persistent sets from the branching prefix. In this way the persistent sets are calculated once at the beginning of the analysis.

The last contribution is actually of independent interest: it establishes a connection between the partial-order approaches based on explicit state representation [8] and partial-order approaches based on the implicit branching-prefix representation of states [7].

2 Time Petri Nets

Time Petri nets are a simple yet powerful formalism for modeling concurrent systems with time constraints. In time Petri nets, transitions are labeled with time intervals. There is an upper and a lower limit for the time a transition can be enabled before firing, which makes it possible to model imperfect timing where the exact durations of events are not known. The following section recalls the basic definitions of time Petri nets, describes a method for enumerating the reachable states, and gives a definition for independence of transitions in a time Petri net.

2.1 Definitions

Definition 2.1 A *time Petri net* is a five-tuple $\mathsf{TPN} = (\mathsf{P}, \mathsf{T}, \mathsf{F}, \mathsf{SI}, \mathsf{m}_0)$, where P is the set of *places*, T is the set of *transitions*, $\mathsf{P} \cap \mathsf{T} = \emptyset$, $\mathsf{F} \subseteq (\mathsf{P} \times \mathsf{T}) \cup$

$(T \times P)$ is the *flow relation*, $SI : T \longrightarrow \mathbb{N} \cup \{\infty\} \times \mathbb{N} \cup \{\infty\}$ is a function called *static interval*, and $m_0 \subseteq P$ is the *initial marking* of the time Petri net. The tuple (P, T, F, m_0) is the *underlying net*. The boundaries of the static interval associated with a transition t are called *earliest firing time* t^ϵ and *latest firing time* t^λ respectively. The *preset* of $x \in P \cup T$ is ${}^\bullet x = \{y \mid yFx\}$ and *postset* is $x^\bullet = \{y \mid xFy\}$.

Time Petri nets were introduced in [12]. The above definition defines time Petri nets as elementary net systems enriched with the static intervals on transitions. A transition t of a time Petri net is *enabled* at marking m iff ${}^\bullet t \subseteq m$. The set of all enabled transitions at marking m is denoted by $\mathtt{enabled}(m)$. In time Petri nets, a marking is not sufficient information to describe a complete state of the system. The state must also include timing information. This is given as a clock function that, for each enabled transition, gives the amount of time that has passed since it has become enabled.

Definition 2.2 A *state* of a time Petri net $TPN = (P, T, F, SI, m_0)$ is a pair $S = (m, c)$, where m is a marking of TPN, and $c : T \longrightarrow \mathbb{R}$ is called the *clock function*. The *initial state* of TPN is $s_0 = (m_0, c_0)$, where $c_0(t) = 0$.

For the firing of a transition to be possible at a certain time, four conditions must be satisfied.

Definition 2.3 A transition t may fire from state $s = (m, c)$ after delay $\delta \in \mathbb{R}$ denoted $\mathtt{fireable}(s, (\delta, t))$ iff

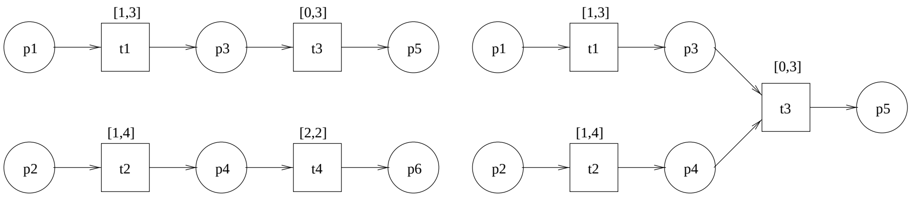
$$t \in \mathtt{enabled}(m), \tag{1}$$
$$(m \setminus {}^\bullet t) \cap t^\bullet = \emptyset, \tag{2}$$
$$t^\epsilon \leq c(t) + \delta, \text{ and} \tag{3}$$
$$\forall t' \in \mathtt{enabled}(m) : c(t') + \delta \leq t'^\lambda. \tag{4}$$

A transition satisfying Equations 1,3 and 4 but not Eqn. 2 is said to be *in contact*. We shall henceforth assume that all our nets satisfy Eqn. 2, ie. are *contact-free*. The set of all transitions that may fire from state s is denoted by $\mathtt{fireable}(s)$. From the definition we see directly that $\mathtt{fireable}((m, c)) \subseteq \mathtt{enabled}(m)$, ie. not all enabled transitions can necessarily fire because of timing constraints. The new marking after a firing is calculated as follows:

Definition 2.4 A transition t *fires after time* δ from state $s = (m, c)$ giving a new state, $s' = (m', c')$, as follows:

$$m' = m \setminus {}^\bullet t \cup t^\bullet, \tag{5}$$
$$c'(t) = \begin{cases} 0 & \text{if } t \in \mathtt{enabled}(m') - \mathtt{enabled}(m), \\ c(t) + \delta & \text{else .} \end{cases} \tag{6}$$

This is denoted by $s' = \mathtt{fire}(s, (t, \delta))$.



(a) The net N_1 (b) The net N_2

Fig. 2. Two example time Petri nets

The behavior of a time Petri net is described in terms of a firing schedule.

Definition 2.5 A *firing schedule* of a time Petri net is a finite or infinite sequence of pairs of transitions and time values $\sigma = \sigma_1, \sigma_2, \sigma_3, \ldots$ with $\sigma_i = (\delta_i, t_i)$, where t_i are transitions and $\delta_i \in \mathbb{R}$ are their *firing delays*. The firing schedule σ is *fireable* from the initial state s_0 if there exist states $s_1, s_2, s_3, \ldots$ such that:

$$s_i = \mathsf{fire}(s_{i-1}, \sigma_i) \;\; i > 0.$$

Given a finite firing schedule σ define $\mathsf{time}_i(\sigma) = \Sigma_{k=0}^{i} \delta_k$.

To exclude from the class of nets under consideration those nets with infinite firing schedules that consume no time, or only a bounded amount of time, we shall require that $\sigma : \Sigma_{k=0}^{\infty} \delta_k \to \infty$.

2.2 Enumerating the reachable states

From the above discussion it is clear that since we consider time to be a continuous quantity the set of states of a time Petri net is infinite. In this section we will show that this infinite state space can be partitioned into equivalence classes that will consist of markings and sets of inequations that constrain the possible occurrence times of the enabled transitions. The idea of constructing equivalence classes of markings was originally introduced in [4] and later refined in [3]. An alternative approach has been proposed in the context of a model-checking algorithm in [14]. Here we shall refine this approach so that it becomes possible to apply partial order reduction techniques that have been developed for untimed systems, directly to timed systems.

Our idea is based on the following observation from [1]: The possible occurrence time of of a transition is fully determined by the occurrence times of its predecessors. Transitions that are causally independent do not affect the occurrence times of each other.

Consider the net N_1 in figure 2(a). At the initial marking $\{p_1, p_2\}$ it is possible to fire both t_1 and t_2. Assume that the system is started at time $\underline{0}$. Then the occurrence time of t_1 is constrained by $\underline{0} + 1 \leq t_1 \leq \underline{0} + 3$, or equivalently $1 \leq t_1 - \underline{0} \leq 3$. Analogously for t_2 we obtain $1 \leq t_2 - \underline{0} \leq 4$. Now suppose that we want to fire t_1. Then it must be the case that t_1 may occur earlier than t_2, ie. $t_1 \leq t_2$. This means that the constraint $t_1 \leq t_2$ should be *consistent* with the other constraints on t_1 and t_2. A set of constraints is consistent iff it has a solution. Let us now fire t_1. The new marking is obtained as usual. The set of constraints now contains a new constraint for the newly enabled transition t_3: $t_1 + 0 \leq t_3 \leq t_1 + 3$, or $0 \leq t_3 - t_1 \leq 3$. Since the transition t_1 now is disabled, we do not need it in the constraint set anymore. We can delete it by algebraic manipulation, and obtain $1 \leq t_3 - \underline{0} \leq 6$, from which we can deduce a constraint $-3 \leq t_3 - t_2 \leq 5$ describing the relation between the possible occurrence times of t_2 and t_3. Not that contrary to [14] we do not add the constraint $t_1 \leq t_2$ to the set of constraints describing the marking $\{p_2, p_3\}$. Thus the state class does not now contain state information. This is motivated by our desire to make the "diamond property" (cf. Fig. 1(a)) hold for the timed state space. If we would add the constraint, then the state space would have the form of a tree (cf. Fig. 1(c)). The disadvantage of this is that the state class now contains solutions that are infeasible. In particular the constrains still allow the solution $t_2 < t_1 < t_3$, which at the marking $\{p_2, p_3\}$ does not make any sense. But fortunately when we are interested only in reachability of markings we can show that this does not matter for the correctness of our algorithm. At this point we can fire t_2, which gives us the new constraint (after deletion of t_2) $3 \leq t_4 - \underline{0} \leq 6$. It is an easy calculation to show that firing the transitions in the order t_2, t_1 leads to the same state class. We thus have the "diamond property".

Another problem however still remains: in the net N_2 of figure 2(b) the firing time of t_3 is determined by the "latest" of the transitions t_1 and t_2. We call a transition with more than 1 immediate predecessors a *synchronising transition*. As was shown in [1], the firing time of a synchronisation transition is determined by the maximum of the firing times of transitions in the set $^{\bullet\bullet}t$. This means that we have to find the maximum of the lower and upper bounds of the occurrence times of transitions in $^{\bullet\bullet}t$, to determine bounds on the occurrence time of t. It is for this reason that we have included the global bounds. This also means that we cannot delete a transition $t' \in {}^{\bullet\bullet}t$ from the set I, before all transitions in $^{\bullet\bullet}t$ have been fired, so that the bounds can be determined.

The above discussion can now be formalized as follows.

A *state class* is a pair (m, I), where $m \subseteq P$, and I is a set of constraints over $T \cup \{\underline{0}\}$. A state class describes the constraints on the possible firing times of the enabled transitions in a marking. The constraints in I are of the form $x - y \leq c$ and we shall require that the set is transitively closed. Such constraints can be represented as matrices, where each matrix entry $M(x, y) = c$ represents an inequation $x - y \leq c$. Using this representation the transitive closure of the constraints can be defined by applying the Floyd-Warshall shortest-path algorithm [15], which defines a normal form for the constraints. A set of constraints I is *consistent* iff it has a solution, otherwise it is *inconsistent*.

To be able to refer to the absolute occurrence times of transitions we introduced an auxiliary transition $\underline{0}$. $\underline{0}$ occurs before any other transition in the net and thus acts as a kind of origo. Constraints $x-\underline{0} \leq c$ express absolute timing constraints from this point. It would then be possible to do without the relative clock constraints $x - y \leq c$ when generating the state space, because these can be deduced. In particular this state space is infinite, because after each iteration of the system the occurrence times of the transitions increase. As we shall see below the relative clock constraints allow us to define a finite state space.

The initial state class is given by:

$$I_0 = \{t - \underline{0} \geq t^{\epsilon} \mid t \in \mathtt{enabled}(m_0)\} \cup \{t - \underline{0} \leq t^{\lambda} \mid t \in \mathtt{enabled}(m_0)\}$$

We denote with the variable t the occurrence time of transition t. A transition t_f is *fireable* in (m, I) iff $t_f \in \mathtt{enabled}(m)$, and we can fire it earlier than the other enabled transitions, ie. $I \cup \{t_f \leq t \mid t \in \mathtt{enabled}(m)\}$ is a consistent set of constraints, this is written $t_f \in \mathtt{fireable}(m, I)$.

Definition 2.6 Given a state class (m_i, I_i) we can fire the transition t_f if $t_f \in \mathtt{fireable}(m_i, I_i)$. The successor state class (m_{i+1}, I_{i+1}) is given by:

$$m_{i+1} = (m_i - {}^{\bullet}t_f) \cup t_f{}^{\bullet} \tag{7}$$

$$I_i' = I_i \cup \{\mathtt{lower}({}^{\bullet\bullet}t, I_i) + t^{\epsilon} \leq t - \underline{0} \leq \mathtt{upper}({}^{\bullet\bullet}t, I_i) + t^{\lambda} \tag{8}$$

$$\mid t \in \mathtt{enabled}(m_{i+1}) - \mathtt{enabled}(m_i)\} \tag{9}$$

$$I_{i+1} = \mathtt{delete}(I_i', (\mathtt{enabled}(m_i) - \mathtt{enabled}(m_{i+1})) \cup {}^{\bullet\bullet}t_f) \tag{10}$$

where $\mathtt{lower}(T, I) = \max\{c \mid t - \underline{0} \geq c \in I, t \in T\}$ and $\mathtt{upper}(T, I) = \max\{c \mid t - \underline{0} \leq c \in I, t \in T\}$ are the upper bounds on the earliest and latest possible occurrence times of transitions in T. The next possible firing time of a transition t is determined by the time it becomes enabled. This enabling time is on the other hand determined by the "latest" transition that precedes it (cf. [1], def. 12). The upper and lower bounds for the enabling time of t can be calculated as the maximum of the upper and lower bounds of occurrence times for transitions in ${}^{\bullet\bullet}t$. To determine these bounds, we introduce the functions $\mathtt{lower}$ and $\mathtt{upper}$ above.

The set I_i' contains a number of variables that are not needed for the calculation of the continued evolution of the system. This includes the set of transitions that were disabled by the firing if t_f, as well as the immediate predecessors of t_f. On the other hand we need to keep in the set those variables that correspond to transitions whose successor have not yet been fired, so that we can calculate the latest possible occurrence time for these successors. The function $\mathtt{delete}(I, V)$ in Eqn. 10 is used for this. It does two things: It first deletes all variables in V from I, which corresponds to taking a projection on the solution space, and then it calculates the transitive closure of the new set of constraints, thus transforming it into normal form.

To show the soundness and completeness of our state space we have to show that to each schedule we can find a corresponding path, and vice-versa.

Theorem 2.7 *Given a schedule* $\sigma = (\delta_1, t_1), (\delta_2, t_2), \ldots$ *with states* $s_i = (\mu_i, c_i)$ *we can construct a path* $\pi = (m_0, I_0) \xrightarrow{t_1} (m_1, I_1) \xrightarrow{t_2} \ldots$ *such that* $\mu_i = m_i \ \forall i \geq 0$.

Proof. *(By suitably modification of the proof of Theorem 1 in [14].*

We construct a function $\mathsf{eval}_i(t, \sigma)$ *that given a schedule* σ *gives us for each* $s_i = (\mu_i, c_i)$ *an assignment to each of the variables* t *in* I_i *such that the assignment is a solution of* I_i.

There are two kinds of variables in I_i*: variables for the occurrence times of transitions that are enabled in* μ*, and variables for occurrence times of transitions that are disabled in* μ*. The latter kind of transitions are transitions that have been fired earlier.*

Let us define a function that gives us a value for each variable t *in the constraint* I_i*:*

$$\mathsf{eval}_i(t, \sigma) = \begin{cases} \max(\mathsf{time}_i(\sigma) + t^{\epsilon} - c_i(t, \sigma), \mathsf{time}_j(\sigma)) & \textit{if } t \in \mathsf{enabled}(\mu) \\ \mathsf{time}_k(\sigma) & \textit{if } t \in \mathsf{disabled}(\mu) \end{cases}$$

where $j \geq i$ *is the time when* t *will become disabled, and* $k \leq i$ *is the time* t *was fired (became disabled). Note that* $\mathsf{time}_j(\sigma) \leq \mathsf{time}_i(\sigma) + t^{\lambda} - c_i(t, \sigma)$*, because* t *must become disabled before* $c(t)$ *reaches* t^{λ}*. Also note that* $\mathsf{eval}_i(\underline{0}, \sigma) = 0$ *for all* i.

We will now show, that given a schedule σ*, we can construct a path* π *s.th.* $\mathsf{eval}_i(\sigma)$ *is a solution of* I_i*, ie. the firing rule respects the firing schedule semantics.*

The base case $(i = 0)$*: For all* $t \in \mathsf{enabled}(\mu_0)$*, we have that:*

$$\mathsf{eval}_0(t, \sigma) = \max(\mathsf{time}_0(\sigma) + t^{\epsilon} - c_0(t, \sigma), \mathsf{time}_j(t, \sigma)) - \mathsf{eval}_0(\underline{0}, \sigma) \geq t^{\epsilon},$$

and

$$\mathsf{eval}_0(t, \sigma) = \max(\mathsf{time}_0(\sigma) + t^{\epsilon} - c_0(t, \sigma), \mathsf{time}_j(t, \sigma)) \leq t^{\lambda},$$

and the only disabled transition is $\underline{0}$.

The induction step*: Assume that* $\mu_i = m_i$ *and* eval_i *is a solution of* I_i*, and that we will fire* t_f *to get to* μ_{i+1}*. Then we must first show that* $t_f \in \mathsf{fireable}(m_i, I_i)$*. Because* $t_f \in \mathsf{enabled}(\mu_i)$ *and* $t_f \notin \mathsf{enabled}(\mu_{i+1})$*, we have that* $\mathsf{eval}_i(t, \sigma) = \mathsf{time}_{i+1}(\sigma)$*. For all other transitions* $t \in \mathsf{enabled}(\mu_i)$ *it is the case that* $\mathsf{eval}_i(t) \geq \mathsf{time}_{i+1}(\sigma)$*. Thus* eval_i *satisfies* $I_i \cup \{t_f \leq t \mid t \in \mathsf{enabled}(\mu_i)\}$*, that is* $t_f \in \mathsf{fireable}(m_i)$.

Let I_{i+1} *be obtained from* I_i *by firing* t_f*. Then we must show that* eval_{i+1} *is a solution of* I_{i+1}*. There are 3 cases:*

(i) t *remains enabled:* $\mathsf{eval}_{i+1}(t,\sigma) = \mathsf{eval}_i(t,\sigma)$ *for all* $t \in \mathsf{enabled}(\mu_i) \cap \mathsf{enabled}(\mu_{i+1})$*, because*

$$\begin{aligned}\mathsf{eval}_{i+1}(t,\sigma) &= \max(\mathsf{time}_{i+1}(\sigma) + t^{\epsilon} - c_{i+1}(t,\sigma), \mathsf{time}_j(\sigma)) \\ &= \max(\mathsf{time}_i(\sigma) + \delta_i + t^{\epsilon} - (c_i(t,\sigma) + \delta_i), \mathsf{time}_j(\sigma)) \\ &= \max(\mathsf{time}_i(\sigma) + t^{\epsilon} - c_i(t,\sigma), \mathsf{time}_j(\sigma)) \\ &= \mathsf{eval}_i(t,\sigma)\end{aligned}$$

Since we do not change the constraint on transitions that remain enabled, $\mathsf{eval}_{i+1}(t,\sigma)$ *is a solution of* I_{i+1}.

(ii) t *is newly enabled:* $\mathsf{eval}_{i+1}(t,\sigma) = \max(\mathsf{time}_{i+1}(\sigma) + t^{\epsilon}, \mathsf{time}_j(\sigma))$ *for* $t \in \mathsf{enabled}(\mu_{i+1}) - \mathsf{enabled}(\mu_i)$*, because* $c_{i+1}(t,\sigma) = 0$*. Now since* t *is newly enabled in* m_{i+1} *it is the case that* $t_f \in {}^{\bullet\bullet}t$*. Thus* $\mathsf{time}_i(t_f,\sigma) = \max(\mathsf{time}_i(t',\sigma))$ *for* $t' \in {}^{\bullet\bullet}t$*, because* t_f *enabled* t*, so*

$$\mathsf{eval}_{i+1}(t,\sigma) - \underline{0} \geq \max(\mathsf{time}_i(t',\sigma)) + t^{\epsilon},$$

for $t' \in {}^{\bullet\bullet}t$*. Using a similar argument we obtain that*

$$\mathsf{time}_j(t,\sigma) - \underline{0} \leq \max(\mathsf{time}_i(t',\sigma)) + t^{\lambda},$$

for $t' \in {}^{\bullet\bullet}t$.

(iii) t *has been fired earlier:* $t \in \mathsf{disabled}(\mu)$*:* $\mathsf{eval}_{i+1}(t,\sigma) = \mathsf{eval}_i(t,\sigma)$*. Since we do not change the constraint on transitions that are disabled,* $\mathsf{eval}_{i+1}(t,\sigma)$ *is a solution of* I_{i+1}.

Since eval_{i+1} *satisfies the inequations generated by the firing rule, and all other inequalities are generated by transitive closure we have that* eval_{i+1} *satisfies* I_{i+1}. □

Because we do not add the constraint $\{t_f \leq t \mid \{t_f \leq t \mid t \in \mathsf{enabled}(m)\}$ to the set of inequations, we do not have the other direction directly, ie. any solution of I does not necessarily determine a feasible clock assignment as was the case in [14], because some solutions may contradict the assumption that t_f fired before any other transition. But with the additional information of the next transition that we will fire, we can show that to each path we can find a corresponding schedule.

Theorem 2.8 *Given a path* $\pi = (m_0, I_0) \xrightarrow{t_1} (m_1, I_1) \xrightarrow{t_2} \ldots$ *we can construct a schedule* $\sigma = (\delta_1, t_1), (\delta_2, t_2), \ldots$ *with states* $s_i = (\mu_i, c_i)$ *such that* $\mu_i = m_i$ $\forall i \geq 0$.

Proof. *Assume that we have a path of state classes, with a solution for each* I_i*. Set* $c_0(t) = 0$ *for all* $t \in T$*. Then set* $c_i(t) = 0$ *if* t *is newly enabled in* μ_i*, else let* $c_i(t) = \mathsf{clock}_{i-1}(t) + c_i$*, where* $c_i = I_{i-1}(t_i) - \mathsf{time}_{i-1}$*.* c_i *is the clock value of transition* t_i *at the time of its firing, while* $I_{i-1}(t_i)$ *is the global firing time of* t_i*. If we only allow solutions* $I_{i-1}(t_1)$ *such that the value* $c_i > 0$*,* *then it is easy to see, that the sequence* $(t_1, c_1), (t_2, c_2), \ldots$ *is a reachable firing schedule.* $\square$

This concludes our argument that the firing rule of Def. 2.6 correctly preserves the semantics of a time Petri net.

As mentioned above, the firing rule will lead to an infinite state space, since the bounds on the constraints of enabled transitions will grow without limit as the system evolves (new instances of the transitions will occur later in time). It is however the case, that after a certain limit the system will enter a state from which state on the behavior will be bisimilar, specifically, the relative constraints will be the same. Thus we can define a bisimulation on state classes as follows:

Definition 2.9 Two state classes (m, I), (m', I') are bisimilar, denote by $(m, I) \simeq (m', I')$ iff

$$m = m' \wedge \mathtt{delete}(I, \underline{0}) = \mathtt{delete}(I', \underline{0}),$$

We then need to show that this really is a bisimulation and that it has finite index.

Theorem 2.10 $\simeq$ *is a bisimulation with finite index.*

Proof. *We need to show that* $\simeq$ *is an equivalence relation, and that if* $\mathtt{sc}_1 \simeq \mathtt{sc}_2$*, then for any* t *such that* $\mathtt{sc}_1 \xrightarrow{t} \mathtt{sc}_1'$ *and* $\mathtt{sc}_2 \xrightarrow{t} \mathtt{sc}_2'$*, we have that* $\mathtt{sc}_2 \simeq \mathtt{sc}_2'$*. Finally we need to show that there are only finitely many distinct equivalence classes.*

(i) $\simeq$ *is an equivalence relation:*
Reflexivity and symmetry are trivial. For transitivity we must show that $(m, I) \simeq (m_1, I_1) \wedge (m_1, I_1) \simeq (m_2, I_2) \Rightarrow (m, I) \wedge (m_2, I_2)$*. The transitivity on markings is evident.*

It remains to show that $\mathtt{delete}(I, \underline{0}) = \mathtt{delete}(I_1, \underline{0}) \wedge \mathtt{delete}(I_1, \underline{0}) = \mathtt{delete}(I_2, \underline{0})$
$\Rightarrow \mathtt{delete}(I, \underline{0}) = \mathtt{delete}(I_2, \underline{0})$*. Assume* $\mathtt{delete}(I, \underline{0}) \neq \mathtt{delete}(I_2, \underline{0})$*. Then we must have a solution* β*, for which* $\mathtt{delete}(I, \underline{0}) \Rightarrow \beta$ *but for which* $\mathtt{delete}(I_2, \underline{0}) \not\Rightarrow \beta$*. But then by assumption (2) above we have that* $\mathtt{delete}(I_1, \underline{0}) \not\Rightarrow \beta$*, which by (1) implies* $\mathtt{delete}(I, \underline{0}) \not\Rightarrow \beta$*, a contradiction.*

(ii) $\simeq$ *is a bisimulation:*
It is enough to show that if $\mathtt{sc}_1 \simeq \mathtt{sc}_2$ *then* $\mathtt{fireable}(\mathtt{sc}_1) = \mathtt{fireable}(\mathtt{sc}_2)$*. We have that* $\mathtt{sc}_1 \simeq \mathtt{sc}_2 \Rightarrow \mathtt{enabled}(\mathtt{sc}_1) = \mathtt{enabled}(\mathtt{sc}_2)$*, because the markings are equal. We need to show that* $\forall t, t' \in \mathtt{enabled}(\mathtt{sc}_1)$ *if* $\{t \leq t'\}$ *is consistent in* I_1*, then it is consistent in* I_2*.*

$\{t \leq t'\} \cup I_1$ *is consistent means that* $\mathtt{sol}(t \leq t') \cap \mathtt{sol}(I_1) \neq \emptyset$*. On the other hand since* $\mathtt{sol}(I_1) \subseteq \mathtt{sol}(\mathtt{delete}(I_1, \underline{0}))$*, we have that* $\mathtt{sol}(t \leq t') \cap \mathtt{sol}(\mathtt{delete}(I_1, \underline{0})) \neq \emptyset$*. Because* $\mathtt{delete}(I_1, \underline{0}) = \mathtt{delete}(I_2, \underline{0})$ *this* *implies that* $\mathsf{sol}(t \leq t') \cap \mathsf{sol}(\mathsf{delete}(I_2, \underline{0}))$. *Now because* $\mathsf{delete}(I_2, \underline{0})$ *and* I_2 *only differ by constraints on* $\underline{0}$, *and* $t \leq t'$ *does not constraint* $\underline{0}$ *it is clear that we also have* $\mathsf{sol}(t \leq t') \cap \mathsf{sol}(I_2) \neq \emptyset$, *from which the result follows.*

(iii) $\simeq$ *has finite index:*
We show that there is an upper bound on the number of transitions that can be fired before firing a specific transition t.

Let t^{λ}_{max} *be the highest* t^{λ} *value in the net. Assume that* $t^{\lambda}_{k} = t^{\lambda}_{max}$ *and that* $t_k \in \mathsf{fireable}(sc_i)$. *The thing to note is that if* $t \in \mathsf{fireable}(sc)$ *and* $sc \xrightarrow{t'} sc'$ *for some other* $t' \in \mathsf{fireable}(sc)$ *then* $t \in \mathsf{fireable}(sc')$. *This means that* t_k *will remain fireable in all successor state classes of* sc_i, *until it is fired or becomes disabled. However if it remains enabled we will eventually reach a state class* sc_n *in which* $t' \leq t_k$ *is not consistent for any* $t' \neq t_k \in \mathsf{fireable}(sc_n)$, *because the upper bounds on constraints* $t_k - t'$ *will decrease and finally become* < 0.

□

Using the bisimulation $\simeq$ we can now define a finite version of the state-space, the *state-class graph* as follows: Let $G(N)$ be the infinite graph obtained by expanding the time Petri net from the initial state using the rule of definition 2.6. By theorem 2.10, the quotient $G(N)/\simeq$ is a finite graph. The state class graph $\mathsf{Scg}(N)$ is obtained from $G(N)/\simeq$ by picking one representative for each state and deleting all constraints on $\underline{0}$.

Our approach differs from the approach proposed in [14] in two ways. In the first place we do not add the constraint $\{t_f \leq t \mid \{t_f \leq t \mid t \in \mathsf{enabled}(m)\}$ to the set of inequations. Doing this would case the state space to branch into a tree, because each state would then contain information about the firing order of transitions. The second difference lies in the technique for determining the firing interval for transitions with more than one predecessor. In our approach the absolute firing intervals are used for this. Since in [14] these are not available the algorithm must assume that any of the predecessors was the "latest", which causes the state space to branch again. The state space in [14] then is a tree, while in the approach proposed here it is a graph. The advantage of our approach will become evident in the next subsection. The disadvantage is that we have to carry around more information in the constraint sets.

2.3 Independent transitions in time Petri nets

The definition of independence of events in a systems is a behavioral notion that captures the following intuition.: Given independent events, it does not matter in which order they are executed, the end result will be the same.

In the untimed case independence of transitions can be formalized in the following definition adapted from [8].

Definition 2.11 Two transitions t_1 and t_2 are independent, iff for all states

s of the state space:

(i) if t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s iff t_2 is also enabled in s'; and

(ii) if t_1 and t_2 are enabled in s, then there is a unique state s' such that $s \overset{t_1;t_2}{\Rightarrow} s'$, and $s \overset{t_2;t_1}{\Rightarrow} s'$

To lift this notion to the timed case we need to take into account the constraints. The way the state classes were defined in 2.2 allows us to state independence for time Petri Nets in terms of the independence in the underlying net.

Theorem 2.12 *Let* t_1, t_2 *be independent in the underlying net. Then given paths* $(m, I) \xrightarrow{t_1} (m_1, I_1) \xrightarrow{t_2} (m_2, I_2)$ *and* $(m, I) \xrightarrow{t_2} (m_1', I_1') \xrightarrow{t_1} (m_2', I_2')$ *in* $\mathsf{Scg}(N)$ *we have that* $m_2' = m_2$ *and* $I_2 = I_2'$.

Proof. *We first look at the situation in* $\mathsf{G}(N)$. *Since* t_1 *and* t_2 *are independent in the underlying net, we have that* $m_2' = m_2$ *and thus* $\mathtt{enabled}(m_2') = \mathtt{enabled}(m_2)$. *The set of variables in* I_2 *and* I_2' *is then* $I - {}^{\bullet\bullet}t_1 - {}^{\bullet\bullet}t_2 \cup \mathtt{enabled}(m_2)$. *Since the set of constraints is transitively closed it remains to show that the bounds on the absolute constraints are equal in* I_2 *and* I_2'. *But this is evident, since the only thing that changes in the firing rule (Def. 2.6) is the absolute constraint on* t_f.

Because of bisimilarity, the statement holds for $\mathsf{Scg}(N)$. □

The importance of the theorem lies in the fact that it allows us to use the algorithms for the calculation of independent sets of transitions developed for untimed systems (stubborn sets, ample sets, etc.) by simply replacing the enabling condition with the condition for fireability. We will now turn to the question of finding independent sets of transitions.

3 Partial-order reduction

In this section we will present a version of the state space traversal algorithm for time Petri nets that uses the branching prefix to guide its search.

Consider the "standard" depth-first state-space search algorithm for time Petri nets given in figure 3. There are two lines in the algorithm that are crucial for its efficiency. First in line 6 we look for all the fireable transitions at the current state class. This introduces branching into the search process, which can be alleviated by using a persistent set as the set of fireable transitions. The other complexity arises in line 12. Here we have to search the set of all previously seen state classes for a state class that is bisimilar to the class we are currently considering.

We propose to alleviate the above problems by exploiting the branching prefix. In particular we will use the untimed branching prefix to detect loops in the system. Then in line 12 we will only have a small list of state classes to search. Moreover we will use the untimed branching prefix to give us a persistent set of transitions directly. Essentially we will show how from the branching prefix it is possible to statically determine a persistent set for each marking, so persistent sets are calculated only once. This approach has the added advantage, that when changing the timing constraints, the branching prefix can be reused. This could be of use eg. when debugging the timing constraints of a real-time system, since then the underlying net will remain unchanged.

There are two ways of applying partial-order methods to time Petri nets that have been reported in the literature.

In [13] a timed version of the branching-prefix is introduced. The authors use the same representation for states as was introduced in [14]. However as was shown in [1] the timing constraints on choices introduce dependencies between events that are not causally related. This problem is addressed in [13] by essentially ignoring it and assuming that all choices will be resolved by the environment. This solution is justified by the specific application area addressed by the algorithm, namely asynchronous circuits. In the intended application area for our algorithm, real-time systems, the full interpretation of choice is needed eg. for modelling timeouts.

On the other hand in [14] the standard state space search is augmented with a form of persistent sets, that are calculated from the current marking. The worst case complexity of this calculation is $\mathcal{O}(|\mathsf{enabled}(m)|^2)$. Calculating persistent sets at each marking is relatively efficient for the untimed case. For the timed case however, the situation is slightly different. First of all the set of fireable transitions is a subset of the set of enabled transitions. Thus the persistent set calculated from the untimed model may contain unnecessary many transitions. Another issue is the fact that in a timed systems the state space will contain several instances of the same marking but with different timing constraints. This means that the same persistent set is calculated several times during the state-space traversal. One way of optimizing this could be to store the persistent set together with the marking.

Below we shall first define the branching prefix, then explain how the basic depth first search algorithm is adapted to run on the branching prefix. Then we will augment it to work on a time Petri net, and finally discuss how to extract persistent sets from the branching prefix.

3.1 The Branching Prefix

In this section we recall the definitions of branching process and branching prefix. The branching prefix is an implicit representation of the state space that was originally introduced by McMillan [11] for the efficient verification of asynchronous circuits. Later it was used by Esparza as a basis for a model-checker for simple safety assertions [7]. The branching prefix is a finite unfolding of the causal structure of the Petri net, in which all loops have been

```
SEARCH(TP : TimePetrinet, M : Marking)
  S : Stack; Seen : Set;
  S.PUSH(< m_0, I_0 >);
  while S ≠ ∅ do
    < m, I >← S.POP();
    foreach e ∈ FIREABLE(< m, I >) do
      < m', I' >← FIRE(< m, I >);
      if m' = M then exit;
      if ¬∃sc ∈ Seen :< m', I' >≃ sc
        then S.PUSH(< m', I' >);
      Seen.ADD(< m', I' >);
    endf
  endw
```

Fig. 3. The basic search algorithm.

expanded. The structure is made finite through the detection of these loops. The beginning of a loop is marked by a *cutoff event*. In the branching prefix markings are not explicitly present, instead the have to retrieved as needed. The model checking algorithm of Esparza gives a procedure for doing this. It can be shown that with a suitable definition of a cutoff event, the size of the branching prefix is never larger than the size of the state space. Indeed it is often substantially smaller.

3.1.1 Branching processes

In a net (S, T, F) elements x_1, x_2 are in *conflict* $(x_1 \# x_2)$ iff $\exists t_1, t_2 : {}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$, $(t_1, x_1) \in F^*$, and $(t_2, x_2) \in F^*$.

Definition 3.1 An *occurrence net* $ON = (B, E, G)$ is a finitary, acyclic net, where

- $\forall b \in B : |{}^\bullet b| \leq 1$, and
- $\forall e \in E : \neg e \# e$

Places B of an occurrence net are called *conditions* and transitions E are called *events*. Preplaces are called *preconditions* and postplaces *postconditions*.

Finitary means that every $x \in B \cup E$ has only a finite number of G-predecessors. We denote by $\leq$ the partial order G^* on $B \cup E$. Event e is said to causally precede e' if $e \leq e'$. We write $Min(ON)$ for the $\leq$-minimal elements of a causal net ON and $Max(ON)$ for the $\leq$-maximal events, and call these *initial elements* and *final elements*, respectively.

Definition 3.2 Let $PN = (P, T, F, m_0)$ be a Petri net. A *branching process* of PN is a pair $\beta = (ON, p)$, where ON is an occurrence net and p is a homomorphism from ON to PN such that

$$\forall e_1, e_2 \in E : {}^\bullet e_1 = {}^\bullet e_2 \wedge p(e_1) = p(e_2) \Rightarrow e_1 = e_1.$$

Engelfriet has shown in [5] that every net has a unique maximal branching process which is called the *unfolding*.

States in a branching process can be described by causally closed, conflict free sets of events, that describe the evolution of the systems to that state. A *configuration* C of a branching process is a set of events, s.th:

$$e \in C \Rightarrow \forall e' \leq e : e' \in C \quad (1)$$

$$e, e' \in C \Rightarrow \neg e \# e' \quad (2)$$

Condition (1) states that C is causally closed, while condition (2) that C is conflict free.

Each branching process β has a set of *maximal configurations* $\mathsf{Max}(\beta)$. Given an event e its *local configuration* $[e]$ is the set $\{e' \mid e' \leq e\}$. For a configuration C, we define a function $\mathsf{Cut}(C) = (C^{\bullet} \cup \mathsf{Min}(N)) \setminus {}^{\bullet}C$. The intuition of a cut is that it represents a state of the system. We denote $p(\mathsf{Cut}(C))$ by $\mathsf{Mark}(C)$. Given a branching process β we define a function $\uparrow(\beta, \mathsf{cut})$ as $\uparrow(\beta, \mathsf{cut}) = \beta - \{x \in \beta \mid \exists y \in \mathsf{cut} : x < y\}$.

3.1.2 The Finite Branching Prefix

It is clear that the representation of the behavior of a system through the maximal branching process may be infinite. However as discovered by McMillan [10], it is possible to find a finite representation, the *branching prefix*, of the branching process of the system that contains all reachable cuts (ie. markings) of the system. This representation was later improved by Esparza, Römer and Vogler [6].

The central observation for the definition of the branching prefix is that a finite 1-safe net generates an infinite branching process only if it has cyclic behavior. This means that we can find a cut in the system from which rest of the branching process will be isomorphic to the branching process starting from the initial cut. Thus it suffices to only take one instance of the cycle. The definition of the branching prefix then relies on the discovery of theses cycles.

Definition 3.3 Let $\prec$ be defined by $C_1 \prec C_2 \Leftrightarrow |C_1| < |C_2|$. An event e is a *cutoff event* of the branching process β iff β contains a local configuration $[e']$ such that:

(i) $p(\mathsf{Cut}([e]) = p(\mathsf{Cut}([e']))$, and

(ii) $[e'] \prec [e]$.

The idea is that we can stop the generation of the prefix if we have found an event e that leads to a marking m that we have already reached through an event e'. However this is not quite enough: the algorithm may terminate too early, resulting in an incomplete prefix. For this we need the second condition: the behavior to e' must be smaller than the behavior to e. The event e' is

```
BPSearch(bp : BranchingProcess, M : Marking)
  S : Stack;
  S.push(Min(bp));
  while S ≠ ∅ do
    cut ← S.pop();
    foreach e ∈ enabled(bp, cut) do
      cut' ← fire(cut);
      if Mark(cut) = M then exit;
      if ¬Seen(e, Mark(cut))
        then S.push(cut);
    endf
  endw
```

Fig. 4. Searching the branching prefix.

the *shift-back* event of e and is denoted by e°. Note that there may be several such events in which case we fix one.

The set of cutoff events of a branching process β is denoted by $\mathsf{Cutoff}(\beta)$. We call the marking $p(\mathsf{Cut}([e]))$ of a cutoff event a *cutoff marking*.

Definition 3.4 Given a net N and its maximal branching process β_m, the finite branching prefix β_f is defined as: $\beta_f = \beta_m - \uparrow(\beta_m, \mathsf{Cutoff}(\beta_m))$

3.2 Searching along the branching prefix

The branching prefix is an acyclic representation of the causal behavior of the net. The states are represented as cuts, while the cyclic behavior is implicitly represented by the cutoff events. Thus we need to modify the standard depth-first search algorithm to to take into account this extra structure, as shown in figure 4.

The set of enabled events at a cut in the branching prefix is easily obtained by looking at the presets of the cut. The actual management of cuts and cutoff events is encapsulated into the FIRE function. There are two cases depending on the type of the event e. If e is not a cutoff event then

$$\textsc{fire}(e, \mathsf{cut}) = \mathsf{cut} - {}^\bullet e \cup e^\bullet.$$

If e is a cutoff event then things are a bit more complicated, because the new cut is not obtained from the postset of the shift-back event e°, but from the cut $\mathsf{Cut}([e^\circ])$. So we need to replace all conditions b in $\mathsf{cut} - {}^\bullet e$ that are instances of places that have instances in $\mathsf{Cut}([e^\circ])$. That is

$$\begin{aligned}\textsc{fire}(e, \mathsf{cut}) = {} & \mathsf{cut} - {}^\bullet e - \{b \in \mathsf{cut} - {}^\bullet e \mid \exists b' \in \mathsf{Cut}([e^\circ]) : p(b) = p(b')\} \\ & \cup \;\; \mathsf{Cut}([e^\circ]).\end{aligned}$$

The detection of loops in the search in done in the SEEN function. The

```
TPNSearch(TP : TimePetrinet, M : Marking)
  S : Stack;
  bp ← BranchingProcess(TP);
  S.push(< Min(bp, I_0 >);
  while S ≠ ∅ do
      < cut, I >← S.Pop()
    foreach e ∈ fireable(bp, cut, I) do
      < cut′, I′ >← Fire(e, cut, I);
      if Mark(cut′) = m then exit;
      if ¬seen(e, Mark(cut), I′)
        then S.push(< cut′, I′ >);
    endf
  endw
```

Fig. 5. The modified state space search algorithm

idea is to exploit the fact that the occurrence of a cutoff event signals that the system is about to loop. Thus we store only markings that occur after cutoff events, and the SEEN function only needs to check a marking in a list of markings indexed by cutoff events, ie.

$$\text{SEEN}(e, \text{MARK}(cut)) = cutoff(e) \wedge cut \in Seen(e).$$

Naturally the search procedure as defined above does not make much sense for an untimed net, because it will traverse the state-space once and stop. However in the state space of a time Petri net a marking is bound to occur several times with a different set of timing constraints, so the fact that we only compare state classes after cutoff events is bound to speed up the search. The search algorithm for time Petri nets is given in figure 5. The function FIREABLE is defined as

$$\text{FIREABLE}(bp, cut, I) = \\ \{e \in bp \mid \exists t \in fireable(Mark(cut), I) \wedge p(e) = h\}.$$

Analogously the FIRE function is defined in terms of the FIRE function above together with the firing rule 2.6 and the SEEN function now needs to compare state classes.

3.3 Exploiting causal independence

The algorithm in figure 5 improves on performance of the search only by exploiting the cutoff events. To add partial-order reduction to the algorithm the function fireable needs to return a subset of fireable transitions, and this subset should be a persistent set. The intuition behind our approach is that we should search each process of the system. In a process all events that are enabled at the same cut are independent. Thus at a cut we only need to make sure that we fire one transition for each process. Let $\beta : E \to$

$2^{\mathsf{Max(bp)}}$ be a function that attaches to each event the set of processes (maximal configurations) that it participates in. With $\beta^{-1} : \mathsf{Max(bp)} \to 2^E$ we denote the function that maps a maximal configuration to its set of events. Then the set of *active* processes at a cut is given by $\mathsf{active(cut)} = \bigcup_{e \in \mathsf{fireable(cut)}} \beta(e)$. When we select a set of events to fire, we have to take care that we select only one representative for each active process at the cut and that we include all conflicting transitions in the set. If it is not possible, then we select the whole set of enabled transitions.

Definition 3.5 Let the set $\mathsf{representative(cut)}$ be a set of events such that

$$\begin{array}{l} \forall c \in \mathsf{active(cut)} : \\ \quad \exists! e \in \mathsf{representative(cut)} : e \in \beta^{-1}(c), \quad \text{and} \\ \forall e \in \mathsf{representative(cut)} : \\ \quad \exists e' \in \mathsf{enabled(cut)} \wedge e' \# e \Rightarrow e' \in \mathsf{representative(cut)} \end{array}$$

If such a set cannot be found, then let $\mathsf{representative(cut)} = \mathsf{enabled(cut)}$.

Now recall the definition of a persistent set from [8]:

Definition 3.6 A set of transitions T enabled in a state s is *persistent in* s iff, for all nonempty sequences of transitions $s = s_1 \xrightarrow{t_1} s_2 \dots s_n \xrightarrow{t_n} s_{n+1}$, from s in the state space and including only transitions $t_i \notin T$, t_n is independent in s_n for all transitions in T.

Then we have the following:

Theorem 3.7 *The set* $\mathsf{representative(cut)}$ *is a persistent set.*

Proof. *If* $\mathsf{representative(cut)} = \mathsf{enabled(cut)}$*, then the result follows trivially.*

Assume that there exists a sequence $s = s_1 \xrightarrow{e_1} s_2 \dots s_n \xrightarrow{e_n} s_{n+1}$*, from* s *in the state space and including only events* $e_i \notin \mathsf{representative(cut)}$*, such that* e_n *is dependent on some events* $t \in \mathsf{representative(cut)}$*. Assume further that this is the shortest of all such sequences. Then for all* $1 \leq i < n$ e_i *is independent in* s_i *for all events in* $\mathsf{representative(cut)}$*.*

The only way that e_n *may be dependent on* e *is because they are in conflict, ie.* $e_n \# t$*. This means that there exists a process* π *that is active in* e_n *but not in* e*. Because process membership is downwards hereditary there must exist an event* $e_i < e_n$ *that belongs to* π*. But by assumption this event* e_i *is independent with* e*. Now since* e_i *does not belong to* $\mathsf{representative(cut)}$*, we must find a* $e' \in \mathsf{representative(cut)}$ *that is the representative for* π*. But this means that* e_i *and* e' *are independent. Since* e_i *and* e *are also independent by transitivity we get that* e' *and* e *are independent. But this means that* $\mathsf{active}(e') \cap \mathsf{active}(e) \neq \emptyset$*, ie. we have more that one representative for a process.* □

Now because of theorem 2.12 we can use the subset of fireable events in the set $\mathsf{representative}$ as a persistent set in the search algorithm in figure 5.

Thus we have, along the lines of [8], that we can use the set $\mathsf{representative}$ to search for *local states* of the system. A local state in this case corresponds to the marking of a place.

Theorem 3.8 *The algorithm in figure 5, where the function* $\mathsf{fireable}(\mathsf{bp}, \mathsf{cut}, I)$ *has been replaced by the function* $\mathsf{representative}(\mathsf{bp}, \mathsf{cut}, I)$ *is a complete and correct decision procedure for determining whether a place* p *is reachable in the time Petri net* N.

Proof. *Along the lines of the proofs in eg. [8]. Note that our algorithm is fair by definition, because the timing constraints will take care that no event is left behind in time for too long.* □

However this is not quite enough. What we want is a procedure that finds a marking m, not just a marked place. The standard trick for finding a marking (a global state) is to reduce it to finding a marked place (a local state) by introducing a event that is enabled exactly at the marking m. However this means that we have to recalculate the branching prefix each time we change the query. To circumvent this we introduce the notion of *visible* events. The set of visible events for a marking m are the events that lead to or from the marking ie. $\mathsf{visible}(m) = {}^\bullet m \cup m^\bullet$. So we need to extend the $\mathsf{representative}$ function to return also all fireable visible events:

$$\begin{aligned} &\mathsf{representative}(\mathsf{bp}, \mathsf{cut}, I, m) = \\ &\quad \mathsf{representative}(\mathsf{bp}, \mathsf{cut}, I) \cup \mathsf{visible}({}^\bullet m \cup m^\bullet). \end{aligned}$$

We then have:

Theorem 3.9 *The algorithm in figure 5, where the function* $\mathsf{fireable}(\mathsf{bp}, \mathsf{cut}, I)$ *has been replaced by the function* $\mathsf{representative}(\mathsf{bp}, \mathsf{cut}, I, m)$ *is a complete and correct decision procedure for determining whether a marking* m *is reachable in the time Petri net* N.

Proof. *Along the lines of the proofs in eg. [8].* □

4 Conclusions

The use of persistent sets is a standard technique in the context of untimed systems (cf. [8]), while for timed systems only a few proposals exists: [2] for timed automata, and [14] for time Petri nets.

In this work we have proposed a new semantics for time Petri nets that does not store the firing order into the timing constraints. This allows us to lift the notion of independence from untimed Petri nets, and it becomes possible to directly apply the theory of partial-order reductions to time Petri nets. We show that our semantics is finite and that it preserves reachable markings. The main difference between our work and [14] is the semantics, which in [14] is finer, because the states contain an implicit ordering of events.

As a second contribution we propose to exploit the cut-off events of the branching prefix originally introduced by McMillan in the state-space search of time Petri nets. This idea, of only comparing state classes that arise after firing a cutoff event, is very similar to the idea of detecting *entry nodes* in [9], in that both approaches try to detect the periodicity of the system by statically analysing the untimed system model.

Finally we also shown how to extract *persistent sets* from the branching prefix.

Although we have not shown so in this paper, we conjecture that much more that just reachability is preserved in our semantics. Indeed it should be possible to show some form of timed bisimulation equivalence between the schedule semantics and our new semantics. Also the exact relationship between the representative sets and other forms of persistent sets should be investigated.

References

[1] T. Aura and J. Lilius. Time processes for time Petri nets. In *Applications and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 136–155. Springer Verlag, 1997.

[2] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi. Partial order reductions for timed system. In *Proceedings of CONCUR'98*, 1998, to appear.

[3] B. Berthomieu and M. Diaz. Modelling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, March 1991.

[4] B. Berthomieu and M. Menasche. A state enumeration approach for analyzing time Petri nets. In *Proc. of 3rd European Workshop on Appl. and Theory of Petri Nets*, pages 27–56, Varenna, Italy, September 1982.

[5] J. Engelfriet. Branching processes of Petri nets. *Acta Informatica*, 28:575–591, 1991.

[6] J. Esparza, S. Römer, and W. Volger. An improvement of McMillan's unfolding algorithm. In *Proc. of TACAS'96*, volume 1055 of *LNCS*, pages 87–106. Springer Verlag, 1996.

[7] J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23:151–195, 1994.

[8] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer Verlag, 1996.

[9] K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structure and state-space reduction. In *Proc. of the 18th IEEE Real-Time Systems Symposium*, pages 14–24, 1997.

[10] K. L. McMillan. A technique of a state space search based on unfolding. *Formal Methods in System Design*, 6(1):45–56, 1995.

[11] K. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Proc. of CAV'92*, volume 663 of *LNCS*, pages 164–177. Springer-Verlag, 1992.

[12] P. M. Merlin and D. J. Farber. Recoverability of communication protocols — implications of a theoretical study. *IEEE Transactions on Communications*, 24(9):1036–1043, September 1976.

[13] A. Semenov and A. Yakovlev. Verification of asynchronous circuits using time petri-net unfolding. In *Proc. ACM/IEEE Design Automation Conference*, 1996.

[14] T. Yoneda and H. Schlingloff. Efficient verification of parallel real-time systems. *Journal of Formal Methods in System Design*, 11(2):187–215, 1997.

[15] S. Yovine. *Méthodes et Outils pur la Vérification Symbolique de Systèmes Temporisés.* PhD thesis, Institut National Polytechnique de Grenoble, 1993.