

Enforcing Transition Deadlines in Time Petri Nets

Haisheng Wang, Liviu Grigore, Ugo Buy, and Houshang Darabi
Department of Computer Science
University of Illinois at Chicago
851 South Morgan Street
Chicago, IL 60607
buy@cs.uic.edu

Abstract

We automatically synthesize supervisory controllers that force a system to perform a certain operation by a given deadline. The operation must be executed by a pre-specified delay λ with respect to the previous execution of the operation. We model both the controlled system and our control supervisors as time Petri nets. Given a target transition and a deadline, our supervisors disable net behaviors in which the firing of the target transition may miss the deadline.

Our method is subject to a merge exclusion assumption on the structure of paths contained in a so-called net unfolding. If a control problem does not satisfy this assumption, we abandon supervisor generation. Preliminary empirical results show that our method is both relatively general and more tractable from a computational standpoint than most other real-time analysis methods.

1 Introduction

Production scheduling of manufacturing systems is known to be quite challenging. Given a set of production tasks, their expected duration, their precedence relationships, and their resource requirements, scheduling must define a production schedule complying with task characteristics. In general one can consider two types of production scheduling problems, namely *off-line* and *real-time* scheduling. In the case of off-line algorithms, scheduling is done before production starts. A disadvantage of this type of schedule is that it will not change during production even if the inputs to production scheduling (i.e., the tasks and their requirements) change. Real-time scheduling algorithms can reflect dynamically varying task durations; however, these algorithms must have low computational complexity.

In this paper we introduce an efficient method that can easily reflect duration changes in production schedules defined off-line. We model both the production tasks and the supervisory controller as time Petri nets [13]. Given

a target net transition t_d —modeling a specific production task—and a deadline λ , our supervisory controllers ensure that t_d is fired at most λ time units since its previous firing. The key aspect of our method is the notion of transition *latency*. In brief, the latency of a net transition t indicates how late t can be fired without missing the deadline on t_d . Our control supervisors will disable a transition t when the time until the expiration of the deadline on the firing of t_d becomes smaller than t 's latency.

The computation of transition latencies relies on a construction called a *net unfolding*. The unfolding of a Petri net is an acyclic Petri net that shows explicitly causal relationships on transition firings in the original net. The unfolded net and the original net have the same sets of reachable markings, after mapping unfolding nodes to nodes in the original net. Thus, a net unfolding is a compact representation of the original net's state space [6], [10], [12], [14]. An advantage of our method is that we unfold the (untimed) ordinary Petri net underlying the time Petri net modeling the controlled system. By considering untimed nets we obtain much smaller unfoldings than in case of timed unfoldings. A disadvantage is that our supervisory controllers are not maximally permissive.

An additional disadvantage is that our method sometimes fails to produce a supervisory controller. This happens when the unfoldings that we obtain do not satisfy the *merge exclusion assumption* on the structure of paths contained in the unfolding. Preliminary empirical results show that a majority of Petri nets we considered satisfies the assumption. Given that our method can handle relatively large time Petri nets—much larger than verification methods and alternative supervisory control methods can handle—we believe that latency-based control is an important step toward the development of practical tools for enforcing real-time deadlines.

We previously introduced a method for defining supervisory controllers for selected transitions in time Petri nets [2]; here we extend that method to arbitrary target transitions. The main contribution of this article compared with our previous results [2], [3], [4] is the intro-

duction of the Discovery algorithm that checks whether the merge exclusion assumption is respected and assigns final latency values to net transitions. This paper is organized as follows. In Section 2 we introduce required definitions. In Section 3, we briefly summarize our framework for supervisor generation. Section 4 discusses our strategy for generating net unfoldings. In Section 5, we discuss the Atlantis and Discovery algorithms for defining transition latencies using net unfoldings. Section 6 discusses supervisor generation.

2 Definitions

An *ordinary Petri net* is a four-tuple $\mathcal{N} = (P, T, F, M_0)$ where P and T are the node sets and F the edges of a directed bipartite graph, and $M_0 : P \rightarrow \mathbb{N}$ is called the *initial marking* of \mathcal{N} , where \mathbb{N} denotes the set of nonnegative integers. In general, a marking or state of \mathcal{N} assigns a nonnegative number of tokens to each $p \in P$.

Given a net $\mathcal{N} = (P, T, F, M_0)$, we use the following notations for the sets of predecessors and successors of a node $x \in P \cup T$. The set of input (output) nodes of x is denoted by $\bullet x = \{y | (y, x) \in F\}$ ($x^\bullet = \{y | (x, y) \in F\}$).

We extend this notation to place subsets and transitions subsets. For instance, given a transition subset $C \subseteq T$, $\bullet C$ denotes the union of the predecessor sets of all transitions in C . Thus, we have $\bullet C = \bigcup_{t_i \in C} \bullet t_i$ and so on.

A transition is *enabled* when all its input places have at least one token. When an enabled transition t is fired, a token is removed from each input place of t and a token is added to each output place; this gives a new marking (state). Petri net $\mathcal{N} = (P, T, F, M_0)$ is *safe* if $M_0 : P \rightarrow \{0, 1\}$, and if all markings reachable by legal sequences of transition firings from the initial marking have either 0 or 1 tokens in every place. Petri net is said to be *level-4 live* if every transition can be fired infinitely often from all reachable net states.

A *time Petri net* \mathcal{N}_T is a five-tuple (P, T, F, M_0, S) where (P, T, F, M_0) is an ordinary Petri net, and S associates a *static firing interval* $\mathcal{I}(t) = [a, b]$ with each transition t , where a and b are rationals in the range $0 \leq a \leq b \leq +\infty$, with $a \neq \infty$ [1], [13]. If transition t with $\mathcal{I}(t) = [a, b]$ becomes enabled at time θ_0 , then transition t must fire in the time interval $[\theta_0 + a, \theta_0 + b]$, unless it becomes disabled by the removal of tokens from some input place in the meantime. The static earliest firing time of transition t is a ; the static latest firing time of t is b ; the dynamic earliest firing time of t is $\theta_0 + a$; the dynamic latest firing time of t is $\theta_0 + b$; the *dynamic firing interval* of t is $[\theta_0 + a, \theta_0 + b]$.

A *firing schedule* for time Petri net \mathcal{N}_T is a finite sequence of ordered pairs (t_i, θ_i) such that transition t_1 is fireable at time θ_1 in the initial state of \mathcal{N}_T , and transition t_i is fireable at time θ_i from the state reached by starting in the initial state of \mathcal{N}_T and firing the transitions t_j for $1 \leq j < i$ in the schedule at the given times.

Henceforth, we will assume that the Petri nets underlying all our time Petri nets are safe. In practice, we found that this assumption does not limit the systems that can be analyzed automatically. (See, for instance, [5].) Moreover, our method can be extended to bounded Petri nets. In a bounded Petri net, the number of tokens that can reside in any reachable net state is at most k , for some integer $k > 1$. Methods for unfolding bounded Petri nets already exist [9], which will make it easy to extend our method. We also assume that untimed nets underlying our time Petri nets are level-4 live. If a Petri net is not live, a deadline on transition firing may not be enforceable. Finally, we assume that our time Petri nets do not contain zero-length cycles (i.e., a non-Zeno assumption). This means that any non-empty firing schedule leading the net from a given marking back to the same marking, must have a strictly positive total duration.

3 Framework

Given a time Petri net $\mathcal{N}_T = (P, T, F, M_0, S)$, define Σ to be the set of all firing schedules of \mathcal{N}_T . Given a schedule $\sigma \in \Sigma$, and a deadline λ on the firing of target transition t_d , we say that σ is *deadline-compliant* with respect to t if the following two conditions are met:

1. The first firing of t_d occurs at time $\theta_0 \leq \lambda$ time units from the start of σ .
2. Any subsequent firing of t_d in σ follows the previous firing of t_d by at most λ time units.

Otherwise, we say that schedule σ is *deadline-violating*.

We generate supervisory controllers in three steps. First, we construct the unfolding \mathcal{U} for the ordinary Petri net \mathcal{N} underlying the plant net \mathcal{N}_T . Second, given \mathcal{U} and a target transition $t_d \in \mathcal{N}$, we compute the latency of each transition t in \mathcal{U} . The latency $l(t)$ of a transition $t \in T$ is an upper bound on the maximum delay between any firing of t and the next firing of t_d , along firing schedules permitted by the supervisory controller for net \mathcal{N}_T , as the deadline on the firing of t_d is approaching. Third, we generate our controller net, which is attached to the plant net. As we will see, this controller net consists of a so-called *clock subnet* \mathcal{C} , and a *supervisor subnet* \mathcal{S} .

Consider, for instance, the time Petri net in Figure 1. Suppose that transition t_4 must be fired every $\lambda = 100$ time units. Given that the static latest firing time of t_4 is 1 time unit, t_4 is guaranteed to fire by the deadline if it becomes enabled at least 1 time unit before the deadline expires and if conflicting transitions are disabled. Thus, we define the latency of transition t_3 , which must fire in order for t_4 to be enabled, to be 1 time unit. Similarly, we can define the latency of transitions t_1 and t_2 to be 8 time units, the sum of t_3 's latency (one unit) and the latest firing time of t_3 (seven units). The latency of t_8 , which enables t_1 , will thus be 12 time units. Defining the latency of transitions t_5 and t_9 is more difficult because the

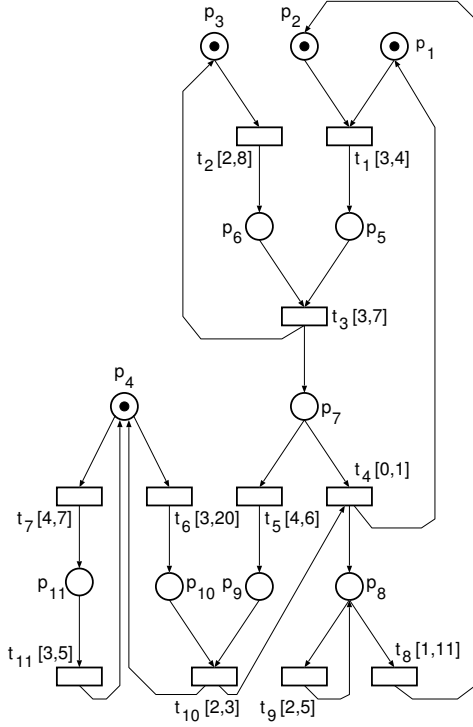


Figure 1. Example of a time Petri net.

firing of transition t_5 disables t_4 , and firing of transition t_9 disables t_8 , thereby preventing t_1 from firing. Thus, t_5 and t_9 must be disabled some time before the deadline on the firing of t_4 . Transition t_7 can also prevent t_4 from firing. Thus, the clock and supervisor subnets for the net shown in Figure 1 must disable transitions t_5 , t_9 and t_7 at a suitable number of time units before the expiration of the deadline. These subnets are not shown in the figure. In the following three sections we explain in detail each of the three steps involved in supervisor generation: (1) unfolding construction, (2) two algorithms for generating transition latencies, and (3) the generation of the supervisory controller.

4 Net unfolding

The unfolding of an ordinary Petri net \mathcal{N} is an unmarked Petri net \mathcal{U} whose places and transitions are mapped to \mathcal{N} 's places and transitions by a homomorphic function h [6], [9], [10], [12]. Unfolding \mathcal{U} has two useful properties relative to the original net. First, \mathcal{U} has the same set of reachable states as net \mathcal{N} after h is applied to the places in \mathcal{U} . Second, \mathcal{U} shows explicitly causal relationships on transition firings that are implicit in \mathcal{N} . Given a target transition t_d in \mathcal{N} , \mathcal{U} shows which \mathcal{N} transitions must be fired first in order for t_d to fire. Thus, by analyzing a net unfolding we can identify sequences of \mathcal{N} transitions that can lead to the firing of t_d . To enforce deadline λ on the firing of t_d we allow only the firing of transition sequences whose overall delay is smaller than

the time left until λ expires.

Our method for supervisor generation unfolds the (untimed) ordinary Petri net $\mathcal{N} = (P, T, F, \mu_0)$ underlying a time Petri net $\mathcal{N}_T = (P, T, F, \mu_0, S)$. We choose untimed unfoldings for two reasons. First, untimed unfoldings allow us to model behaviors that are not present in the original net due to conflicts among transitions with non-overlapping firing intervals. For instance, a transition t_i may be prevented from firing in a timed net by a conflicting transition t_j with a shorter dynamic delay than t_i . However, our supervisory controllers can force t_i to fire by disabling conflicting transitions, such as t_j . By unfolding the untimed ordinary net underlying the original time Petri net, we can capture all potential behaviors of the controlled net, including behaviors in which t_i is forced to fire. Second, timed unfoldings are likely to be much bigger than untimed unfoldings [14].

The definitions below are similar to previous definitions [6], [9], [12]; however, our conditions for terminating unfolding construction are different from previous definitions, as we explain below.

DEFINITION [Node precedence] Consider nodes x and y in a Petri net. Node x precedes y , denoted by $x < y$, if there is a directed path from x to y in the net.

DEFINITION [Node conflict] Nodes x and y are in conflict, denoted by $x \# y$, if the Petri net contains two distinct paths originating at the same place p that diverge immediately after p and lead to x and y . When $x \# x$ holds, we say that node x is in self-conflict. Also, we say that x and y are concurrent if they are not in conflict with each other and neither node precedes the other.

DEFINITION [Net homomorphism] Given two ordinary Petri nets $\mathcal{N}_1 = (P_1, T_1, F_1, \mu_{0,1})$ and $\mathcal{N}_2 = (P_2, T_2, F_2, \mu_{0,2})$, a net homomorphism from \mathcal{N}_1 to \mathcal{N}_2 is a function $h : (P_1 \cup T_1) \rightarrow (P_2 \cup T_2)$ mapping \mathcal{N}_1 nodes into \mathcal{N}_2 nodes, subject to the following conditions:

1. Function h preserves node types (transitions and places). Thus, we have $h(P_1) \subseteq P_2 \wedge h(T_1) \subseteq T_2$.
2. Function h preserves the input and output relation among nodes. Thus, we have $\forall x \in (P_1 \cup T_1)$ and $\forall y \in \bullet x, h(y) \in \bullet(h(x))$. Likewise, we have $\forall z \in x^\bullet, h(z) \in (h(x))^\bullet$.
3. Function h preserves initial markings. Thus, we have $\forall p_1 \in \mu_{0,1}, h(p_1) \in \mu_{0,2}$.

DEFINITION [Unfolding] An unfolding $\mathcal{U} = (\mathcal{O}, h)$ of net \mathcal{N} is a pair containing an unmarked Petri net $\mathcal{O} = (P_{\mathcal{O}}, T_{\mathcal{O}}, F_{\mathcal{O}})$ and a net homomorphism h mapping \mathcal{O} into \mathcal{N} . Net \mathcal{O} is subject to following conditions: (1) $\forall p \in P_{\mathcal{O}} |\bullet p| \leq 1$; (2) \mathcal{O} is acyclic; (3) each node $u \in P_{\mathcal{O}} \cup T_{\mathcal{O}}$ is finitely preceded; and (4) no node in \mathcal{O} is in self-conflict. The initial places of an unfolding, denoted by μ_{min} , have no input transitions; these places correspond to μ_0 , the initial places of \mathcal{N} .

discussed elsewhere [6]. Although the size of an unfolding is at worst exponential in the size of the original net, in practice unfoldings tend to be much smaller, that is, polynomial in the size of net \mathcal{N} .

5 Computing transition latencies

We compute latencies of places and transitions contained in unfolding \mathcal{U} in two steps. First, the Atlantis algorithm assigns so-called *basic latencies* to nodes in \mathcal{U} . Next, the Discovery algorithm assigns so-called *full latencies* to \mathcal{U} nodes. Before discussing these two algorithms in detail, we introduce definitions specific to our method.

Let $\mathcal{U} = (P_U, T_U, F_U, \mu_{min}, h)$ be the unfolding of the untimed Petri net \mathcal{N} underlying \mathcal{N}_T . Let t_d denote the target transition representing the system event for which we want to enforce periodic deadline λ , with $\lambda \in \mathbb{N}^+$.

DEFINITION [Full cause of a node] Let $w \in P_U \cup T_U$ be a node in unfolding \mathcal{U} . The *full cause* of w is the set of nodes preceding or equal to w : $[[w]] = \{w' \in P_U \cup T_U \mid w' \leq w\}$.

DEFINITION [Fringe place] Let $t \in T_U$ be a cutoff transition. Any place $p \in P_U$ belonging to the cut of t 's cause (i.e., $p \in cut([t])$) is called a *fringe place*.

DEFINITION [Matching transition and place] Let $t \in T_U$ be a cutoff transition in unfolding \mathcal{U} . If there exists $\tilde{t} \in T_U$ such that $h(cut([t])) = h(cut([\tilde{t}]))$ and $[\tilde{t}] \subset [t]$, we call \tilde{t} the *matching transition* of t , denoted by $\tilde{t} \approx t$. In this case, if $h(p) = h(\tilde{p})$, where $p \in cut([t])$ and $\tilde{p} \in cut([\tilde{t}])$, we call \tilde{p} the *matching place* of p , denoted by $\tilde{p} \approx p$. We note that the concept of a matching transition was discovered concurrently and independently with us by Giua and Xie, which they call a *mirror transition* [8].

DEFINITION [Precedes-by-match] Let $w \in P_U \cup T_U$ be a node in the unfolding \mathcal{U} and places $p, \tilde{p} \in P_U$ such that p is a fringe place, $p \approx \tilde{p}$, and $\tilde{p} \in [[w]]$. For all $w' \in P_U \cup T_U$ such that $w' \in [[p]] - [[w]]$, we say that w' *precedes-by-match* w and we shall denote that by $w' <_{match} w$.

The precedes-by-match relation can be informally explained using the following logic. If a node w' , $w' \notin [w]$, precedes fringe place p , that matches place \tilde{p} , and if place \tilde{p} precedes arbitrary node w , then w' precedes by match w : $w' < p, p \approx \tilde{p}, \tilde{p} < w \rightarrow w' <_{match} w$. For instance, in Figure 2 we have that $p_8 <_{match} p_7$ because $p_8 < p'_5$, $p_5 \approx p'_5$, and $p_5 < p_7$. We define the *extended-precedes* relation to be the transitive closure of the precedes-by-match relation.

DEFINITION [Extended path] We also say that an extended path σ_w exists between two nodes w and w' if w and w' are in an extended-precedes relation, if σ_w contains at most one occurrence of each node in unfolding \mathcal{U} , and if the number of matching places contained in σ_w is minimal. (This means that there is no other extended path between w and w' that contains fewer pairs of matching places than σ_w .)

DEFINITION [Basic latency] Let u be a node in unfolding \mathcal{U} . We define the *basic latency* $b(u)$ of u to be a function mapping u into a nonnegative integer as follows:

$$b(u) : P_U \cup T_U \rightarrow \mathbb{N}, b(u) = \max_{\sigma_u} \left\{ \sum_{t \in \sigma_u \cap T_U - \{u\}} lft(t) \mid \sigma_u = \{u, \dots, t_d\} \text{ and } t_d \in T_D \right\}$$

In the above σ_u denotes an extended path starting with node u and ending with some t_d occurrence in \mathcal{U} . Also, set T_D denotes all occurrences of the target transition in \mathcal{U} : $T_D = \{t_{di} \in T_U \mid h(t_{di}) = t_d\}$.

Thus, we define the basic latency of a node u to be the maximum length of all extended paths leading from u to all occurrences of t_d in \mathcal{U} . In addition, we define the length of one such path σ_u to be the sum of the latest firing times of all transitions in $\sigma_u - \{u\}$, the sequence obtained by removing u from σ_u .

The basic latency of a node u is not sufficient in order to determine the time that a token may take to travel from u to a target transition occurrence because the token could incur additional delays while going through transitions with multiple input places.

DEFINITION [Synchronization delay] Given a transition $t \in \mathcal{U}$, such that t has multiple input places p_1, \dots, p_n , the synchronization delay $\delta(p_i)$ of one such place p_i is an upper bound on the time that a token may reside in p_i while transition t is not enabled.

The synchronization delay of a place p accounts for the fact that a token may be delayed while waiting for transition $t = p^\bullet$ to become enabled. Of course, this delay could be indefinite, for instance, if another input place of t never receives a token. Our liveness assumption prevents this situation from taking place.

DEFINITION [Full latency] Let u be a node in unfolding \mathcal{U} . We define the *full latency* $l(u)$ of u to be the sum of the basic latency of u and the synchronization delays along extended paths from u to t_d occurrences in \mathcal{U} .

$$l(u) : P_U \cup T_U \rightarrow \mathbb{N}, l(u) = \max_{\sigma_u} \left\{ \sum_{t \in \sigma_u \cap T_U - \{u\}, p \in \sigma_u} lft(t) + \delta(p) \right\}$$

with $\sigma_u = \{u, \dots, t_d\}$ and $t_d \in T_D$

If the first node u of σ_u is a transition, we deliberately exclude the latest firing time of u from the computation of the length of σ_u . This is so because the disabling and firing of transitions are zero-delay events. If u is a transition, it will be disabled when its latency becomes greater than the time left before the deadline on t_d expires.

5.1 The Atlantis algorithm

This algorithm iteratively performs various backward sweeps in unfolding \mathcal{U} . The node sets considered in each sweep are disjoint. Atlantis takes as input an unfolding

of untimed net \mathcal{N} underlying $\mathcal{N}_{\mathcal{T}}$, and pairs of matching places discovered during unfolding construction. Atlantis's output is the basic latency of each \mathcal{U} node.

Each sweep consists of two phases. The first sweep starts from occurrences of target transition t_d in \mathcal{U} . For each such occurrence t_{di} , during the first phase we find all nodes in $[[t_{di}]]$ by a linear sweep backward from t_{di} and we mark the nodes that we find. We then set the latency of each t_{di} to zero. During the second phase, we compute the latency of each marked node by exploring backward from each t_{di} .

When exploring back from a place $p \in P_U$, we consider the single transition in $t_i \in \bullet p$. This transition is assigned the same basic latency as p , but only provisionally. This is so because t_i could be discovered multiple times during the same backward sweep. The basic latency value of t_i is the highest such value. If t_i has one or more out arcs to place(s), p_j , marked in this sweep whose basic latency is still unknown, we must wait until we re-discover t_i from each such place p_j before settling the basic latency value of t_i . In this case, we also postpone exploration of the predecessors of t_i as the basic latency of t_i may yet be increased, requiring us to increase also the basic latency of nodes in $[[t_i]]$.

When exploring back from a transition $t \in T_U$, we consider all places $p_i \in \bullet t$. The basic latency of each p_i is set to the sum of the basic latency of t and the latest static firing time of t . This basic latency value signifies that a token may reside in p_i for as long as the latest firing time of t before moving past t . Again, this value is assigned only provisionally, as place p_i may be reached along multiple paths originating at different occurrences of t_d in \mathcal{U} .

The first sweep is complete when the initial places of the unfolding that are in $[[t_{di}]]$, for some t_{di} , have their basic latency defined. In the unfolding of Figure 2, we start the first sweep from t_4 , the only occurrence of the target transition in this unfolding. The nodes in $[[t_4]]$ are marked in the first phase, namely $t_4, p_7, t_3, p_6, p_5, t_2, t_1, p_3, p_2$ and p_1 . In the second phase, we first set the basic latency of t_4 to zero. Next, we explore the places in $\bullet t_4$. The basic latency of p_7 is set to one, the sum of the latest firing time and the basic latency of t_4 . When exploring backward from p_7 , we discover transition t_3 . Although t_3 has another output place, p'_3 , in addition to p_7 , we can settle the basic latency of t_3 because p'_3 was not marked during Phase 1 of this sweep. Thus, the basic latency of t_3 is set permanently to one. In this sweep we have these results: $b(p_6) = 8, b(p_5) = 8, b(t_2) = 8, b(t_1) = 8, b(p_3) = 16, b(p_2) = 12$ and $b(p_1) = 12$.

After the completion of the i -th sweep, we decide whether additional sweeps are needed based on the existence of matching places for the places discovered during the sweep. If at least one such place exists, then an additional sweep is needed. In this case, the starting nodes for sweep $i + 1$ will be all and only the matching places of places discovered during the i -th sweep. In Figure 2,

p_1, p_2, p_3, p_5 and p_7 are matched by $p'_1, p'_2, p'_3, p'_5, p'_7$ and p''_7 . Thus, we must conduct an additional backward sweep using the latter six nodes as starting nodes.

Before we conduct sweep $i + 1$, we must first check whether our *merge exclusion assumption* holds for the supervisory control problem under consideration. This assumption is based on the notion of an *out-of-sweep node* for sweep $i - 1$. In brief, node u is an out-of-sweep node if (1) u is discovered by Atlantis during sweep i , and (2) at least a node in $\bullet u$ is discovered by Atlantis in sweep $i - 1$. For instance, transitions t_5 and place p'_3 are out-of-sweep nodes for sweep 1 of the unfolding in Figure 2. The assumption states that unfolding paths originating at out-of-sweep nodes $u_1 \dots u_n$ for sweep $i - 1$ may not merge with each other or with conflicting paths contained in sweep i if at least one of $u_1 \dots u_n$ is a transition. If this assumption is violated, we abandon the generation of a supervisory controller.

In Figure 2, the restriction of the starting nodes for the second sweep contains nodes $p'_7, t'_3, p'_5, p'_6, t'_1, t'_2, p'_3, p'_2, p'_1, t_8, p_8, p''_7, t_{10}, p_{10}, p_9, t_6, t_5$ and p_4 . The basic latencies of the six starting nodes is set to the basic latencies computed earlier for their matching places. Next, we check whether our merge assumption holds for Sweep 2. The out-of-sweep nodes for sweep 1 are t_5, p_8, p'_1 and p'_3 . Since paths originating at the only out-of-sweep transition, t_5 , do not merge with paths originating at other out-of-sweep nodes or with conflicting paths in Sweep 2, the merge assumption is met. After the second sweep is complete, we conduct a third sweep starting from places p''_3, p'_4, p''_4 and p'_8 . All these places match places discovered during Sweep 2. The merge assumption is again checked successfully and the basic latencies of nodes in the third sweep are defined.

5.2 The Discovery algorithm

The Discovery algorithm computes full latency values for key nodes contained in an unfolding \mathcal{U} by taking into account synchronization delays that may occur within each sweep. Consider a sweep- i transition t with multiple input places; the following three kinds of nodes could be contained in the full cause $[[t]]$ of t :

1. Out-of-sweep nodes for sweep $i - 1$;
2. Initial unfolding nodes discovered in sweep i ; and
3. Places p_i such that p_i is matched by a fringe node p'_i that is a starting place for sweep $i + 1$ preceded by an out-of-sweep place p_o for sweep i .

When $[[t]]$ contains multiple nodes from the three above categories, we set the full latency of out-of-sweep nodes in $[[t]]$ to be the maximum of the following three kinds of nodes (whichever apply):

1. The basic latency of sibling out-of-sweep places for sweep $i - 1$ contained in $[[t]]$;

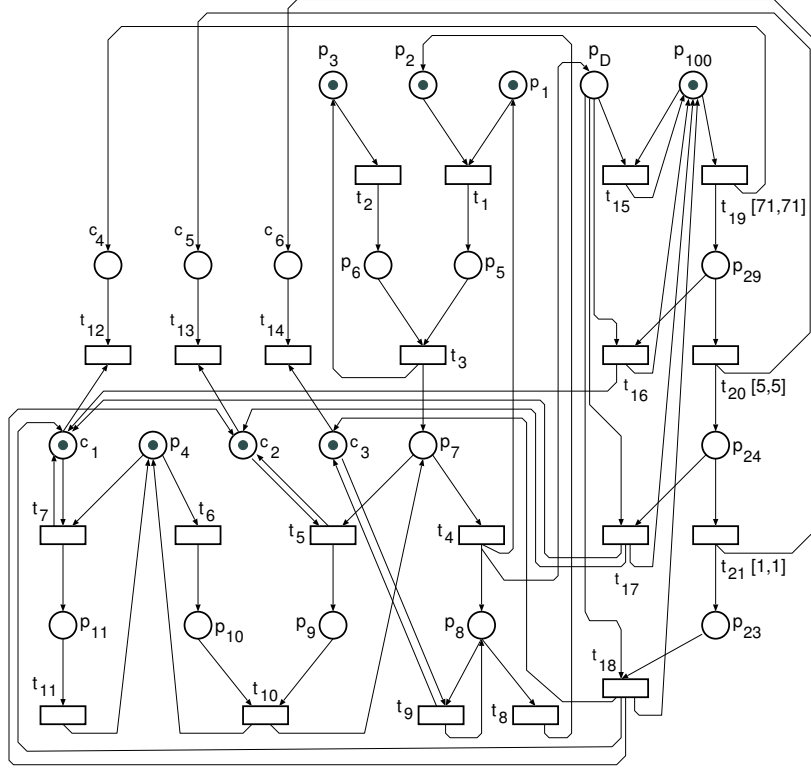


Figure 3. Controller of time Petri net in Figure 1.

2. The basic latency of initial unfolding nodes discovered in sweep i and contained in $[[t]]$; and
3. The full latency of out-of-sweep places for sweep i , such that these places lead to at least one fringe place p'_i matching a place $p_i \in [[t]]$.

In Figure 2, out-of-sweep transition t_5 and initial unfolding place p_4 join at transition t_{10} , meaning that t_5 and p_4 may cause a synchronization delay at t_{10} . Thus, we set the *full latency* of t_5 to 24, the maximum of 4, 24 and 24, the basic latencies of t_5 , p_4 and p'_4 . The resulting full latencies of all nodes in the unfolding are shown in Figure 2.

6 Supervisor generation

The purpose of our supervisory controllers is to disable transitions whose firing may lead to a violation of the deadline on the firing of a target transition. To this end, it is sufficient to disable out-of-sweep transitions when their full latency becomes greater than the time left until the expiration of λ . Note that out-of-sweep transitions may have multiple occurrences in an unfolding. When generating the supervisory controller, we consider the greatest full-latency value of all occurrences of an out-of-sweep transition. Transitions in \mathcal{N}_T are enabled again immediately after t_d fires. In the sequel, we assume that $\lambda \leq l(t) \forall t \in T$. If, however, for some transition $t_s \in T$

we have $l(t_s) < \lambda$, t_s will never be enabled by our control supervisors, meaning that the value of λ is too low for our supervisors to guarantee that t_d is fired in timely fashion after t_s has fired. In this case, a portion of net \mathcal{N}_T may still be live.

Our supervisory controllers consist of two subnets, a *clock subnet* and a *supervisor subnet*. The formal definitions of these subnets appear elsewhere [2]. The clock subnet keeps track of the elapsing of time until the expiration of λ . Transitions in this subnet must force the disabling of transitions in \mathcal{N}_T , as the deadline approaches. The supervisor subnet actually controls the enabling and disabling of out-of-sweep transitions in \mathcal{N}_T . For instance, in the net appearing in Figure 1 it is sufficient to disable transitions t_7 at 29 time units, t_5 at 24 time units and t_9 at 23 time units before the expiration of the deadline in order for target transition t_4 to fire by the deadline.

The right portion of Figure 3 shows the clock subnet for the time Petri net appearing in Figure 1 and deadline $\lambda = 100$ on the firing of transition t_4 . The clock subnet consists of transitions t_{15} through t_{21} and places p_D , p_{100} , p_{29} , p_{24} and p_{23} , along with arcs incident upon these places and transitions. The supervisor subnet consists of transitions t_{12} through t_{14} and places c_1 through c_6 . A token in place p_{100} signifies that there are at least 29 time units, but no more than 100 time units, until the deadline on t_4 expires. A token in place p_{29} signifies that there are no more than 29 time units left until the deadline's expiration. Place p_{100} is initially marked. After 71

time units, transition t_{19} moves the token to place p_{29} . The firing of this transition will also disable transition t_7 , whose full latency is in fact 29. Transitions t_{15} through t_{18} reset a token in place p_{100} immediately after the firing of t_4 ; these transitions are enabled by place p_D .

Place c_2 in Figure 3 controls t_5 , which must be disabled 24 time units before the deadline on t_4 expires. This event is captured by transition t_{20} . When t_{20} fires c_2 loses its token disabling t_5 . Place c_2 becomes marked again (re-enabling t_5) as soon as t_d fires. In this case, place p_D receives a token, which enables one of the transitions in the set t_{15} through t_{18} . When one of these transitions fires, transitions t_5 , t_7 , and t_9 are enabled. Transitions t_7 and t_9 are controlled similarly.

We have implemented the two most crucial aspects of our method for supervisor generation, namely unfolding construction and the Atlantis algorithm. We have applied the resulting software prototypes to the analysis of various Petri nets. The largest net that we considered models a manufacturing plant containing a conveyor belt, a working station, a buffer, and various sensors and actuators.

7 Conclusions and future work

We presented a supervisory control method for enforcing a deadline on the execution of an event in a real-time system. We model both the controlled system and the generated controllers as time Petri nets. An advantage is that our method avoids enumerating explicitly the states of the controlled net. Instead, our method uses net unfolding, an implicit state space representation of the untimed Petri net underlying our controlled (timed) nets. This is clearly beneficial because untimed state-space representations are likely to be much smaller than the corresponding timed representations.

Preliminary results with our prototype set confirm that the complexity of our method is dominated by unfolding construction. In the future, we will continue our empirical studies. In addition, we will consider enforcing multiple deadlines on different net transitions simultaneously. We are currently investigating ways to relax the merge exclusion assumption on our plant nets. We are also investigating extensions to the case of bounded, rather than safe, Petri nets. Finally, we are investigating the integration of our supervisory control methods with methods for enforcing liveness properties. (See, e.g., [7], [11].)

References

- [1] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Trans. Softw. Eng.*, 17(3):259–273, Mar. 1991.
- [2] U. Buy and H. Darabi. Deadline-enforcing supervisory control for time Petri nets. In *CESA'2003 – IMACS Multiconference on Computational Engineering in Systems Applications*, Lille, France, July 2003. Available on CD-ROM.
- [3] U. Buy, H. Darabi, M. Lehene, and V. Venepally. Supervisory control of time Petri nets using net unfolding. In *Proceedings 29th IEEE Computer Software and Applications Conference (COMPSAC-05)*, volume 2, pages 97–100, 2005.
- [4] U. Buy, M. Lehene, and H. Darabi. Latency-based supervisors for enforcing deadlines in time Petri nets. In *Proceedings IEEE/NASA 29th Annual Software Engineering Workshop (SEW 29)*, Greenbelt, Maryland, Apr. 2005.
- [5] S. Duri, U. Buy, R. Devarapalli, and S. M. Shatz. Application and experimental evaluation of state space reduction methods for deadlock analysis in Ada. *ACM Trans. Software Engineering and Methodology*, 3(4):340–380, Oct. 1994.
- [6] J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. *Formal Methods in System Design*, 20(3):285–310, May 2002.
- [7] M. P. Fanti and M. C. Zhou. Deadlock control methods in automated manufacturing systems. *IEEE Trans. on Systems, Man, and Cybernetics*, 2004.
- [8] A. Giua and X. Xie. Control of safe ordinary Petri nets with marking specifications using unfolding. In *Proc. 7th IFAC Work. on Discrete Event Systems (WODES04)*, Reims, France, Sept. 2004.
- [9] K. X. He and M. D. Lemmon. Liveness-enforcing supervision of bounded ordinary Petri nets using partial order methods. *IEEE Transactions on Automatic Control*, 47(7):1042–1055, July 2002.
- [10] V. Khomenko. *Model Checking Based on Prefixes of Petri net unfoldings*. PhD thesis, University of Newcastle Upon Tyne, Newcastle, United Kingdom, 2003.
- [11] Z. Li and M. C. Zhou. Elementary siphons of Petri nets and their applications to deadlock prevention in flexible manufacturing systems. *IEEE Trans. on Systems, Man, and Cybernetics*, 2004.
- [12] K. L. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design*, 6(1):45–65, Jan. 1995.
- [13] P. M. Merlin and D. J. Farber. Recoverability of communication protocols—implications of a theoretical study. *IEEE Trans. Communications*, COM-24(9):1036–1043, Sept. 1976.
- [14] A. Semenov and A. Yakovlev. Verification of asynchronous circuits using time Petri net unfolding. In *Proceedings of the 33rd Design Automation Conference (DAC96)*, pages 59–62, Las Vegas, Nevada, June 1996.