Sidestepping verification complexity with supervisory control

Ugo Buy
Dept. of Computer Science
University of Illinois at Chicago
buy@uic.edu

Houshang Darabi
Dept. of Mechanical and Industrial Engineering
University of Illinois at Chicago
hdarabi@uic.edu

September 12, 2003

Abstract

While the goal of verification is to check whether a model of the system under consideration has a desired property, supervisory control achieves correctness by adding a so-called supervisor that prevents the occurrence of incorrect behaviors to the original system. Supervisory control methods are appealing because they can be much more tractable than the corresponding verification problems. Here we first examine two supervisory control algorithms, one for enforcing mutual exclusion properties and the other for enforcing real-time deadlines on Petri net models of the controlled system. Next, we argue that use of supervisory control methods may lead to a simpler and more effective coding style for embedded software than current practices. Finally, we highlight research issues that must be addressed in order to permit widespread application of supervisory control methods.

1 Introduction

Embedded systems often exhibit features typical of concurrent and real-time systems. The automatic verification of concurrency and timing properties has been studied extensively for over two decades; however, progress in this area has been slow. One reason for this state of affairs is the computational intractability of most verification problems. Here we suggest that supervisory control can be a more practical approach than automatic verification for a broad class of embedded systems. While verification seeks to determine whether a model of the system under consideration has a desired property, supervisory control achieves correctness by adding a so-called supervisor that inhibits incorrect behaviors to the original system.

The supervisory control methods discussed here are suitable for embedded systems that can be modeled as a discrete event system (DES). We are specifically interested in control systems for discrete manufacturing processes, although the same supervisory control techniques are generally applicable to other DES models as well. A discrete manufacturing plant consists of machines for producing, moving, and assembling parts on a shop floor. Control systems for such plants must enforce a variety of correctness properties, including traditional concurrency and timing properties.

Supervisory control methods for discrete event systems typically employ finite state automata or Petri nets to model a DES [4, 13, 14, 17]. Here we focus on Petri-net-based models for two reasons. First, Petri nets support computationally-tractable methods for supervisor synthesis. We summarize two such methods below. These methods use concepts specific to Petri nets, such as P-invariants and net unfolding. Thus, these methods are not applicable to discrete event systems modeled by finite-state automata. Existing supervisory control methods for automata models usually resort to the cross-product of finite-state automata for supervisor synthesis, which is likely to lead to state-space explosion. Second, Petri nets are used extensively for specification and analysis of discrete manufacturing systems. For instance, the language of Sequential Function Charts (SFCs) is a straightforward extension of Petri nets. SFCs are part of the IEC 61131 standard for manufacturing control languages; they are supported by popular commercial products such as Matlab and RSLogix 5000 [8].

Given a Petri net and a set of correctness properties, *supervisory control* methods can enforce the given properties by cleverly disabling net transitions that could lead to a violation of the properties. Thus, the supervisory controller of a Petri net $\mathcal{N}$ is a subnet $\mathcal{S}$ that is added to $\mathcal{N}$ in order to enforce the properties of interest. In this case, $\mathcal{N}$ is said to be the *controlled net* [6]. A supervisor is said to be *maximally permissive* if it does not disable any behavior that satisfies the property of interest while preventing the occurrence of all behaviors that violate the property.

The first method that we survey uses Petri net P-invariants to enforce a broad variety of mutual exclusion constraints of the controlled net [4, 11, 17]. An advantage of this method, which has been studied extensively in the past decade, is that its worst-case computational complexity is polynomial in the size of the controlled system. This complexity is dramatically lower than the complexity of the corresponding verification problems [16]. An additional advantage is that the method generates maximally permissive supervisors [17].

The second mehod uses the concept of *transition latency* to enforce real-time deadlines in time Petri nets [2]. This is a new method; however, it is one of few existing techniques for enforcing real-time properties in timed models. In brief, the latency of a transition t is the latest time when t can be fired while guaranteeing that the given deadline is met. Transition latencies are computed by *unfolding* the ordinary Petri net underlying the time Petri net that models the controlled system [3, 9, 15]. The complexity of this method is dominated by the unfolding, which is at worst exponential in the size of the controlled net [3]. However, we believe that the average-case complexity will be polynomial.

On the positive side, supervisory control methods not only provide a tractable alternative to intractable verification problems. Supervisory controllers can also lead to a novel programming paradigm for embedded and real-time systems. In contrast with current practices, in the new paradigm a programmer would first code an embedded system without the burden of building desired correctness properties (e.g., compliance with mutual exclusion or timing constraints) directly into the code. The programmer would then submit this code, along with a control specification, to a *supervisor generator*, which would augment the programmer's code with a supervisor capable of enforcing the properties contained in the specification.

On the negative side, several issues may adversely affect the applicability of supervisory controllers. For instance, events in the controlled system may not be *observable* and *controllable* to the extent needed for supervisor generation. Informally, an event is said to be observable if its occurrence can be detected by the supervisor. An event is controllable if its occurrence can be inhibited by the supervisor. Moreover, the integration of supervisors for different properties in order to guarantee correctness with respect to all properties considered must be explored.

This paper is organized as follows. In Section 2 we introduce some required definitions. Section 3 summarizes a method for enforcing mutual exclusion properties of generalized Petri nets. In Section 4, we present our paradigm for generating deadline-enforcing supervisors in time Petri nets. In Section 5, we discuss the potential advantages and disadvantages of supervisory control methods in software development for embedded systems.

2 Definitions

An *ordinary Petri net* is a four-tuple $N = (P, T, F, M_0)$ 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 N assigns a nonnegative number of tokens to each $p \in P$.

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 $N = (P, T, F, M_0)$ 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.

A *generalized* Petri net associates a positive weight *w* with each arc $f \in F$. If f goes from an input transition t_1 to an output place p, then *w* tokens are deposited in p whenever t_1 fires. If f goes from input place q to output transition t_2, then at least *w* tokens are needed in q in order for t_2 to be enabled. In this case, the firing of t_2 removes *w* tokens from q.

A *time Petri net* [1, 10] 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 $a \neq \infty$.

Static intervals change the behavior of a time Petri net with respect to an ordinary Petri net in the following way. 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 $[\theta_0 + a, \theta_0 + b]$.

The state of a time Petri net is a triple (M, Θ, I), where M is the marking of the underlying untimed Petri net, Θ is the global time, and I is a vector containing the dynamic firing interval of each transition enabled by M. The initial state of a time Petri net consists of its initial marking, time 0, and a vector containing the static firing interval of each transition enabled by this marking.

A *firing schedule* for time Petri net $\mathcal{N}$ 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 $\mathcal{N}$, and transition t_i is fireable at time θ_i from the state reached by starting in the initial state of $\mathcal{N}$ and firing the transitions t_j for $1 \leq j < i$ in the schedule at the given times.

3 Enforcing mutual exclusion

This supervisory control method enforces sets of linear mutual exclusion constraints on the reachable markings of the controlled net $\mathcal{N}$. For instance, if $\mathcal{N}$ has m transitions

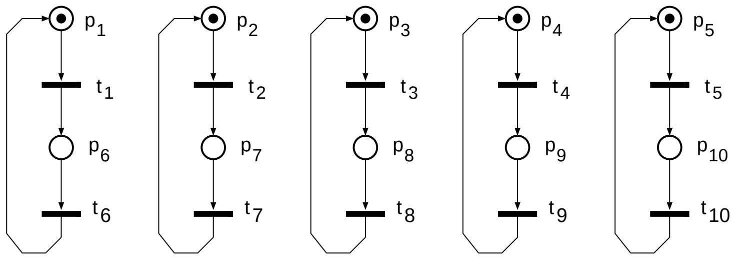

Figure 1: Example of controlled Petri net for readers and writers example.

and n places, each constraint may take the following form:

$$\sum_{i=1}^{n} l_i \cdot \mu_i \leq \beta \tag{1}$$

Variable μ_i represents the marking of place p_i, l_i is an integer coefficient, and β is an integer constant [4, 11, 17].

Given controlled net $\mathcal{N}$ and a set of linear constraints similar to inequality (1) above, this supervisory control method exploits a property of Petri net P-invariants. A P-invariant is a subset P_I of $\mathcal{N}$'s place set P such that the weighted sum of the tokens residing in places $p_i \in P_I$ remains constant in all reachable markings of $\mathcal{N}$. Inequality (1) can be transformed into a P-invariant equality by adding a slack variable μ_C:

$$\sum_{i=1}^{n} l_i \cdot \mu_i + \mu_C = \beta \tag{2}$$

Variable μ_C represents the marking of a control place p_C that enforces inequality (1). Consequently, a set of k linear inequalities can be enforced by a supervisory controller consisting of k control places and zero transitions.

The arc subset connecting P_C, the set of control places, to P, the place set in the controlled net, can be easily computed by a simple matrix multiplication. Let D be the $n \times m$ incidence matrix of a Petri net N with m transitions and n places. Entry $d_{i,j}$ is a positive (negative) integer w if N contains a weight w arc from transition t_j to place p_i to (from p_i to t_j). In general, the place invariants of N are the integer solutions to the following vector equation:

$$x^T \cdot D = 0^T \tag{3}$$

Here x^T is a transposed n-vector representing the integer coefficients of the net's place invariants and 0^T is a transposed m-vector filled with zeros.

Therefore, the P-invariants induced by a set of k inequalities (1) must satisfy the following equation:

$$[L \quad I] \cdot D = 0 \tag{4}$$

where L represents a $k \times n$ matrix containing the coefficients of inequality constraints (1), I is the unit matrix of size k and D is the incidence matrix of the net consisting of $\mathcal{N}$ and the supervisory controller. Clearly D has $n + k$ rows, where n is the number of places in $\mathcal{N}$, and m columns, one for each transition in $\mathcal{N}$:

$$D = \begin{bmatrix} D_N \\ D_C \end{bmatrix} \tag{5}$$

Here D_N is the $n \times m$ incidence matrix of $\mathcal{N}$ and D_C is the $k \times m$ incidence matrix of the supervisor net $\mathcal{S}$. From equations (4) and (5), we can find D_C as follows:

$$D_C = -L \cdot D_N \tag{6}$$

Thus, the desired supervisory controller can be found by a simple matrix multiplication involving the incidence matrix of the controlled net and the coefficients appearing in the inequality constraints to be enforced. The elements of D_C will be integers, as required, because L and D_N are integer matrices. Yamalidou et al showed that the supervisory controllers generated in this fashion are maximally permissive [17].

We illustrate the potential benefits of P-invariant-based supervisory control by applying this method to the readers and writers problem, a classical example of mutual exclusion. We consider a version with 3 readers and 2 writers. As usual, multiple readers are allowed in the buffer when no writer is in the buffer; however, each writer excludes both other writers and all readers from the buffer. Figure 1 shows a Petri net for a version in which the mutual exclusion constraints are not enforced. Places p_1, p_2 and p_3 represent the three readers in the idle state. When p_6, p_7 and p_8 have a token, the three readers are in the buffer. Likewise, places p_4 and p_5 represent the idle states of the two writers; a token in place p_9 or p_{10} means that a writer is in the buffer. To enforce the mutual exclusion constraints we write the following three inequalities:

$$\begin{aligned} p_6 + p_9 + p_{10} &\leq 1 \\ p_7 + p_9 + p_{10} &\leq 1 \\ p_8 + p_9 + p_{10} &\leq 1 \end{aligned}$$

The first constraint stipulates that at most one of the first reader and the two writers can be in the buffer simultaneously. The remaining two constraints stipulate the

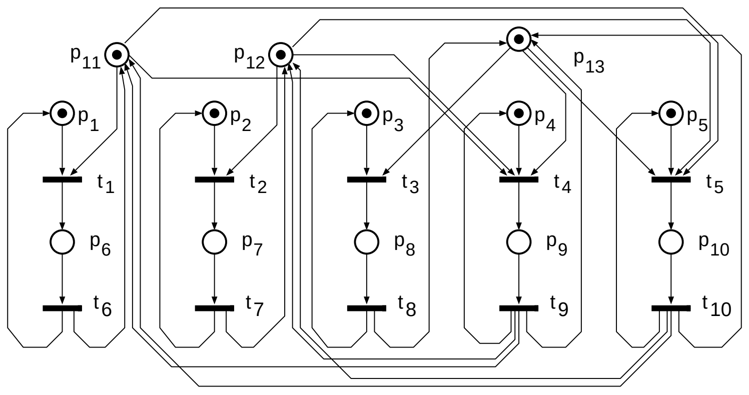

Figure 2: Petri net for readers and writers example with supervisory controller.

same condition for the second and third reader. Figure 2 shows the Petri net for the readers and writers example with the supervisory controller obtained by formula (6) above. Places p_{11}, p_{12}, p_{13}, and their incident arcs are the supervisor. In particular, place p_{11} enforces the mutual exclusion between the first reader and the two writers. Places p_{12} and p_{13} play a similar role for the second and third reader. The mutual exclusion among writers follows from each of the three constraints above.

The significance of this method is that it can enforce mutual exclusion constraints of a Petri net model in time polynomial in the size of the controlled net and the supervisor net. Therefore, the overall complexity will be polynomial whenever a mutual exclusion problem can be translated into a linear system containing a number of inequalities polynomial in the size of the controlled net. This performance is in sharp contrast with the verification of mutual exclusion properties, for which no general-purpose polynomial-time algorithm is known. When it is applicable, the approach based on supervisory control is likely to be vastly more scalable than verification. This method has also been extended to the case of Petri nets with unobservable and uncontrollable transitions [11]. Yamalidou et al defined other extensions including the case of "greater-than" constraints, constraints expressed as logical formulas, and constraints involving the firing vectors of the controlled net [17]. Finally, Iordache et al defined a method for enforcing net liveness (e.g., freedom from deadlock) in the controlled net [7].

4 Enforcing real-time deadlines

We report preliminary results on a method for enforcing real-time deadlines in time Petri nets [10]. Given a time Petri net $\mathcal{N} = (P, T, F, M_0, S)$, a net transition t_D, and a deadline λ, our method seeks to generate a supervisory controller that forces t_D to fire no more than λ time units since the latest of the previous firing of t_D and the beginning of a firing sequence. Throughout this section we assume that $\mathcal{N}$ is a safe and live time Petri net.

Our paradigm for generating deadline-enforcing supervisory controllers consists of three steps. First, we compute a so-called *transition latency* for each transition t in $\mathcal{N}$. Given a time Petri net $\mathcal{N} = (P, T, F, M_0, S)$, the latency $l(t)$ of a transition $t \in T$ is 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 $\mathcal{N}$. Thus, the latency of t is an upper bound on the time required for t_D to fire after t fires.

Second, we define a so-called *clock net* $\mathcal{C}$, a time Petri net whose places correspond to transition latencies identified earlier. A place in a clock net is used to disable dynamically transitions whose firing may prevent t_D from meeting λ. Of course, a transition t should be allowed to fire only when t's latency is no greater than the time left until the deadline on the firing of t_D expires.

Third, we synthesize a supervisory controller $\mathcal{S}$ based on nets $\mathcal{N}$ and $\mathcal{C}$. Controller $\mathcal{S}$ disables transitions in $\mathcal{N}$ based on the marking of places in $\mathcal{C}$. In particular, $\mathcal{S}$ dynamically disables transitions whose latency is greater than the time left until the deadline on the firing of t_D.

Consider, for instance, the time Petri net appearing in Figure 3. Suppose that target transition t_7 must be fired within 51 time units from the initial state. In order for t_7 to fire, transition t_2 must fire first. Since t_2 is in conflict with t_1, a supervisory controller must disable t_1 some time before the deadline expires. In this case, we can set the latency of t_2 to 25 time units, the sum of the static latest firing delays of t_3, t_5, and t_7. After t_1 fires, transitions t_3, t_4, t_6, t_2, t_3, t_5 and t_7 must be fired in order for the deadline to be met. The sum of their static latest firing times is 48 time units. Thus, it is safe to fire t_1 if at least 48 time units remain until t_7 must be fired.

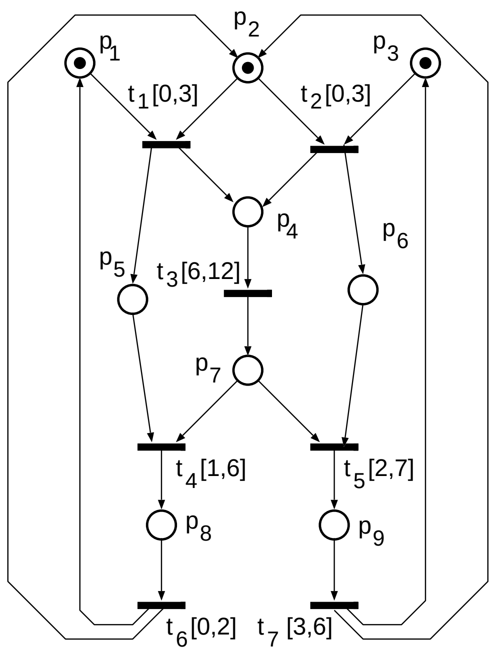

Figure 3: Example of a time Petri net.

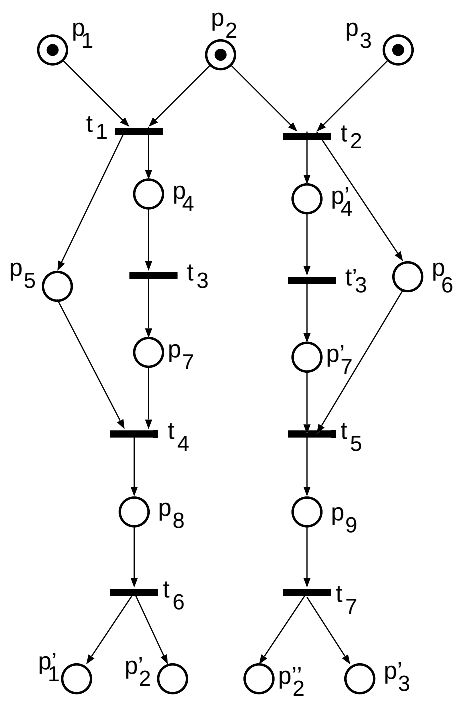

Figure 4: Unfolding of the Petri net appearing in Figure 3.

4.1 Computing transition latencies

We believe that various approaches can be followed when defining the latency of $\mathcal{N}$ transitions. Here we discuss a technique called *net unfolding* [3, 5, 9, 15]. We choose this technique for two reasons. First, net unfolding explicitly captures the causal relationship on transition firings for the Petri net under consideration. Thus, by unfolding net $\mathcal{N}$ we can define reasonably tight latency values. Second, unfolding $\mathcal{N}$ allows us to identify $\mathcal{N}$ transitions that need not be disabled in order for deadline λ to be met. In general, λ can be enforced by disabling only a small subset of transitions in the controlled net. For instance, it is sufficient to disable transition t_1 in a timely manner in order to force transition t_7 to fire in Figure 3. This fact can lead to reductions in the size of subnets $\mathcal{C}$ and $\mathcal{S}$ below.

We require the following definitions. Consider nodes x and y in an (untimed) ordinary Petri net. Node *x precedes y*, denoted by $x < y$ if there is a directed path from x to y in the Petri net. 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, node x is said to be in *self-conflict*. Nodes x and y are *concurrent* if they are not in conflict with each other and neither node precedes the other.

An *occurrence net* is an unmarked ordinary Petri net $\mathcal{O} = (P_O, T_O, F_O)$ subject to these conditions [3]:

1. $\forall p \in P_O$, p has at most one input arc.
2. $\mathcal{O}$ is acyclic.
3. Each node $x \in P_O \cup T_O$ is finitely preceded, meaning that the number of nodes $y \in P_O \cup T_O$ such that $y < x$ is finite.
4. No node $x \in P_O \cup T_O$ is in self-conflict.

Given a controlled net $\mathcal{N}$, consider $\mathcal{M}$, the ordinary Petri net underlying $\mathcal{N}$, so that $\mathcal{M} = (P, T, F, M_0)$. An *unfolding* of $\mathcal{M}$ is a marked, labeled occurrence net $\mathcal{U} = (P_U, T_U, F_U, M_{0U}, l_U)$, where l_U is a function mapping each node $x \in P_U \cup T_U$ to a node $l_U(x)$ in $\mathcal{M}$. In brief, each element of $\mathcal{U}$ is an "occurrence" of its image in $\mathcal{M}$. The formal definition of a net unfolding can be found elsewhere [3] along with algorithms for generating unfoldings of ordinary Petri nets. Here we simply report an example of a net unfolding.

Figure 4 shows an unfolding of the ordinary Petri net underlying the time Petri net in Figure 3. Places p_1, p_2, and p_3, which are initially marked, are mapped to the homonymous places in $\mathcal{N}$. Transitions t_1 and t_2 are mapped similarly. However, place p_4 in $\mathcal{N}$ is in self-conflict because it can be reached from p_2 either through transition t_1 or t_2. Thus, this place is represented by two places, p_4 and p_4', in Figure 4. Transition t_3 and place p_7 are also split into two nodes for the same reason. Finally, places p_1', p_2', p_2'', and p_3', represent the so-called *cut-off* points of the unfolding. When these places are marked, the net returns to its initial state.

We define transition latencies from the unfolding of the untimed net underlying controlled net $\mathcal{N}$. First, for each transition $t_u \in T_U$, the transition set of unfolding $\mathcal{U}$, we associate the static latest firing time of $l_U(t_u)$, the image of t_u in $\mathcal{N}$, with t_u. Second, we examine backward paths from each occurrence of t_D in $\mathcal{U}$ to the initial places of $\mathcal{U}$ and forward paths from t_D occurrences to the cut-off places of $\mathcal{U}$. We add the static latest firing times of the transitions that we find along these paths and we associate the partial sums with such transitions. Define c to be the longest backward path from a t_D occurrence in $\mathcal{U}$ to the initial places of $\mathcal{U}$. Define d to be the longest forward path from a t_D occurrence to the cut-off places of $\mathcal{U}$. Third, we consider paths from initial places to the cut-off places that do not include $\mathcal{U}$ transitions mapping into t_D. These paths may correspond to cyclic behaviors of net $\mathcal{N}$ in which t_D is not fired (e.g., if t_D is disabled along these paths). We add c to the combined delays along such paths. The resulting values yield the latencies for $\mathcal{N}$ transitions.

In the example appearing in Figure 4, we associate delays as follows: $t_1 \rightarrow 3$, $t_2 \rightarrow 3$, $t_3 \rightarrow 12$, $t_4 \rightarrow 6$, $t_5 \rightarrow 7$, $t_6 \rightarrow 2$, and $t_7 \rightarrow 6$ during the first phase of the algorithm. Next, we consider paths originating at target transition t_7. Since t_7 feeds directly into cut-off places p_2'' and p_3', we discard paths toward cut-off places. Backward paths from t_7 to initial places p_2 and p_3 yield the following latencies: $t_7 \rightarrow 0$, $t_5 \rightarrow 6$, $t_3 \rightarrow 13$, and $t_2 \rightarrow 25$. Finally, we consider the cycle involving firing sequence $\sigma = t_1, t_3, t_4, t_6$. The sum of the static latest firing delays along σ is 23. We add the latency of transition t_2 and the latest firing delay of t_2 to the delays computed on the cycle. This yields the following latency values: $t_1 \rightarrow 48$, $t_3 \rightarrow 36$, $t_4 \rightarrow 30$, $t_6 \rightarrow 28$. Transition t_3 is seemingly given two different latency values because this transition is in self-conflict. When this happens, we define the latency to be the least value.

4.2 Clock nets

We compute a clock net $\mathcal{C} = (P_C, T_C, F_C, M_{0C}, S_C)$ for a controlled net $\mathcal{N} = (P, T, F, M_0, S)$ based on the transition latencies and choice points previously defined with net unfolding. We specifically consider a subset $T_H \subseteq T$ of $\mathcal{N}$ transitions that are involved in choice points (i.e., because they share at least one input place).

First, we add to P_C one place for each distinct element in the set of latency values $L = \{v \mid \exists t \in T_H \text{ and } v = l(t)\}$. Given a latency value v, we denote the place corresponding to v by p_v. In addition, P_C contains a place p_λ corresponding to deadline λ and a place p_D for resetting the clock net after the firing of t_D. Figure 5 shows the clock net for the controlled net appearing in Figure 3. Here set T_H consists of transitions t_1 and t_2; places p_{25} and p_{48} map the latencies of these transitions in the clock subnet. Place p_{51} models deadline λ.

We define the transition set T_C, static delay intervals S_C, and flow relation F_C of $\mathcal{C}$ as follows. First, we insert a transition between pairs of clock net places with consecutive index values. The static delay of each such transition is the difference between the index values of its input and output place. In Figure 5 this yields transitions t_8 and t_9 with delays of 3 and 23. Next, we define an arc from t_D to p_D, and we add a group of $|P_C| - 1$ zero-delay transitions to T_C, one transition for each place $p_k \in P_C$, except for p_D. A token in p_D enables one of these transitions immediately after t_D fires. The transition removes the token from p_D and from one of the other places in P_C; it deposits a token in p_λ and in a suitable number of $\mathcal{S}$ control places described below. This completes the resetting of $\mathcal{C}$ and $\mathcal{S}$. In Figure 5, transitions t_{10}, t_{11}, and t_{12} reset the clock subnet and supervisor after t_7 fires. Additional details can be found elsewhere [2].

4.3 Supervisory controllers

Supervisory controllers enforce deadline λ on the firing of transition t_D in net $\mathcal{N} = (P, T, F, M_0, S)$ with clock net $\mathcal{C} = (P_C, T_C, F_C, M_{0C}, S_C)$. Let $P_V = P_C - \{p_D\}$. The supervisory control constraint is expressed by:

$$\text{Disable } t \in T_H \text{ if } p_v \in P_V \text{ marked with } l(t) = v \quad (7)$$

We note that by construction, for any marking of clock net $\mathcal{C}$, all places in P_V combined will always contain exactly one token. Unless t_D is fired, the token in P_V will always move toward places with lower index values. Constraint (7) above states that a transition $t \in \mathcal{N}$ is disabled whenever the token in P_V moves to a place p_v whose index v is equal to t's latency $l(t)$. Therefore, control constraint (7) above will disable all transitions that might delay the firing of t_D by more than v units, the index of the marked state of $\mathcal{C}$. Moreover, once t is disabled, t is not allowed to fire again until after target transition t_D has been fired. As a result, all the transitions that may result in the violation of deadline λ on the firing of t_D are disabled.

We implement constraint (7) above by defining two places, q_1 and q_2, and one transition r for each $t \in T_H$. The rules for defining arcs incident on q_1, q_2 and r are discussed elsewhere [2].

Figure 5 shows the supervisory controller for the net in Figure 3. This controller disables transitions t_1 and t_2 when transitions t_8 and t_9 are fired. For instance, when t_8 fires place c_3 becomes marked, which enables transition t_{13}. The firing of t_{13} causes the removal of the token from place c_1; this action disables transition t_1. Transition t_9

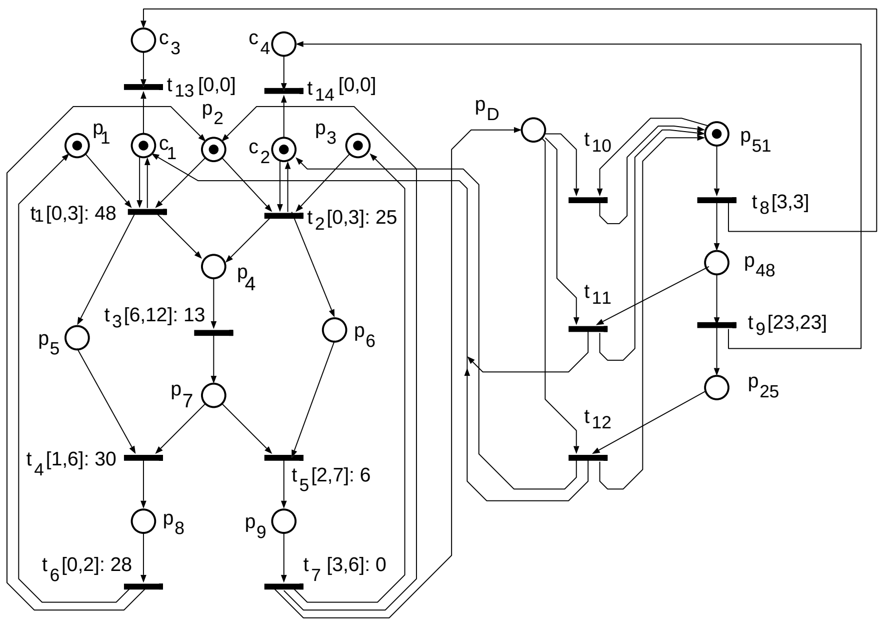

Figure 5: Supervisory controller and clock net of time Petri net appearing in Figure 3.

similarly disables transition t_2. Additional details can be found elsewhere [2].

5 Assessment

The two methods discussed earlier indicate that supervisory control may have significant benefits on the development of software for concurrent and real-time systems. The most significant advantage of supervisory control is reduced computational complexity with respect to the corresponding verification algorithms. For instance, in Section 3 we saw that a broad variety of mutual exclusion constraints can be enforced in time polynomial in the size of the system under consideration. This is in sharp contrast to the verification of mutual exclusion properties, which is computationally intractable. Although we lack empirical data on the real-time method discussed in Section 4, we believe that on average this method will also be tractable. The verification of real-time systems is generally considered even more complex than the case of untimed concurrent systems. When they are applicable, supervisory control methods may provide greater help to developers of embedded systems than existing techniques.

However, the full potential of supervisory control in software development is more far-reaching than just guaranteeing that certain correctness properties are met. The availability of supervisory control tools could free programmers from the need to build compliance with correctness properties directly into their code. The version of the readers and writers example shown in Figure 1 is a case in point. This version could be obtained by translation from software that was deliberately written without paying attention to its mutual exclusion constraints. However, a supervisory controller can subsequently enforce these and other properties that are expressed as linear equalities and inequalities on net markings and firing vectors.

Thus, the use of supervisory control methods could lead a new programming paradigm for concurrent and real-time systems. In this paradigm, a programmer would first write a version of the program without being concerned about complying with mutual exclusion and real-time properties. Next, the programmer would submit this program along with a control specification to a supervisory control tool. The tool would then translate the program into a Petri-net model and generate suitable supervisors. Finally, the tool would add code that enforces the control specification to the original code.

While supervisory control methods hold considerable promise for the development of concurrent and real-time systems, the widespread application of these methods also faces formidable obstacles. Petri net transitions may not be observable and/or controllable to the extent needed for supervisor definition. This could happen, for instance, in wireless sensor networks, special kinds of embedded systems [12]. These networks often lack the ability for a node (i.e., an embedded system equipped with sensors) to know instantaneously and control events in different nodes.

Additional obstacles may arise when attempting to integrate multiple supervisory control methods in order to enforce different properties. For instance, it is currently unclear whether the two methods that we discussed earlier can be effectively combined in an effort to guarantee simultaneously mutual exclusion *and* real-time properties.

Liveness properties, such as freedom from deadlock, pose additional challenges to the application of supervisory control methods. Although freedom from deadlock is an intractable verification problem, this is considered the "easiest" property to check through verification. The same does not hold in the world of supervisory control; the definition of supervisors for enforcing Petri net liveness is much more challenging than, say, enforcing mutual exclusion properties expressed as linear constraints [7]. A method by He and Lemmon, who use net unfoldings to enforce liveness, seems especially promising [5].

To date, several research issues must be investigated in order to answer some of the questions regarding the applicability of supervisory control to software development. First, additional supervisory control methods must be defined for enforcing different properties. In the case of the readers and writers example, it is quite conceivable to define versions in which the readers or the writers have priority or in which read and write requests should be handled in FIFO order. Supervisory control strategies for these kinds of specifications are generally not available yet. Second, we must collect empirical data on the applicability of supervisory control in software development. At the very least, we should find out how often mutual exclusion constraints can be expressed through a small number of linear constraints in the form (1). The complexity of net unfolding when applied to real-world software problems must also be assessed empirically because this techique is crucial both to liveness-enforcing and deadline-enforcing supervisors.

6 Conclusions

We briefly summarized two supervisory control methods for concurrent and real-time systems. Although these methods have not reached the level of maturity needed to permit the creation of tools for software development, they hold considerable promise because they are generally more tractable than the corresponding verification algorithms. For these reasons, we should investigate research directions that may lead to widespread applications of supervisory control in software development for concurrent and real-time systems, such as embedded systems.

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] 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.

[4] A. Giua, F. DiCesare, and M. Silva. Generalized mutual exclusion constraints for nets with uncontrollable transitions. In *Proceedings IEEE Int. Conf. on Systems, Man, and Cybernetics*, pages 974–979, Chicago, Illinois, Oct. 1992.

[5] 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.

[6] L. E. Holloway, B. H. Krogh, and A. Giua. A survey of Petri net methods for controlled discrete event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 7:151–190, Apr. 1997.

[7] M. V. Iordache, J. O. Moody, and P. J. Antsaklis. Synthesis of deadlock prevention supervisors using Petri nets. *IEEE Transactions on Robotics and Automation*, 18(1):59–68, 2002.

[8] R. W. Lewis. Programming industrial control systems using IEC 1131-3. Technical report, The Institution of Electrical Engineers, 1998.

[9] K. L. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design*, 6(1):45–65, Jan. 1995.

[10] 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.

[11] J. O. Moody and P. J. Antsaklis. Petri net supervisors for DES with uncontrollable and unobservable transitions. *IEEE Transactions on Automatic Control*, 45(3):462–476, Mar. 2000.

[12] G. J. Pottie and W. J. Kaiser. Wireless integrated network sensors. *Commun. ACM*, 43(5):51–58, May 2000.

[13] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal of Control and Optimization*, 25(1):206–230, 1987.

[14] A. S. Sathaye and B. H. Krogh. Supervisor synthesis for real-time discrete event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 8, 1998.

[15] A. Semenov and A. Yakovlev. Verification of asynchoronous circuits using time Petri net unfolding. In *Proceedings of the 33rd Design Automation Conference (DAC96)*, pages 59–62, Las Vegas, Nevada, June 1996.

[16] R. N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Inf.*, 19:57–84, 1983.

[17] K. Yamalidou, J. Moody, M. Lemmon, and P. Antsaklis. Feedback control of Petri nets based on place invariants. *Automatica*, 32(1):15–28, 1996.