DEADLOCK DETECTION WITH STUBBORN UNFOLDINGS

Haisheng Wang and Ugo Buy
Department of Computer Science
University of Illinois at Chicago
851 South Morgan Street
Chicago, IL 60302

ABSTRACT
Stubborn sets and model unfolding are two well-known methods for sidestepping the complexity of state-space exploration based on formal models of concurrent programs, such as Labeled Transition Systems and Petri nets. Here we define a deadlock detection strategy that combines these two methods. We also show that the resulting method, *stubborn unfoldings*, can produce smaller state space representations than both previous methods.

KEY WORDS
Formal methods; Verification and validation; Petri nets; Deadlock detection; Stubborn sets; Unfolding.

1 Introduction

Concurrent systems contain multiple asynchronous processes that can be executed in parallel. Examples of these systems are found in client-server applications, in software for embedded systems and sensor networks, and in programs running on multi-processor architectures. The behavior of these systems is often subject to random factors, such as arbitrary delays over communication lines, the impredictable occurrence of interrupts on a given processor, and the duration of processor computations. The resulting nondeterminism complicates testing and debugging of concurrent software. Additional complications arise from special error conditions that may arise in the case of concurrent systems, such as *deadlocks*. For these reasons automatic verification is viewed as an attractive alternative to testing in the case of concurrent software.

Concurrency verification is usually aimed at discovering potential violations of concurrency properties, such as freedom from deadlock and compliance with mutual exclusion constraints. Concurrency verification is typically based on formal models, such as finite-state automata and Petri nets. These models abstract away irrelevant details of program behavior while focusing on aspects that can affect compliance with desired concurrency properties. A drawback of these models is that their state space is often subjected to the notorious *state explosion problem*. The number of states reachable in the model often grows exponentially in the size of the model.

Stubborn sets and *unfoldings of transition systems* are two well-known methods for sidestepping the complexity of state-space exploration. Both methods rely on the notion of *independent* transitions contained in a formal model, such as a set of communicating finite state automata or a Petri net. Two transitions are considered independent if the execution of one transition does not affect the execution of the other transition and vice versa. Independent transitions can often be executed in an arbitrary order while reaching the same final state.

Stubborn sets and unfoldings differ in the way they exploit the notion of independent transitions to achieve reductions in the resulting state space representations. The goal of stubborn sets is reduced state space generation. This method produces a reachability graph that does not include all the reachable states. Graph nodes represent reachable states; arcs represent state transitions. Given a set of transitions that can be executed concurrently and independently of each other, it is often sufficient to explore just one possible interleaving of these transitions to check a property of interest. This is true, for instance, when only the final state reached by executing all independent transitions is relevant to the property. Significant state space reductions are sometimes obtained by avoiding the explicit enumeration of all intermediate states [8].

Similar to stubborn sets, model unfolding exploits the parallelism contained in the original model to sidestep the complexity of state space representations; however, unfoldings do not represent states explicitly. Instead, unfoldings capture the possible execution sequences of the processes contained in a concurrent program. Unfoldings have several benefits for concurrency analysis. First, unfoldings are acyclic graphs, making it possible to use efficient algorithms for the analysis of Petri nets based on linear-algebra [6, 7]. Second, unfoldings show explicitly the causal relationships on the execution of transitions in the underlying model. This property supports efficient analysis methods such propositional solvers [2] and supervisory control of real-time systems [9]. Third, unfoldings are typically much smaller than explicit state space representations of their underlying models.

Here we define *stubborn unfoldings*, a new analysis method that combines the strengths of the two existing methods, and we show that stubborn unfoldings can in fact produce smaller state space representations than both stubborn sets and the original unfoldings. Our goal here is to detect global deadlocks in the original Petri net using stub-

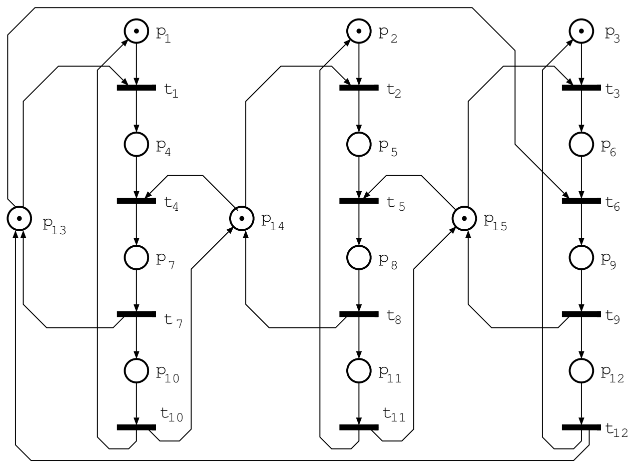

Figure 1. Example of Petri for dining philosphers example.

born unfoldings.

This paper is organized as follows. In Section 2 we define the concept of a Petri net and show an example. In Section 3 we summarize stubborn sets. In Section 4 we briefly discuss existing unfolding methods. In Section 5 we define our algorithm for creating stubborn unfoldings. Finally, in Section 6 we give some conclusions and outline future research directions.

2 Petri Nets

Petri nets are a formal, graph-based model of concurrent systems. The Petri net formalism can capture naturally the main features of these systems including parallel computation, flow of control, inter-process communication and synchronization, and nondeterministic choice [7]. Several variations of Petri nets exist; here we use models called ordinary Petri nets.

An *ordinary Petri net* is a directed, bipartite graph with two disjoint node sets, namely places and transitions. Places can hold so-called tokens, which mimic flow of control in asynchronous processes. Formally, a Petri $\mathcal{N}$ is a 4-tuple $\mathcal{N} = (P, T, F, M_0)$, where P is a finite set of places, T is a finite set of transitions, F is an arc set, and M_0 is an initial marking, that is, an initial assignment of tokens to each place $p \in P$. Given an arc f from a place p (a transition t) to a transition t (a place p), p is said to be an input (output) place for t, and t is an output (input) transition for p.

State changes are carried out by firing *enabled* transitions. 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 of t; this gives a new state (i.e., a new marking). A net marking is *dead* if there are no enabled transitions. A Petri net is *safe* if no place has more than one token in all reachable markings.

Figure 1 shows an example of an ordinary Petri net for the well-known dining philosophers example with three philosophers. In this example, three philosophers are sitting around a table. A fork is placed between each pair of philosophers. Philosophers alternate between thinking and eating, for which they need both adjacent forks. Places p_1, p_4, p_7, and p_{10} on the left of the figure model the first philosopher. Places in the middle and right of the figure model the second and third philosophers. Places p_{13}, p_{14}, and p_{15} model the three forks. Initially, these three places are marked to indicate that the forks are available. Transition t_1 is enabled in the initial marking. When this transition fires, tokens are removed from p_1 and p_{13}, and a token is added to p_4 signifying that Philosopher 1 has picked his left fork. Now, transition t_4 will be enabled. The firing of this transition removes tokens from places p_4 and p_{14}. A token is added to p_7 signifying that Philosopher 1 has picked up his right fork as well. The firing of transitions t_7 and t_{10} returns tokens to p_1, p_{13}, and p_{14}, signifying that Philosopher 1 has put down both forks.

The dining philosophers problem contains a deadlock, which occurs when all philosophers pick up their left fork. In the Petri net, this is captured by the state in which places p_4, p_5, and p_6 are marked. No transitions are enabled.

3 Stubborn Sets

The goal of the stubborn set method is the automatic generation of a reduced state space of a variable/transition system, such as a Petri net. The reduction in state space size is achieved by partitioning, in each reachable state, the transition set T of the Petri net into two independent subsets, namely the stubborn set and the nonstubborn set. When expanding a graph node *n* during reachability graph construction, only enabled transitions that belong to the chosen stubborn set are considered. As a result, only a portion of the state space of the Petri net is explicitly included in the resulting reachability graph.

Formal conditions for a transition subset to be stubborn in a net state are given elsewhere [8]. In brief, a stubborn set must contain at least one enabled transition. Also, an enabled transition contained in a stubborn set cannot be disabled by the firing of nonstubborn transitions (i.e., transitions in the nonstubborn set). Finally, a disabled transition contained in a stubborn set should not become enabled by firing only nonstubborn transitions.

In general, a given net state can be associated with multiple stubborn sets (i.e., multiple ways of partitioning the transition set into a stubborn and nonstubborn set). In the net shown on the left-hand side of Figure 2, the singleton sets $\{t_1\}$, $\{t_2\}$, and $\{t_3\}$ are all stubborn because each set satisfies the criteria that we just summarized. When a state has multiple stubborn sets, one of the stubborn sets is selected arbitrarily, and only the enabled transitions contained in that set are fired. This strategy leads to the state space reduction shown in Figure 2. The middle portion of Figure 2 shows the full state space of the net shown on the

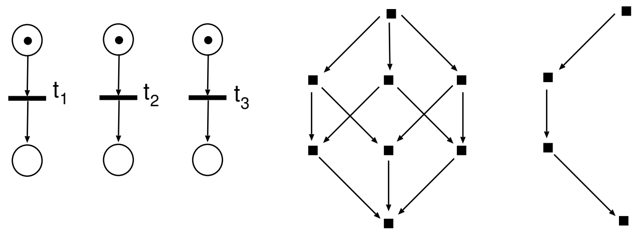

Figure 2. Petri net example along with its unreduced and reduced state spaces.

left. The initial state has three possible successors, depending on whether transition t_1, t_2 or t_3 is fired. The overall state space has a number of reachable states exponential in the number of transitions in the original Petri net. The right-hand side shows a state space that could be obtained by applying the stubborn set method. This state space has a number of states linear in the number of transitions in the Petri net.

An important strength of the stubborn set method is that it can take advantage of the parallelism that might exist in the specification of a concurrent system when building its state space. Independent transitions in a Petri net model generally correspond to events that can occur concurrently in the underlying system. When a model contains a high degree of parallelism, this method can result in a dramatic reduction in the number of states that are generated explicitly.

4 Net Unfoldings

The unfolding of a Petri net is similar to the concept of unfolding a general graph. In short, the unfolding of a graph is a (generally infinite) tree structure starting from an initial node and showing all possible node paths in the graph. Subsequent to McMillan's original definition [5], Esparza et al. exploited properties of Petri nets to reduce the size of net unfoldings [3]. Khomenko et al. [4] defined generalized conditions for truncating unfoldings. Finally, Esparza and Heljanko gave a comprehensive discussion of unfoldings and their applications [2]. Our definition follows McMillan's [5].

We require the following definitions. Consider nodes x and y in an ordinary Petri net. Node *x precedes y*, denoted by $x < y$, if there is a directed path from x to y in the net. Nodes x and y are *in conflict* with each other, denoted by $x\#y$, if the Petri net contains two distinct paths that start at the same place p but diverge immediately after p and lead to x and y. When $x\#x$ holds, we say that node x is in *self-conflict*. Nodes x and y are *concurrent* if they are not in conflict with each other and neither one precedes the other.

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

1. $\forall p \in P_O$ the in-degree of p is at most 1.
2. $\mathcal{O}$ is acyclic.
3. No node $x \in P_O \cup T_O$ is in self-conflict.

Given a Petri net $\mathcal{N} = (P, T, F, M_0)$, an *unfolding* $\mathcal{U}$ of $\mathcal{N}$ is a pair consisting of a marked, labeled occurrence net and a homomorphic function l_U mapping each node in $\mathcal{U}$ to the corresponding node in $\mathcal{N}$. Given an unfolding $\mathcal{U} = (P_U, T_U, F_U, M_{0U}, l_U)$, l_U maps $x \in P_U \cup T_U$ to a node $l_U(x)$ in $\mathcal{N}$. The formal properties of l_U are discussed elsewhere [9]. In brief, each node of unfolding $\mathcal{U}$ is an "occurrence" of its image in $\mathcal{N}$. Here we just show an unfolding example.

Figure 3 shows an example of a simple Petri net on the left and its unfolding on the right. Unfolding nodes mapped to a given Petri net node are shown by their labels. For instance, unfolding places p_1, p_2, and p_3 on the right are mapped to the homonymous places in the Petri net on the left. Unfolding transitions t_1 and t_2 are mapped similarly. Petri net nodes p_4, t_3, and p_7 on the left are in the self-conflict because they can be reached through multiple paths originating at p_2. Thus, the unfolding on the right uses multiple copies of these nodes in order to prevent self-conflicts. For instance, Petri net place p_4 is mapped by two unfolding places, p_4 and p'_4, in Figure 3.

An important property of unfoldings is that an unfolding has the same set of reachable states (markings) as the corresponding Petri net, modulo the mapping of unfolding places to Petri net places. This is clearly the case for the net shown in Figure 3 and its unfolding. Finally, transitions t_6 and t_7 are two cutoff points for the unfolding. Consider the unfolding marking obtained by firing transition t_6 and transitions preceding t_6 in the unfolding. In general this place set is called the *cut* of an unfolding transition. The cut of t_6 contains places p'_1, p'_2, and p_3. This marking is identical, modulo homomorphism, to the initial marking of the unfolding in which unfolding places p_1, p_2 and p_3 are marked. In general, unfolding construction can stop when a repeated marking is reached. Formal conditions for terminating unfolding construction are discussed elsewhere [5].

5 Stubborn Unfoldings

The goal of our method is to produce a reduced unfolding by combining the traditional rules for unfolding construction with reduced state space exploration [3, 8]. Traditional algorithms for unfolding construction start by adding

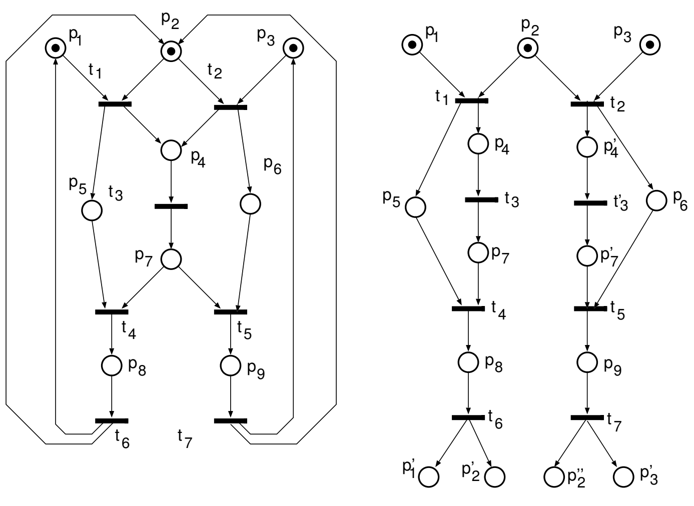

Figure 3. Example of Petri net (on the left) and its unfolding (on the right).

initially-marked places to the unfolding. Next, the transitions enabled by those places are identified and added to the unfolding, along with their output places. Each time a transition t is added to an unfolding, the transition is checked for the cutoff condition. In general, if the cut of t (i.e., the unfolding marking obtained by firing t and all transitions preceding t in the unfolding) is identical—modulo homomorphism—to either the initial unfolding marking or the cut of a transition preceding t, t is a cutoff transition and its output places have no successors. If, however, t is not a cutoff transition, all transitions enabled by newly-added places are identified; these transitions are generally included in the unfolding unless its input places include an output place of a cutoff transition. Unfolding produced using the traditional definitions have all and only the states of the corresponding Petri net, after unfolding places are mapped to Petri net places.

Our stubborn unfoldings differ from traditional unfoldings in that we do not add all transitions enabled in the unfolding under construction to the unfolding. Instead, we partition such enabled transitions into a stubborn set and a non-stubborn set in a way similar to the stubborn set method for reduced state space enumeration. When multiple partitions are possible, we use a heuristic algorithm to select one stubborn set and we add only enabled transitions contained in that stubborn set to the unfolding.

Figure 4 illustrates our algorithm for detecting global deadlocks using stubborn unfoldings. The algorithm uses a stack S of open transitions to be explored. Each item in the stack contains two items: (1) a state (marking) $\overline{m}$ in the Petri net underlying unfolding $\mathcal{U}$, and (2) the enabled transitions contained in a stubborn set of $\overline{m}$. In general, all transitions contained in each stack item must be included in an unfolding.

Initially, unfolding $\mathcal{U}$ is empty. According to the first step in Figure 4, the initially marked Petri net places (i.e., the places contained in M_0) are added to $\mathcal{U}$. Next, we find the transitions enabled in this state and compute a stubborn set for this state. In general, this stubborn set will contain fewer enabled transitions than the total number of enabled transitions in the state. Only the stubborn, enabled transitions are pushed on stack S, along with state M_0, for inclusion in the unfolding. If, however, M_0 contains no enabled transitions, the initial Petri net state is a deadlock state; unfolding construction is complete; and the deadlock is reported.

Step 3 in Figure 4 begins the main loop of the unfolding construction algorithm. This step first checks stack S. If S is empty, unfolding construction terminates successfully and the Petri net is free of deadlock. Otherwise, the transition set in the top item of S is checked. If this set is empty, the stack is popped and we repeat Step 3. Otherwise, we remove a transition t from the top item and we add t to the unfolding in Step 5, unless t was already included in a previous step. In Step 6 we check whether t is a cutoff transition. In this case, we return to Step 3. Otherwise, in Step 7 we compute the new Petri net marking m obtained by firing t from the state, $\overline{m}$, in which t was enabled. In Step 8 we compute a stubborn set for m and we push a new item on the stack. This item is a pair consisting of m and all enabled transitions in the stubborn set. If, however, no transition is enabled, the stubborn set cannot be computed and a dead state is found. Finally, we return to the start of the main loop in Step 9.

The most complex steps in Figure 4 are (1) the computation of the stubborn sets for a given net marking and (2) the check whether an unfolding transition is a cutoff transition. For the computation of the stubborn sets we extend heuristics outlined by Valmari [8]. In general there are multiple ways of partitioning transitions enabled in a state into a stubborn set and a non-stubborn set. The size of the overall reduced state space depends on the choice of a particular

1. Add images of places initially marked in Petri net to unfolding $\mathcal{U}$; find transitions enabled in initial marking.
2. Find a stubborn set of transitions in the initial marking and push these transitions and initial marking on stack S. If no stubborn set is found, no transition is enabled in the initial state, the Petri net is dead, and the algorithm is exited.
3. Check stack S. If S is empty, exit algorithm with success. If S is not empty, check whether the top item of S contains any enabled transitions. If there are no enabled transitions, pop S and repeat Step 3. Otherwise, remove a transition t from the top item and copy the corresponding Petri net marking $\overline{m}$.
4. If t is currently not in the unfolding, add a t image along with output places to $\mathcal{U}$.
5. Check whether t is a cutoff transition. If t is a cutoff transition, go to Step 3. If t is not a cutoff transition go Step 6.
6. Find the new Petri net marking m obtained by firing t in the state enabling t based on $\overline{m}$ and the input/output arc sets of t.
7. Find a stubborn set of transitions for m and push these transitions along with m on S. If no stubborn set is found, no transition is enabled in m, a deadlock state is reported, and the algorithm is exited.
8. Go back to Step 3.

Figure 4. Algorithm for constructing stubborn unfoldings.

stubborn set during expansion of each state. Similar to Valmari, we choose a stubborn set with a minimal number of enabled transitions. Although this does not guarantee that we will always obtain the smallest unfolding possible, in general we do obtain such an unfolding.

We check for cutoff transitions as follows. Following McMillan [5], we say that an unfolding transition t is a cutoff transition if its cut is identical—modulo homomorphism—to either the set of initial unfolding places, or the cut of a transition preceding t in the unfolding. Whenever we add a transition t to $\mathcal{U}$, we store t's cut in a hash table. When a transition t is added to an unfolding along with its output places, it is easy to compute t's cut from the cut of transitions preceding t in the unfolding and to check this cut against existing cuts stored in the hash table.

Figure 5 shows the stubborn unfolding of the dining philosophers example appearing in Figure 1. The stubborn unfolding contains 12 transitions and 24 places. This unfolding is much smaller than the full unfolding, which contains 21 transitions and 39 places according to the PEP toolset [2]. Better yet, the full unfoldings of the dining philosophers example grow exponentially in the number of philosophers and forks in the example [6]. In contrast with full unfoldings, our stubborn unfoldings grow linearly in the number of philosophers and forks. This result also beats the traditional stubborn set method for reduced state space enumeration, which yields a quadratic growth in state space size [8]. (The full, unreduced state space is $O(3^n)$ where n is the number of philosophers and forks.)

Similar results can be obtained from the example of Figure 3. Although the full unfolding of the Petri net as shown is relatively small, a slightly modified version of this net has exponentially-sized unfoldings. Suppose, for instance, that the arc from transition t_6 to place p_2 in the left-hand side of Figure 3 is replaced by an arc from transition t_4 to p_2. Likewise, replace the arc from transition t_7 to p_2 with an arc from transition t_5 to p_2. In this case, the size of traditional unfoldings explodes as the example is scaled up; however, our stubborn unfoldings have linear growth.

We note that the algorithm in Figure 4 does not always seek to complete unfolding construction. This is so because the purpose of the algorithm is to detect whether a global deadlock is possible or not in the original Petri net. For instance, the algorithm is exited as soon as a deadlock state is found. Alternatively, the algorithm is also exited when a transition sequence is found that can fire indefinitely and that cannot be disabled by other net transitions. In this case, a global deadlock cannot occur even though some net transitions may no longer be fireable, for instance, because of a partial deadlock.

A proof of correctness of our algorithm for constructing a stubborn unfolding is under active investigation. In brief, the correctness of the algorithm is tied to the fact that our stack S explores a set of Petri net states similar to the original stubborn set method. However, the state space representation based on unfolding can be much smaller than explicit state space enumerations that are used in the original stubborn set method.

6 Conclusions

We have presented a method for combining effectively two existing methods for reduced state space exploration, namely stubborn sets and unfoldings of transition systems. The results of our preliminary investigations are quite encouraging. Stubborn unfoldings are nearly always smaller (sometimes much smaller) and never bigger than traditional unfoldings. In addition, stubborn unfoldings can be much smaller than reduced state spaces (i.e., reachability graphs) obtained with stubborn sets. An implementation of the stubborn unfolding method is currently under way.

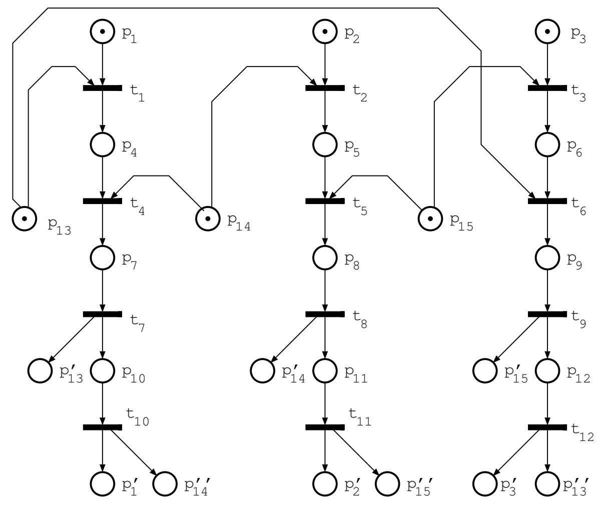

Figure 5. Stubborn unfolding of dining philosopher Petri net shown in Figure 1.

When the implementation is complete, we plan to conduct extensive empirical studies to compare the efficiency of our method with the original stubborn sets and unfolding method. In addition, we are investigating the possibility of integrating additional partial-order methods such as ample sets and persistent sets [1].

References

[1] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *Software Tools for Technology Transfer*, 2(3):279–287, 1999.

[2] J. Esparza and K. Heljanko. *Unfoldings: A Partial-Order Approach to Model Checking*. Springer-Verlag, Berlin, Germany, 2008.

[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] V. Khomenko, M. Koutny, and W. Vogler. Canonical prefixes of Petri net unfoldings. *Acta Informatica*, 40(2):95–118, October 2003.

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

[6] S. Melzer and S. Romer. Deadlock checking using net unfoldings. In O. Grumberg, editor, *Computer Aided Verification: 9th Internat. Conf., CAV 97*, volume 1254 of *LNCS*, pages 352–363. Springer-Verlag, 1997.

[7] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr. 1989.

[8] A. Valmari. A stubborn attack on state explosion. In E. M. Clarke and R. P. Kurshan, editors, *Computer-Aided Verification: 2nd Internat. Conf., CAV '90*, LNCS 531, pages 156–165. Springer-Verlag, 1991.

[9] H. Wang, L. Grigore, U. Buy, and H. Darabi. Enforcing transition deadlines in time Petri nets. In *Proc. 12th IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA 2007)*, pages 604–611, Patras, Greece, Sept. 2007.