In: *Proceedings of the 1994 Internatinal Symposium on Software Testing and Analysis (ISSTA '94)*, pp. 228–239, August 1994, ACM Press.

Analysis of Real-Time Programs with Simple Time Petri Nets

Ugo Buy and Robert H. Sloan
Electrical Engineering and Computer Science (M/C 154)
University of Illinois
851 South Morgan Street
Chicago, Illinois 60607
Email: buy|sloan@eecs.uic.edu

Abstract

We present a first report on our PARTS toolset for the automated static analysis of real-time systems. The PARTS toolset is based upon a timed extension of Petri nets.

Our *simple time Petri nets* or STP nets are specifically aimed at facilitating real-time analysis. Our analysis approach uses the state space of an STP net in order to answer queries about the concurrency and timing behavior of the corresponding system. An attractive feature of STP nets is that they support a variety of techniques for controlling the number of states that must be explicitly enumerated. These techniques were originally defined for the analysis of concurrency properties of untimed systems, and in this paper we discuss the extension of each to the timed domain.

We also report on some preliminary experimental results that we obtained by running our toolset on examples of real-time systems.

1 Introduction

Real-time systems are characterized by the presence of timing constraints on program behavior. The correctness of these systems depends not only on their functional behavior, but also on their ability to meet the timing constraints. Because real-time systems are frequently used in safety-critical applications, it is crucial that these systems be correct. Unfortunately, the behavior of a real-time system is often affected by the presence of multiple asynchronous processes, and the resulting nondeterminism complicates testing and debugging. Furthermore, testing and debugging cannot guarantee that a system is correct; instead, they merely provide probabilistic evidence of correctness. For these reasons static analysis, whose goal is to *guarantee* that a real-time system satisfies its timing constraints, is an important area of investigation.

In this paper we present a preliminary report on the Petri-net-based Analyzer of Real-Time Software, or PARTS, toolset for automatic analysis of real-time programs. The PARTS toolset takes an Ada program and a set of queries about its timing and concurrency behavior as inputs, translates to an internal Petri-net-based model, and answers the query using reachability analysis. Here we describe the heart of the toolset, which must perform reachability analysis on a large timed extension of a Petri net, and therefore must overcome the infamous *state explosion problem*.

Many timed extensions of Petri nets have been defined [3, 7, 11, 13, 22]. As we shall explain, we believe that our *simple time Petri nets* (or STP nets) are especially well-suited for real-time analysis [3]. The PARTS toolset is based on the construction of the state space of an STP net describing a real-time system. The state space is represented as a weighted, directed graph whose nodes correspond to reachable program states and whose arc weights capture the time taken by transitions from one state to the next.

Using the state space, we can answer user timing queries of the form "What is the longest/shortest path from state α to state β?" as well as user-specified safety queries of the form, "Does the system satisfy (untimed) property P in all states?"

In order to circumvent the state explosion problem we have investigated a set of optimization techniques. The common goal of these techniques is to reduce the number of states that must be explicitly generated during reachability-graph construction. To date we have considered the following three techniques: Net reduction, stubborn sets, and net symmetry. These techniques were originally introduced for the analysis of safety properties of (untimed) concurrent systems [16, 18, 20]. As past experiments have shown, these techniques are very successful at reducing state space size in Petri-net-based deadlock analysis of Ada tasking programs [6].

The PARTS toolset combines real-time versions of all three of the above techniques. None of these techniques was designed for real-time applications; much of our work has been extending them to the real-time context. While extending the net symmetry method [16] to real-time analysis is relatively straightforward, the net reduction method and the stubborn set method require special attention for the following two reasons. First, we need to guarantee that the techniques do not change properties of interest in the system being analyzed. Second, we seek to exploit the structure of our STP nets in order to achieve additional state space reductions with respect to the original techniques. Our investigations have led to real-time extensions of the optimization techniques we considered. In addition, we have implemented the resulting techniques in the PARTS toolset for STP-net-based analysis.

In this paper, we discuss how net reduction, stubborn sets, and net symmetry can be extended to real-time analysis using STP nets. We also report on preliminary experimental data that we have obtained by applying our prototype set to various examples of real-time systems. Our results indicate that none of the techniques that we considered can single-handedly make reachability-based analysis of real-time systems tractable. In most examples, however, the various techniques are mutually beneficial for the overall efficiency of the analysis. We conclude that all these techniques should be used in combination in order to make reachability-based analysis tractable.

We note that the problem of real-time analysis is generally intractable in the worst case. Our method for real-time analysis can be forced to solve the PSPACE-complete problem, "Given a safe untimed Petri net, is a given marking reachable?" [10]. Moreover, any reasonably general analysis method that includes the ability to perform even untimed concurrency analysis will be solving *NP*-hard problems [17]. Our goal here is simply to find a combination of heuristics that has good average-case performance on those problems that arise in practice.

This paper is organized as follows. In Section 2 we introduce the STP-net model and discuss the basic approach to real-time analysis using this model. The net reduction technique for STP nets is presented in Section 3. The main aspects of real-time stubborn sets are discussed in Section 4. In Section 5 we briefly summarize the net symmetry method. Our experiments are discussed in Section 6. In Section 7 we evaluate the effectiveness of the optimization methods that we used. In Section 8 we compare STP nets with alternative Petri net models, including time Petri nets [11] and time-basic nets (or TB-nets) [7]. Finally, some future research directions are given in Section 9.

2 Simple time Petri nets

2.1 Definitions

We require the following definitions. An *ordinary Petri net* is a 4-tuple $N = (P, T, F, M_0)$, where P and T are the node sets of a directed bipartite graph, F is the arc set, and M_0 assigns a nonnegative integer to each element of P. We call P the set of *places*, T the set of *transitions*, F the *flow relation*, and M_0 the *initial marking* of N. In general, a marking or state of N associates a nonnegative integer number of markers or *tokens* with each place.

The predecessors of a place p (transition t) are called its *input transitions (input places)* and are denoted by ${}^\bullet p$ (${}^\bullet t$). Also, the successors of a place p (transition t) are called its *output transitions (output places)* and are denoted by $p^\bullet$ ($t^\bullet$).

A transition t is *enabled* in a given state when all of its input places have at least one token. An enabled transition t can be fired. When this happens, a token is removed from each input place of t and a token is added to each output place; this gives a new state.

Many timed extensions of Petri nets have been proposed. Our investigations suggest, however, that most of those extensions are too general to permit efficient automated analysis. We discuss this matter further in Section 8. Thus, we will restrict our attention to *Simple Time Petri Nets*, or *STP nets* [3], which are a special case of Time Petri nets [2, 11]. An STP net is a 5-tuple (P, T, F, M_0, D) where (P, T, F, M_0) is an ordinary Petri net, and D is a length $|T|$ vector of nonnegative rational numbers that associates a *static delay*, $\sigma(t)$, with each transition t.

Static delays change the behavior of an STP net with respect to an ordinary Petri net in the following way. Whenever a transition t becomes enabled (i.e., when all its input places have a token), that transition must fire precisely $\sigma(t)$ time units later, unless it becomes disabled by the removal of tokens from an input place in the meantime. Observe that ordinary, untimed Petri nets are now a special case of STP nets with all static delays being zero. At any given global time, each transition t also has a *dynamic delay*, $\delta(t)$, which tells how much time is now left until t fires. Note that for every transition t, at all times we have $0 \leq \delta(t) \leq \sigma(t)$.

The state of an STP net is a pair (M, I), where M is the state of the underlying untimed Petri net, and I is a vector containing the dynamic firing time for each transition enabled by M and $\perp$ for each disabled transition. The initial state of STP net (P, T, F, M_0, D) consists of M_0 and a vector containing the static firing interval of each transition enabled by M_0 and $\perp$ for those transitions that are disabled by M_0.

We assume that our STP nets are *safe*; that is, a place can hold at most one token in every reachable state. Our experience with (untimed) concurrency analysis indicates that restricting our attention to safe Petri nets will not affect our ability to model real-time systems [6].

We say that a transition t is *fireable* in state (M, I) if t is enabled (M associates at least one token with all of ${}^\bullet t$) and it is legal to fire t next. This will be true if and only if transition t has minimal dynamic delay among all enabled transitions. Of course, in a given net state s, there may be several fireable transitions. (This happens if all such transitions are enabled in s, and if their dynamic delays are both identical and minimal in s.) If multiple fireable transitions are in conflict (i.e., they all share an input place), then one of these transitions is nondeterministically chosen for firing.

Let s' be the state derived from firing fireable transition t in state s. We write $s - t \rightarrow s'$, and also $s \Rightarrow s'$. State s' has the marking obtained by considering only the underlying untimed Petri net; its vector of dynamic delays is as follows. For every transition u that was disabled in s but is enabled in s', transition u has dynamic delay equal to $\sigma(u)$. If t—the transition that fired—is enabled again in s', then $\delta(t) = \sigma(t)$; otherwise $\delta(t) = \perp$. If transition $u \neq t$ was enabled in s and remains enabled in s', then the dynamic delay of u in state s' is its previous dynamic delay minus the dynamic delay of transition t in state s. Finally, if u is not enabled in s', then in s' we set $\delta(u) = \perp$.

We say that *state s leads to state s'* whenever there exists a finite sequences of states $s_1, s_2, \ldots, s_n$ such that

$$s \Rightarrow s_1 \Rightarrow s_2 \Rightarrow \cdots \Rightarrow s_n \Rightarrow s'.$$

The *state space* of STP net N is the set of all states s of N such that the initial state s_0 leads to s.

A *(firing) schedule of STP net N from state s_0* is a finite sequence of ordered pairs (t_i, s_i), for $1 \leq i \leq n$, such that for all $1 \leq i \leq n$ we have $s_{i-1} - t_i \rightarrow s_i$. We will often denote such a schedule by $s_0 - t_1 \rightarrow s_1 - t_2 \rightarrow \cdots - t_{n-2} \rightarrow s_{n-2} - t_{n-1} \rightarrow s_{n-1} - t_n \rightarrow s_n$. The *time* of such a sequence is the sum of the dynamic delays of each t_i in state s_{i-1} for $1 \leq i \leq n$.

We say that a transition t is *current* in state s if either t is fireable or t has a static delay of 0. Otherwise we say that t is *delayed*. The idea is that any delayed transition cannot possibly fire until some positive time after the currently fireable transition(s).

2.2 Reachability analysis

Our approach to real-time analysis is based on the following two steps. First, we construct the state space of the STP net modeling the system under consideration. We represent the state space of an STP net as a weighted, directed graph. Each node corresponds to a unique state consisting of a net marking and of a vector of dynamic firing delays. Each arc represents a change from a state x_i to a state x_j resulting from the firing of a transition t. The weight on the arc corresponds to the time elapsed between x_i and x_j, that is, the dynamic firing delay of t in x_i.

After the reachability graph has been built, we apply suitable algorithms to the graph in order to answer queries about the behavior of the system being analyzed. We typically cast queries involving minimum (or maximum) delays among events in the system as shortest (longest) path problems on the graph. We next discuss our algorithms for graph analysis and for reachability graph construction.

The problem of finding the longest or shortest path in an *acyclic* graph (or exhibiting a cycle if the graph is not acyclic) can be solved in linear time. (See, for instance, [5].) In practice, the most common timing queries are of the form "What is the longest possible time between state α and state β?" There are three possibilities for such a query:

1. The portion of the reachability graph reachable from α without passing *through* β is acyclic, so we can answer the question in linear time.
2. There is a positive weight cycle between α and β in the reachability graph. This means that the answer to the timing question is "Unbounded," and all we need to do is exhibit the cycle.[1]
3. There is a zero weight cycle between α and β. Our view is that such cycles in the state space represent an error in the STP net (or the protocol it is modeling), and again we simply report the cycle.

[1]Some would say that in this case the scheduler should be enforcing some fairness conditions that do bound the time required. We feel that if the correctness of a protocol depends on the scheduling policy, then that scheduling policy should be an explicit part of the model being verified.

In all our experiments to date, answering such queries has always taken negligible time compared to the time required to build the reachability graph.

For queries of the form, "What is the shortest possible time between states α and β?" we first try the linear time algorithm for shortest paths in an acyclic graph. If we detect a cycle, we resort to Dijkstra's algorithm, which is reasonably fast.

The basic algorithm for the construction of the reachability graph of a safe STP net is relatively simple. The algorithm first creates the root node n_0 of the graph based on the initial marking M_0 and on the vector D containing the static delays of each transition in the net. The following expansion procedure is subsequently applied to every created node n_i, starting from n_0. First, the transitions that are enabled in n_i are found. If no such transitions exist, n_i is a terminal node, which may represent either a deadlock condition or normal program termination. Otherwise, at least one of the enabled transitions must be fireable. For each fireable transition $t_{i,j}$, we visit the node n_j that is reached by firing $t_{i,j}$ from n_i. The marking of n_j is obtained from the marking of n_i by removing a token from every input place of $t_{i,j}$ and adding a token to every output place of $t_{i,j}$. The dynamic firing delays of n_j are obtained by applying the rules we discussed earlier. If a node n_k with the same marking and dynamic delays as n_j is already in the graph, an arc from n_i to n_k is added to the graph; otherwise, n_j is added to the graph with an arc from n_i to n_j. In either case, the weight of the arc is the dynamic firing delay of $t_{i,j}$ in n_i.

The reachability graph construction ends when all created nodes have been expanded. Note that STP nets have finite reachability graphs, because STP nets are a special case of time Petri nets, and time Petri nets have finite reachability graphs [2]. Thus, our algorithm for reachability-graph construction always terminates. In general, however, the graph can be quite large (i.e., the size of the graph is frequently exponential in the number of processes contained in the input program). Thus we continue by discussing several methods to make reachability graphs smaller.

3 Net reduction

If we start with a smaller STP net, we of course wind up with a smaller reachability graph. The *net reduction method* seeks to shrink an STP net, while preserving properties of interest, such as the possibility of deadlock, in the reduced net. The reduction is obtained by repeated application of a set of reduction rules until no rules can be applied. Berthelot originally proposed a large set of reduction rules for general (untimed) Petri nets [1]. Berthelot's method was subsequently extended and applied to the problem of deadlock detection in Ada tasking programs by Tu et al. [18].

We have further extended the net reduction method to real-time analysis using STP-net models. Here we discuss the most significant aspects of our extensions; an extended discussion focusing on reduction rules for arbitrary time Petri nets may be found in [14].

As with the previous methods, our reduction method is based on a set of reduction rules, which are iteratively applied to a Petri net until no rules can be applied. The goal is that the reduced Petri net should be equivalent to the original one with respect to some properties of interest. A significant difference between our work and previous work is that the goal of previous researchers was only to preserve the possibility of deadlock, whereas we are also interested in preserving timing properties, such as minimum and maximum delays. In order to preserve deadlock properties, one must simply ensure that the reduced net has terminal states if and only if the original net does. The case of minimum and maximum delays is more difficult because in general one must ensure that paths of a certain length are in the state space of the reduced net whenever they are present in the state space of the original net.

For this reason, each of our reduction rules guarantees the following notion of equivalence. Given an STP net N, let N_R be the net obtained by applying a reduction rule to N. The reduction rules are such that N_R is obtained from N by removing a subset S_1 of N's places and transitions and by replacing it with a (possibly empty) set S_2 of new places and transitions. In brief, we require that the set of all firing schedules of N be the same as the set of all firing schedules of N_R, if firings of transitions in S_1 and S_2 are not taken into account.

We now consider an example of a reduction rule—the transition prefusion rule—in order to illustrate the difficulties introduced by the explicit modeling of time in our STP nets. This rule was originally defined by Berthelot [1]. It stipulates that a place p and a transition t_1 can be removed if the following conditions are met for some nonempty set of transitions S.

1. Place p is unmarked in the initial marking.
2. ${t_1}^\bullet = \{p\}$. (Place p is the unique output place of t_1.)
3. For all $t \in S$ place $p \in {}^\bullet t$. (Place p is an input to all transitions in S.)
4. Place p is disconnected from all transitions other than t_1 and those in S.
5. Transition t_1 does not share any of its input places with any other transition.

In this case place p is removed from the net and transition t_1 is merged with each transition in S. Figure 1 illustrates how this rule works when S contains a single transition, t_2. Transition t_f is the result of merging t_1 and t_2. (The delays associated with each transition are shown in square brackets.)

When the rule is extended to STP nets, a conflict arises involving the delay of transition t_f. Suppose that place p_3 receives a token before p_1 does in the original net (left-hand side of Figure 1). In this case, place p_4 receives a token $(a + d)$ time units after a token arrives in p_1. Thus, it appears that the delay on t_f should be $(a + d)$ in order for our equivalence condition to be satisfied. Suppose, however, that place p_3 receives a token d time units after a token has arrived in p_1 in the unreduced net. In this case, a token is deposited in p_4 only a time units after a token arrives in p_3. If the reduced net must behave similarly, the delay of t_f should be a.

We resolve the above conflict by introducing an additional condition on the applicability of the rule. The condition states that either the delay of t_1 is zero $(d = 0)$ or, if $d > 0$, then for every transition $t \in S$, for every $q \in {}^\bullet t$ with $q \neq p$ we have nice-tree(t, q, d), where nice-tree(t, q, d) is defined by:

1. $q^\bullet = \{t\}$.
2. For each $t' \in {}^\bullet q$, first $t'^\bullet = \{q\}$, and additionally either $d \leq \sigma(t')$, or for each place $q' \in {}^\bullet t'$, nice-tree$(t', q', d - \sigma(t'))$. (Recall that $\sigma(t')$ is the static delay of t'.)

The nice-tree condition says that for every transition $t_2 \in S$ we can find a tree embedded in the STP net with root t_2 and leaves that are transitions, and the sum of the static delays of the transitions along any root-to-leaf path is at least d. This tree must include as nodes all input places of any non-leaf transition except for the place p itself, and all output places of every transition in the tree.

If the additional condition is satisfied, place p and transition t_1 are removed. Every place in ${}^\bullet t_1$ becomes an input place of every transition in S. If $\sigma(t_1) = d$ with $d \neq 0$, then add d to the time interval of every transition in S, and adjust the times of the transitions in the trees found by the nice-tree condition as follows: For each transition t that is an input transition to an input place of some transition in S, subtract $m = \min(d, \sigma(t))$ from $\sigma(t)$. If $m < d$, then for each t' that is an input transition to some place in ${}^\bullet t$, subtract $\min(d - m, \sigma(t'))$ from $\sigma(t')$. Continue subtracting out times along the transitions in the tree in this fashion until the leaves are reached (which will occur when the full d has been subtracted out).

A proof that this rule satisfies the conditions for the equivalence between the reduced net and the original net appears in [14]. Additional reduction rules are also discussed. In particular, we show that a slightly modified version of the serial fusion rule proposed by Tu et al. can also be applied to STP nets. Berthelot's conditions for the redundancy of a place carry over directly (i.e., without restrictions) to STP nets [1]. In all cases, we can apply reduction rules only if they do not remove places that our timing or safety queries refer to.

To date, we have generally obtained only small state-space reductions from the reduction rules in the PARTS toolset. This is not surprising, because we have not yet finished implementing the translator from Ada to STP nets, so we have been working with hand-built STP nets. Based on previous experience with untimed Petri-net-based analysis [6], we expect that the greatest benefit from the net reduction method will come on machine-generated nets, in much the same way that the greatest gains from an assembly language optimizer come on compiled code. Nevertheless, we will see an example in Section 6 below where the reduction method (in fact a combination of the prefusion rule and another very simple rule) decreases a state space size from exponential to linear

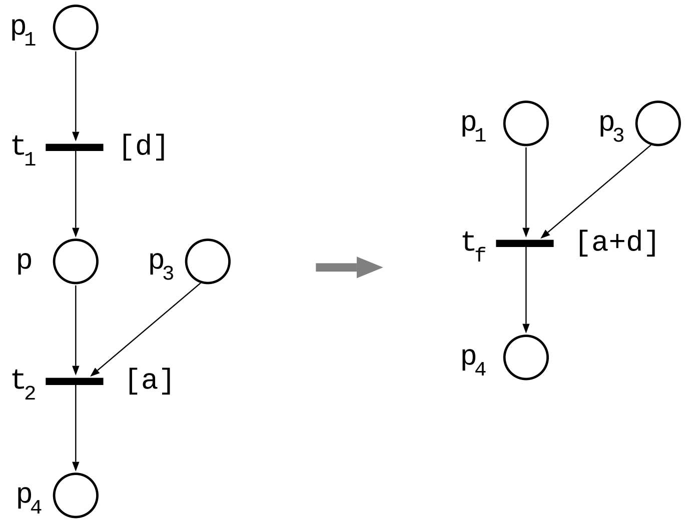

Figure 1: Illustration of pre-fusion reduction rule.

in the number of processes involved.

4 Stubborn sets

We now consider the *stubborn set method*, another key state-space reduction method of the PARTS toolset. Imagine that instead of considering all possible interleavings of n concurrent events we considered only one particular ordering. This might reduce the size of our reachability graph from order 2^n to order n. This is the central idea behind the *stubborn set method* [19, 20]. Instead of exploring the whole state space of a net, we only consider a certain subset of the fireable transitions, called the *stubborn set*, for each state. Of course, as with all our analysis methods, this subset must be chosen based on what queries one later intends to ask.

Valmari invented this method and proved its validity for untimed concurrent systems, first in the context of Petri nets and then for arbitrary transition systems, and obtained impressive speedups in the analysis of many concurrent systems [19, 20]. Subsequently we showed that this method combines synergistically with other methods for state-space reduction of concurrent programs [6].

For the PARTS toolset, we have extended the stubborn set method to STP nets. We note that some of our results could be obtained directly from Valmari's work on stubborn sets for transition systems [20], because one can view STP nets as a particular kind of transition system. Our contributions to the stubborn set method are two-fold. First, we exploited timing properties of our STP nets in order to obtain smaller stubborn sets than we could obtain by simply viewing nets as transition systems. Second, we proved that the stubborn set technique gives proper answers to timing queries, as well as to untimed safety and liveness queries. (Once we converted STP nets to arbitrary transition systems, Valmari's results only guaranteed that *untimed* queries would be answered correctly, and substantial work was required to show that timing properties were also preserved. The interested reader is referred to [15] for a complete discussion of our extensions.)

The definition of a stubborn set of transitions for an STP net is as follows.

DEFINITION. Set A of the transitions of STP net $N = (P, T, F, M_0, D)$ is *stubborn* (in state s) if A contains at least one fireable transition and for every transition $t \in A$:

1. Transition t is current.
2. If t is not enabled, then for some empty $p \in {}^\bullet t$, each transition in ${}^\bullet p$ is either in A or delayed.
3. If t is fireable, then for each $t' \in T - A$ either ${}^\bullet t \cap {}^\bullet t' = \emptyset$, or t' is delayed.

We assume that for every STP net N we have a *stubborn set generator*, a function *TS*(N, s) that will pick out a stubborn set for each net state s. In general it is quite easy to pick out a reasonably small stubborn set. Valmari gives specific algorithms; in practice we currently try building a stubborn set around each fireable transition until either we find a stubborn set with just one transition or we have tried all possibilities. We can typically find smaller stubborn sets for STP nets than for untimed Petri nets because our definition allows us to stop increasing the stubborn set whenever we reach a delayed transition.

When we wish to verify timing properties, we will in general be asking questions of the form "What is the longest time (or shortest time) path from given STP net state s_0 to any state whose marking is in some specified set of markings $\mathcal{C}$?" We must obviously be careful that when we generate only part of the state space of the net that we do not discard the desired path.

Following Valmari [20] we define a transition t to be *visible with respect to the set of net markings* $\mathcal{C}$ if there are two states s and s' such that precisely one of them has a marking in $\mathcal{C}$, and $s{-}t{\rightarrow}s'$. For any specified set of markings $\mathcal{C}$ we assume we are provided with a set of transitions vis$(\mathcal{C})$ that contains all transitions that are visible with respect to $\mathcal{C}$. (The set vis($\mathcal{C}$) does not have to be the smallest such set.)

We build our stubborn set generator so that for each state s of the state space of our STP net either

1. there are no fireable transitions in vis($\mathcal{C}$) $\cap$ *TS*(s), or
2. all current transitions in vis($\mathcal{C}$) are in *TS*(s).

If our stubborn set has this property, then we will get the same answer to all timing queries by examining only the stubborn fireable transitions in each net state instead of all fireable transitions. (We can also, of course, answer safety and liveness queries not involving time.) A proof of this fact appears in [15].

For some versions of the (untimed) Dining Philosopher's problems, the original stubborn set method reduces the size of the state space from exponential to linear [19]. We can add times to that problem and thus achieve the same reduction with our real-time stubborn set method. In Section 6 below, we consider an example of a real-time "divide and conquer" problem that Corbett and Avrunin used as one of the benchmarks for their constrained-expression system for real-time analysis [4], and show that the real-time stubborn set method reduces the state space for that problem from exponential to quadratic in size.

5 Symmetry

The goal of the symmetry method is to exploit symmetries inherent in a Petri net when building the state space of the net. We use a straightforward extension of Starke's method for safety and liveness analysis of untimed Petri nets [16]. His method and our extension both employ a *symmetry function* on a Petri net's node set (the union of the transition and place sets of the Petri net). We define a symmetry function ϕ to be a bijection that maps the node set of an STP net to itself and that satisfies the following conditions:

1. The image $\phi(x)$ of a node x is the same kind of node (either a place or a transition) as x.
2. For every place p in the net, the initial markings of p and $\phi(p)$ are the same.
3. Given two net nodes x and y connected by an arc a, there is also an arc connecting $\phi(x)$ to $\phi(y)$.
4. For every transition t in the net, the static firing delays of t and $\phi(t)$ are the same.

Our first three conditions are the same as Starke's conditions for untimed nets [16]. We added the last condition to reflect the presence of firing delays in our STP nets. Note that if ϕ is a symmetry function, then so is its inverse ϕ^{-1}.

We use symmetry mappings to partition the node set of a Petri net into equivalence classes. Two nodes, x and y, belong to the same equivalence class if and only if there is at least one symmetry function ϕ that maps x to y. (Note that in this case y is mapped to x by ϕ^{-1}.) In addition, symmetry mappings can be used to partition the set of reachable states of Petri net. We say two states s_1 and s_2 of an STP net N are *equivalent* if there is a symmetry mapping ϕ on the node set of N such that:

1. For each place $p \in N$ the marking of p in s_1 is the same as the marking of $\phi(p)$ in s_2.
2. For each transition $t \in N$, the dynamic firing delay of t in s_1 is the same as the dynamic firing delay of $\phi(t)$ in s_2.

Equivalence classes of reachable net states are exploited during reachability graph construction by including only one state from each equivalence class in the reduced state space. Starke has defined algorithms for computing all symmetry mappings of a Petri net, for partitioning the node set into equivalence classes, and for generating a reduced state space [16].

Starke's definitions provide an excellent theoretical foundation for reduced state space generation. However, a program containing symmetrical processes can have a very large number of symmetry mappings. For instance, if a program contains n identical instances of a given process, in general the corresponding Petri net has $n!$ possible symmetry mappings.

In order to apply Starke's method in practice, we have defined a heuristic that allows us to avoid generating symmetry mappings when checking whether two states s_1 and s_2 are equivalent (or symmetrical, in our terminology). First, we identify the equivalence classes of places and transitions using the algorithm described in [16]. As we noted above, two places (or transitions) belong to the same class if they can be mapped into each other by suitable symmetry mappings. Call these two sets of equivalence classes $\mathcal{P}$ and $\mathcal{T}$. Next, we check three sufficient conditions on the existence of a symmetry mapping between s_1 and s_2. If the conditions are satisfied, s_1 and s_2 are concluded to be symmetrical.

The first (structural) condition is based on the notion a "process" within an STP net. Roughly speaking, a process is a set of net places such that at most one place in the set has a token in any reachable net state. Suppose that every place of an STP net is associated with a unique process. Then, the structural condition states that each process can have at most one place in each of the equivalence classes in set $\mathcal{P}$. The second (behavioral) condition for two states s_1 and s_2 to be equivalent is that each place class in $\mathcal{P}$ has the same number of tokens in s_1 and s_2. The third (behavioral) condition states the following for each transition class T_i in $\mathcal{T}$: The vector of dynamic firing delays for T_i transitions in s_1 is a permutation of the corresponding vector in s_2. In all of our examples, we have been able to use these three conditions in order to determine whether two states are equivalent or not.

6 Experimental results

In order to assess the effectiveness of our analysis techniques, we automatically analyzed several examples of real-time systems using the PARTS toolset. At present the toolset consists of three prototypes. The net reduction tool takes as input an STP net and a query, and outputs a reduced STP net.

The tool for reachability graph construction takes as input an STP net and a query, and returns a reachability graph for the net. Stubborn sets and net symmetry can be applied during this construction to produce a reduced state space. Finally, a prototype for the analysis of our graph structures takes a reachability graph and a query as input, and returns an answer to the query as output. All tools are implemented in Common LISP.

We considered the following real-time systems: a simple *divide-and-conquer* example, a simple *network router* example, a generalized version of Leveson's *railroad crossing* example, and an Ethernet *bus protocol* [4, 9, 12]. For each example we ran several experiments using different problem sizes and different combinations of optimization methods. Here we report on the most difficult part of the analysis—reachability graph construction. Every experiment starts from an STP-net model of the problem at hand and automatically generates a (possibly reduced) reachability graph for the net. All experiments were run on a Sun SPARC Station 10 with 32 Mbytes of memory.

In the divide-and-conquer example n computing elements are given [4]. When a problem must be computed, the first element nondeterministically chooses whether to conquer the problem (which requires a lot of time), or to pass the problem to the next computing element (which requires a little time on the part of the first element). The next $(n-2)$ elements behave similarly to the first element; the last element must conquer the problem.

In Table 1 we report experimental results for various problem sizes and combinations of optimization methods. Each entry in the table has three numbers. The first two numbers give the number of nodes (i.e., unique states) and arcs (i.e., generated states) in the reachability graph. The last number gives the CPU time taken by reachability graph construction in seconds. We do not report the times required by net reduction (when applicable), and by shortest and longest path computations because these times are much smaller than the corresponding times for reachability graph construction.

The first column in Table 1 gives the number of computing elements in each version we analyzed. The remaining columns correspond to different combinations of optimization methods. In particular, the second column has data for full state space generation. The next three columns correspond to stubborn sets, net symmetry and net reduction. Finally, the last two columns report data obtained by combining net reduction with either stubborn sets or net symmetry.

Note that the growth in state space size is exponential in the number of computing elements when no optimization method is used. In addition, the symmetry method fails to achieve any state space reductions because there are no symmetries in the net. However, the growth in state space size becomes quadratic when stubborn sets are used. In this case the reachability graph has $(n^2 + 3n - 2)$ nodes. Moreover, state space size is linear in n when net reduction is used. In this case, we obtain $(3n - 1)$ nodes. Note that the growth in the time for state space generation is always worse than linear. We believe that this phenomenon is due primarily to inefficiencies in our current implementation of reachability graph construction. Still, our results compare favorably to the results obtained by Corbett and Avrunin using the constrained expression toolset. The largest version they analyzed also has 100 computing elements, and they used 595 seconds of CPU time on a DECstation 5000/125 for that analysis [4].

The network router example consists of a $(2 \times n)$ matrix of routing elements [4]. Each routing element that is located in a column i with $1 < i < n$ is connected to the four elements located in adjacent columns. Elements in the first column are connected on the right to the two elements in the second column, and to a packet sender on the left. Elements in column n, are connected to the two elements in column $(n-1)$ on the left, and to a packet receiver on the right. Different delays are associated with the links between the $2n$ routing elements. A packet must travel through n routing elements from the sender to the receiver. When a routing element receives a packet, it nondeterministically chooses to send the packet to either of the two elements to its right. The problem is to find the shortest and the longest time that a packet can take while going from the sender to the receiver.

We analyzed two versions of this problem. One version has 50 pairs of routing elements, and the other version has 100 pairs. In both versions each link between two routing elements was randomly assigned an integer delay between zero and nine. Our results are reported in Table 2. The format of this table is similar to the previous table. Our state space sizes are always linear in the number of routing elements. In particular, when no optimization method is used, we obtain $(4n + 1)$ nodes. The stubborn set method provides no reductions because the example does not contain any parallelism. The net reduction method achieves some reductions in the STP net by removing places and transitions corresponding to routing elements with delay equal to zero. Finally, the net symmetry method takes advantage of columns whose two routing elements have identical delays. As with the divide-and-conquer example, our experimental results compare well with analogous results obtained using the constrained expression toolset [4]; that toolset required 2,032 CPU seconds for the 100 element network router.

The generalized railroad crossing (GRC) system operates a gate at a railroad crossing containing n tracks [9]. The area of the crossing lies within a region of interest. When a train enters this region on one of the tracks, the GRC system is notified through a set of sensors. Sensors are also used to signal the GRC system when a train leaves the crossing. We have verified a safety property and a utility property for this example. The safety property stipulates that the gate is always closed when a train is in the crossing. The utility property states that the gate should be open as much as possible when no trains are in the region of interest.

In Table 3 we report data for one-track, two-track, and three-track versions that satisfy both properties. This table differs from the previous two in that we have added a column corresponding to the combination of stubborn sets and net symmetry, and a column corresponding to the combination of all three optimization methods. In all cases the net reduction technique can merge few places and transitions, which results in a small reduction of the full state spaces. (Compare, for

Problem	Optimization methods: none	stub	sym	nr	nr+stub	nr+sym
D & C 5	54	38	54	14	14	14
	76	37	76	17	17	17
	1	1	1	1	1	1
D & C 10	1096	128	1096	29	29	29
	4179	127	4179	37	37	37
	64	3	74	1	1	1
D & C 20	—	458	—	59	59	59
	—	457	—	77	77	77
	—	30	—	2	2	5
D & C 40	—	1718	—	119	119	119
	—	1717	—	157	157	157
	—	408	—	13	14	38
D & C 60	—	—	—	179	179	179
	—	—	—	237	237	237
	—	—	—	41	43	123
D & C 100	—	—	—	299	299	—
	—	—	—	397	397	—
	—	—	—	179	185	—

Table 1: Experimental data showing state space reduction for the divide-and-conquer example. Each entry gives the number of states and arcs in the corresponding reachability graph, and the time required to generate the graph in seconds. A "—" indicates that the experiment was not run to termination because it was taking too long.

Problem	Optimization methods: none	stub	sym	nr	nr+stub	nr+sym
Router 50	201	201	193	192	192	183
	298	298	286	289	289	277
	53	58	65	47	50	56
Router 100	401	401	381	378	378	358
	598	598	568	575	575	546
	405	418	440	361	373	385

Table 2: Experimental data showing state space reduction for network router example.

instance, the columns labeled *none* and *nr* in Table 3.) This technique results in greater reductions when we also use the stubborn set method. This fact seems to indicate that even small reductions in an STP net can significantly benefit the stubborn set method.

In the two-track version, the net symmetry method can reduce the state space because of the symmetry between the tracks. In this version and in the three-track version, the best results are obtained when all methods are used in combination. Unfortunately, however, the growth in state space size remains exponential even after all the methods are applied.

A significant contributor to the state explosion problem in the GRC example is the *time granularity* of transitions in the corresponding STP-net model. Suppose that a net state s_1 enables some transitions with a small (or zero) dynamic delay and a transition, t_L, with a large delay. Of course, the transitions with small delay must be fired before t_L. Also suppose that, after firing a set of transitions with a small delay, a state s_2 is reached. State s_2 has the same marking and dynamic firing delays as s_1, except that the delay of t_L is decreased by a small amount. In this case, transitions that were fireable in s_1 are again fireable in s_2, which results in a replication of the subgraph originating in s_1. Moreover, the subgraph under s_1 may have to be replicated many times depending on the delay of t_L relative to the length of the path from s_1 to s_2. We are currently investigating techniques for detecting and avoiding unnecessary replications of portions of a reachability graph caused by the time granularity problem.

The last example we considered is the well-known CSMA/CD (carrier sense, multiple access with collision detection) protocol for an Ethernet bus [12]. This example has *n* stations sending packets through a bus. The bus does not perform buffering of packets, and there is a propagation delay σ on the sending of packets. Suppose that the bus is available for some time before a sender S_1 sends a packet at time θ_0. For the next σ time units, the other senders sense that the bus is still available. At time $(\theta_0 + \sigma)$ the bus signals the other senders that the bus is busy. If a sender S_2 decides to send a packet but senses that the bus is busy, then S_2 waits for a random time between 0 and 2σ before attempting to send again. If, however, S_2 senses that the bus is still available and sends a packet after S_1 has sent a packet, a collision occurs. When this happens, the bus waits for σ units and then signals all senders that there was a collision and that the packets were lost. In this case, each sender randomly waits for up to 2σ time units before retrying. Suppose that no collision occurs

	Optimization methods							
Problem	none	stub	sym	stub+sym	nr	nr+stub	nr+sym	nr+stub+sym
GRC 1	207	83	207	83	205	82	205	82
	417	94	417	94	414	93	414	93
	3	1	3	1	3	1	3	1
GRC 2	5058	1908	2493	1120	5004	1510	2465	847
	11836	2174	5800	1290	11703	1716	5732	963
	107	27	63	19	99	21	59	14
GRC 3	—	—	—	—	—	—	—	14669
	—	—	—	—	—	—	—	16702
	—	—	—	—	—	—	—	364
Ethernet	642	293	338	160	534	296	282	161
	1406	406	751	223	1178	448	632	245
	12	4	8	3	9	4	6	3

Table 3: Experimental data showing state space reduction for generalized railroad crossing example and for Ethernet bus protocol.

after S_1 sends a packet. In this case, the packet is delivered λ time units after the packet was sent, with $\lambda \gg \sigma$. Moreover, the bus waits for σ time units after the completion of the send before signaling the other senders that the bus is free.

We analyzed a version of the bus protocol example with two senders and delays $\lambda = 10$ and $\sigma = 1$. Our experimental results are reported in the last row of Table 3. As the data indicate, state space construction for this example is relatively fast. We observe two significant limitations of our approach, however. First, we cannot capture time intervals, such as random delays used by senders, using STP nets. Instead, we are forced to discretize time intervals contained in a problem specification. Second, as with the GRC example our analysis is very sensitive to the time granularity problem. For instance, our toolset cannot complete state space construction for a two-sender version in which parameters σ and λ are increased to 15 and 200. Nicollin et al. have analyzed two-sender versions with various values of λ and σ [12]. An advantage of their approach is that their analysis is not significantly affected by the time granularity problem; however, when σ and λ are 1 and 10, our toolset is much faster.

7 Evaluation of experimental results

To date we have conducted a relatively small number of experiments with the PARTS toolset. These experiments do not enable us to draw definitive conclusions about the strength of our STP-net-based approach; however, we can make a few observations about the optimization techniques that we used.

A first consideration is that, in general, the net reduction technique has not significantly reduced either our STP nets or the corresponding state space sizes. This is so because all STP nets in our experiments were designed manually rather than generated automatically by a translator. Of course, we strived to avoid including unnecessary complexity and redundancies in our STP nets. An exception is given by the divide-and-conquer example (see Table 3). In this case, the net reduction method can by itself reduce the size of our state spaces from exponential to linear in the number of computing elements. This phenomenon is due to the possibility of repeated forking and merging of computations performed by different processes in this example. As a result, the transition pre-fusion rule, which is discussed in Section 3, plays a crucial role in reducing the STP nets for this example.

In addition to successfully taming the divide-and-conquer example, the net reduction technique also provided some, albeit small, state space reductions for the other three examples (see Tables 1, 2, and 3). We expect that the reductions provided by this technique will increase significantly when we consider STP nets automatically generated by a translator. In fact, this technique has consistently given dramatic reductions in the (untimed) deadlock analysis of ordinary Petri nets automatically generated from Ada programs [6]. Therefore, we believe that the net reduction technique will play a significant role in reachability-based analysis of real-time systems.

Experiments involving the stubborn set technique show that the effectiveness of this method varies widely depending on the amount of parallelism inherent in the example being analyzed. Consider, for instance, the analysis of the unreduced STP net for the divide-and-conquer example, which has a high degree of parallelism. In this case the stubborn set method reduces the size of the reachability graph from exponential to quadratic in the number of computing elements (see Table 1). In the network router example, however, stubborn sets do not achieve any state space reductions, because this example has no parallelism (see Table 2).

The stubborn set method is sensitive to slight variations in our STP-net models. On the one hand, this method fails to provide any reductions in state space size when applied to the reduced net for the divide-and-conquer example (see Table 1). This is in contrast to the success of stubborn sets when they are applied to the unreduced net. Evidently, the net reduction method removes all parallelism from this example. On the other hand, in the GRC example, the stubborn set method is more effective when combined with net reduction than without net reduction. In this case, reduced and unreduced nets are very similar, however, stubborn sets work more effectively with the reduced net.

The effects of stubborn sets in the Ethernet example seem to be inconsistent: The combination of net reduction and stubborn sets gives a slightly larger state space than the stubborn set method alone. We believe that this apparent inconsistency is caused by the nondeterminism inherent in the stubborn set method. It is known that the effectiveness of stubborn sets depends on nondeterministic choices made during the application of this method [20].

As we expected, the net symmetry method is effective only when analyzing STP nets containing symmetries. Thus, this method gives no reductions in the divide-and-conquer example and in the one-track GRC example. The method gives good reductions in the GRC example with multiple tracks and in the Ethernet protocol (i.e., because of the symmetry between the two senders). We believe that this method will play an important role in the analysis of problems containing symmetrical processes; however, additional experiments are needed in order for us to assess the full power of this method.

A general conclusion from our experiments is that all the optimization methods that we considered should be used in combination in order to make reachability-based analysis more tractable. Empirically, no one method could single-handedly make real-time analysis tractable for all problems in the relatively small collection of problems discussed in this paper. Intuitively, different methods exploit different aspects of a real-time system in order to reduce state space size. As a result, the smallest state spaces are often obtained by combining all methods (e.g., the GRC example).

It is also clear that additional methods must be devised to further enhance the effectiveness of the analysis. We are currently investigating techniques for addressing the time granularity problem, and for exploiting the modular structure of a real-time system by means of compositional analysis methods.

8 Comparison to other Petri net models

We do not think programmers will ever spend time constructing proofs by hand; therefore we have worked on building a completely automated toolset for real-time analysis. In this section we explain why we feel STP nets are the best type of timed Petri net for automated analysis. In short, we think they reach the best trade-off between expressive power and analyzability.

Some research has been done on even less expressive timed Petri nets [22]. We believe the positive results we present here show that using STP nets does not cause analysis to become intractable.

Many more researchers have looked at timed Petri nets with more expressive power than STP nets (see, for instance, [2, 7, 11, 21]).

Time Petri nets [2, 11] are somewhat more expressive than STP nets. We will explain in some detail why we think automatic analysis is likely to be impractical for Time Petri nets; these remarks apply *a fortiori* to more powerful timed Petri nets such as Time Basic nets [7].

8.1 Time Petri nets

Time Petri nets (TPNs) are like STP nets except that they allow for firing *intervals* instead of firing delays. That is, each transition t is associated with an interval $[a, b]$ with $0 \leq a \leq b \leq +\infty$ such that once t becomes enabled it must fire from a to b time units later, unless it becomes disabled in the meantime.

The tremendous advantage of TPNs is that they have more expressive power, and in particular they can represent dense time. A real limitation of our approach is that our time must ultimately be discretized. To model a random delay of zero to two time units with STP nets, we included a nondeterministic choice among three places: one an input to a transition of static delay 0, one an input to a transition of static delay 1, and one an input to a transition of static delay 2. With a TPN, one models this situation with a single transition with interval $[0, 2]$.

Of course, the states of a TPN are more complicated than those of an STP net [2]. They now contain a dynamic firing interval for each enabled transition. In addition, for every pair of enabled transitions t_i and t_j, there may be a constraint of the form "(firing time t_i) $-$ (firing time t_j) $\leq c$" for some $c \geq 0$ depending on the particular state. (For an explanation of how those constraints arise, see Berthomieu and Diaz [2].)

Simply generating the reachability graph of a TPN is very time consuming. For STP nets, generating the next state once we have picked out the transition we wish to fire takes time linear in the number of enabled transitions (typically the number of "processes"), each of which must have its delay updated. For TPNs, we have a quadratic number of constraints that must be updated, and the updating is complex. The most recent work on real-time analysis with TPNs requires time $O(n^3)$, where n is the number of enabled transitions, to compute the next state [21].

At the time of this writing it is unclear whether TPNs can be the foundation for a really practical tool. Yoneda et al. have obtained very promising preliminary results with a TPN-based approach for real-time analysis [21]. However, the only analysis they report is for a system with a relatively simple state space. If similar results can be obtained for problems of at least the complexity of those discussed above in Section 6, then TPNs may well be preferable to STP nets. Further work is need to determine whether this is possible, given the high complexity of state space generation for TPNs.

8.2 Time Basic nets

Time Basic nets (TB-nets) [7] are a strict generalization of TPNs. Now the firing interval of a transition can be bounded by functions instead of constants. TB-nets have tremendous modeling capabilities, but of course the problems we outlined above for TPNs apply to TB-nets as well.

In fact, reachability analysis is even more difficult for TB-nets. In general, the state space of a TB-net is infinite, so one can only verify bounded safety and bounded liveness properties for these nets.

One of the many useful modeling features provided by TB-nets are *weak time semantics*. When a transition becomes enabled, it may either fire during its firing interval or *expire*; it is not compelled to fire. STP nets and TPNs have *strong time semantics*; once a transition becomes enabled it must fire at the prescribed time unless it is disabled by the removal of tokens in the meantime.

Both weak and strong time semantics are needed to model real-time systems. Weak time semantics accurately capture a situation where some event either occurs by a certain time or not at all; strong time semantics accurately model timeouts.

Imitating weak time semantics with STP nets (or TPNs) is straightforward. For instance, with STP nets one can make a transition t obey weak time semantics by adding a second transition with the same inputs and delay as t and no outputs (a "sink transition").

The methods for obtaining strong time semantics in nets with weak time semantics are more ungainly, and in addition completely destroy partial order methods such as stubborn sets. For example, Ghezzi et al. [7] suggest, for a high-level net that in fact is a generalization of TB-nets, that one can simulate strong time semantics by adding a new place that is an input place and output place of *every* transition. With such a place in a net, the stubborn set method would never find any reductions, and most of the net reduction rules would become inapplicable.

9 Future work

We have described the core of the PARTS toolset for the static analysis of real-time systems. In many cases we have significantly outperformed previously published results on real-time analysis; in almost all cases we have done as well as previous systems. We believe a key to our success is the combination of many different techniques for controlling state space size.

Currently we are pursuing a number of different research ideas to further reduce state space size. Most importantly, we are working on a method to detect and eliminate state space blowup caused by time granularity problems. We hope this will allow us to analyze the Ethernet protocol described above in Section 6 for arbitrary values of the network propagation delay. (Currently our analysis runs very quickly for small values of that parameter, but very slowly as that parameter grows.)

We are also working to extend Godefroid and Wolper's *sleep set method* [8] to the real-time context. This is another partial-order method that is complementary to the stubborn set method.

We hope to eventually apply the PARTS toolset to large real-world Ada programs. An implementation of a translator that converts real-time code written in Ada into STP nets is under way. Finally, we are starting to investigate compositional methods, because we suspect that the analysis of truly large programs may have to be done in pieces.

References

[1] G. Berthelot. Checking properties of nets using transformations. In G. Rozenberg, editor, *Advances in Petri Nets 1985*, pages 19–40. Springer-Verlag, 1987.

[2] 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.

[3] U. Buy and R. Sloan. A Petri-net-based approach to real-time program analysis. In *Proc. Seventh Internat. Workshop on Software Specification and Design*, pages 56–60, Dec. 1993.

[4] J. C. Corbett and G. S. Avrunin. A practical technique for bounding the time between events in concurrent real-time systems. In *Proc. 1993 Internat. Sympos. on Software Testing and Analysis*, pages 110–116. ACM, June 1993.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[6] S. Duri, U. Buy, R. Devarapalli, and S. M. Shatz. Using state space reduction methods for deadlock analysis in Ada tasking. In *Proc. 1993 Internat. Sympos. on Software Testing and Analysis*, pages 51–60. ACM, June 1993.

[7] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzé. A unified high-level Petri net formalism for time-critical systems. *IEEE Trans. Softw. Eng.*, 17(2):160–172, Feb. 1991.

[8] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In K. G. Larsen and A. Skou, editors, *Computer-Aided Verification: 3rd Internat. Conf., CAV '91*, LNCS 575, pages 332–342. Springer-Verlag, 1992.

[9] C. Heitmeyer, R. Jeffords, and B. Labaw. A benchmark for comparing different apporaches for specifying and verifying real-time systems. In *Proc. Tenth Internat. Workshop on Real-Time Operating Systems and Software*, May 1993.

[10] N. D. Jones, L. H. Landweber, and Y. E. Lien. Complexity of some problems in Petri nets. *Theoretical Comput. Sci.*, 4:277–299, 1977.

[11] P. Merlin and D. Faber. Recoverability of communication protocols—implications of a theoretical study. *IEEE Trans. Communications*, COM-24(9):381–404, Sept. 1976.

[12] X. Nicollin, J. Sifakis, and S. Yovine. Compiling real-time specifications into extended automata. *IEEE Trans. Softw. Eng.*, 18(9):794–804, Sept. 1992.

[13] C. Ramchandani. Analysis of asynchronous concurrent systems using timed Petri nets. Research Report MAC-TR 120, Massachusetts Institute of Technology, Feb. 1976.

[14] R. H. Sloan and U. Buy. Reduction rules for time Petri nets, 1994. Submitted for publication to *Acta Informatica*.

[15] R. H. Sloan and U. Buy. Stubborn sets for real-time Petri nets. Submitted to 15th IEEE Real-Time Systems Symposium, 1994.

[16] P. H. Starke. Reachability analysis of Petri nets using symmetries. *Syst. Anal. Model. Simul.*, 8:293–303, 1991.

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

[18] S. Tu, S. M. Shatz, and T. Murata. Applying Petri net reduction to support Ada-tasking deadlock analysis. In *Proc. Eleventh Internat. Conference on Distributed Computing Systems*, pages 96–103, Paris, France, 1990.

[19] A. Valmari. Error detection by reduced reachability graph generation. In *Proc. of the Ninth European Workshop on the Application and Theory of Petri Nets*, pages 95–112, Venice, Italy, 1988.

[20] 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.

[21] T. Yoneda, A. Shibayama, B.-H. Schlingloff, and E. M. Clarke. Efficient verification of parallel real-time systems. In C. Courcoubetis, editor, *Computer-Aided Verification: 5th Internat. Conf., CAV '93*, LNCS 697, pages 321–332. Springer-Verlag, 1993.

[22] W. M. Zuberek. Timed Petri nets: Definitions, properties, and applications. *Microelectronics and Reliability*, 31(4):627–644, 1991.