Manuscript accepted by *Automated Software Engineering*.

Automatic Real-Time Analysis of Reactive Systems with the PARTS Toolset

U. Buy and R. 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

Real-time systems are becoming increasingly widespread, often in safety-critical applications. It is therefore crucial that these systems be correct; however, there are few automated tools for analyzing concurrency and timing properties of these systems. The PARTS toolset uses a Petri-net-based reachability analysis to analyze program specifications written in an Ada-83 subset. Our *simple time Petri nets* are specifically aimed at facilitating real-time analysis. In order to control the state-explosion problem, PARTS employs several optimization techniques aimed at state-space reduction. In this paper we discuss our approach and we report on extensive experiments with several examples of real-time specifications based on Ada 83. When possible, we also compare our experimental results with results obtained by other approaches to real-time analysis.

1 Introduction

Real-time software is becoming increasingly widespread, including in safety-critical applications such as traffic-control systems, navigation systems and patient-monitoring systems. Real-time software is characterized by the existence of timing constraints on its behavior, such as *deadlines* on the completion of program computations. In many cases, real-time software is also physically distributed, meaning that different software units (e.g., processes) execute on different processors that communicate through a network. Because real-time software is often used in safety-critical applications, it is essential that this software be dependable; however, the development of reliable real-time programs is extremely difficult.

One reason for this state of affairs is that the behavior of a real-time system is often affected by unpredictable factors, such as communication delays and the arbitrary interleaving of computations performed by different processes. The resulting nondeterminism significantly complicates system testing and debugging. Testing is difficult because erroneous behaviors may not manifest themselves during test runs even if the appropriate input test data are used. After an error is detected, it may be difficult to reproduce the sequence of events causing the error; this phenomenon complicates debugging. An additional difficulty in testing real-time software is the so-called *probe effect*: The instrumentation for measuring the timing behavior of code is likely to affect the behavior being measured [53]. Because testing and debugging of real-time software are so difficult, the development of static analysis techniques is highly desirable. Unfortunately, to date tools supporting static analysis are quite scarce.

Here we are specifically interested in the automatic analysis of real-time specifications written in a subset of Ada 83. We use a timed extension of Petri nets, which we call a *Simple Time Petri net (STP net)*, to model the behavior of a program specification. For our analyses, we apply model-checking techniques to state-space representations of our STP nets. Although the state space of an STP net is always finite, the state explosion problem can often prevent effective application of enumerative techniques such as ours. We employ various state-space reduction techniques in order to control the state explosion problem. Most of these techniques were originally defined by other researchers for untimed concurrency analysis [9, 57, 62]; we extended them to the analysis of timed models [55, 56].

Our verification method is based on the following paradigm. Given an Ada-like program specification, we first translate the specification into an STP-net model. Next, we generate a reduced state space of the net. Finally, we use graph-theoretic algorithms to answer questions about the behavior of the system. Our method has lead to the implementation of the Petri-net-based Analyzer of Real-Time Systems (PARTS) toolset. In this paper we report experimental results obtained by the toolset on several examples of real-time systems.

We also compare the performance of PARTS with other methods for real-time analysis [3, 10, 23, 35, 51, 65].

To date we have considered a subset of Ada 83 as the basis for our real-time specifications mostly for historical reasons. The Ada-83 subset that we are considering does not include dynamic process creation because this feature cannot be captured easily with finite-state models. An extension to include Ada-95 constructs is under active investigation.

During the translation from Ada-like specifications into STP nets, we apply a zero-time assumption on the time taken by program computations. Although STP nets can handle arbitrary delays on computations, this assumption simplifies the translation process. In addition, this assumption allows us to sidestep the problem of estimating the time taken by the execution of Ada statements on a given architecture. In this way, we can concentrate on a discussion of our analysis method, which is the focus of this paper. If the time consumed by program computations is in fact significant, then it could be captured in our specifications by *delay* statements, which we do implement. Our zero-time assumption is similar to the case of so-called *synchronous* real-time languages, such as Lustre [32] and Esterel [8].

Our analysis method relies on a *maximal parallelism* assumption. This means that an enabled computation is never delayed by the lack of an available processor. This assumption follows from the zero-time assumption above; it simplifies our analyses because we are not required to model processor contention and processor scheduling in our STP nets. Most other work in real-time analysis also makes this assumption [10, 23, 50, 65].[1]

The use of STP nets is an important feature of PARTS. To date, many different timed extensions of Petri nets have been defined [30, 47, 52, 68]. STP nets associate each net transition t with a constant delay d; t fires if it is continuously enabled for d time units. The expressive power of our temporal model is roughly equivalent to a timed version of the constrained expression model for concurrency [3]. Other Petri-net models, such as time Petri nets [47] and time-basic nets [29] are more expressive than STP nets. Time Petri nets associate a delay interval bounded by two constants, rather than a constant delay, with each net transition. Time-basic nets are even more expressive than time Petri nets, mostly because they allow the delay of transition t to be expressed as a function of the firing time of other transitions in the net. However, the advantage of STP nets is that they allow us to automatically analyze larger programs than those two Petri-net models.

A preliminary report on PARTS appeared in [17]. Here we extend our previous results in several significant ways. First, we have added a tool for translating program specifications based on a subset of Ada 83 into STP nets. Now we carry out a fully automated analysis starting from an Ada-based program specification

[1] Exceptions include the method of Avrunin et al. [3] and Corbett's model for Ada tasking [21].

and a query on the behavior of the specification. Typical queries include questions about safety properties (e.g., mutual exclusion) and timing properties (e.g., minimum and maximum delays between events) in a real-time system.

The addition of a translator adds strength to the experimental results obtained by PARTS. Although the results that we obtained in the past were quite encouraging, they were based on hand-written Petri nets [17]. Could comparable results be obtained by automatic translation from program specifications? This question is important because, as other authors have observed, the kind of input provided to a verification toolset can have a significant effect on the performance of the toolset [22]. Our current results have answered this question in a positive manner.

An additional extension with respect to our previous work is that we have enhanced our reduction methods and improved our algorithms for state-space generation. As a result, we have obtained much better empirical results than we reported previously, even though our analysis now starts from Ada-like specifications, rather than STP nets.

Finally, we have extended the set of examples that we considered in order to evaluate the performance of PARTS relative to other toolsets for verification. Our empirical results show that our performance results are quite reasonable compared to other approaches to automated verification.

This paper is organized as follows. STP nets are formally described in Section 2. Section 3 discusses the translation of real-time Ada-like specifications into STP nets. Our algorithms for state-space construction and for graph analysis are described in Section 4. The optimization techniques we used are summarized in Section 5. Section 6 reports on experimental results that we obtained by applying PARTS to various examples of real-time systems. In Section 7 we evaluate the effectiveness of the optimization techniques that we used and of our integrated approach. In Section 8 we compare our STP nets with other analysis methods. Finally, some conclusions and directions for future work are given in Section 9.

2 Simple time Petri nets

We require the following definitions. An *ordinary Petri net* is a four-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 marking if 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 marking.

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. (This subject is discussed in detail in Subsection 8.1.) Our STP nets [16] are a special case of Time Petri nets [10, 47]. An STP net is a five-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. 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:

$$0 \leq \delta(t) \leq \sigma(t) \tag{1}$$

The state of an STP net is a pair (M, I), where M is the marking of the underlying untimed Petri net, and I is a vector containing the dynamic firing delay 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 delay 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. This assumption does not restrict our ability to model real-time systems because the subset of Ada 83 that we consider for analysis does not include dynamic process creation or variables with unbounded domains. In general, dynamic process creation is considered a questionable—and potentially harmful—practise in the case of real-time systems [42].

As a result of our assumptions, the real-time systems that we analyze automatically have finite state spaces. As Murata et al. observe [49], it is relatively straightforward to build translators generating safe Petri nets for these kinds of systems from, say, Ada tasking code. In line with our safety assumption, we restrict M_0 to be a vector of zeros and ones of length $|P|$ in the case of STP nets.

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 if 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 there are multiple fireable transitions, then one is nondeterministically chosen for firing.

Let s' be the state derived by 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 $\delta(u) = \perp$.

Observe that ordinary, untimed Petri nets are now a special case of STP nets with all static delays being zero. When this condition is true of an STP net, the dynamic delays of all enabled transition must always be zero because of (1) above. Thus, an enabled transition is always fireable, as with ordinary Petri nets.

The equivalence between zero-delay STP nets and ordinary Petri nets may not be obvious because in an STP net an enabled transition is sometimes "forced" to fire; this never happens in ordinary Petri nets. In particular, in an STP net an enabled transition *t* must be fired before any other enabled transitions with greater dynamic delays than *t*. However, when all transitions in an STP net have zero static delay, an enabled transition is never forced to fire, because all dynamic delays will always be zero. Thus, an STP net with all static delays being zero behaves exactly like the underlying ordinary Petri net.

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 $s_{i-1} - t_i \rightarrow s_i$. 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).

3 Translation into STP nets

Our analysis of real-time systems starts from program specifications based on a subset of Ada 83. This is in contrast with our earlier experiments, which used STP nets as the input language to PARTS [17]. We have found that Ada-like specifications are much easier to write and to understand than the corresponding STP nets. An additional benefit is that these specifications give us an opportunity to test the effectiveness of our reduction method for STP nets [55]. This is so because hand-generated Petri nets are usually of minimal or near-minimal size, whereas nets generated automatically by a translator are usually much larger than they need to be. Of course, prospective users of tools like ours should not need to write their problem descriptions using Petri nets in order to obtain good empirical results. Finally, the use of Ada-based specifications shows that our analysis method is applicable to a high-level language, such as Ada 83.

In this section we discuss the characteristics of the program specifications that we have automatically analyzed using the PARTS toolset. We also describe our translation from Ada-like specifications to STP nets.

3.1 Program specifications

In general, real-time analysis requires that the time taken by the execution of each program statement on a given hardware and software architecture be known before the analysis is performed. A program model must incorporate the time of program computations. Our STP-net model supports this kind of analysis because computation time can be easily captured by delays associated with Petri net transitions.

In this article, however, we are not interested in the problem of estimating the time required for the execution of individual Ada statements on a given architecture. Instead, we are interested in the broader picture, that is, the analysis of real-time systems in which computer components monitor physical processes that are subject to timing constraints [7]. In general, the time required by physical processes, such as mechanical processes, is orders of magnitude greater than the time required for program execution. For this reason, we assume that the time taken by program computations is negligible when compared to the time required by the physical processes monitored. This assumption is consistent with the model of time adopted by synchronous real-time languages such as Lustre and Esterel [8, 32]. Of course, when estimates for the times required by process computations are both relevant and available, we can easily incorporate them in our models, as we did for the network router and the divide-and-conquer examples discussed in Section 6.

To simplify our experiments with STP-net-based analysis, we restrict our attention to a relevant subset of Ada 83. This subset includes the *task* construct, which specifies an asynchronous process, the *rendezvous* mechanism, which allows for interprocess communication and synchronization, and statements involving time, such as *delay* statements and *timed entry calls*. A rendezvous is a synchronous, bidirectional communication between a task accepting a so-called entry call and a task that issued the call. The rendezvous mechanism also allows a task to wait simultaneously for calls on multiple possible entries by means of a so-called *selective wait statement*. A delay statement is used to suspend execution of a task for a specified period of time. As we show in the railroad crossing example below, we often use delay statements to capture the elapsing of time required by certain activities, such as the closing of a gate at the crossing. Finally, a timed entry call allows a task issuing the call to set a limit on the time spent waiting for the call to be accepted. A similar timeout mechanism can also be included in selective wait statements.

In addition to constructs supporting concurrency and timing properties, the Ada 83 subset that we consider includes also the main iteration and alternation constructs of the language (e.g., if-then-else, while-do, and for-do). Finally, we explicitly model boolean variables and finite-range integer variables that affect the flow of control in the system being analyzed. A reader with some familiarity with Ada-83 tasking constructs should be able to understand the specification example below without problems. A full discussion of Ada language constructs is beyond the scope of this article [60].

Figure 1 shows an example of an Ada-like specification, the one-track generalized railroad crossing (GRC) system [34, 39]. (Two additional examples of program specifications, the divide-and-conquer and delayed-reactor-trip systems, appear in the Appendix.) The GRC system operates a gate at a railroad crossing containing *n* tracks. The railroad crossing lies within a region of interest. When a train enters this region on any 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. An implicit assumption is that trains traveling through the crossing are far enough from each other that only one train can be in the region of interest at any point in time. (In Subection 6.3 we will also consider multitrack GRC versions in which one train for each track can be in the crossing.)

The one-track GRC version shown in Figure 1 contains three tasks. Task *track* models sensors that signal when a train enters or exits the region of interest around the crossing. Task *gate* represents the gate, and task *control* monitors the whole crossing. When a train enters the region of interest, *track* calls entry *control.enter*; likewise, a call on entry *control.leave* signifies that the train has left the crossing. When the gate must be lowered, *control* calls entry *gate.close*. Finally, *control* calls entry *gate.open* to raise the gate.

We use delay statements to model the time required by physical components to perform some action. For instance, task *gate* captures the time for lowering the gate by delaying its own execution by 5 time units. The selective wait statement in *gate* models the gate opening. This action is slightly more difficult than the previous one because a train may arrive while the gate is opening. In this case, the gate raising should

```
task GATE is
   entry OPEN;
   entry CLOSE;
  end GATE;

task CONTROL is
   entry ENTER;
   entry LEAVE;
  end CONTROL;

task TRACK;

task body GATE is
begin
  accept CLOSE;
  delay 5;
  loop
    accept OPEN;
    select
      accept CLOSE;
      delay 5;
    or
      delay 10;
      accept CLOSE;
      delay 5;
    end select;
  end loop;
end;
```

```
task body CONTROL is
begin
  loop
    select
      accept ENTER
        do GATE.CLOSE; end ENTER;
    or
      accept LEAVE;
        do GATE.OPEN; end OPEN;
      end select;
    end loop;
  end CONTROL;

task body TRACK is
begin
  loop
    if x = 0
    then delay 1;
    else CONTROL.ENTER;
         if y = 0
         then delay 10;
         else delay 20;
         end if;
           CONTROL.LEAVE;
    end if;
  end loop;
end;
```

Figure 1: Specification of one-track generalized railroad crossing example.

be canceled and the gate should be immediately closed to prevent the new train from reaching the crossing before the gate is completely closed.[2]

Task *gate* models the possibility of a train arriving at the intersection while the gate is opening by use of an Ada selective wait statement with a delay alternative. This statement follows an accept statement on entry *open*; a rendezvous on this entry signifies that the gate should be raised. We assume that the gate will then take 10 time units to open. If no train arrives during this time, the delay alternative is taken in the code, and the gate becomes open. Note that the two select alternatives specify similar behaviors of *gate*. We introduced the two alternatives in order to model explicitly states in which the gate is completely open. Task *gate* will then wait for a call on entry *close* before starting to close the gate again. If, however, a train arrives before 10 time units elapse, then the accept alternative on entry *close* is taken, which will close the gate within 5 time units.[3]

Task *track* shown in Figure 1 uses two nondeterministic *if* statements to model trains' arrival in the region of interest, and the time trains take to reach the crossing after entering this region. This time may vary because trains travel at different speeds. Our current translation tool generates an STP-net segment

[2]We also analyzed a version of the GRC example in which the gate must finish opening before it can start closing. In this case, PARTS properly detected the possibility that a train might be in the intersection while the gate was still closing.

[3]The time to close the gate may actually be lower in this case, depending on the actual position of the gate when a train enters the intersection. We do not model intermediate gate positions in our specification.

that models a nondeterministic *if* statement whenever the *if* statement's condition contains an undeclared variable. Consider, for instance, the first *if* statement appearing in *track*. The condition on this statement is the expression $(x = 0)$, which contains the undeclared variable x. The resulting STP-net fragment nondeterministically chooses between the two *if* alternatives. The first alternative delays *track* by 1 time unit, and then repeats the test. The second alternative calls entry *control.enter*, signifying that a train has entered the intersection. The behavior of the second *if* statement is similar; this statement models a nondeterministic delay between a train entering the region of interest and reaching the crossing.

Alternative analysis methods have used different notations for expressing nondeterministic *if* statements. For instance, the constrained expression approach uses ellipses instead of an actual condition [2].

3.2 Translation into STP nets

Translating an Ada-like specification into a safe STP net that captures the task interactions and timing properties of the specification is relatively straightforward. Our translation approach is based on an earlier method for generating (untimed) ordinary Petri nets from Ada tasking programs [54]. That method has been applied to deadlock detection in Ada code [26].

Our translation method significantly extends the previous method. First, the presence of time in our STP nets allows us to represent additional Ada constructs, such as delay statements, timed entry calls, and delay alternatives in selective wait statements. Second, we explicitly model counter variables (i.e., integer variables with finite ranges) that are used to control tasking behavior. Finally, we model parameter passing from the entry caller to the callee during a rendezvous. Our method has been implemented in the Ada translator system (ATS) [63].

We do not report on all the aspects of our translation method here. Instead, we simply discuss the representation of an Ada construct involving time, that is, a timed entry call. This is one of several new aspects of our translation method. Additional translation details can be found elsewhere [26, 54, 63].

A timed entry call consists of an entry call and a delay alternative, which specifies a delay d. In Ada, when task T_1 calls an entry E in task T_2, T_1 is blocked until T_2 executes an accept statement for E. If, however, T_1 issues a timed entry call, then T_1 is blocked for at most d time units. If T_2 does not accept the call within this time, then T_1's call is canceled, and T_1 executes the delay alternative instead.

Figure 2 shows a net fragment illustrating our modeling of timed entry calls. Each place is labeled by a place name, and each transition is labeled by a transition name, followed by a static firing delay in square brackets. Places p_1, p_2, and p_3 represent the flow of control in three tasks, T_1, T_2, and T_3. Tasks T_1 and T_3 both issue timed entry calls on T_2 with delays of 10 and 8 time units. In particular, a token in p_1 signifies

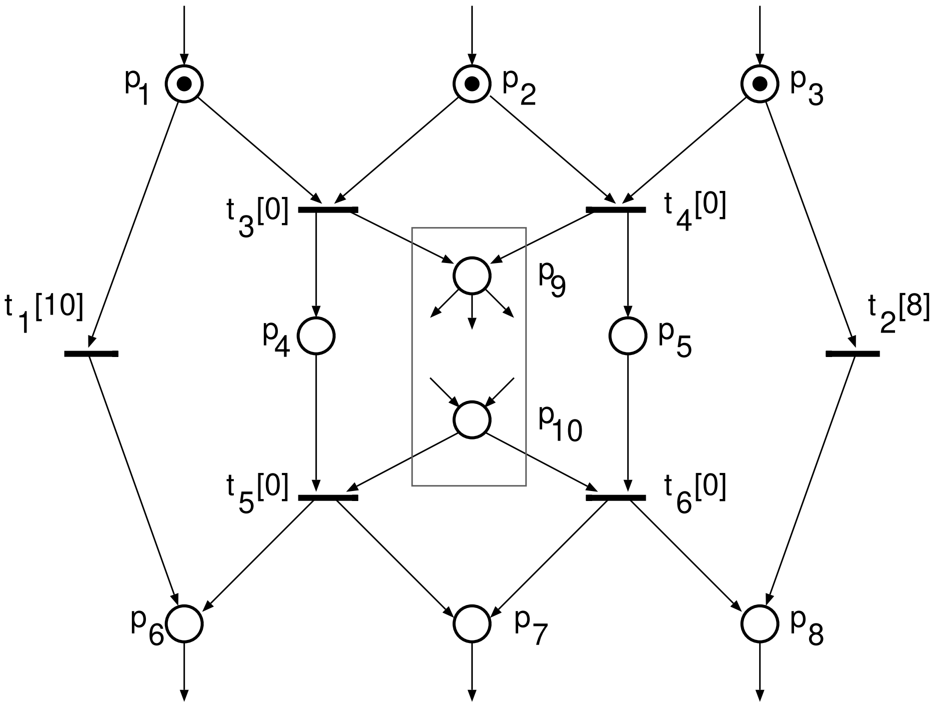

Figure 2: STP-net fragment modeling Ada's timed entry call statement.

that task T_1 has issued the call. A token in p_2 signifies that T_2 is ready to accept a call. Places p_4 and p_5 signify that a rendezvous has been engaged between T_1 and T_2 or between T_2 and T_3, respectively. The subnet in the dashed box models the body of an accept statement in T_2.

Transition t_3 fires as soon as both p_1 and p_2 have tokens. When t_3 fires, tokens are removed from p_1 and p_2, and place p_4 receives a token. A token is also added to place p_9, which starts the execution of the accept body. When the accept body is complete (meaning that place p_{10} has a token), transition t_5 fires. As a result, tokens are removed from places p_4 and p_{10}, and added to places p_6 and p_7.

Transition t_1 corresponds to the timeout for task T_1. If place p_1 holds a token for 10 time units, and if transition t_3 does not fire during this time, then t_1 fires. In this case, the token is removed from p_1, signifying that the entry call has been canceled. The case of task T_3 is symmetric to T_1, except that T_3 waits only up to 8 time units before canceling the call on T_2.

In addition to the above net, we have developed nets for modeling the delay statement and the selective wait statement with delay alternatives. We have also defined techniques for modeling integer variables over finite ranges in STP nets. Finally, we have developed a technique for modeling the binding of arguments and parameters in entry calls.

We show an example STP net produced by our translation tools in Figure 3. We obtained this STP net by applying first ATS and then our net reduction tool to the Ada code for the one-track GRC example

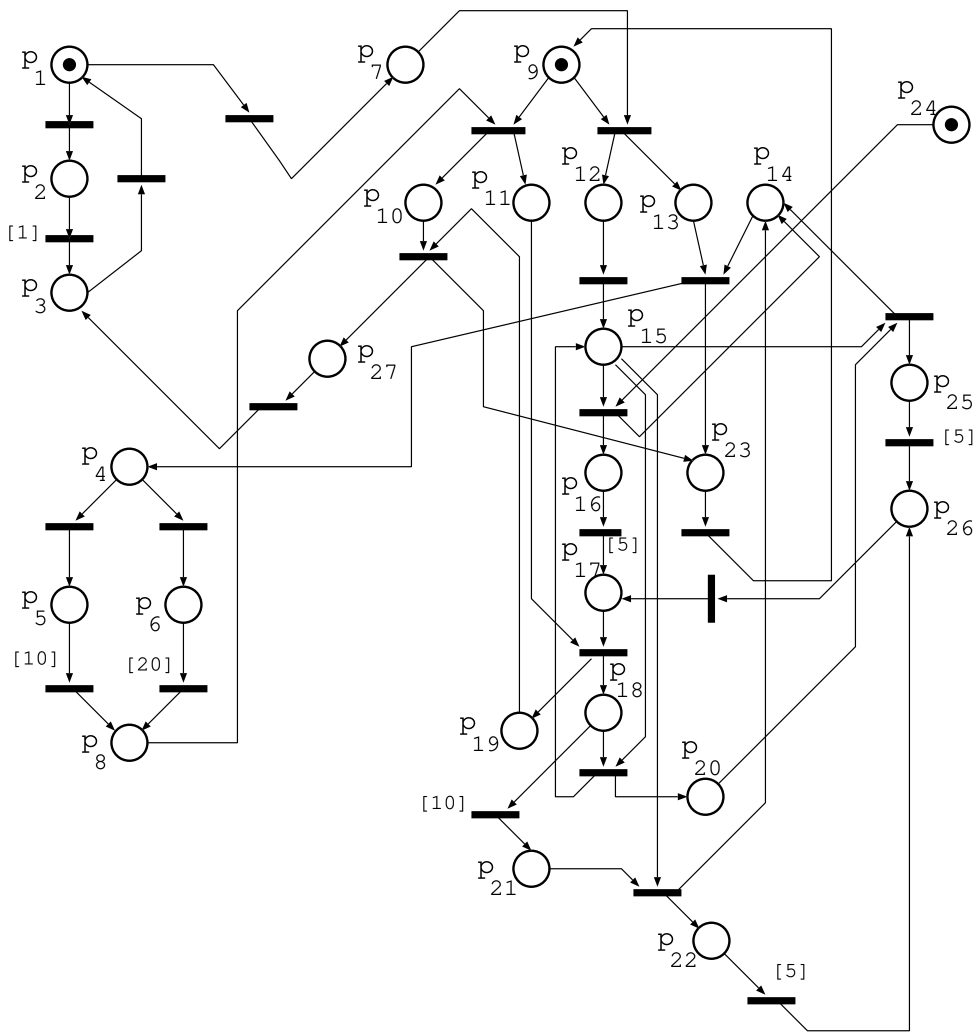

Figure 3: Example of STP net produced by our translation tools from the Ada tasking code shown in Figure 1. To avoid clutter, we labeled only net places using a subscript notation and we included only nonzero transition delays in square brackets.

appearing in Figure 1.[4] Places that are initially marked are shown with a token. These places represent the initial location in the flow of control of tasks *track*, *control* and *gate*. To avoid clutter we omitted zero static delays.

Places p_1 through p_8 roughly correspond to the *track* task. In particular, place p_1 models the beginning of the loop body, and places p_2 and p_3 model the *then* clause in the outer *if* statement in *track*'s body. The transition with unit delay captures the corresponding *delay* statement. A token in place p_7 signifies that a train has entered the region of interest. In the Ada source, this happens when *track* calls entry *control.enter*. When *control* accepts the entry, tokens are deposited in places p_{12} and p_{13}. A token in place p_4 signifies

[4]The unreduced STP net is too large to include here.

that the corresponding rendezvous has been completed. At this point, *track* nondeterministically chooses whether to wait for 10 or 20 time units, as denoted by a token in either place p_5 or p_6. A token in place p_8 means that *track* has called entry *control.leave*. This entry captures the action of a train leaving the crossing. When the entry is accepted, tokens are deposited in places p_{10} and p_{11}. A token in place p_{27} signifies that the rendezvous has been completed; control can now return to the beginning of the *track* loop.

Place p_9 models the beginning of task *control*. At this point the task can accept either entry *enter* or *leave*. This nondeterministic choice is captured by the two transitions in $p_9^\bullet$. When *enter* is accepted, *control* calls entry *gate.close*. This rendezvous is captured by the firing of the transition between p_{15} and p_{16}. After this firing, places p_{13}, p_{14}, and p_{16} are marked. The firing of the transition in $p_{13}^\bullet$ signifies that the rendezvous on *enter* has been completed. Next, a token is deposited in place p_{17}, meaning that the gate has been closed. The gate remains closed until the train leaves the crossing, which is modeled by a token in place p_{11}. A token in place p_{18} means that the gate is opening; in the Ada code this happens after *control* has accepted entry *open*. While the gate is opening, two courses of action are possible that correspond to the two alternatives in *gate*'s selective wait statement. If a train arrives before the gate has finished opening (i.e., the first alternative), a token is deposited in place p_{20}, and the gate is closed again. If, however, no train arrives while the gate is opening, place p_{21} receives a token, signifying that the gate has been opened completely.

A token will remain in place p_{21} until a new train arrives, as is modeled by an accept statement on entry *close* in task *gate*. Once this happens, a token in place p_{22} indicates that the gate is closing. A token in place p_{26} or p_{17} means that the gate is closed. Again, the gate will remain closed until place p_{11} is marked, signifying that a train has left the intersection. The resulting Petri net has 27 places and 25 transitions. Before reduction, the net had 47 places and 41 transitions.

The size of the reduced and unreduced nets in the single-track GRC example makes a strong case for automating the translation from Ada-like specifications to STP nets. Even this relatively small example yields an STP net too large for human users to manage efficiently. Evidently, Ada-like specifications are much easier to use than the corresponding STP nets.

4 State-space-based 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 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. In the next three subsections, we discuss our algorithms for reachability-graph construction and for graph analysis, and our query language.

4.1 Reachability graph construction

The 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*, that contains 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. (Recall that an enabled transition is fireable if it has minimal dynamic delay.)

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 in Section 2. 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 [10]. 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). In Section 5 we discuss several methods for making reachability graphs smaller.

4.2 Graph analysis

Our analysis seeks to answer different kinds of questions regarding the behavior of real-time systems. Some questions involve safety properties of the system under consideration. Examples of these properties include freedom from deadlock and compliance with mutual exclusion constraints. These properties are usually verified by searching the nodes in a reachability graph. To verify, for instance, that a system is free from global deadlocks, we search for all terminal graph nodes (i.e., nodes with no outgoing edges). We verify that a system satisfies a mutual exclusion constraint by checking that no reachable state violates the constraint. The violation of a mutual exclusion constraint is expressed as a predicate on the reachable markings of the STP net representing the system under consideration. For instance, we have applied this technique in the GRC example to ensure that a car and a train cannot be in the crossing at the same time.

As we explain below, the predicates expressing safety properties involve places in an STP net. We identify the places involved in a given predicate by examining the STP net. This process is facilitated by the fact that the name of each place encodes the underlying semantics of the place with respect to the original Ada-like specification. Should this need arise, it would be relatively straightforward for us to build a query processor translating queries based on Ada specifications into queries on STP-net places.

We also automatically answer questions involving real-time properties of a system being analyzed. These questions are expressed in terms of maximum or minimum delays between events in the system under consideration. To answer these questions we look for the longest path or the shortest path between pairs of graph nodes. In either case, the answer to the query is the total length of the path, which is the sum of the labels on the arcs in the path. (Recall that arc labels represent dynamic delays associated with state transitions.)

The problem of finding the longest or shortest path in a directed acyclic graph (or exhibiting a cycle if the graph is not acyclic) can be solved in time linear in the number of nodes and arcs in the graph. (See, for instance, [24].) 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.

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.

We use three graph algorithms to answer timing queries about a program specification. The first algorithm answers queries about the maximum delay between events by computing the longest path between nodes in a reachability graph. This algorithm, which is linear in the number of graph nodes and arcs, is a variation of a well-known linear-time algorithm for computing single-source shortest or longest paths in directed, acyclic graphs (see, e.g., [24]).

In our implementation, the algorithm takes as input two graph nodes, s and *goal*, and returns a vector containing the lengths of the longest path from node s to every node reachable from s without crossing *through goal*. We accomplish this by removing all arcs out of *goal* before running the single-source longest paths algorithm. If there are any cycles in the portion of the graph reachable from s, then instead our algorithm terminates with a report that a cycle was found (and tells whether the particular cycle found had zero or nonzero length).

The key aspect of our longest path algorithm is that it first performs a topological sort of the nodes contained in a reachability graph. Next, we iteratively apply a simple procedure to the sequence of graph nodes produced by the topological sort. The overall complexity of this algorithm is $O(N + A)$, where N and A are the node set and the arc set of the reachability graph.

When answering minimum-delay queries, we first apply a shortest-path algorithm that is a simple variation of the previous algorithm for longest paths in acyclic graphs. As with the first algorithm, the shortest path algorithm either returns a vector containing the length of the shortest paths from node s to every node reachable from s without crossing through *goal*, or signals the presence of loops and returns an empty vector in linear time. Details of this algorithm are reported elsewhere [24].

If the linear-time shortest-path algorithm detects a cycle while answering a minimum-delay query, we apply Dijkstra's algorithm to compute the shortest path from node s to every other node in a reachability graph [25]. Unlike the linear-time algorithm, Dijkstra's algorithm works even when a graph has cycles; however, Dijkstra's algorithm has worse computational complexity than the previous algorithm. (Our implementation is quadratic in the number of graph nodes.)

4.3 Query language

The reachability graph of an STP net supports the analysis of various kinds of properties. Here we are interested in verifying both safety properties of graph nodes and properties involving minimum and maximum delays between events in the underlying real-time system.

When checking node properties, we use queries consisting of two components. The first component is an arbitrary boolean formula on a set V of boolean variables. Each variable $v_i \in V$ corresponds to a place

p_i of the STP net. Variable v_i is true in a given net state if and only if the corresponding place p_i is marked in that state. For instance, the predicate below is true of states in which either place p_1 is marked and p_2 is not marked, or both places p_3 and p_4 are marked:

$$(v_1 \wedge (\neg v_2)) \vee (v_3 \wedge v_4)$$

The second (optional) component of a node query involves the dynamic delays of enabled transitions in the states of an STP net. Given a graph node n, each enabled transition t_j in state n is associated with a nonnegative numeric variable z_j, which gives the dynamic delay of t_j in n. The second component of a node query is an arbitrary predicate on a set of variables such as z_j. For instance, the following predicate is true of states (i.e., nodes) in which the dynamic delay of transition is t_1 is strictly less than 10, and the combined delay of t_2 and t_3 is greater than 8:

$$(z_1 < 10) \wedge ((z_2 + z_3) \geq 8)$$

We use the following algorithms for answering node queries. Given a reachability graph $\mathcal{G}$ and a query involving a safety property P, PARTS returns a set of $\mathcal{G}$ nodes that satisfy P. To answer questions about mininum and maximum delays, we use the three algorithms discussed earlier. For longest path queries, if the linear time algorithm detects a cycle between α and β, we report that the longest path between the two nodes is infinite. For queries of the form, "What is the shortest possible time between states α and β?" we try the linear time algorithm for shortest paths in an acyclic graph first and Dijkstra's algorithm next.

Note that all our graph algorithms involve finding the minimum or maximum delay from a single state to other states. In some cases, we are also interested in finding out the delays from many states (e.g., all states satisfying a given predicate) to other states. When this happens, we first run a safety query to find all the states that satisfy the predicate. Subsequently, we run a timing query for each state returned by the first query. Thus far, the need to run multiple timing queries has not been a problem because the number of source states of interest is generally small. Of course, we could easily switch from single-source shortest and longest path algorithms to multiple-source or all-source algorithms, should the need arise.

5 Reducing state space explosion

We considered the following optimization methods in an effort to control the state explosion problem in our reachability graphs: a net reduction technique [55], the stubborn set method [56], and a net symmetry method [57]. All techniques extend existing methods that were originally defined for controlling the state explosion problem in the analysis of untimed concurrent systems [57, 59, 62]. A detailed discussion of the original methods and of our extensions is beyond the scope of this paper. Here we summarize only the main aspects of each method. The interested reader is referred elsewhere [26, 55, 56] for additional details.

5.1 Net reduction

One way to reduce the size of a Petri net's state space is by reducing the size of the net. The *net reduction method* seeks to shrink an STP net, while preserving properties of interest, such as the possibility of deadlock and timing properties, 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 [9]. Berthelot's method was subsequently extended and applied to the problem of deadlock detection in Ada tasking programs by Tu et al. [59]. Their method exploits Ada semantics to speed up the application of Berthelot's rules, some of which have high computational complexity. Moreover, their reduction method played a significant role in their obtaining successful analysis results [26].

Our reduction method extends both Berthelot's original method and the Ada-based technique of Tu et al. Similar to those methods, our reduction method is based on a set of reduction rules, which are iteratively applied to a Petri net until no rule can be applied. STP nets generated by ATS exhibit a uniform structure because similar subnets model different occurrences of a given construct in a specification. Similar to Tu et al., we exploit this fact by associating semantic information with each place in an STP net and we use this semantic information to carry out net reductions.

As with previous reduction methods, our goal is that the reduced Petri net should be equivalent to the original one with respect to some specified properties of interest. A significant difference from previous work is that the goal of previous researchers was to preserve only the possibility of deadlock, whereas we are also interested in preserving timing properties, such as minimum and maximum delays. 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 any given length are in the state space of the reduced net whenever they are present in the state space of the original net.

We have extended the net reduction method in several significant ways, so as to preserve timing properties in a reduced Petri net model. First, we have introduced a formal notion of equivalence between a time Petri net and a reduced time Petri net.[5] Second, we have considered all reduction rules defined by previous

[5] Because our STP nets are a special case of time Petri nets, this notion of equivalence is applicable to STP nets as well.

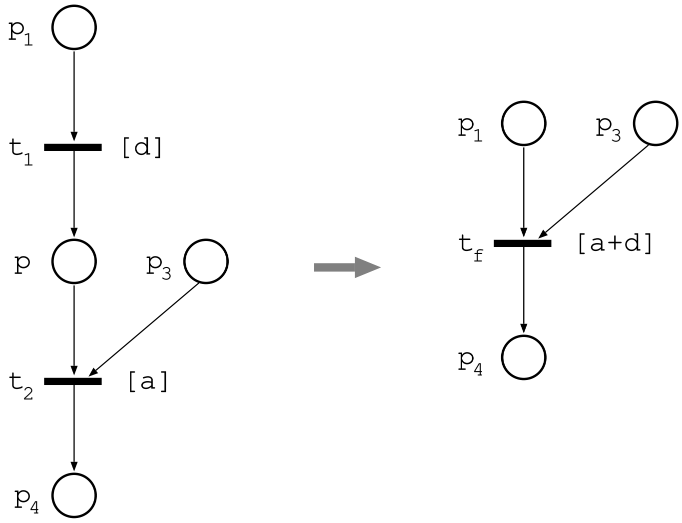

Figure 4: Illustration of pre-fusion reduction rule. Here the set $S = \{t_2\}$.

investigators [9, 59]. Rules that could be fruitfully applied to time Petri nets have been suitably extended [55]. Finally, we have defined new reduction rules specifically for time Petri nets.

Each of our reduction rules guarantees the following notion of equivalence. Given a time Petri 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. For the two nets to be equivalent 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 pre-fusion rule—to illustrate the difficulties introduced by the explicit modeling of time in our STP nets. This rule was originally defined by Berthelot [9] for untimed Petri nets. It says 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 4 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. Figure 4 also shows a delay associated with each transition in square brackets. Of course, these delays are irrelevant to Berthelot's definition of the rule because that definition is based on ordinary (untimed) Petri nets.

When the rule is extended to STP nets, a difficulty 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 4). 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 of 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 (or more) 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. $q^\bullet = \{t\}$, and $t \notin {}^\bullet q$.
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). [6]

[6] The pre-fusion rule discussed here is a special case of the pre-fusion rule that we defined for time Petri nets [55]. In the general case, the additional condition requires that the delay interval of transition t_1 be either $[0, 0]$ or $[d, d]$ (with the nice-tree condition), for some nonnegative constant d.

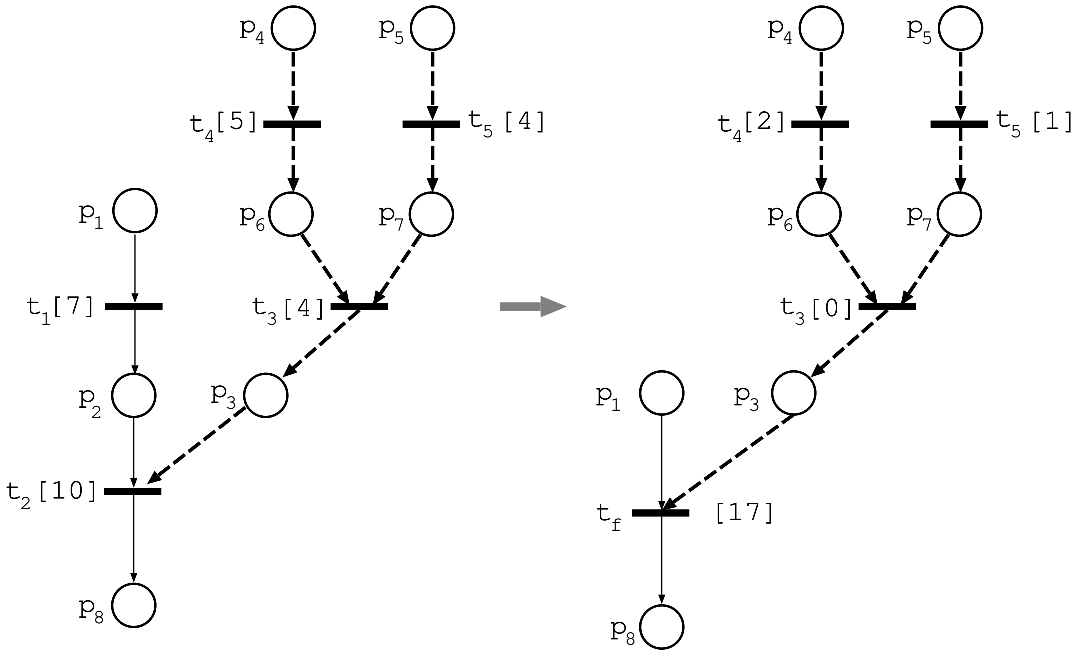

Figure 5: Example of nice-tree condition *nice-tree*$(t_2, p_3, 7)$ in the pre-fusion rule. The nice tree is highlighted by the dashed arcs.

We show an example of the nice-tree condition in Figure 5. In this case, the appearance of a token in place p_3 must be advanced by seven time units, the delay of transition t_1. This is accomplished first by reducing the delay of transition t_3, which is an input to p_3, from four to zero. Three additional time units are obtained by reducing t_4's delay from five to two and t_5's delay from four to one. Of course, transitions t_3, t_4, and t_5 must not have any output places outside the nice tree in order for the rule to work.

A proof that this rule satisfies the conditions for the equivalence between the reduced net and the original net appears in [55]. 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 [9] carry over directly (i.e., without restrictions) to STP nets. In all cases, we can apply reduction rules only if they do not remove places that our timing or safety queries refer to. Additional details on our reduction method can be found elsewhere [55].

5.2 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 $\Theta(2^n)$ to $\Theta(n)$. This is the central idea behind the *stubborn set method* [61, 62]. Consider, for instance, the Petri net in Figure 6. This

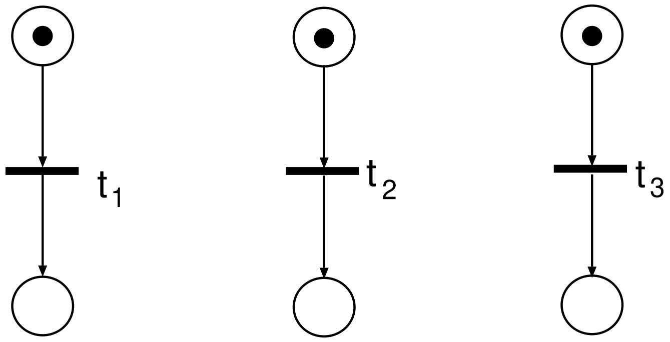

Figure 6: Example of a Petri net containing independent transitions.

net has six possible firing sequences (e.g., $t_1t_2t_3$ and $t_2t_3t_1$). All sequences lead to the same state, however. If only this state is of interest, it is therefore sufficient to explore only one of the possible sequences.

Valmari invented the stubborn set method and proved its validity for untimed concurrent systems, first in the context of Petri nets and then for arbitrary variable/transition systems. He obtained impressive results in the analysis of many concurrent systems [61, 62]. Subsequently, Duri et al. showed that this method combines synergistically with other methods for state-space reduction of concurrent programs [26].

The stubborn set method exploits the notion of *independence* among fireable transitions. Roughly speaking, two transitions are independent when firing one transition has no effect on the fireability of the other transition and vice versa. For instance, all three transitions in Figure 6 are independent of each other. To use this method, in each net state *x* we must partition the transition set of the Petri net into two independent subsets, called the *stubborn* set and the nonstubborn set. Only fireable transitions in the stubborn set need to be fired to find the successors of *x* in the reachability graph of the net. Of course, as with all our analysis methods, this subset must be chosen based on what queries one later intends to ask.

Valmari defined formal conditions for a transition subset to be stubborn in a net state [62]. In brief, the conditions are aimed at enforcing the following criteria. First, a stubborn set must contain at least one enabled transition. Second, 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 set and a nonstubborn set). In the state shown in Figure 6, 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 fireable transition contained in that set are fired. In the example, one of t_1, t_2, and t_3 is fired.

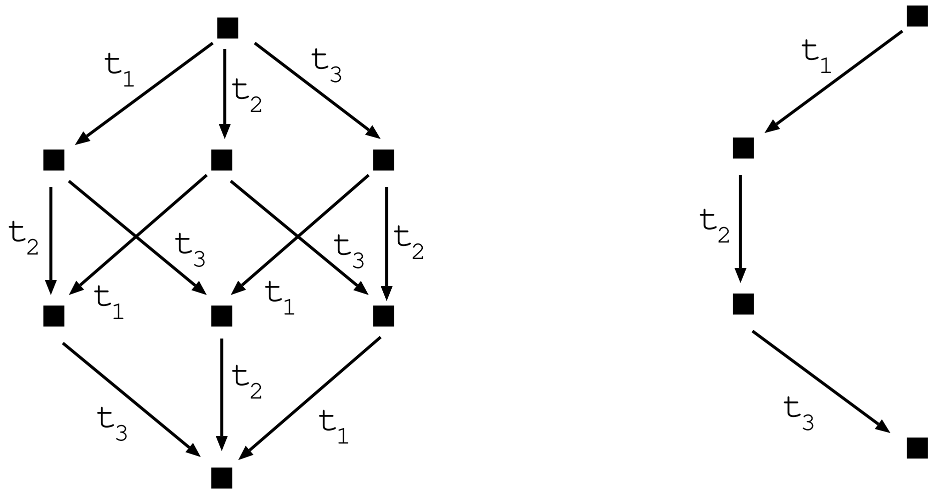

Figure 7: Unreduced and reduced state spaces of the Petri net shown in Figure 6.

Figure 7 illustrates the effects of the stubborn set method on the Petri net shown in Figure 6. The left-hand side of Figure 7 shows the full state space, which contains all six sequences of transition firings. The right-hand side, which contains only one such sequence, shows a reduced state space.

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 [62], 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 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 guaranteed only that *untimed* queries would be answered correctly, and substantial work was required to prove that timing properties were also preserved.) The interested reader is referred to [56] for a complete discussion of our extensions.

5.3 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 [57]. 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). In brief, we use symmetry functions to partition the node set of an STP net into equivalence classes. We then use the equivalence classes of places to identify equivalent states in the reachability graph, as we discuss below. We accomplish a reduction in state space size by including in the reachability graph only one state from a set of equivalent states.

Here is a more detailed discussion. 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 an arc a from node x to node y, 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 [57]. 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. In addition, symmetry mappings can be used to partition the set of reachable states of a 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 [57].

Starke's definitions provide an excellent theoretical foundation for reduced state space generation. In particular, Starke's method guarantees that a reduced state space is equivalent with respect to deadlock to the full state space. However, it is difficult to apply that method in practice because of its high computational complexity.

In order to apply Starke's method, 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 symmetric, in our terminology). Our heuristics are a straightforward extension to the timed domain of similar heuristics that were used by

Duri et al. [26]. First, we identify the equivalence classes of places and transitions using Starke's algorithm [57]. 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 conditions for the existence of a symmetry mapping between s_1 and s_2. In all the examples we considered, if these conditions are met, s_1 and s_2 are equivalent states. Although we have not proved the sufficiency of our three conditions for state equivalence, we conjecture that the conditions are indeed equivalent based on extensive empirical studies with PARTS and with the TOTAL toolset for deadlock detection [26].

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. (A token corresponding to a given process captures the flow of control in the process.) 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 $\mathcal{P}$. [7] The second (behavioral) condition for two states s_1 and s_2 to be equivalent stipulates that each place class in $\mathcal{P}$ must have 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.

We sometimes apply the net symmetry method in combination with stubborn sets. In this case, the algorithm for node expansion behaves as follows. First, we identify a stubborn set of minimal size and we consider each fireable transition in the set. If the firing of a stubborn transition leads to a state equivalent to an existing state, we do not include the new state in the graph. This means that we apply stubborn sets followed by net symmetry when expanding each graph node. The opposite order would result in unnecessary work because we would end up exploring all fireable transitions, including non-stubborn transitions, if we applied net symmetry before stubborn sets. Evidently, a computed state reached by following a non-stubborn transition would not be included in resulting state space even if net symmetry failed to eliminate this state.

6 Experimental results

We ran our experiments on the PARTS toolset for real-time analysis. Figure 8 shows the flow of information in the toolset. An Ada-like specification to be analyzed is input into ATS, which generates an STP-net model [63]. The net output by ATS is then fed into the timed net reduction (TNR) tool, which produces

[7] Our concept of a process is different from a Petri net place invariant or P-invariant [48]. A P-invariant is a place subset such that the sum of the tokens associated with places in the subset remains constant in all reachable net states. The sum of the tokens in a process can add up to zero or one, depending on the net state.

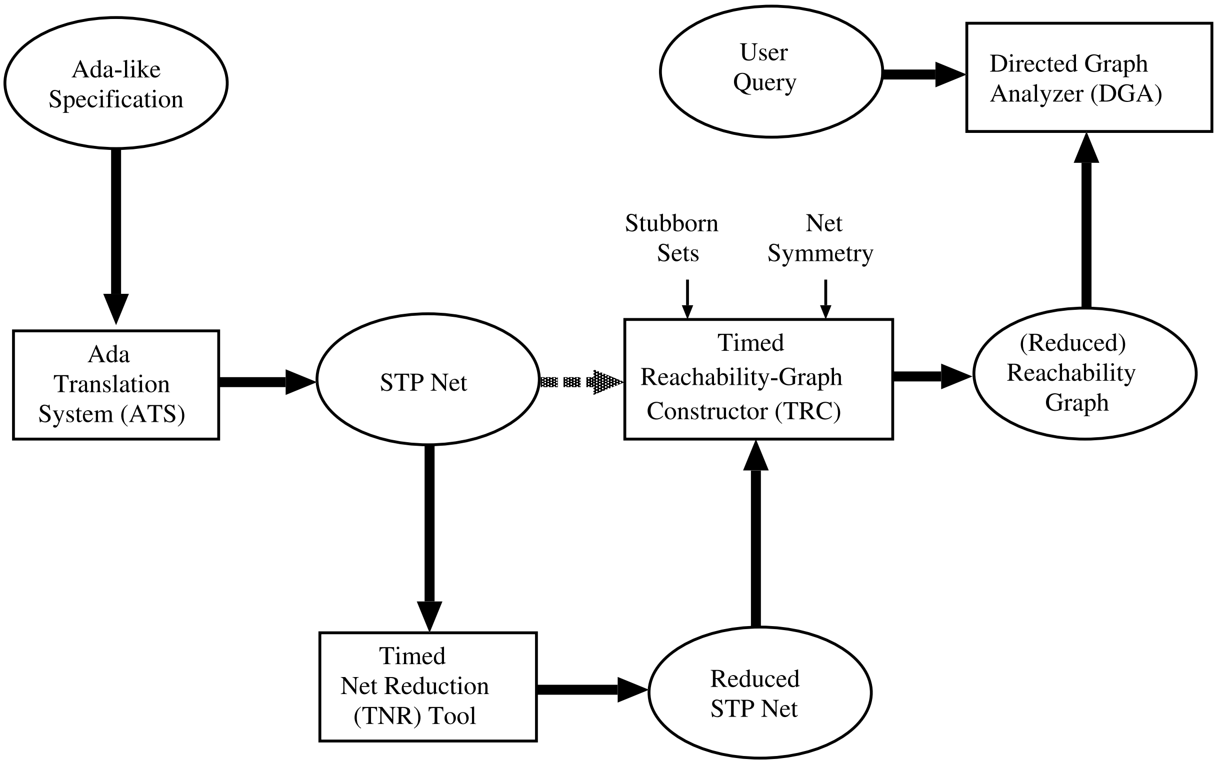

Figure 8: Diagram of PARTS toolset.

a reduced STP-net model. The Timed Reachability-graph Constructor (TRC) tool takes as input an STP net and generates the state space of the input net. This tool also implements the net symmetry method and real-time stubborn sets. These methods can be optionally used during reachability graph construction to reduce state space. Finally, the directed graph analyzer (DGA) tool computes shortest and longest paths in the reachability graphs. Note that an STP net output by ATS can be fed directly into TRC for experiments involving unreduced STP-net models of input specifications. All tools were implemented in the language CommonLISP except for ATS, which was written in C++.

The goals of our experiments are twofold. First, we seek to assess the overall strength of our integrated approach to real-time analysis of Ada-like specifications. Second, we wish to investigate the effects of the optimization methods that we considered on the effectiveness of the analysis.

For our experiments we considered a set of traditional examples of real-time systems and we coded them using our specification language. The examples include a simple divide-and-conquer problem [23], a simple network router [23], a generalized version of Leveson's *railroad crossing* example [34], a traffic-light system [20], and Ostroff's delayed reactor trip [51].

For each example we ran several experiments using different versions and different combinations of optimization methods. Every experiment starts from an Ada-like specification and automatically generates a (possibly reduced) reachability graph for the problem. All experiments were run on a Sun Ultra Sparc Station 1 with 64 Mbytes of memory, except for the translation into STP nets, which was performed on a Sun Sparc Station 10 with 32 Mbytes of memory.

Here we report on the most expensive part of the analysis—reachability-graph construction. For each experiment, we give the number of nodes and arcs in the reachability graph of the corresponding STP net, and the time required for reachability graph construction and (when applicable) net reduction on the Sun Ultra Sparc Station. We do not report on the time required by translation into STP nets, or the time required for graph analysis, because these times are much smaller than the times for net reduction and reachability-graph construction. For instance, the longest time for any translation is 16.1 seconds for the network router problem with 300 columns of routing elements (i.e., 602 Ada tasks) on the Sun Sparc Station 10. In this case, the best analysis time is over 700 seconds on faster hardware, our Sun Ultra Sparc Station 1. Times for graph analysis are even smaller than for translation.

6.1 Divide-and-conquer

The divide-and-conquer example models n computing elements [23]. When a problem must be computed, the first element nondeterministically chooses whether to solve the problem directly (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 solve the problem. The Ada-like specification for a small version of this problem is given in Appendix A.

The most relevant aspect of the divide-and-conquer example is given by the computation times for solving versus passing the problem to the next element. To capture this example, we relax our assumption that computation times are negligible for this example. As we mentioned earlier, we use *delay* statements in our specifications to define the amount of time required by computations.

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 seconds of CPU time taken by reachability-graph construction and (when applicable) net reduction. As we mentioned before, Table 1 does not include the times for translation from Ada-like specifications to STP nets and for shortest and longest path computations, because these times are much smaller than the corresponding times for reachability-graph construction and for net reduction.

The first column in Table 1 gives the number of computing elements in each version we analyzed. The following column denotes the variable being measured. The remaining columns correspond to different

Problem		Optimization methods: none	stub	sym	nr	nr+stub	nr+sym	nr+stub+sym
D & C 10	*nodes*	8,653	290	8,653	29	29	29	29
	arcs	37,134	289	37,134	37	37	37	37
	secs	34.1	0.5	69.2	0.2	0.2	0.3	0.3
D & C 20	*nodes*	—	1,085	—	59	59	59	59
	arcs	—	1,084	—	77	77	77	77
	secs	—	3.2	—	0.5	0.5	1.7	1.7
D & C 40	*nodes*	—	4,175	—	119	119	119	119
	arcs	—	4,174	—	157	157	157	157
	secs	—	27.0	—	1.7	1.8	10.0	9.8
D & C 60	*nodes*	—	9,265	—	179	179	179	179
	arcs	—	9,264	—	237	237	237	237
	secs	—	124.5	—	3.8	3.8	30.8	30.5
D & C 80	*nodes*	—	16,355	—	239	239	239	239
	arcs	—	16,354	—	317	317	317	317
	secs	—	403.3	—	6.0	5.9	71.2	70.1
D & C 100	*nodes*	—	—	—	299	299	299	299
	arcs	—	—	—	397	397	397	397
	secs	—	—	—	8.7	8.9	135.6	135.7
D & C 200	*nodes*	—	—	—	599	599	599	599
	arcs	—	—	—	797	797	797	797
	secs	—	—	—	34.2	34.2	1,030.0	1,072.7
D & C 500	*nodes*	—	—	—	1499	1499	—	—
	arcs	—	—	—	1997	1997	—	—
	secs	—	—	—	208.8	217.4	—	—

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 either because it was taking longer than 20 minutes of CPU time or because LISP address space was exhausted.

combinations of optimization methods. In particular, column 3 (none) has data for full state space generation. Column 4 (stub) corresponds to stubborn sets. Column 5 (sym) corresponds to net symmetry. Column 6 (nr) corresponds to net reduction. Column 7 (nr+stub) gives the data for the combination of net reduction and stubborn sets. Column 8 (nr+sym) has the data for the combination of net reduction and net symmetry. Finally, column 9 corresponds to the combination of all three optimization methods.

Note that the growth in state space size is exponential in the number of computing elements when no optimization method is used. In this case, we obtain $(\frac{17}{2}2^n - 4n - 11)$ nodes. The symmetry method never achieves 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 $(\frac{5}{2}n^2 + \frac{9}{2}n - 5)$ nodes.

State space size becomes linear in *n* when net reduction is used. In this case, we obtain $(3n - 1)$ nodes.

Note, however, that the growth in the time for state space generation is roughly quadratic, rather than linear. The reason for this is that the time for generating a new state is usually proportional to the size of the STP net, which is linear in n in this case. The largest version that we analyzed has 500 computing elements.

Our results compare favorably with the results of Corbett and Avrunin's constrained expression toolset [23]. That toolset uses two methods for providing upper bounds on the time between events in a distributed real-time system based on a maximal parallelism assumption. The largest version that they analyze has 100 computing elements.

Avrunin et al. [3] discuss timing analysis of concurrent systems running on a uniprocessor architecture. For that analysis they used a modified version of the constrained expression toolset. Their empirical results are very similar to ours. The largest version they analyze has 500 computing elements, and this analysis takes 555 seconds of CPU time on a Sparc Station 10.

Finally, our current results show substantial improvements over our earlier, preliminary results [17]. This is in spite of the fact that our STP nets are now generated from translation rather than being hand-coded. The largest version that we analyzed with an earlier version of PARTS had only 100 computing elements [17].

6.2 Network Router

The network router example consists of a $(2 \times n)$ matrix of routing elements [23]. 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. According to Avrunin et al., this is the structure of a fault-tolerant telephone network in urban India [3]. The problem is to find the shortest and the longest time that a packet can take while going from the sender to the receiver.

In all of our experiments with the network router example we have used the same delays that Avrunin et al. use for their experiments with the constrained expression toolset [3]. Thus, the delays on packet transmission are 10 time units for each transmission, except for the transmissions from elements 5, 10, 25, ... in the first row to elements 6, 16, 26, ... in the second row, which were assigned a delay of 20 time units. In addition, the transmissions from elements 6, 16, 26, ... in the second row to elements 7, 17, 27, ... in the first row were also assigned a delay of 20 time units. Figure 9 illustrates the case of a router problem

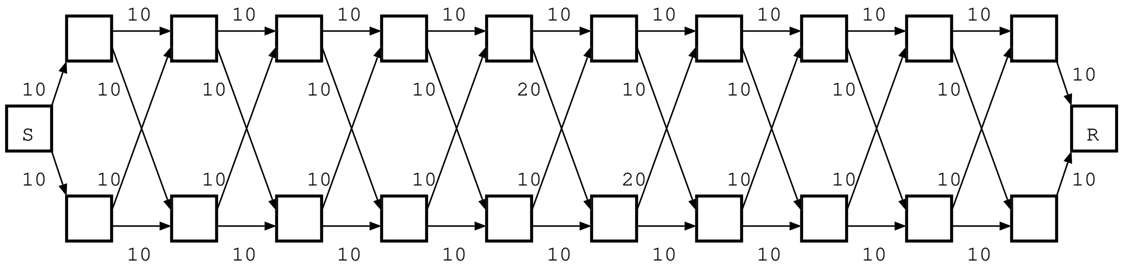

Figure 9: Network router problem with 10 columns of routing elements.

with 10 columns of router elements.

In Table 2 we report our experiments with the network router example. This table is similar to Table 1, except that we added a column for the combination of stubborn sets and net symmetry. We considered various sizes of this problem. The first column in the table gives n, the number of columns of routing elements that we considered. The other columns are as in the previous table. Each problem size we considered has $(2n + 2)$ Ada tasks, namely, a sender, a receiver, and two elements for each column.

The state space for the network router grows exponentially when no optimization method is used. Thus, we could not complete state space construction even for the case of 10 columns of routing elements. The stubborn set method dramatically improves our analyses. Even when applied to an unreduced STP net, this method can reduce state-space growth to linear in the number of routing elements. The state space now has $(32n - 6)$ states. As with the divide-and-conquer example, the growth in the times is roughly quadratic because the time to generate each state also grows linearly in the size of the STP net.

This problem exhibits some symmetries among routing elements that are located in the same column and that use identical delays to send packets to routing elements in the next column to the right. However, the growth in state space size remains exponential when the net symmetry technique is used by itself.

Experiments with a combination of net symmetry and stubborn sets show an additional problem of the symmetry method. When these two methods are combined, smaller state spaces are obtained than in the case of the stubborn set method alone. However, the times required by state space generation are much larger when the symmetry method is also used. The additional time is for finding equivalence classes of places and transitions. Owing to the size of our unreduced nets for this example, this time grows quite rapidly as the number of routing elements is increased. The same effect happens when the net symmetry method is applied to reduced STP nets, as is shown in the last two columns of Table 2.

The PARTS toolset does much better when net reduction is used. In this case, state space growth becomes linear in the number of routing elements even if stubborn sets and net symmetry are not used. In particular, we obtain $(34n - 34)$ states with net reduction alone. If the stubborn set method is combined with net

n		none	stub	sym	stub+sym	nr	nr+stub	nr+sym	nr+stub+sym
		Optimization methods							
10	*nodes*	—	314	—	164	306	174	134	87
	arcs	—	332	—	176	456	192	212	99
	secs	—	1.5	—	11.3	0.9	0.6	4.2	3.9
20	*nodes*	—	634	—	323	646	354	269	172
	arcs	—	672	—	348	976	392	432	197
	secs	—	4.2	—	86.0	3.0	1.8	29.1	27.7
50	*nodes*	—	1,594	—	—	1,666	894	674	427
	arcs	—	1,692	—	—	2,536	992	1,092	491
	secs	—	33.5	—	—	22.2	12.1	641.4	624.4
100	*nodes*	—	3,194	—	—	3,366	1,794	—	—
	arcs	—	3,392	—	—	5,136	1,992	—	—
	secs	—	163.5	—	—	108.2	52.5	—	—
150	*nodes*	—	4,794	—	—	5,066	2,694	—	—
	arcs	—	5,092	—	—	7,736	2,992	—	—
	secs	—	438.3	—	—	280.7	134.0	—	—
200	*nodes*	—	6,394	—	—	6,766	3,594	—	—
	arcs	—	6,792	—	—	10,336	3,992	—	—
	secs	—	921.6	—	—	596.2	251.2	—	—
300	*nodes*	—	—	—	—	—	5,394	—	—
	arcs	—	—	—	—	—	5,592	—	—
	secs	—	—	—	—	—	738.5	—	—

Table 2: Experimental data showing state-space reduction for the network router example. Again, secs indicates CPU seconds required to generate the state space, and a "—" indicates that the state-space generation was terminated because it exhausted some computational resource.

reduction, state space sizes are further reduced to $(18n - 6)$ nodes. The largest version we analyzed has 300 columns of routing elements and 602 Ada tasks (see Table 2).

For the network router example we also checked the maximum time that a packet can take to go from the sender to the receiver. In particular, we considered the reduced STP net of the router version with 100 elements, and we ran the DGA tool on the state space of this net. DGA found 1210, the correct answer, in 0.2 CPU seconds on a Sun Sparc Station 10.[8]

Our experimental results on the network router example are comparable with the results obtained by the constrained expression approach. Recently, Corbett and Avrunin [23] reported experimental results with the toolset version that conforms to the maximal parallelism assumption, similar to PARTS. In one instance, that toolset analyzed a network router problem with 100 columns in 2,032 seconds of CPU time on a DECstation 5000/125. Avrunin et al. also report on results obtained with the toolset version that uses a uniprocessor

[8]This is the correct answer because the delays in the net cause a packet to zigzag between the two rows every ten columns of routing elements. Thus, the longest delay for sending a packet from the sender through the first 10 adjacent columns is 120 units. Subsequently, the same delay occurs each time the packet travels through 10 adjacent columns. Finally, sending the packet from the last column to the receiver adds 10 more units to the total delay.

model of computation [3]. In one instance, that toolset analyzed a router problem with 300 columns in 486 seconds of CPU time on a Sun Sparc Station 10.

Finally, our current experimental results show a definite improvement with respect to our previous results [17]. The largest version that we analyzed with an earlier version of PARTS has 100 columns. Our improvement is especially significant considering that our previous results were obtained with hand-generated Petri nets. Those nets were much smaller than the nets generated automatically by our Ada translator. For instance, our (unreduced) hand-generated STP net for the 100-column version of the network router had 401 places and 598 transitions, but the STP net that ATS automatically generates from Ada code has 3001 places and 2597 transitions. (In this case, the reduced STP net output by TNR has 1797 places and 1794 transitions.) This discrepancy happens because the structure of the network router problem can be modeled in a straightforward manner with STP nets, but using Ada tasks for each routing element significantly increases the complexity of the models.

6.3 Generalized Railroad Crossing

The generalized railroad crossing (GRC) system is explained in Subsection 3.1. In general, there may be n parallel tracks in 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.

To verify the safety property, we searched the GRC's state spaces for states satisfying queries of the form $\neg v_0 \wedge (v_1 \vee \ldots \vee v_n)$. Propositional variable v_0 is true when the place indicating that the gate is closed has a token. Variables v_1 through v_n indicate that a train is in the crossing for each of the n tracks. Of course, in a correct GRC system no states satisfy the query.

We verified the utility property by using our shortest and longest path algorithms on GRC's state spaces. First, we computed the minimum delay between a gate reaching the closed position and a train entering the crossing. Next, we computed the maximum delay between a train leaving the crossing and the gate starting to open, assuming that no other trains are in the crossing. Both values must be zero in a correct GRC system.

In Table 3 we report data for one-track, two-track, and three-track versions that satisfy both properties. This table is similar to Table 2. The net reduction technique and stubborn sets provide similar reductions in state-space size. (Compare the columns labeled *none*, *stub*, and *nr* in Table 3.) Additional reductions are obtained when these two methods are combined. In the versions with multiple tracks, the net symmetry method can reduce the state space even further because of the symmetry among the tracks. Thus, in these versions, the best results are obtained when all methods are used in combination. Unfortunately, however,

Problem		Optimization methods							
		none	stub	sym	stub+sym	nr	nr+stub	nr+sym	nr+stub+sym
GRC 1	*nodes*	243	184	243	184	114	103	114	103
	arcs	331	204	331	204	146	122	146	122
	secs	0.2	0.1	0.5	0.3	0.1	0.1	0.1	0.2
GRC 2	*nodes*	10,616	4,063	5,418	2,237	4,371	2,652	2,237	1,452
	arcs	19,701	4,421	10,112	2,474	7,049	3,078	3,646	1,747
	secs	12.9	4.0	12.6	4.3	3.8	2.6	3.7	2.5
GRC 3	*nodes*	—	132,926	80,505	25,037	166,285	86,265	29,571	16,267
	arcs	—	143,564	175,762	27,377	298,738	99,243	53,956	19,109
	secs	—	294.6	355.9	63.0	269.3	183.2	70.7	35.8

Table 3: Experimental data showing state-space reduction for generalized railroad crossing example. Again, secs indicates CPU seconds required to generate the state space, and a "—" indicates that the state-space generation was terminated because it exhausted some computational resource.

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 dynamic delay and one transition, t_L, with a large dynamic 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 (i.e., the amount of time taken by firing the transitions with small delays). In this case, transitions that were fireable in s_1 are again fireable in s_2, which results in a replication of the reachability subgraph originating at s_1. Moreover, this subgraph 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.

Other authors have considered the GRC problem. For instance, Yang et al. report on analysis results of two one-track versions using symbolic model checking [64]. Both analyses are completed in 0.4 seconds of CPU time on a Sun Sparc Station 1. Their approach is not afflicted by the time granularity problem; however, it is unclear whether their approach scales up to multiple-track versions. Finally, Henzinger reports on the analysis of a version of one-track GRC problem with the HyTech toolset [35]. (See discussion in Subsection 8.2.)

6.4 Traffic light

This example models a traffic light placed at the intersection of two roads. It was originally defined by Cha et al. to illustrate an experimental methodology to aid in the development of time-critical systems [20]. They described the traffic light system as follows.

> A traffic light control system at an intersection consists of four (identical) sensors and a central controller. The sensors in each direction detect cars approaching the intersection. If the traffic light currently is not green, the sensor notifies the controller so that the light will be changed. A car is expected to stop and wait for a green light. If the light is green already, the car may pass the intersection without stopping. The controller accepts change requests from the four sensors and arbitrates the traffic light changes. Once the controller changes the light in one direction (East-West or South-North) to green, it maintains the green signal for five seconds so that other cars in the same direction may pass the intersection without stopping. Before the green light in any direction becomes red, it should remain in yellow for one second so that any car present in the intersection may clear. The light then turns to red while the light in the other direction turns green.

Cha et al. also give a sample Ada implementation of the traffic light system [20]. The system has four sensor tasks and a controller task. We could not input their code directly into ATS, because that code relies on a global variable for communication among tasks. The variable is set by the controller task and read by the sensor tasks. Given that ATS does not currently model global variables, we are forced to use the rendezvous mechanism for communication between tasks. In our code each of the four sensor tasks performs an indefinite loop in which entry *notify* of the controller task may be called nondeterministically. The direction of traffic (i.e., East-West vs. North-South) monitored by each sensor is passed as a parameter in the calls to *notify*. As usual, we model the time intervals during which the light must remain yellow or green with *delay* statements.

We analyzed a correct and an incorrect version of this example. The incorrect version is the result of a deliberate error, which was introduced by Cha et al. for the purpose of illustrating verification features of their development methodology [20]. When a car approaches the intersection from a direction with a red light, the controller sometimes switches the light to green in that direction and to red in the perpendicular direction without first switching to yellow in the perpendicular direction. As a result, cars can collide in the intersection. This cannot happen in the correct version of this example.

Our experimental results with the traffic light example are shown in Table 4. This example is somewhat simpler than the larger versions of the previous examples. Thus, PARTS can build the full state space even when no heuristic methods are applied. The net reduction method seems to shrink the state space by a factor of about five, except when it is combined with stubborn sets. In this case, net reduction is much less effective. This fact seems to indicate that there is some overlap in the reductions that are obtained by net reduction and stubborn sets. The stubborn set method is extremely effective in this example; this method causes state

Problem		none	stub	sym	stub+sym	nr	nr+stub	nr+sym	nr+stub+sym
		Optimization methods							
correct	*nodes*	92,998	2,923	25,042	1,111	19,764	1,905	5,506	706
	arcs	354,788	3,421	96,396	1,315	69,310	2,627	19,488	996
	secs	480.9	5.5	137.1	4.8	43.9	3.4	25.6	3.4
incorrect	*nodes*	101,171	2,265	25,883	844	22,840	1,593	5,901	569
	arcs	400,458	2,697	103,898	1,023	84,330	2,273	22,108	842
	secs	494.9	4.4	152.3	4.8	52.2	3.0	29.0	3.4

Table 4: Experimental data showing state space reduction for traffic light example.

space reductions exceeding an order of magnitude both in case of reduced and unreduced STP nets. Finally, the symmetry method also causes some state space reductions because of the symmetry between pairs of sensor tasks. (The two sensors in each traffic direction are identical.) The best results are obtained when all heuristics are applied in conjunction. In this case, we obtain 706 nodes and 996 arcs for the correct version, and 569 nodes and 842 arcs for the incorrect version. We consider these to be excellent results, given that the full state spaces are two orders of magnitude greater than the reduced state spaces.

6.5 Delayed Reactor Trip

The Delayed Reactor Trip (DRT) example models a shutdown system in a nuclear reactor. The DRT system monitors the pressure and power levels in the reactor. When both values exceed a given threshold, the DRT system opens a relay that shuts down the reactor. This example was first introduced by Lawford and Wonham [45]; Ostroff analyzed this example using the Statetime toolset [51].

The behavior of the system should meet the following requirements, aimed at avoiding unnecessary shutdowns that can result in financial loss for the company operating the reactor [51]. When both the power and pressure of the reactor exceed their acceptable safety limits, the DRT system waits for 3 seconds. If the power level remains higher than its safety limit after 3 seconds, the relay is opened (i.e., regardless of the pressure level). In this case, the relay must remain open for at least 2 seconds. After this, the relay must be closed as soon as the power reaches its safe level. Finally, the controller should never deadlock.

We considered two versions of this example. Each version has four tasks, namely two sensors, a controller, and a relay. In the first version we faithfully reproduced Ostroff's coding of the example, which turns out to be unnecessarily complicated in Ada. In particular, the controller executes an iteration of its main loop every 100 milliseconds and uses two integer variables as timers. The first timer, which ranges from 0 to 30, measures the time elapsed since power and pressure were found to exceed their safe levels. The second timer, which ranges from 0 to 20, measures the time elapsed since the relay was opened. The main differ-

Problem		Optimization methods							
		none	stub	sym	stub+sym	nr	nr+stub	nr+sym	nr+stub+sym
counters	*nodes*	109,127	9,791	—	9,791	22,443	5,512	22,443	5,512
	arcs	269,360	10,190	—	10,190	47,132	5,911	47,132	5,911
	secs	1,342.3	41.7	—	203.1	59.2	22.6	241.3	168.5
no counters	*nodes*	12,281	1,866	12,281	1,866	2,407	1,099	2,407	1,099
	arcs	36,497	2,179	36,497	2,179	5,560	1,527	5,560	1,527
	secs	27.4	2.5	52.3	5.0	3.7	1.6	5.7	2.7

Table 5: Experimental data showing state space reductions for Delayed Reactor Trip (DRT) example.

ence between our and Ostroff's code is that we must use Ada's rendezvous mechanism for communication among the four tasks, whereas Ostroff uses shared structures for this purpose [51].

In the second version we take advantage of Ada's *delay* statement to avoid using explicit clock variables. This version, whose Ada-like specification appears in Appendix B, is more natural and concise than the previous one. In both versions one time unit in our Ada code (and the resulting STP nets) corresponds to 100 milliseconds of elapsed time.

The results of our experiments with the DRT example are shown in Table 5. The PARTS toolset can analyze both versions even when no heuristics are applied. As we expected, the version with counters is more difficult to analyze than the version without counters. Yet, we are encouraged by PARTS's ability to complete the analysis of an example that includes two integer counters with much larger ranges than the other examples we considered. As with the traffic light example, the stubborn set method gives substantial state space reductions, even when combined with net reduction, especially in the version with counters. In contrast with the traffic light, the symmetry method does not produce any state-space reductions because the DRT example does not contain any symmetries. Consequently our best results are obtained by using net reduction and stubborn sets. In this case, PARTS generates the state space of the two versions in 22.6 and 1.6 seconds of CPU time.

These results are competitive with Ostroff's results. His Statetime toolset first translates visual models called TTMcharts into so-called Timed Transition Models (TTMs). These models are analyzed using either a reachability-based tool called *Verify* or Stanford's STeP theorem prover which uses deductive reasoning [11, 51]. Ostroff reports on the analysis of various properties of a correct and an incorrect version of the DRT system, both with counters. The most expensive analysis (checking that the relay is properly opened 3 seconds after power and pressure are found to exceed their safe values) takes over 35 minutes of CPU time on a Sun Ultra Sparc Station 1 with 160Mbytes of memory using Verify; STeP completes the same analysis in about 2 minutes of CPU time on similar hardware. However, STeP cannot verify freedom from deadlock, whereas Verify completes this analysis in about 2 minutes of CPU time. PARTS generates the state space in 22.6 seconds; it can then analyze various properties in negligible time.

7 Evaluation of experimental results

Our empirical results with the PARTS toolset carry some good news and some bad news on Petri-net-based real-time analysis of Ada-like specifications. On the positive side, our heuristics are quite effective at reducing state-space size. In the network router and the divide-and-conquer examples, each of the net reduction and the stubborn set methods can reduce the growth of the state space from exponential to polynomial in problem size. In the other examples we considered, we still observe dramatic reductions in state-space size when our heuristics are applied together. As a result, PARTS can usually handle most of the verification examples found in the literature in relatively short time on our Sun Ultra Sparc Station [3, 20, 23, 34, 51].

On the negative side, our experiments make it clear that we must overcome two current limitations of Petri-net-based real-time verification before we can apply this technology to large-scale industrial systems. First, we must overcome the time granularity problem that we discussed in Section 6.3. This problem leads to unacceptable dependencies between the static delays associated with net transitions and state-space size. Second, we must be able to conduct our analysis in a compositional fashion in order to handle real-world problems containing more than a few tasks. On the one hand, the issue of compositionality is common to both timed and untimed analyses. On the other hand, the time granularity problem is unique to real-time analysis; this problem did not arise when performing deadlock analysis of untimed models [26]. On the whole, our heuristic techniques for real-time analysis appear to be at least as effective as the original techniques for untimed concurrency analysis [26], except when the time granularity problem blows up the state-space size of timed models.

Here is a detailed discussion of each heuristic technique that we applied to real-time analysis. Our first observation is that our net reduction technique is always at least somewhat helpful. This contrasts with our earlier results where the net reduction technique sometimes gave dramatic improvements, but the rest of the time it did not help at all [17]. The difference is that we are now analyzing Ada-like specifications automatically translated into STP nets. In our preliminary work, we hand coded STP nets for the systems we analyzed.

The STP nets produced by the automatic translation of Ada-like specifications can almost always be made smaller. We expect this to be true for any system that automatically translates from a higher-level programming language to an internal representation. This situation is analogous to compilation. If you want to get compact assembly language code, you can either write in assembler or you can use a compiler that includes some optimization. Here, the net reduction technique is the optimization we apply to STP nets (assembly language) produced by our Ada to STP-net compiler.

The net reduction technique was especially effective in the divide-and-conquer and network router examples (see Tables 1 and 2). For instance, in the divide-and-conquer example net reduction alone reduced 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 discussed in Subsection 5.1 plays a crucial role in reducing the STP nets for this example [55].[9]

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 case of the network router example, the stubborn set method reduces the state space of unreduced STP nets from exponential to linear in the number of routing elements. Stubborn sets also obtain impressive state-space reductions in the traffic light and the DRT examples (see Tables 4 and 5).

On the negative side, it is difficult to predict the performance of stubborn sets. If we first apply net reduction to the divide-and-conquer example, then the stubborn set method does not provide any further reductions in state-space size (see Table 1). Even in the other examples the reduction factor provided by stubborn sets is greater when this method is applied to unreduced STP nets. Evidently, the net reduction method can remove at least some of the parallelism contained in the nets generated by translation.

As we expected, the net symmetry method is effective only when analyzing STP nets containing symmetries. Thus, this method gives no reductions for either the divide-and-conquer example or the one-track GRC example. The method gives significant reductions in the GRC example with multiple tracks and in the traffic-light example.

Notice that the symmetry method can be very time-consuming, particularly when it does not achieve any reduction. (Compare, for instance, the *nr* and *nr+sym* columns of Table 1.) A general method for finding symmetry mappings in nets must solve the *graph automorphism* problem. No polynomial time algorithm is known for this problem, and indeed one may not exist. (See Köbler et al. [41] for current complexity-theoretic results on graph automorphism.) Our heuristics tend to find a symmetry mapping quickly when

[9] In fact, even when we created the nets by hand, net reduction gave us an exponential-to-linear reduction in state-space size.

one exists, but they often have a very long running time if no symmetry mapping exists. This behavior suggests that we may consider a time-bounded algorithm that discontinues the search if a mapping is not found within a certain amount of time. We have not implemented this algorithm yet because our current priorities are enhancing our state-space reduction techniques and defining additional techniques.

In general, we conclude that using all three optimization methods that we considered in combination makes reachability-based analysis more tractable. Empirically, no one method could single-handedly make real-time analysis tractable for even the relatively small collection of problems discussed in this paper. Intuitively, different methods exploit different aspects of a real-time system to reduce state space size. As a result, the smallest state spaces are often obtained by combining all methods (e.g., the GRC example). The exception, as we said, is that the symmetry method should only be used where it helps. In practice, this means that the analyst will need to try analyzing his real-time program both with and without that technique. However, when all three methods are applied in combination, the order of application of the methods is defined by our paradigm: The net reduction technique is applied first (i.e., before the reachability graph is generated); stubborn sets and net symmetry are applied during the expansion of each graph node.

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 with other methods for real-time analysis

In recent years, numerous approaches for automated analysis and verification of real-time systems have been defined [3, 4, 10, 14, 18, 19, 23, 28, 30, 33, 35, 44, 47, 50, 51, 58, 65, 66]. Despite these efforts, we believe that the field of real-time analysis and verification is still in its infancy. Experimental data on each of these approaches are generally scarce or, in some cases, nonexistent. The lack of available data often makes it difficult to compare our analysis approach with other methods, especially methods that do not use Petri nets for system modeling. When experimental data are indeed available, direct comparisons between approaches are still difficult because of the differences in the hardware platforms used to collect the data, in the systems being verified automatically, and in the representation used by the various approaches. In next two subsections we first discuss other Petri net models, followed by a comparison with approaches that do not use Petri nets.

8.1 Comparison with other Petri net models

Many more researchers have looked at timed Petri nets with more expressive power than STP nets (see, for instance, [10, 30, 47, 65]).

Time Petri nets (TPNs) [10, 47] are somewhat more expressive than STP nets. We will explain in some detail why we think automatic verification of sizable real-time systems is likely to be impractical for TPNs; these remarks apply *a fortiori* to more powerful timed Petri nets such as Time Basic nets [4, 30].

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 0 to 2 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 [10]. 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 [10].)

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. Recent work on real-time analysis with TPNs requires time $\Theta(n^3)$ to compute the next state, where n is the number of enabled transitions [65]. Thus, for a system such as the divide-and-conquer example, where in many states there is a fireable transition for a constant fraction of all the processes, TPNs will require at least $\Omega(n^4)$ time for state space generation. Note that this lower bound is for the *reduced* state space.

At the time of this writing it is unclear whether TPNs can be the foundation for a really practical verification tool. Yoneda et al. have obtained very promising preliminary results with a TPN-based approach for real-time analysis [65]. However, the one system they analyzed has a much smaller number of processes than, say, our network router example.

Time Basic nets (TB-nets) [30] are a strict generalization of TPNs. Now the firing interval of a transition can be bounded by functions instead of constants. TB-nets have excellent modeling capabilities, but of course the problems we outlined 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 verify only bounded safety and bounded liveness properties for these nets.

An additional complication is that TB-nets provide *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 certain optimization methods such as stubborn sets.

The Design/CPN toolset developed at the Univerisity of Aarhus, Denmark uses a different model of time from time Petri nets and STP nets [43]. In particular, Design/CPN associates nonnegative integer timestamps with so-called timed tokens in a colored Petri net. In brief, colored Petri nets differ from ordinary Petri nets in that tokens are associated with data values [40]. The firing of transitions defines the data values of the tokens deposited in the transition's output places, based on the data values found in the transition's input tokens. In order to model time, Design/CPN associates a global clock with a colored Petri net; the clock is initialized to zero and incremented by one unit to model the elapsing of time. The key idea underlying Design/CPN's approach is that a timed token cannot be used to enable a transition if the value of the token's timestamp is greater than the current value of the global clock. In order to model the time taken by a computation or physical phenomenon, a transition can set the value of the timestamps in its output tokens to a value higher than its input tokens. The transition can use an arbitrary function, including a random function, to compute the difference in the values of the timestamps in the output tokens relative to the input tokens.

Design/CPN's concept of time can be viewed as a generalization of Ramchandani's timed Petri nets [52]. It is well-known that those nets are not as expressive as Merlin and Farber's time Petri nets [29]. However, this limitation is avoided in Design/CPN by the use of functions to define the timestamps on tokens produced by a transition. The use of functions to define the value of timestamps output by a transition is similar to the case of TB-nets. In addition, Design/CPN can support probabilistic analyses through simulation and state-space enumeration as well as traditional verification. The resulting toolset is quite powerful; it has been successfully applied to the analysis of numerous real-world systems [43]. However, it is unclear how well verification done by Design/CPN scales up with the size of the systems analyzed.

8.2 Comparison with non-Petri-net models

Many other authors have defined analysis and verification approaches that do not use Petri nets for system modeling (e.g., [3, 14, 18, 28, 33, 50, 58, 64]).

In recent years, Campos et al. [18, 19] have defined an extension of symbolic model checking techniques [15] to the timed domain. These techniques have yielded excellent results in the analysis of untimed concurrent systems (see, e.g., [22]). Their extension uses Binary Decision Diagrams (BDDs) [13] to encode the transition relation in a real-time system. As with our method, their model of time associates a constant delay with each system transition; however, their model is subject to the additional restriction that each transition must take exactly one time unit. Campos et al. also define a technique for handling different durations [19], but it seems that their technique cannot easily capture the execution of multiple transitions in parallel.

Campos et al.'s method has been fruitfully applied to the analysis of systems running on uniprocessor architectures, including an aircraft navigation system [19] and a robot-mounted sensing device [18]. In both cases, various tasks of different priority levels must be scheduled on the processor. Schedulability of the tasks is guaranteed a priori by use of the rate monotonic scheduling algorithm [46]. Two interesting aspects of these examples are the presence of periodic tasks and the possibility of processor preemption based on task priority. Campos et al.'s analyses uncover interesting information about the behavior of these systems, including minimum and maximum delays on the completion of a task. However, at this time it is unclear whether their method can be applied to such classes of real-time systems as distributed systems and reactive systems.

We have not analyzed the systems defined by Campos et al. [18, 19] because our STP nets cannot easily capture processor preemption. This feature can be modeled by Petri nets with inhibitor arcs, which are not implemented by the current version of our PARTS toolset. The implementation of a PARTS extension with inhibitor arcs is currently under way. An additional difficulty is that ATS, our translator, uses a zero-time assumption on computations taking place in the system being analyzed. In order for us to analyze uniprocessor systems with rate-monotonic scheduling, we will either have to write STP nets by hand or use a different translation scheme.

Stuart [58] defines a method for analyzing real-time specifications written in a subset of Modechart [38] and applies the method to a one-track GRC example. Stuart's method seeks to construct a structure called a *computation graph* from a Modechart [38] specification. The graph shows feasible transition sequences induced by the specification. A potential benefit of Stuart's method is that it is applied to high-level specifications rather than to code-level specifications, as is PARTS. Because of its higher abstraction level, a program specification is likely to have a smaller state space than the corresponding code. Moreover, analyzing specifications rather than code leads to an early detection of potential errors, lowering the cost of correcting such errors.

A potential disadvantage of Stuart's method is the apparent complexity of the graph construction method. First, a computation tree is built from a specification. This tree is similar to a computation graph; however, the tree may also contain edges that do not correspond to actual transitions, because these edges violate the transition ordering induced by the underlying specification. To identify these edges Stuart generates a weighted graph, called a separation graph, for each distinct path in the computation tree. The presence of positive weight cycles in a separation graph identifies edges to be removed. The resulting tree is further simplified by collapsing nodes corresponding to the same state; this activity yields the computation graph of the specification.

Another potential disadvantage of Stuart's method is that queries on the behavior of a specification must conform to few predefined formats. For instance, it is possible to ask whether the intervals associated with two events are disjoint or whether one interval is contained in the other. However, the method does not provide tight bounds to questions on, say, the minimum or the maximum delay between arbitrary events. Moreover, the activity of answering queries requires an examination of all simple cycles in a computation graph. Given that computation graphs are typically quite large and that the number of simple cycles in a graph is often exponential in the number of nodes of the graph, the resulting computations are likely to be prohibitively expensive. It is unclear whether Stuart's method can scale up to larger versions of the "toy" examples, let alone analyze real-world examples.

Ostroff's Statetime toolset uses two verification tools, namely Verify and STeP [51]. Verify, which is coded in Prolog, uses reachability analysis. An advantage of Verify with respect to PARTS is that Verify uses transition delay intervals, rather than point delays. In addition, Verify can be run without specifying a time interval during which a given property is supposed to hold. In this case, Verify finds the time bounds of this interval through reachability analysis. By way of comparison, to find one such time bound we would have to perform a binary search by running PARTS repeatedly with different delays on the relevant transition. A disadvantage of Verify is that it does not use the state-space reduction techniques we discuss here. This may explain why, using net reduction and stubborn sets, PARTS can analyze both versions the DRT example for deadlock faster than Verify. But, of course, one example does not provide a sufficient basis for comparing the performance of two analysis toolsets.

Statetime also uses Stanford's STeP theorem prover [11] for verification. This tool can verify untimed properties. The verification of timed properties is a bit more cumbersome. Ostroff reports on the construction of a so-called *Observer* automaton that is composed with the DRT system in order to detect violations of timing requirements [51]. In this case, STeP verifies that the relay is opened properly in a little over 2 minutes of CPU time on a Sun Ultra SPARC station 1 with 160 MBytes of memory.

Numerous existing verification methods use extensions of Alur and Dill's *timed automata* for modeling real-time processes [35, 44, 66]. For instance, Henzinger's HYTECH toolset uses symbolic model-checking for real-time analysis of so-called *hybrid automata*. Hybrid automata use real-valued variables to specify conditions upon which automata transitions can be taken. The conditions are expressed as equalities and inequalities involving the variables and their first derivatives. HYTECH has been applied to the analysis of a thermostat system and of a version of the one-track GRC example [35]. In addition, Corbett uses HYTECH for verification of his models of Ada 95 code [21].

Evidently, HYTECH's model of time is much, much richer than the case of PARTS, essentially because of HYTECH's ability to use differential equations over continuous variables to describe the behavior of clocks. At this time, however, it is unclear how well HYTECH fares in terms of scalability. Corbett expresses similar doubts on the scalability of HYTECH after analyzing several examples of Ada 95 programs with this tool.

Similar considerations apply to the UPPAAL toolset jointly developed by Larsen et al. at the University of Uppsala and the University of Aalborg [44]. As with HYTECH, UPPAAL is a mature tool available for distribution through the world-wide web. UPPAAL represents a real-time system as a collection of timed automata models [1] extended with integer and boolean data variables. In addition, UPPAAL can also handle linear-hybrid automata, which include drifting clock rates, by translating these automata into timed automata. UPPAAL supports various kinds of analysis techniques. For instance, it uses symbolic model-checking techniques to verify reachability properties and bounded liveness properties. Recently, UPPAAL underwent substantial improvements including performance enhancements and new graphical user interface [6]. To date, UPPAAL has been successfully applied to the verification of numerous real-world timed systems, including various communication protocols; however, its scalability is unclear.

The KRONOS toolset also uses an extension of timed automata in order to model real-time processes [12, 66]. That toolset supports various kinds of analyses, including model-checking of timed properties expressed in the logic TCTL [37, 50]. KRONOS also supports bisimulation of timed-abstracted automata by invoking the ALDEBARAN toolset [27].

KRONOS has been applied to the verification of various systems, including the CSMA/CD (carrier sensed, multiple access with collision detection) protocol for an Ethernet bus. We analyzed the same example with an earlier version of PARTS [17]. Our experimental results indicate that, at least in this case, KRONOS is much less sensitive to the time granularity problem than PARTS. Using a symbolic model-checking approach similar to KRONOS's, Nicollin et al. analyzed successfully three versions of the CSMA/CD protocol [50]. PARTS could analyze much faster than reported by Nicollin et al. a version in which two delay constants were reduced to facilitate analysis. However, PARTS failed to complete the analysis when the original time constants were used. This result suggests that our heuristics for state-space reduction should be combined with techniques for mitigating the time granularity problem in order for Petri-net verification to be truly effective. Additional information about verification methods based on timed-automata can be found in Yovine's excellent survey [67].

Finally, Belluomini and Myers defined an interesting model for hardware systems called *Timed Event/Level Structures* (TEL) [5]. TEL has been proved to be at least as expressive as safe Petri nets. In addition, it supports the specification of interval delays, similar to time Petri nets. Consequently, Belluomini and Myers use a system of difference constraints similar to that defined by Berthomieu and Diaz for time Petri nets [10] in order to keep track of dynamic firing delays during state space exploration. An interesting aspect of the TEL model is that it supports partial-order state reductions similar to our real-time stubborn sets [55] and the method of Yoneda et al.'s [65]. At present, it is unclear how well this method scales up with the size of the systems analyzed. Similar to the case of Berthomieu and Diaz, the generation of the next state in a TEL specification is likely to be at least cubic in the number of enabled transitions (rules and events, in TEL's terminology). Thus, it is unclear whether this method can handle systems with a large number of enabled transitions.

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 the state-space explosion caused by the time granularity problem. We are also working to extend Godefroid and Wolper's *sleep set method* [31] to the real-time context. This is another partial-order method that is complementary to the stubborn set method. In addition, we are investigating extensions of our specification language and translation method in order to handle constructs defined in Ada 95, such as protected objects and remote procedure calls. Finally, we are beginning to investigate compositional methods, because we suspect that the analysis of truly large programs may have to be done in pieces.

References

[1] R. Alur and D. Dill. Automata for modelling real-time systems. *Theoretical Computer Science*, 126(2):183–235, Apr. 1994.

[2] G. S. Avrunin, U. A. Buy, J. C. Corbett, L. K. Dillon, and J. C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Trans. Softw. Eng.*, 17(11):1204–1222, Nov. 1991.

[3] G. S. Avrunin, J. C. Corbett, L. K. Dillon, and J. C. Wileden. Automated derivation of time bounds in uniprocessor concurrent systems. *IEEE Trans. Softw. Eng.*, 20(9):708–719, Sept. 1994.

[4] C. Bellettini, M. Felder, and M. Pezzè. Merlot: A tool for analysis of real-time specifications. In *Proceedings of the Seventh International Workshop on Software Specification and Design*, pages 110–119, Redondo Beach, California, Dec. 1993.

[5] W. Belluomini and C. J. Myers. Verification of timed systems using POSETs. In *Proc. 1998 Computer-Aided Verification, CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998. Springer-Verlag.

[6] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, Y. Wang, and C. Weise. New Generation of UPPAAL. In *Int. Workshop on Software Tools for Technology Transfer*, June 1998.

[7] G. Berry. Reactive programming. In J. J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 2, pages 1005–1006. John Wiley & Sons, Inc., New York, New York, 1994.

[8] G. Berry and G. Gonthier. The synchronous programming language Esterel, design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, Nov. 1992.

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

[10] 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.

[11] N. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H. Sipma, and T. E. Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *Computer-Aided Verification: 8th International Conference, CAV '96*, LNCS 1102, pages 415–418. Springer-Verlag, July 1996.

[12] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In *Proc. 1998 Computer-Aided Verification, CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998. Springer-Verlag.

[13] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[14] G. Bucci and E. Vicario. Compositional validation of time-critical systems using communicating time Petri nets. *IEEE Trans. Softw. Eng.*, 21(12):969–992, Dec. 1995.

[15] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.

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

[17] U. Buy and R. H. Sloan. Analysis of real-time programs with simple time Petri nets. In *Proc. 1994 Internat. Sympos. on Software Testing and Analysis*, pages 228–239, Aug. 1994.

[18] S. Campos, E. Clarke, W. Marrero, and M. Minea. Verus: A tool for quantitative analysis of finite-state real-time systems. In *Second ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 75–84, June 1995.

[19] S. Campos, E. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *IEEE Real-Time Systems Symposium*, 1994.

[20] S. S. Cha, N. G. Leveson, and T. J. Shimeall. Safety verification in MURPHY using fault tree analysis. In *Proceedings of the Tenth International Conference on Software Engineering*, pages 377–386, Apr. 1988.

[21] J. C. Corbett. Constructing abstract models of concurrent real-time software. In S. J. Zeil, editor, *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA)*, pages 250–260, San Diego, California, January 1996.

[22] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Trans. Softw. Eng.*, 22(3):161–179, Mar. 1996.

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

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

[25] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[26] S. Duri, U. Buy, R. Devarapalli, and S. M. Shatz. Application and experimental evaluation of state space reduction methods for deadlock analysis in Ada. *ACM Trans. Software Engineering and Methodology*, 3(4):340–380, Oct. 1994.

[27] J. C. Fernandez and L. Mounier. On the fly verification of behavioral equivalences and preorders. In K. G. Larsen and A. Skou, editors, *Computer-Aided Verification: 3rd Internat. Conf., CAV '91*, LNCS 575, pages 181–191. Springer-Verlag, 1992.

[28] R. Gerber and I. Lee. A layered approach to automating the verification of real-time systems. *IEEE Trans. Softw. Eng.*, 18(9):768–783, Sept. 1992.

[29] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzè. A general way to put time in Petri nets. In *Proc. 5th Internat. Workshop on Software Specifications and Design*, pages 60–67, Pittsburgh, Pennsylvania, May 1989.

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

[31] 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.

[32] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language Lustre. *IEEE Trans. Softw. Eng.*, 18(9):785–793, Sept. 1992.

[33] M. G. Härbour, M. H. Klein, and J. P. Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE Trans. Softw. Eng.*, 20(1):13–28, Jan. 1994.

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

[35] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1997. An earlier version appeared in [36].

[36] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: a symbolic model checker for hybrid systems. In O. Grumberg, editor, *Computer Aided Verification: 9th Internat. Conf., CAV 97*, LNCS 1254, pages 460–463. Springer-Verlag, 1997.

[37] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:193–244, 1994.

[38] F. Jahanian and A. K. Mok. Modechart: A specification language for real-time systems. *IEEE Trans. Softw. Eng.*, 20(12):933–947, Dec. 1994.

[39] F. Jahanian and D. A. Stuart. A method for verifying properties of Modechart specifications. In *Proc. Real-Time Systems Sympos.*, pages 12–21, Dec. 1988.

[40] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 3 of *Monographs in Theoretical Computer Science*. Springer-Verlag, New York, New York, 1997.

[41] J. Köbler, U. Schöning, and J. Torán. Graph isomorphism is low for PP. *Computational Complexity*, 2(4):301–330, 1992.

[42] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*, volume 395 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, Massachusetts, 1997.

[43] L. M. Kristensen, S. Christensen, and K. Jensen. The practitioner's guide to coloured Petri nets. *Software Tools for Technology Transfer*, 2(2):98–132, 1998.

[44] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997.

[45] M. Lawford and W. M. Wonham. Equivalence preserving transformations for timed transition models. *IEEE Transactions on Automatic Control*, 40(7):1167–1179, July 1995.

[46] C. L. Liu and J. W. Layland. Scheduling algorithms for multi-programming in a hard real-time environment. *J. ACM*, 20(1):46–61, Jan. 1973.

[47] 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.

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

[49] T. Murata, B. Shenker, and S. M. Shatz. Detection of Ada static deadlocks using Petri net invariants. *IEEE Trans. Softw. Eng.*, 15(3):314–326, 1989.

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

[51] J. S. Ostroff. A visual toolset for the design of real-time discrete-event systems. *IEEE Transactions on Control Systems Technology*, 5(3), 1997.

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

[53] W. Schütz. Fundamental issues in testing distributed real-time systems. *Real-Time Systems*, 7(2):129–157, Sept. 1994.

[54] S. M. Shatz, K. Mai, C. Black, and S. Tu. Design and implementation of a Petri net-based toolkit for Ada tasking analysis. *IEEE Transactions on Parallel Distributed Systems*, 1(4):424–441, Oct. 1990.

[55] R. H. Sloan and U. Buy. Reduction rules for time Petri nets. *Acta Inf.*, pages 687–706, 1996.

[56] R. H. Sloan and U. Buy. Stubborn sets for real-time Petri nets. *Formal Methods in System Design*, 11(1):23–40, July 1997.

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

[58] D. A. Stuart. Implementing a verifier for real-time systems. In *Proceedings 11th Real-Time Systems Symposium*, pages 62–71, Lake Buena Vista, Florida, December 1990. IEEE.

[59] 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.

[60] U. S. Department of Defense, Washington, D. C. *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A edition, Jan. 1983.

[61] 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.

[62] 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.

[63] E. Vlachogiannis. On translating Ada programs into Petri nets: Issues and an object-oriented implementation. Master's thesis, University of Illinois, Chicago, Illinois, 1994.

[64] J. Yang, A. K. Mok, and F. Wang. Symbolic model checking for event-driven real-time systems. In *Proc. Real-Time Systems Symposium*, pages 23–32. IEEE, Dec. 1993.

[65] 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.

[66] S. Yovine. A verification tool for real-time systems. *Software Tools for Technology Transfer*, 1(1–2):123–133, Dec. 1997.

[67] S. Yovine. Model-checking timed automata. In *Embedded Systems*, volume 1494 of *Lecture Notes in Computer Science*. Springer-Verlag, Oct. 1998.

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

A Divide-and-Conquer example

Here is the specification for 3-element version of the divide-and-conquer example that we verified automatically. Larger versions of this example are omitted for reasons of space. They can be obtained from the authors of this article.

```
task ELEMENT1 is
  entry END1;
  end ELEMENT1;

task ELEMENT2 is
  entry IN2;
  entry END2;
  end ELEMENT2;

task ELEMENT3 is
  entry IN3;
  end ELEMENT3;

task body ELEMENT1 is
begin
    if (x = 0)
      then delay 5;
      else ELEMENT2.IN2;
           delay 1;
           accept END1;
      end if;
  end ELEMENT1;

task body ELEMENT2 is
begin
    accept IN2;
    if (x = 0)
      then delay 5;
      else ELEMENT3.IN3;
           delay 1;
           accept END2;
      end if;
    ELEMENT1.END1;
  end ELEMENT2;

task body ELEMENT3 is
begin
    accept IN3;
    delay 5;
    ELEMENT2.END2;
  end ELEMENT3;
```

B Delayed Reactor Trip

Here is the Ada source code for one of the two versions the delayed reactor trip that we verified automatically. This is the version in which we use Ada's *delay* statement in order to avoid reprensenting clocks explicitly. The other version, which is similar, is omitted for reasons of space. It can be obtained from the authors of this article.

```
task CONTROL is
   entry POWER_HIGH;
   entry PRESSURE_HIGH;
   entry POWER_LOW;
   entry PRESSURE_LOW;
  end CONTROL;

task RELAY is
   entry OPEN;
   entry CLOSE;
  end GATE;

task POWER_SENSOR;

task PRESSURE_SENSOR;

task body RELAY is

begin
  loop
    select
      accept CLOSE;
    or
      accept OPEN;
    end select;
  end loop;
end RELAY;

task body POWER_SENSOR is

begin
  loop
    if x = 0
      then delay 1;
      else if y = 0
             then CONTROL.POWER_HIGH;
             else CONTROL.POWER_LOW;
           end if;
    end if;
  end loop;
end;

task body PRESSURE_SENSOR is

begin
  loop
    if x = 0
      then delay 1;
      else if y = 0
             then CONTROL.PRESSURE_HIGH;
             else CONTROL.PRESSURE_LOW;
           end if;
    end if;
  end loop;
end;
```

```
task body CONTROL is

  High_Power:  INTEGER RANGE 0..1 := 0;
  High_Pressure:  INTEGER RANGE 0..1 :=0 ;

begin

  loop

    select

      accept POWER_HIGH do
        High_Power := 1;
        if High_Pressure = 1
        then
          select
            accept POWER_LOW;
            High_Power := 0;
          or
            delay 30;
            RELAY.OPEN;
            delay 20;
          end select;
        end if;
      end POWER_HIGH;
    or
      accept PRESSURE_HIGH do
        High_Pressure := 1;
        if High_Power = 1
        then
          select
            accept POWER_LOW;
            High_Power := 0;
          or
            delay 30;
            RELAY.OPEN;
            delay 20;
          end select;
        end if;
      end PRESSURE_HIGH;
     or
       accept POWER_LOW do
        High_Power := 0;
        RELAY.CLOSE;
       end POWER_LOW ;
     or
       accept PRESSURE_LOW do
        High_Pressure := 0;
       end POWER_LOW ;

    end select;
  end loop;
end CONTROL;
```