In: *Proceedings of the Seventh International Workshop on Software Specification and Design*, pp. 56–60, December 1993, IEEE Computer Society Press.

A Petri-Net-Based Approach to Real-Time Program Analysis

Ugo Buy* Robert Sloan†
Dept. of Elect. Eng. and Computer Science (M/C 154)
University of Illinois at Chicago
851 S. Morgan St. Room 1120 SEO
Chicago, IL 60607-7053

Abstract

We propose to extend existing Petri-net-based tools for concurrency analysis to real-time analysis. The goal is to create a fully automated system, which starts from code in a higher level language for real-time programming, and answers programmers' queries about timing properties of the code.

The key difficulty with all reachability-based approaches is that the state space quickly becomes intractably large. To circumvent this state explosion problem, we propose using a combination of several heuristics for model reduction and state space reduction that have been effective for untimed concurrency analysis.

1 Introduction

The analysis of real-time software is very difficult. Indeed, the activities of design, implementation and testing are costly and complex even for traditional software, considerably more costly and complex for untimed concurrent systems, and much worse still for real-time software.

Real-time software is characterized by the presence of timing constraints on computations, such as deadlines on process execution. Consequently, the correctness of real-time software depends both on its *functional correctness* and on its *time performance*. We propose extending techniques that have proven successful for the analysis of concurrent programs to real-time systems.

There are several reasons why real-time software is so troublesome. First of all, programmers must ensure that timing constraints are met by a software system. Second, because real-time software systems are usually physically distributed, their behavior often depends on such factors as random delays over communication networks and the unpredictable occurrence of interrupts. The resulting nondeterminism complicates program testing and debugging because program executions may not be reproducible easily. Testing and debugging are further complicated by the fact that it is often impossible to measure the timing performance of a real-time system without affecting this performance. The magnitude of the family of all possible interleavings of computations performed by asynchronous processes often makes traditional testing and simulation techniques impractical. Finally, the presence of multiple asynchronous processes introduces the possibility of such erroneous conditions as deadlock and starvation, which must be taken into account by software developers.

Our view is that existing methods for analyzing concurrent systems can be profitably extended to real-time systems. In the past, excellent results have been obtained for deadlock analysis of Ada programs using Petri-net-based models [3]. We believe that these techniques should be extended to the analysis of timing properties of concurrent programs. Our goal is to produce tools and techniques for the automatic analysis of real-time properties of Ada code. Our techniques will use a Petri-net-based model with suitable extensions for representing time information.

The remainder of this paper is organized as follows. In the next section, we briefly review our results for concurrent systems. In Section 3 we describe how Petri nets have been extended to model time. Finally, in Section 4 we outline our approach to real-time analysis.

2 Previous work

Our approach to real-time analysis is built around the notion of a reachability analysis of a Petri-net

*Supported in part by NSF grant CCR-9109231. Net address: buy@eecs.uic.edu.

†Supported in part by NSF grant CCR-9108753. Net address: sloan@eecs.uic.edu.

model of a concurrent program [13]. The general goal of reachability analysis is to enumerate the set of reachable states for the program under consideration. The algorithm underlying this analysis is quite simple: An expansion procedure is applied to every generated state, starting from the initial program state. The procedure first finds all transitions that are possible (i.e., enabled) in the state and then generates a new state for each enabled transition. A disadvantage of reachability-based approaches is that the state space thus generated is usually quite large (i.e., exponential in the number of processes contained in the program analyzed).

In the past, several techniques for controlling the complexity of reachability-based deadlock detection have been investigated [3, 6, 9, 10, 11, 12]. This work showed that *model reduction* approaches could be successfully combined with *reduced state space generation* approaches in order to control the complexity of reachability-based analysis of safety properties. Model reduction techniques are aimed at transforming a model of a concurrent program, such as a Petri net, into a smaller model having the same deadlock features as the original model [11]. Reduced state space generation techniques are aimed at generating only a portion of the reachability tree, subject to the condition that any deadlock states contained in the full state space are in the reduced state space as well [6, 9, 10, 12]. The improvements provided by these two kinds of approaches complement one another, which makes their combination very attractive.

The Petri-net reduction technique of Tu, Shatz and Murata [11] was implemented in the TOTAL toolkit for the analysis of Ada code along with three techniques for reduced state space generation: The stubborn set method [12], the sleep set method [6], and Starke's symmetry method [10]. Extensive experiments with this toolkit have given excellent results on various toy examples (e.g., dining philosophers, readers and writers, and gas station) and on some real world problems (e.g., a border defense system and a system monitoring a host buoy at sea) [3].

3 Adding time to Petri nets

We believe the same methods should work for analyzing real-time programs, but first one must extend Petri nets to model time. Several interesting extensions of Petri nets to the timed domain have been defined [5, 7, 8]. In this section we discuss a particular timed extension of Petri nets that we believe will lead to good practical results for real-time analysis.

We require the following definitions. An *ordinary Petri net* is a directed bipartite graph that contains two kinds of nodes, namely *places* and *transitions*. Directed arcs connect places to transitions and vice versa. A network state associates a nonnegative number of tokens (or markers) with each place. Formally, an ordinary Petri net N is a four-tuple $N = (P, T, A, M_0)$, where P is a finite set of places, T is a finite set of transitions, $A \subset (P \times T) \cup (T \times P)$ is an arc set, and M_0 is an initial marking, that is, an initial assignment of tokens to each place $p \in P$. Given an arc a from a place p (a transition t) to a transition t (a place p), p is said to be an input (output) place for t and t is said to be an output (input) transition for p.

State changes are carried out by firing *enabled* transitions. In an ordinary net a transition is enabled when all its input places have at least one token. When an enabled transition t is fired, a token is removed from each input place of t and a token is added to each output place of t; this gives a new state.

Our approach to modeling time with Petri nets is based on the notion of a *simple time Petri net* or *STP net*. An STP net differs from an ordinary Petri net in that a firing delay is associated with each transition t. As a result, an STP net N is a five-tuple $N = (P, T, A, M_0, D)$, where P, T, A and M_0 are the same as for ordinary Petri nets, and D is a vector of nonnegative real numbers associating a firing delay τ_t with each transition $t \in T$.

Firing 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 τ_t time units later, unless it becomes disabled by the removal of tokens from its input places in the interim. Note that ordinary, untimed Petri nets are the special case of STP nets with all firing delays being zero.

The STP nets proposed here are a special case of Merlin's time Petri nets [2, 7]. Merlin allows the firing delay associated with a transition to vary within an interval, whereas in STP nets this delay is a fixed number. Merlin's model has greater expressive power than STP nets; however, as we discuss below, it is less amenable to automated analysis than our model.

STP nets provide a convenient way of representing several program properties that we are interested in modeling. First of all, we have found that interactions among asynchronous processes can be suitably modeled by ordinary Petri nets. In particular, these nets can capture the relevant aspects of Ada tasking behavior [3, 11]. Second, the inclusion of firing delays

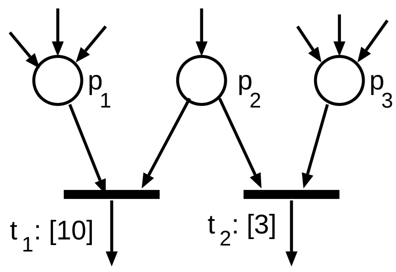

Figure 1: Example of STP net modeling a nondeterministic choice.

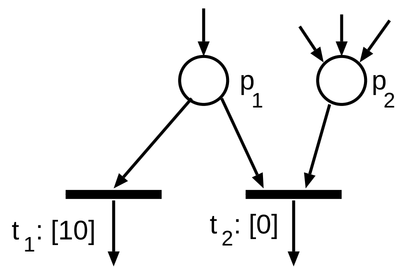

Figure 2: Example of STP net modeling a timeout clock.

for transitions in STP nets allows us to model several timing properties. For instance, if a transition t represents a certain computation, its firing delay τ_t can be used to represent the time required by the computation. Also, this delay can be used to capture the fact that the validity of some data has a limited duration (e.g., a value measured by a sensor). An additional advantage of STP nets is that the generation of the state space of these nets is relatively straightforward. Our algorithm for state space generation is much simpler and faster than the algorithm for Merlin's time Petri nets [2]. In summary, we believe that STP nets are a reasonable compromise between modeling power and ease of analysis.

Some Petri-net extensions have been defined that provide even more expressive power than Merlin's time Petri nets [4, 5]; however, the additional expressive power is obtained at the cost of more complicated notations for expressing firing intervals. For instance, in TB-nets (Time Basic nets) [4] and in E/R nets (Environment/Relation nets) [5] the upper and lower bounds of transition firing intervals are expressed with monotonic functions of time rather than with constants. The use of these functions makes analysis more difficult, however.

An important distinction underlying many modeling techniques for real-time systems, including Petri nets, is between *weak time semantics* and *strong time semantics*. In a weak time model, an enabled transition *may* fire but is not forced to do so when its firing delay expires. In strong time models, such as STP nets, an enabled transition *must* fire once its firing delay expires, unless it has become disabled in the interim.

Weak and strong time semantics can suitably model different aspects of real-time systems. On the one hand, weak semantics is better suited than strong semantics for modeling a nondeterministic choice among alternative flows of control in a concurrent system. On the other hand, strong semantics is better suited than weak semantics for capturing the behavior of timeout clocks, which are required for modeling such things as Ada's select construct with a delay clause.

Consider, for instance, the STP net fragment shown in Figure 1. Suppose that places p_1, p_2 and p_3 represent locations in the flow of control of three asynchronous processes, say, *P_1, P_2,* and *P_3*. Also, suppose that transitions t_1 and t_2 represent the nondeterministic choice of a communication between two of the three processes. A communication between *P_1* and *P_2* requires 10 times units (i.e., the delay of t_1), and a communication between *P_2* and *P_3* requires 3 time units (i.e., the delay of t_2). If, however, p_1 and p_3 already have a token when a token appears in p_2, t_1 cannot be fired with strong time semantics because its delay is greater than the delay of t_2. In this case, weak time semantics allows for both t_1 and t_2 to be fired. Therefore, this kind of semantics can properly model the nondeterministic choice in this example. However, we can also model such nondeterministic choices using strong time semantics, at the price of including additional places and transitions in the net.

Now consider the STP net fragment shown in Figure 2. Here places p_1 and p_2 represent two locations in processes *P_1* and *P_2*, and transition t_2 represents a communication between the two processes. In this case, however, *P_1* does not wish to wait for longer than 10 time units in order to communicate with *P_2*. The timeout is suitably captured by transition t_1 using strong time semantics. If p_2 receives a token before p_1 does or within 10 time units after p_1 does, then t_1 (i.e., the timeout computation) is not fired, and t_2 is fired instead. If, however, p_2 does not receive a token within 10 time units after p_1 receives a token, t_1 fires, thus preventing the firing of t_2. This behavior cannot be easily modeled using weak time semantics.

We have chosen strong time semantics for our STP nets essentially because nondeterministic choice can still be modeled using this kind of semantics, whereas weak time semantics cannot suitably capture the behavior of timeout clocks.

Ghezzi et al. have argued that strong time semantics is inferior because of the *non-locality* of firing conditions [5]. On the one hand, in an ordinary Petri net the conditions allowing a transition t to fire are local to t. All one must check is that all input places of t are suitably marked. The same is true of Petri nets with time but with weak time semantics. On the other hand, in our STP nets one must check not only that a transition has its input places marked, but also that its remaining delay time is minimal among all enabled transitions.

While this non-locality of transition firings would be harmful if we were running a distributed simulation of an STP net, it is unlikely that this phenomenon will adversely affect reachability-based analysis of the net. For each net state x, the search for the successors of x requires finding all transitions that are enabled in x. Consequently, the enablement status of all transitions in the net must be checked even if weak time semantics is used.

4 Analysis techniques

4.1 State space generation for STP nets

Our approach to real-time analysis is based on the generation of the state space of an STP net. This state space is a directed graph whose nodes correspond to net states and whose arcs correspond to state transitions. This approach is similar to the construction of a reachability graph of an ordinary Petri net when concurrency analysis is performed. A significant difference is that each state x of an STP net consists of two components, a net marking and a distinct delay for each transition enabled in x, whereas for an ordinary Petri net a net marking alone suffices.

An additional difference is that only enabled transitions with minimal delays are fired in order to find the successors of a state of an STP net. Suppose, for example, that a state x_1 enables three transitions, t_1, t_2 and t_3, with delays of 2, 5, and 2. In this case, t_1 and t_3 must be fired before t_2 because the delay of t_2 is greater than the delay of the other two transitions. Thus, x_1 has two successors, x_2 and x_3, corresponding to the firing of t_1 and t_3.

Finally, in an STP net the firing delays of enabled transitions must be updated whenever a transition is fired. In the example above, suppose that transition t_2 stays enabled after transitions t_1 and t_3 have fired. Then, the delay of t_2 in both state x_2 and state x_3 is 3.

4.2 Comparison to time Petri nets

Our state space generation algorithm for STP nets is considerably simpler than the corresponding algorithm for Merlin's time Petri nets [2]. Suppose that a state x_1 of a time Petri net enables two transitions, t_1 and t_2, and that their firing intervals are $[1, 3]$ and $[2, 5]$. If t_1 is fired first, generating a new state x_2, the firing interval of t_2 in x_2 must be updated to reflect the time elapsed before the firing of t_1. This phenomenon is a serious problem because in general there are an infinite number of possibilities for the new t_2 interval, depending on the time at which t_1 fires. For instance, if t_1 fires at time θ_1 such that $2 \leq \theta_1 \leq 3$, then in the next state the firing interval for t_2 becomes $[0, (5-\theta_1)]$. Thus, there is a distinct state for each value of θ_1.

Berthomieu and Diaz [2] define an interesting technique for sidestepping the problem of infinity in firing intervals and give an algorithm for state space generation. A disadvantage this algorithm is that for each net state x the firing intervals of transitions enabled in x are represented using a system of linear equalities and inequalities. As a result, the time required to generate a state is much longer than in the case of STP nets.

4.3 Optimization techniques

We intend to apply some of the optimization techniques that have proven successful in deadlock analysis [6, 10, 11, 12] to our framework for state space construction in STP nets. Thus far, we have been investigating the Petri net reduction technique of Tu, Shatz, and Murata [11] and the stubborn set method [12]. Here we summarize our preliminary results with these two methods.

The *net reduction* method iteratively applies a set of reduction rules that transform a Petri-net model of an Ada program into a net with a smaller state space and the same deadlock properties [1, 11]. Some of these rules can be extended to real-time analysis. Consider, for instance, the portion of a Petri net shown in Figure 3(a). If the purpose of the analysis is the detection of terminal states, this subnet can be simplified to the net in Figure 3(b), in which place p_2 is removed and transitions t_1 and t_2 are fused into one transition, t_3. In general, this rule can also be applied in the case of Merlin's time Petri nets or STP nets.

The *stubborn set* method attempts to avoid generating the full state space of a Petri net by exploiting

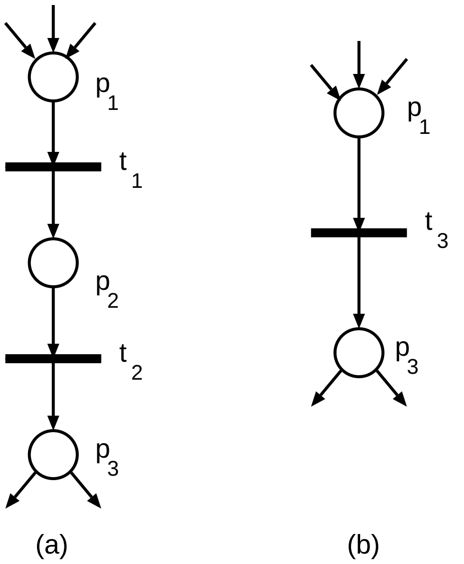

Figure 3: Example of Petri-net reduction rule.

parallelism inherent in the net [12]. In a given net state, this method seeks to find a set of enabled transitions that are "independent," in the sense that one need explore only one possible order of firing these transitions, instead of all possible methods. The stubborn set method was originally defined for so-called variable/transition systems, a broad class of systems which includes ordinary Petri nets, Merlin's time Petri nets, and STP nets. Our preliminary investigations have shown, however, that the method cannot be fruitfully applied to the reduction of the state space for time Petri nets, because it concludes that there are *no* independent transitions. For STP nets, however, we have found that the stubborn set method does help.

References

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

[2] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Trans. Softw. Eng.*, 17(3):259–273, Mar. 1991.

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

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

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

[6] 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.

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

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

[9] 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.

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

[11] 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.

[12] 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.

[13] M. Young and R. N. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Trans. Softw. Eng.*, 14(10):1499–1511, Oct. 1988.