Modeling and Analysis of Real-Time Ada Tasking Programs*

James C. Corbett
Department of Information and Computer Science
University of Hawaii at Manoa
corbett@hawaii.edu

Abstract

We propose a model for real-time Ada tasking programs that naturally represents such features as processor sharing, priority preemption, and process suspension. We describe a semi-decision procedure for proving properties of the model that uses linear programming to determine the feasibility of paths explored during a state-space search of the program. We demonstrate the feasibility of this procedure by applying a prototype analyzer to several examples.

1 Introduction

Control software for real-time embedded systems is often written in Ada. Many of these systems are safety-critical and must be subjected to the most rigorous quality assurance possible. Concurrency, although a useful and natural construct in many such systems, makes timing analysis of the software extremely difficult. Most safe subsets of Ada used for safety-critical systems do not include tasking constructs due to the unpredictability they introduce. Before developers of certifiably safe software can take advantage of these features, rigorous analysis techniques to analyze their timing properties must be developed.

Timing analysis of sequential code is already a hard problem without the addition of concurrency. Here, we will focus on models and techniques for dealing with the concurrent constructs of Ada, realizing that any complete analysis scheme would have to incorporate existing methods for bounding the execution time of Ada's sequential constructs.

In this paper, we make three contributions. First, we propose a formal model for real-time Ada tasking programs that combines and extends existing models of real-time specifications to more easily represent real software. Second, we present an analysis algorithm for this model that allows automatic verification of very general timing properties. Third, we describe the results of several experiments with a prototype analyzer.

Model. One might question the need for a new model of real-time systems given the abundance of such models, including: timed Petri nets [4, 6], timed automata [1, 13, 14, 17], timed process algebras [5, 16], and real-time logics [7, 10]. The difficultly is that real programs usually have resource constraints that are absent in these more abstract models and are difficult to represent within them. This is not a shortcoming of these models—most are intended for representing specifications, not implementations. Gerber's CCSR [5] is one exception, and indeed we borrow his notion of priority and preemption with shared resources, though we use a continuous model of time rather than his discrete one. Models used in schedulability analysis [9] do account for resource sharing, but are restricted to scheduling highly structured units of computation with limited interaction. Our model is essentially a restricted form of hybrid automata [2], with the addition of features for representing priority and resource sharing. It combines the generality of the specification models with features needed to represent a running program. Although our model restricts hybrid automata, it still inherits undecidability of the state reachability problem, which is equivalent to the verification of invariance properties. Our model is presented in Section 2.

Analysis. A formal model is of little use if it does not facilitate analysis. We present a semi-decision procedure for state reachability that can be used to verify very general timing properties expressed as timed automata. The algorithm involves exploring all paths backwards from the state and testing for the feasibility of the timing constraints along each path using linear programming (LP). Section 3 describes the analysis algorithm.

Experiments. Since our analysis algorithm does not always terminate, its utility must be evaluated empirically by application to sample systems. In Section 4, we apply a prototype analyzer to a range of examples from the literature. We also compare the performance of our tool with that of the HyTech verifier for hybrid systems [2].

*This research was supported by National Science Foundation grant CCR-9308067.

2 Model

We want to construct an abstract model of a real-time Ada tasking program that is sufficiently *accurate* and yet *analyzable*. The model must include enough detail to verify properties of interest, but not so much detail that the analysis becomes intractable. To model a real-time Ada tasking program accurately, we feel the model must include:

Processor sharing. We should be able to represent multiple tasks assigned to the same processor and multiple processors assigned to the same task (i.e., the task may run on any processor in the set, but can run on only one at a time).

Priority and preemption. Each task action has an associated priority and processors will elect to execute actions of higher priority over actions of lower priority. Priority provides a powerful mechanism for implementing many different scheduling policies.

Interruptible actions. Each action of a task will model the execution of a block of code. As such, actions take a certain amount of CPU time to complete (bounded from above and below), but can be interrupted (e.g., when preempted by a higher priority action) and resumed any number of times.

Fine-grained time. We should assume that the bounds on the execution time of actions are real numbers (or very large integers—e.g., the number of clock cycles), not small integers. In particular, we would like the size of the model and the complexity of its analysis to be largely independent of the magnitude of the durations.

Our model will abstract away some details of the real program, but only in such a way that all possible behaviors of the real program are represented in the model. For example, a certain block of code might take exactly 5, 10, or 15 time units to execute. In our model, we would represent this code using an action whose duration could be any real value in the interval $[5, 15]$. If we now prove that no behaviors of the model have a certain anomaly, then we have shown that no behaviors of the real program do either. Note that even if the model has the anomaly, the real program may not; this is the price we pay for abstraction.

2.1 Syntax

For a set of real-valued variables X, let $Cond(X)$ be the set of all boolean functions comprised of boolean combinations of linear inequalities in X (e.g., $(x_1 \geq 1) \wedge (x_1 + x_2 < 5)$). A *timed automaton* M is a 10-tuple $(S, \Sigma, \Delta, X, s_0, Delay, Pri, Enable, Reset, Running)$ where

- S is a set of *states*
- Σ is a set of *actions*
- $\Delta \subseteq S \times \Sigma \times S$ is a *transition relation*
- X is a set of real-valued *clocks*
- $s_0 \in S$ is the *start state*
- $Delay : S \to Cond(X)$ assigns a *delay condition* to each state; the automaton may remain in a state s only while $Delay(s)$ is true
- $Pri : \Delta \to \{1, 2, \ldots\}$ assigns a *priority* to each transition
- $Enable : \Delta \to Cond(X)$ assigns an *enabling condition* to each transition; a transition δ may occur only when $Enable(\delta)$ is true
- $Reset : \Delta \to 2^X$ assigns to each transition a set of clocks to *reset*
- $Running : \Delta \to 2^X$ assigns to each transition a set of clocks that are *running*

A timed automaton may represent a single task or a group of tasks executing concurrently (i.e., a program).

Given n timed automata, $M_1, \ldots, M_n$, we may compose the automata in parallel into a new timed automaton $M = M_1 \parallel \ldots \parallel M_n$ where $S = \prod_{i=1,\ldots,n} S_i$, $\Sigma = \bigcup_{i=1,\ldots,n} \Sigma_i$, $X = \bigcup_{i=1,\ldots,n} X_i$, $s_0 = (s_{01}, \ldots, s_{0n})$, $((s_1, \ldots, s_n), a, (s'_1, \ldots, s'_n)) \in \Delta$ iff

$$\forall i.(a \in \Sigma_i \wedge (s_i, a, s'_i) \in \Delta_i) \bigvee (a \notin \Sigma_i \wedge s_i = s'_i)$$

$Delay((s_1, \ldots, s_n)) = \bigwedge_{i=1,\ldots,n} Delay_i(s_i)$, and each transition property $p = Pri, Enable, Reset, Running$ is defined with an aggregation operator $\sqcup$ on the components of the transition: $p((s_1, \ldots, s_n), a, (s'_1, \ldots, s'_n)) =$

$$\bigsqcup_{i=1,\ldots,n} \begin{cases} Ident(\sqcup) & a \notin \Sigma_i \\ p_i((s_i, a, s'_i)) & a \in \Sigma_i \end{cases}$$

where Pri is aggregated with Max, $Enable$ is aggregated with $\wedge$, and $Reset$ and $Running$ are aggregated with $\cup$. $Ident(\sqcup)$ denotes the identity of $\sqcup$ (e.g., $Ident(\cup) = \emptyset$).

2.2 Semantics

Time. There are two fundamentally different kinds of transitions needed to model a task in real-time program. One kind of transition represents the execution of a sequence of machine instructions within the task; the other kind of transition represents a change in the state of the task (e.g., from blocked to ready) caused by the run-time system. We call transitions of the former kind *busy* and transitions of the latter kind *nonbusy*. Execution of code is modeled by busy transitions; timeouts and wakeup alarms are modeled by nonbusy transitions. Only busy transitions require that a processor be allocated to the task.

To model the timing constraints for these two different kinds of transitions, we use two different kinds of time: *real time* and *CPU time*. The delay before a nonbusy transition (e.g., a timeout) is measured in real time, but the delay before a busy transition (representing the time to execute its action) cannot be measured in real time if the task containing this transition may be preempted. Instead, this delay must be measured in CPU time, which advances only when the task is running. We use two different kinds of clocks to measure these two different kinds of time. The set of clocks X is partitioned into a set X_G of *global clocks*, which measure real time, and a set X_L of *local clocks*, which measure CPU time.

The above dichotomy is represented in our model using the $Running$ map. To model the execution of a program, CPU time is allocated to transitions, which represent actions of the program. If a clock x is in $Running(\delta)$ for $\delta \in \Delta$, the meaning is that x will accumulate any CPU time allocated to δ. This implies two things: δ is a busy transition (i.e., a transition that will have CPU time allocated to it), and x is a local clock (i.e., a clock that measures CPU time instead of real time). Thus we define a transition $\delta \in \Delta$ as busy if and only if $Running(\delta) \neq \emptyset$, and we define a clock to be local if and only if it appears in $Running(\delta)$ for some $\delta \in \Delta$. Note that a local clock may be running (i.e., in $Running(\delta)$) for multiple transitions and thus may advance faster than real time if those transitions execute concurrently on a multiprocessor.

Preemption. A timed automaton representing a program must be *mounted* onto a set of processors in order to be run. Processors are grouped into *clusters* that share the same ready queue and each task is statically assigned to a cluster. Formally, a mounting of a timed automaton M is given by a quadruple $(M', C, Size, Host)$ where M' is a timed automaton described below, C is a set of clusters, $Size : C \to \{1, 2, \ldots\}$ gives the number of processors in each cluster, and $Host : \Delta \to C$ assigns each transition of M to a cluster. The sharing of processors by multiple tasks may result in the preemption of certain transitions due to resource constraints. This preemption is reflected in M', which is identical to M except that $\Delta' \subseteq \Delta$. To define Δ', we require the notion of an *activity*. In a state s, each busy transition out of s represents an action that is ready to execute on a processor. We call such transitions *ready* and denote the set of such transitions $Ready(s)$. At any time, there will be a subset of these transitions whose actions are currently executing on the processors. These transitions are referred to as *active* and such a set of ready transitions is called an activity. For any set of transitions $A \subseteq \Delta$ and any cluster $c \in C$, let $A_c = \{\delta \in A | Host(\delta) = c\}$ be the restriction of A to the transitions hosted on c. Formally, an activity in a state s is a set $A \subseteq Ready(s)$, such that:

- $\forall c \in C. |A_c| \leq Size(c)$

  Each cluster is executing no more actions than it has processors.

- $\forall \delta \in Ready(s).(\delta \in A) \bigvee (Host(\delta) = c \wedge |A_c| = Size(c) \wedge \forall \delta' \in A_c.Pri(\delta') \geq Pri(\delta))$

  Every busy transition out of s is either active or preempted by equal or higher priority transitions on its cluster.

Let $Activities(s)$ denote the set of activities in state s. A busy transition out of a state is preempted if and only if there is no possible activity in that state containing it: $\delta = (s, a, s') \in \Delta'$ iff

$$\delta \in \Delta \wedge (Running(\delta) = \emptyset \vee \exists A \in Activities(s).\delta \in A)$$

Note that only busy transitions preempt and are preempted.

Trace. We define a trace of a mounted timed automaton $(M, C, Size, Host)$ as follows. A *valuation* is a function $X \to \mathbf{R}^+$ that assigns a nonnegative real to each clock. Let $(v :=_{X'} 0)$ for $X' \subseteq X$ denote the valuation v' such that

$$\forall x \in X'.v'(x) = 0 \bigwedge \forall x \in X - X'.v'(x) = v(x)$$

and let $(v +_{X^*} t)$ for $X^* : X_L \to \{0, 1, 2, \ldots\}$ a multiset of X_L and $t \in \mathbf{R}^+$ denote the valuation v' such that

$$\forall x \in X_L.v'(x) = v(x) + X^*(x) \cdot t$$
$$\bigwedge \forall x \in X_G.v'(x) = v(x) + t$$

For each activity A, we define $Run(A) : X_L \to \{0, 1, 2, \ldots\}$ as the multiset of local clocks running during A: $Run(A)(x) = |\{\delta \in A | x \in Running(\delta)\}|$. We must use a multiset rather than a set for $Run(A)$ since a local clock may accumulate CPU time for multiple transitions that run concurrently (i.e., in the same activity). Given a condition $e \in Cond(X)$ and a valuation v, we write $e(v)$ to denote the boolean value of the condition with each variable $x \in X$ given the value $v(x)$. Let V be the set of all valuations and let v_0 be the function that assigns 0 to all clocks. A *configuration* is an element of the set $S \times V$. The initial configuration of the machine is (s_0, v_0).

The machine may evolve from one configuration to another in two ways: by taking an instantaneous transition, or by letting time pass while performing some activity. The machine may take a transition $\delta = (s, a, s') \in \Delta$, denoted $(s, v) \xrightarrow{\delta} (s', v')$, iff

- $Enable(\delta)(v)$

  The enabling condition is true in the current valuation.

- $v' = (v :=_{Reset(\delta)} 0)$

  The right clocks are reset.

The machine may let time $t \in \mathbf{R}^+$ pass while performing activity $A \in Activities(s)$, denoted $(s, v) \stackrel{t,A}{\rightarrow} (s, v')$, iff

- $\forall t' \in [0, t].Delay(s)(v +_{Run(A)} t')$

  The delay condition for the state remains true throughout the interval.

- $v' = (v +_{Run(A)} t)$

  The global clocks are advanced by the elapsed real time t, while each local clock accumulates the total CPU time allocated to active transitions for which it is running.

A *trace* is a finite sequence of configurations, each evolving from the previous, starting from the initial configuration. Since the machine may engage in several activities before leaving a state, we write $(s, v) \stackrel{t}{\rightarrow} (s, v')$ for $t \in \mathbf{R}^+$ iff

$$\exists A_1, \ldots, A_n \in Activities(s), t_1, \ldots, t_n \in \mathbf{R}^+.$$
$$t = \sum_{i=1}^{n} t_i \wedge (s, v) \stackrel{t_1,A_1}{\rightarrow} \ldots \stackrel{t_n,A_n}{\rightarrow} (s, v')$$

2.3 Representing Ada Tasking Programs

Each task in an Ada program is represented by a timed automaton. States in this automaton correspond to points of control within the task before and after synchronizations (e.g., entry calls/accepts, delay statements). We partition task actions into three classes: rendezvous, internal computations, and timeouts. The representation of each kind of action is described below. We assume that techniques like [12, 15], closely tied to a compiler and architectural information, would be used to determine bounds on the CPU time for all code regions between synchronizations. We also assume that other kinds of overhead (e.g., context switch time, timer interrupts) can be accounted for in these bounds. In this paper, we ignore such complex issues as dynamic tasks, exceptions, abort statements, dynamic priorities, and the effect of variables on control flow, although all of these constructs can be represented in our model to some extent.

As an example, we present a variant of the distributed robot system of [5] in which sensor data are integrated to periodically send a command to a robot. There are two sensor tasks and one integrator task. The code for one sensor task is shown in Figure 1; the other sensor task is similar. Each sensor task constructs a reading and sends it to the integrator. Since the environment is constantly changing, the sensor reading expires if not accepted within a certain time and a new reading is immediately constructed. When the data has been sent successfully, the sensor task sleeps for a period of time. The integrator task is shown in Figure 2. It accepts readings from the two sensors in either order and then computes and sends a command to the robot. To make matters more difficult, there is a proximity requirement: the readings used to construct a command must have been received within a bounded interval. If integrator receives a reading from one sensor and then does not receive the other sensor's reading within a certain time, the first sensor's data expires. Possible delays and computation duration bounds (in milliseconds) are given in Figure 3. Using this example, we will discuss how computations, rendezvous, and timeouts are represented with timed automata.

The execution of a block of code with no synchronizations is represented using an action `a`, a state s, a local clock x which measures the amount of CPU time allocated to a, and a transition $\delta = (s, \mathtt{a}, s')$ which is taken instantaneously when `a` completes. Clock x is reset by all transitions into s and $x \in Running(\delta)$. If the duration of `a` is in the real interval $[L, U]$, then $Enable(\delta) = (x \geq L)$ and $Delay(s) = (x \leq U)$. The simplest case is where the state has a single busy transition. An example is the start state of the automaton in Figure 1, which has only a transition on `read1`, an action with duration $[0.5, 1.1]$. Each state s is labeled with $Delay(s)$ (multiple inequalities are implicitly conjoined) and each transition label has the form: $Enable(\delta) \rightarrow \mathtt{action}, Running(\delta), Reset(\delta) := 0$. We use $x_1, x_2, \ldots$ for local clocks and $y_1, y_2, \ldots$ for global clocks throughout the paper. When there are multiple computations the task might select, one computation must get its required amount of CPU time, and all other computations must get no CPU time. To achieve this, we use a different local clock $x_1, \ldots, x_n$ for each busy transition and conjoin the delay conditions we would have used for the transitions individually. Then we add an *exclusion constraint* to the delay condition of the state that requires at most one of these local clocks be nonzero: $\bigvee_i (\sum_{j \neq i} x_j \leq 0)$.

A rendezvous `r` is represented using a sequence of two transitions on actions `r*` and `r` in both the calling and accepting task automata. The action `r*` represents the beginning of the synchronization; the action `r` represents its end. The transitions representing the entry call are nonbusy and have no timing constraints; the transitions representing the accept are busy and account for the CPU time to execute the accept body. An entry call is implemented by putting the caller (and any parameters) on an entry queue. The overhead of doing this should be placed in a previous transition of the calling task automaton (an additional tran-

```
task body sensor1 is
  data : reading;
begin
  loop
    READ: loop
      ReadSensor1(data);
      select
        integrator.Send1(data);
        exit READ;
      or
        delay Sensor1Expire;
      end select;
    end loop READ;
    delay PeriodLength;
  end loop;
end sensor;
```

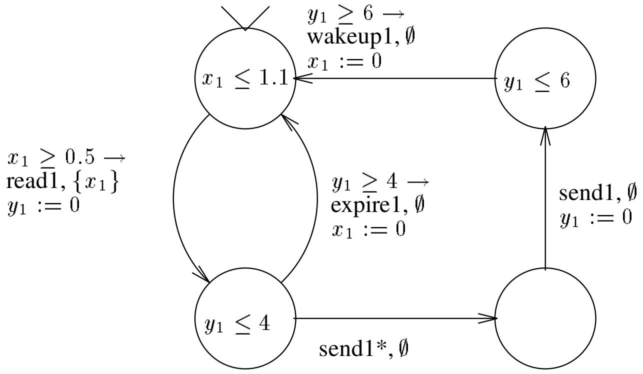

Figure 1: Sensor task

```
task body integrator is
  data1, data2 : reading;
  com : command;
begin
  loop
    GATHER: loop
      select
        accept Send1 (data : reading) do
          data1 := data;
        end Send1;
        select
          accept Send2 (data : reading) do
            data2 := data;
          end Send2;
          exit GATHER;
        or
          delay ProximityExpire;
        end select;
      or
        accept Send2 (data : reading) do
          data2 := data;
        end Send2;
        select
          accept Send1 (data : reading) do
            data1 := data;
          end Send1;
          exit GATHER;
        or
          delay ProximityExpire;
        end select;
      end select;
    end loop GATHER;
    Compute(data1,data2,com);
    Signal(com);
  end loop;
end integrator;
```

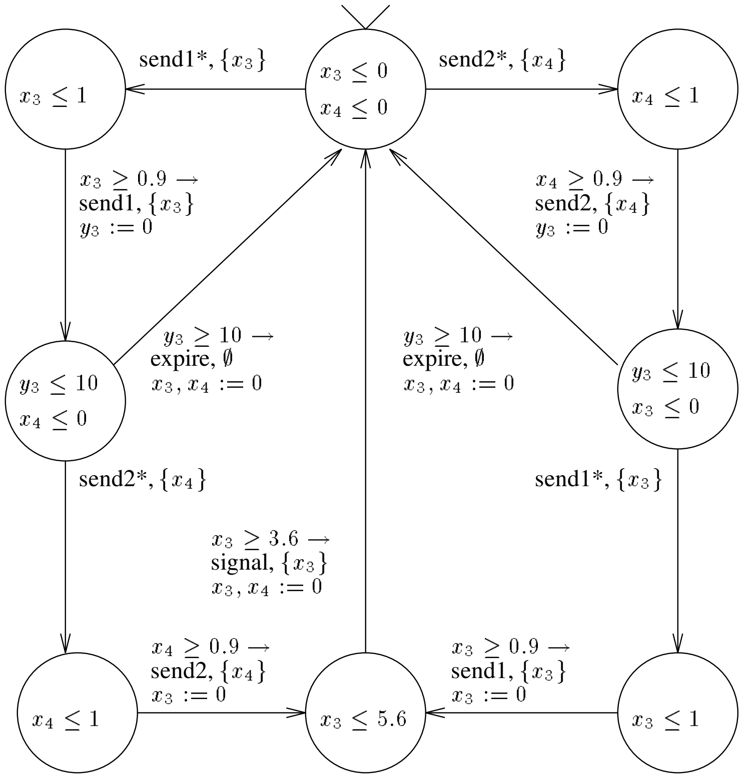

Figure 2: Integrator task

Delay	Real time	Action	CPU time
Sensor1Expire	4	ReadSensor1	$[0.5, 1.1]$
Sensor2Expire	8	ReadSensor2	$[1.5, 2]$
ProximityExpire	10	accept Send$i$	$[0.9, 1]$
PeriodLength	6	Compute+Signal	$[3.6, 5.6]$

Figure 3: Delays and durations for robot example

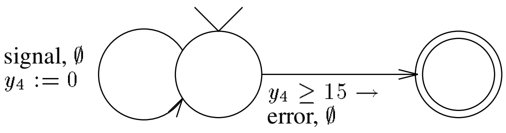

Figure 4: Response property automaton

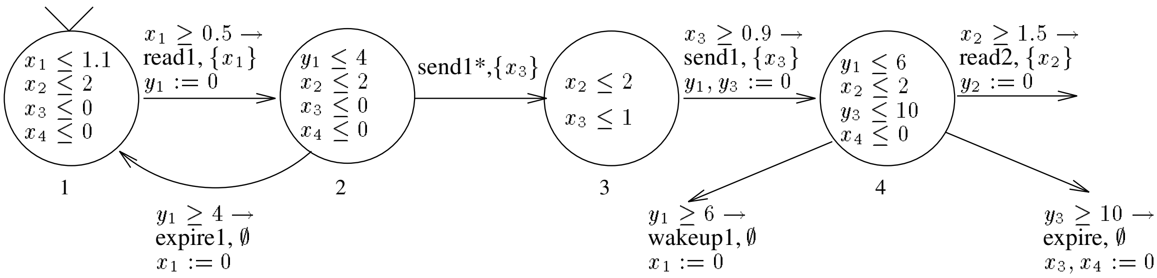

Figure 5: Part of product automaton for robot example

sition is not necessary if the call immediately follows some internal computation of the task, as is often the case). When ready, the acceptor executes the entry body and then wakes up the caller. Both tasks are then simultaneously ready to execute their next actions. The CPU time for the accept body is accounted for entirely in the `r` action; the `r*` transition simply allows any timeouts (discussed below) on the synchronization to be canceled. The `send1` rendezvous between the sensor and the integrator tasks is illustrated by the `send1*` and `send1` actions in Figures 1,2. Any nested rendezvous made within the accept body of `r` can be placed between the `r*` and `r` transitions in the accepting task automaton. We note that, in the absence of nested rendezvous or timeouts on synchronization, the `r*` transitions may be omitted to reduce the size of the automata. The use of Max to aggregate priorities in the product reflects priority inheritance in Ada.

Ada allows only one kind of timeout, although our model allows more general timeouts to be expressed. In Ada, a task waiting for a communication $\delta = (s, \texttt{a*}, s')$ may impose an upper bound T on its wait time. We achieve this by using a global clock y which measures the time that the task has been waiting for the communication, and a nonbusy transition $\delta_t = (s, a_t, s_t)$. Clock y is reset by all transitions into s. We set $Enable(\delta_t) = (y \ge T)$ and $Delay(s) = (y \le T)$. The timeout is cancelled by the `a*` transition if the rendezvous begins. For example, the timeout for the timed entry call `integrator.send1` is represented by the `expire1` transition in Figure 1.

Ada's `delay` statement can also be used outside a `select` statement to put a task to sleep for the indicated number of time units. The representation of this construct is similar to a timeout, and is illustrated in Figure 1 with the `wakeup1` transition. In this paper, we assume the argument for a `delay` statement is a constant, although the use of an expression involving the current clock time is useful for creating periodic tasks (see Section 9.7.1 of the Ada Reference Manual). A more complex analysis of the source code can be used to represent variable delays, though this is beyond the scope of this paper.

Once each task has been represented as a timed automaton, we set the priority of each transition in the automaton to the static priority of the task. The product of these timed automata, mounted onto a set of processors, represents the behavior of the program run on some machine. Each transition in the product automaton is assigned to the cluster to which its associated task was assigned, with transitions representing rendezvous being assigned to the cluster of the accepting task. For this example, we assume that all three tasks share a cluster of size one and that their priority ordering, from highest to lowest, is: sensor 1, sensor 2, integrator.

In Figure 5, we show part of the product automaton for the robot example; each state has a number below it for identification. The automaton for the sensor 2 task is not shown, but is structurally identical to the sensor 1 automaton, with analogous action and clock names (e.g., `read2`, x_2) and slightly different durations and delays (as shown in Figure 3). In state 1, the sensor tasks are both ready to perform their respective `read` actions. Since there is only one processor and sensor 1 has higher priority, the `read2` transition is preempted. The integrator task is ready to begin a rendezvous on either `send1*` or `send2*`, but since neither sensor task is ready for this rendezvous, there are no such transitions in this product state. Since $\{(1, \texttt{read1}, 2)\}$ is the only possible activity, the program may remain in this state at most 1.1 ms before taking the `read1` transition to state 2, though this transition may occur any time after 0.5 ms have passed.

State 2 inherits the delay condition $y_1 \le 4$ from the component state of the sensor 1 task, imposing an upper bound of 4 ms on the real time that can be spent in the state. In this state, the sensor 2 task is still ready to perform the `read2` action, while the sensor 1 and integrator tasks are ready to perform the `send1*` action jointly. Although the computation for the `send1*` action is part of the integrator task over which the sensor 2 task has priority, the `send1*` action inherits the priority of the sensor 1 task and thus the `read2` action is preempted again. The only possible activity in this state is $\{(2, \texttt{send1*}, 3)\}$, but since this activity advances x_3 and the delay condition requires $x_3 \le 0$, time cannot advance in the state and the `send1*` transition must be taken immediately (since the `expire1` transition is not yet enabled). The automaton must then pass between 0.9 and 1.0 ms in the state 3, in which the only activity is $\{(3, \texttt{send1}, 4)\}$. Note that the timeout (delay condition $y_1 \leq 4$) was canceled by the `send1*` transition in the sensor 1 task.

In state 4, the high priority sensor 1 task has only the nonbusy transition on `wakeup1`, which does not preempt the `read2` transition. The sensor 1 task does impose an upper bound of 6 ms on the delay in this state ($y_1 \leq 6$). If the `read2` action could take 8 ms instead of 2, then the machine would be forced to take the `wakeup1` transition after 6 ms into a state where the `read2` action would once again be preempted by the `read1` action. Eventually the machine would enter a state where the sensor 1 task is asleep the `read2` action would again have the highest priority. Since the local clock x_2 is not advanced nor reset by the transitions of the other task automata, the `read2` computation would then continue from where it left off. With appropriate durations and delays, such preemption might be repeated many times before the `read2` action completes.

The resulting model captures all possible behaviors of the Ada program with a preemptive priority scheduling scheme and any discipline to schedule equal priority actions. FIFO within priorities is the only language-defined task dispatching policy in Ada 9X and may be modeled on clusters of size one by using exclusion constraints to serialize all equal priority actions.

3 Analysis

Although most properties of our model are undecidable, we present a semi-decision procedure for determining whether a given state, called the *goal state*, is reachable (i.e., appears in some trace). The procedure does not apply to all timed automata, but only to those that are *delay monotonic*: for each state $s \in S$, the delay condition $Delay(s)$ is always true on entry to s, and every delay condition is a monotonically decreasing function of X:

$$Delay(s)(v) \wedge \forall x \in X . v'(x) \leq v(x) \Rightarrow Delay(s)(v')$$

All automata we used to model Ada programs are delay monotonic by construction.

The procedure performs a breadth first search of all paths backward from the goal state, testing the feasibility of the timing constraints along each path. A path $\delta_1 \ldots \delta_n$ with $\delta_i = (s_i, a_i, s_{i+1})$ is *timed feasible* iff:

$$\exists v_1, \ldots, v_{n+1}, v'_1, \ldots, v'_n \in V, t_1, \ldots, t_n \in \mathbf{R}^+ .$$
$$(s_1, v_1) \xrightarrow{t_1} (s_1, v'_1) \xrightarrow{\delta_1} (s_2, v_2) \xrightarrow{t_2} \ldots \xrightarrow{\delta_n} (s_{n+1}, v_{n+1})$$

Before applying the procedure, we add an additional *initial state* to the automaton and a transition from the initial state to the start state s_0 which resets all clocks. A state is reachable if and only if it lies on some timed feasible path from the initial state. If all paths backward from the goal state eventually become infeasible, then clearly there is no timed feasible path from the initial state to the goal state and the procedure returns false. If a timed feasible path from the initial state to the goal state is found, then the goal state is reachable and procedure displays this path and returns true.

The problem of determining whether a path is timed feasible can be reduced to a linear programming problem as follows. We represent the possible valuations $v_1, \ldots, v_{n+1}, v'_1, \ldots, v'_n$ symbolically using *readings* $r_1, \ldots, r_{n+1}, r'_1, \ldots, r'_n$. Each reading is a map that assigns to each clock a sum of nonnegative *delay variables*, of which there are three kinds. For each $i = 1, \ldots, n$, there is a variable d_i that represents t_i, the real time spent in state s_i just before δ_i. For each $i = 1, \ldots, n$ and each $\delta \in Ready(s_i)$, there is a variable $d_{i,\delta}$ that represents the amount of CPU time allocated to δ in state s_i just before δ_i (i.e., the amount of real time δ was active in s_i). For each clock x there is a variable d_x that represents the value of the clock at the start of the path: $r_1(x) = d_x$ for all $x \in X$. When time passes before δ_i, delay variables are added to clocks in r_i to produce r'_i:

$$r'_i(x) = \begin{cases} r_i(x) + d_i & x \in X_G \\ r_i(x) + \sum_{\delta \in A} d_{i,\delta} & x \in X_L \end{cases}$$

where $A = \{\delta \in Ready(s_i) | x \in Running(\delta)\}$. When δ_i is taken, certain clocks are reset to produce r_{i+1} from r'_i:

$$r_{i+1}(x) = \begin{cases} 0 & \text{if } x \in Reset(\delta_i) \\ r'_i(x) & \text{otherwise} \end{cases}$$

In each state s_i for $i = 1, \ldots, n$, we generate a set of constraints, denoted $Pass(s_i)$, on the delay variables for s_i to insure that the amount each clock is advanced is consistent with some sequence of activities in s_i:

$$\exists A_1, \ldots, A_m \in Activities(s_i), t_1, \ldots, t_m \in \mathbf{R}^+ .$$
$$t = \sum_{i=1}^{m} t_i \wedge v'_i = (\ldots(v_i +_{Run(A_1)} t_1) \ldots +_{Run(A_m)} t_m)$$

To enforce this, for each cluster $c \in C$ we generate the equation

$$\sum_{\delta \in Ready(s_i)_c} d_{i,\delta} = d_i \cdot Min(Size(c), |Ready(s_i)_c|)$$

constraining the total CPU time expended on busy transitions in a cluster to be the product of the real time spent in the state times the size of the cluster or the number of ready transitions, whichever is smaller. $Ready(s_i)_c$ is the set of ready transitions in s_i hosted on cluster c. We further constrain each $d_{i,\delta}$ to be bounded from above by d_i since the CPU time expended on a particular action in a state cannot exceed the real time spent in the state (these constraints may be omitted for clusters of size one). Let D be the set of nonnegative delay variables and let $R_i : Cond(X) \rightarrow Cond(D)$ be the function that substitutes $r'_i(x)$ for x. Then the path is timed feasible if and only if the linear system

$$\bigwedge_{i=1,\ldots,n} Pass(s_i) \wedge R_i(Delay(s_i) \wedge Enable(\delta_i))$$

has some solution.

This procedure always terminates if the state is reachable, but is not guaranteed to terminate if the state is not reachable. Since state reachability is undecidable in our model (even for delay monotonic automata), we cannot construct a decision procedure for it, though this procedure may fail on very simple timed automata containing a timed feasible cycle of unreachable states. Our belief that the procedure will often terminate, supported by the experiments described in Section 4, is based on the observation that many systems are *self synchronizing*—even if started in an unreachable configuration, the system will evolve to some reachable configuration within a bounded number of transitions. On all such systems, the procedure will terminate.

General timing properties can be expressed as timed automata that may reach their accept state only if the corresponding properties are violated. We may verify such properties by including their automata in the product and then testing for the reachability of all product states in which the property automaton is accepting [8]. Figure 4 shows an automaton expressing the real-time response property "a signal must be generated at least every 15 ms". Integration properties (e.g., "a task spends less than 10 percent of its time in its critical section") can be expressed using property automata with local clocks.

Even when the procedure terminates, its complexity is high. We must solve an LP problem for each path explored. The LP problems we generate for Ada programs are purely conjunctive (the exclusion constraints we mentioned in Section 2.3 can be enforced in the analysis without the disjunction by moving them into the enabling conditions). Purely conjunctive LP is in P and our experience with the technique indicates that the LP problems we generate are easily solved (no LP problem has taken more than 0.2 seconds of CPU time on a SPARCstation 10). The real expense of the algorithm lies in the number of paths that must be explored. Given m tasks each with k states, the number of paths of length n through the product automaton is $O(k^{2mn})$. Several factors mitigate this expense, however. First, many paths are not timed feasible, and if a path is not timed feasible, then no further extensions of that path are timed feasible. Second, by starting from a state violating the property, we focus the search and may not have to explore much of the state space in order to prove that the state is unreachable. Still, the algorithm will be practical only when the number of timed feasible paths backwards from the goal state is manageable.

4 Experiments

We have implemented part of a prototype analyzer in Lisp to test the practicality of our analysis algorithm. The analyzer accepts as input a collection of timed automata representing tasks, a specification for how these tasks are to be mounted, and a timed automaton representing a property to be verified. The analyzer mounts the tasks as it constructs the product automaton, pruning preempted transitions as they are generated. It then explores all paths backward from states in which the property automaton is accepting. The search actually proceeds in a depth-first manner since this improves efficiency by allowing the LP problems to be constructed incrementally. The search is stopped at a user specified depth bound, which may be iteratively extended from the frontier states (which are saved) until the search completes or until the user's patience is exhausted. The LP problems are solved with the MINOS optimization package which uses a simplex-based method.

We applied our tool to versions of several real-time specifications from the literature, including: the Fischer timed mutual exclusion protocol in [2], the nuclear control rods [3], the train crossing [2, 3], a timed producer-consumer system [11], and the distributed robot system [5]. The first four examples all involved proving certain states unreachable (e.g., states where the train is in the crossing and the gate is not down). Our analysis algorithm terminated successfully for all of these examples except the Fischer protocol, the specification of which in [2] contains a timed feasible cycle of unreachable states. When we added an upper bound on the time a process can remain in its critical section, however, this cycle was no longer timed feasible and our analysis algorithm terminated successfully. While this extra condition should not be necessary to prove the protocol enforces mutual exclusion, a real program would likely have such a bound. Table 1 gives the performance of the analyzer on these problems, showing: the problem name, the number of tasks or components, the number of states/transitions in the product automaton, the number of paths explored, the CPU time in seconds on a SPARCstation 10 Model 51, and the result of the analysis.

Problem	Tasks	Product Machine States	Product Machine Transitions	Paths Explored	CPU time	Result	HyTech
Fischer protocol (original)	2	29	53			failed	8
Fischer protocol (CS bounded)	2	29	53	17	1	verified	9
Nuclear control rods	3	28	61	52	1	verified	12
Train crossing	3	23	48	73	1	verified	10
Timed Producer/Consumer	3	19	31	83	1	verified	18
Robot (1 cluster, original)	3	97	166	6140	141	disproved	1152
Robot (1 cluster, integrator priority)	3	95	163	2699	35	verified	150
Robot (2 clusters)	3	97	179	6970	99	verified	824
Robot (1 cluster of size 2)	3	97	183	10697	158	verified	1416

Table 1: Performance results for prototype analyzer

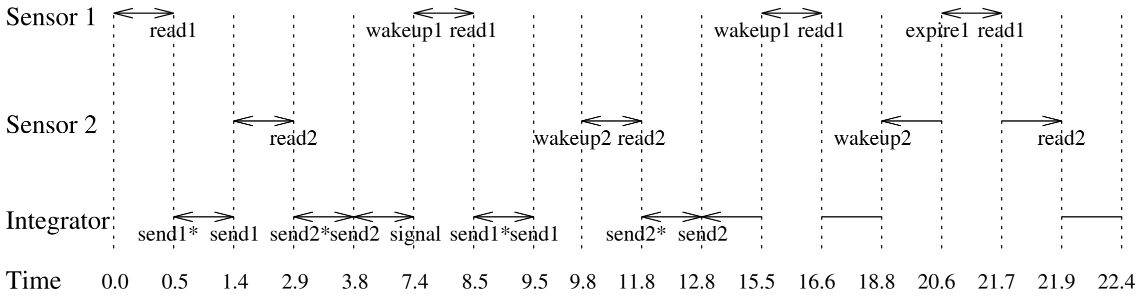

Figure 6: Path violating response property

While these examples are interesting, they do not contain the features of real software (e.g., processor sharing, priority preemption) that our model was created to address. The last example we present is a variant of the distributed robot system of [5] with two sensor tasks and one integrator task, each of which has been slightly modified so that the whole example illustrates as many real-time features of Ada as possible. The system in [5] has many more components, but each action takes exactly one time unit. Our version was described in Section 2.3. The requirement we must prove is that the robot will receive a command at least every 15 ms, the property automaton for which is shown in Figure 4.

For this example, unlike the previous ones, the idea of proving the property by hand is daunting, even though the program is relatively small. In fact, the property does not hold. With very short durations on the first iteration and very long durations on the second, the sensor tasks can steal enough CPU time from the integrator task to cause it to miss its deadline. The analyzer finds the timed feasible path in Figure 6 illustrating this. The analyzer was able to show that the property does hold if any of the following changes are made: the integrator task is given the highest priority, the integrator task is assigned to its own cluster, or another processor is added to the existing cluster.

For comparison, we also include in Table 1 the performance of the HyTech verifier for hybrid systems [2] on the same problems. We wrote a simple translator to generate the input for HyTech automatically from the mounted product automaton produced by our tool, which is essentially a restricted kind of hybrid automaton. By encoding the current activity into the state, the rates of change of all clocks in each state can be determined. With some assistance from HyTech's author Pei-Hsin Ho, we were eventually able to analyze all of our examples with HyTech. Although our analyzer is nearly an order of magnitude faster on these problems, HyTech is more general than our tool, analyzing a much richer model and using a more powerful analysis algorithm (as evidenced by its success on the Fischer protocol, on which our algorithm fails). Also, HyTech is written in Mathematica, which is probably slower than Lisp. HyTech is currently being rewritten in C, which should improve its performance and may render it a better analysis engine for our model than our current analyzer.

Finally, to show that our technique is not sensitive to the absolute magnitudes of the durations, we ran the train crossing example increasing all durations by a factor of 10. The number of paths explored and the analysis time were unchanged. Changing the relative magnitudes of the durations, however, can impact the performance of the algorithm by changing the number of paths that are timed feasible.

5 Conclusion

We have proposed a model for real-time Ada tasking programs that extends existing models of real-time specifications to more easily represent the resource constraints faced by such programs. We have given a procedure for proving properties of the model, and have demonstrated its feasibility by applying a prototype analyzer to several examples, including a representation of a small but complex Ada program. Our algorithm might be compared to state enumeration approaches to concurrency analysis in that we exhaustively examine all possible allocations of CPU time to task actions. Though the analysis can quickly become intractible as the problem size grows, the technique can still be useful on moderately sized programs that are impractical to analyze manually. The next step is to investigate techniques for improving the efficiency of the analysis while retaining its applicability to real software.

Acknowledgements

Thanks are due to George Avrunin for comments on earlier drafts of this paper and to Pei-Hsin Ho for assistance in using the HyTech verifier.

References

[1] R. Alur and D. Dill. Automata for modeling real-time systems. In *Proc. 17th Int. Colloq. Aut. Lang. Prog.*, 1990.

[2] R. Alur, T. Henzinger, and P. Ho. Automatic symbolic verification of embedded systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1993.

[3] J. M. Atlee and J. Gannon. Analyzing timing requirements. In T. Ostrand and E. Weyuker, editors, *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, pages 117–127, New York, June 1993. ACM Press.

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

[5] R. Gerber and I. Lee. A layered approach to automating the verification of real-time systems. *IEEE Trans. Softw. Eng.*, 18(9):768–784, Sept. 1992.

[6] C. Ghezzi, D. Mandriolli, S. Morasca, and M. Pezzé. A unified high-level Petri net model for time-critical systems. *IEEE Trans. Softw. Eng.*, 17(2), Feb. 1991.

[7] C. Ghezzi, D. Mandriolli, and A. Morzenti. Trio: A logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 12(2):107–123, May 1990.

[8] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In K. G. Larsen and A. Skou, editors, *Computer Aided Verification, 3rd International Workshop Proceedings*, volume 575 of *Lecture Notes in Computer Science*, pages 332–242, Aalborg, Denmark, July 1991. Springer-Verlag.

[9] 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, 1994.

[10] F. Jahanian and A. Mok. Safety analysis of timing properties in real-time systems. *IEEE Trans. Softw. Eng.*, 12(5):890–904, 1986.

[11] I. Lee, R. Gerber, and A. Zwarico. Specifying scheduling paradigms for time dependent processes. In *Proceedings of the IEEE Workshop on Real-time Operating Systems and Software*, 1988.

[12] A. Mok. Evaluating tight execution time bounds of programs by annotations. In *Proceedings of the IEEE Workshop on Real-time Operating Systems and Software*, pages 74–80, May 1989.

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

[14] J. S. Ostroff. Deciding properties of timed transition models. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):170–183, 1990.

[15] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, pages 48–57, May 1991.

[16] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, June 1988.

[17] A. C. Shaw. Communicating real-time state machines. *IEEE Trans. Softw. Eng.*, 18(9):805–816, 1992.