Analyzing Partially-Implemented Real-Time Systems

George S. Avrunin, *Member*, *IEEE Computer Society*, James C. Corbett, and Laura K. Dillon, *Member*, *IEEE Computer Society*

Abstract—Most analysis methods for real-time systems assume that all the components of the system are at roughly the same stage of development and can be expressed in a single notation, such as a specification or programming language. There are, however, many situations in which developers would benefit from tools that could analyze partially-implemented systems, those for which some components are given only as high-level specifications while others are fully implemented in a programming language. In this paper, we propose a method for analyzing such partially-implemented real-time systems. Here we consider real-time concurrent systems for which some components are implemented in Ada and some are partially specified using regular expressions and Graphical Interval Logic (GIL), a real-time temporal logic. We show how to construct models of the partially-implemented systems that account for such properties as run-time overhead and scheduling of processes, yet support tractable analysis of nontrivial programs. The approach can be fully automated, and we illustrate it by analyzing a small example.

Index Terms—Real-time, concurrency, static analysis, Ada, temporal logic, hybrid systems, Graphical Interval Logic.

1 Introduction

The correctness of a real-time computer system depends not only on the logical results of its computations but also on whether those computations satisfy certain timing requirements. Developers of such systems, which are increasingly embedded in safety-critical applications such as air traffic control or patient monitoring, need to check that their systems do the right computations at the right times. Testing, executing the system with inputs chosen to reflect the operational profile or to exercise various components and execution paths, is an essential part of this process, but testing alone can consider only a relatively small subset of the possible behaviors of the system and is not adequate even for sequential systems. Many embedded systems are naturally concurrent, and the nondeterministic behavior typically introduced by concurrency means that the same set of inputs may produce different behavior at different times. For concurrent systems, developers must supplement testing with static analysis methods that consider all possible behaviors of the system, rather than executing a small subset of those behaviors.

Most of the static analysis methods that have been proposed for use with real-time systems assume that all the components of the system are at roughly the same stage of development and can be naturally expressed in a single notation, such as a specification or programming language or a mathematical formalism such as Petri nets. It is a software engineering commonplace that analysis should begin at the early design stages of software development—there is some evidence that the majority of errors are introduced at this stage, and it is certainly true that errors caught at the design stage can be corrected much more easily and cheaply than if they are discovered in the later stages of development. Analysis tools that work with specification and design languages would, therefore, seem to be most appropriate. The performance of a real-time system, however, may depend critically on implementation details that cannot be captured in designs (e.g., the scheduling of processes on the available processors), suggesting that analysis of fully-implemented systems is more appropriate, at least for real-time properties. The complexity of analyzing fully-implemented systems is, however, daunting even for relatively small systems, and a full timing analysis of a large and complex system is almost certainly infeasible.

Thus, neither analysis of designs nor analysis of fully-implemented systems is adequate for real-time systems. In practice, developers of real-time systems typically restrict the architectures of their systems so that system components are highly structured and interact in very limited ways (e.g., periodic tasks with precedence constraints). These restrictions allow the use of special scheduling techniques and algorithms, such as rate monotonic scheduling [1], to guarantee that a system's timing requirements are satisfied [2], [3].

In fact, there are many situations in which developers would benefit from tools that could analyze partially-implemented systems, those for which some components are given only as high-level specifications while others are fully implemented in a programming language. These include:

- **Initial Development of Complex Systems**. The development of the various components of large systems

- *G.S. Avrunin is with the Department of Mathematics and Statistics, Box 34515, University of Massachusetts, Amherst, MA 01003. E-mail: avrunin@math.umass.edu.*
- *J.C. Corbett is with the Department of Information and Computer Science, University of Hawaii, Honolulu, HI 96822. E-mail: corbett@hawaii.edu.*
- *L.K. Dillon is with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824. E-mail: ldillon@cse.msu.edu.*

Manuscript received 23 Sept. 1997; revised 1 May. 1998.
Recommended for acceptance by A. Fuggetta and R. Taylor.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 106769.

seldom proceeds at a uniform rate. Some components will have been fully implemented while others remain only partially specified. Analysis at this stage, before the system has been completely implemented, can make use of the detailed information about implemented components to verify that the system will meet its requirements if the unimplemented components meet their specifications, or point out the need for modifications in the specifications or implementations.

- **Evolutionary Development**. In order to understand the implications of proposed modifications to an existing system, developers necessarily confront a combination of fully-implemented components, those that will not be modified, and specifications for the new or modified components. Indeed, specifications of the original components of the system may not be available at all during maintenance, so analysis using high-level specifications of all components may be impractical.
- **Compositional Analysis**. Although the performance of the system may depend on some of the details of the implementation, it may be possible to abstract much of the implementation detail away without affecting the analysis of a particular aspect of the system. In this case, some of the components of a fully-implemented system could, for the purposes of analysis, be represented by a high-level specification of their interfaces with the rest of the system. Such compositional techniques can make analysis practical for systems that would otherwise be far too large for existing analysis methods.
- **Modeling the Environment of the System**. Most real-time systems are *reactive*—they interact repeatedly with their environments, rather than simply computing a value and terminating. Although the environment may consist largely of other computer systems, it may involve components such as sensors that will not be implemented in software. For the purpose of analyzing the behavior of the system, it may be more appropriate to express the possible behavior of the environment in a suitable high-level specification than to fully implement a software model with the same behaviors.

In this paper, we propose a method for analyzing partially-implemented real-time systems. We consider real-time concurrent systems for which some components are implemented in Ada and some are partially specified using regular expressions and Graphical Interval Logic (GIL) [4], a real-time temporal logic with an intuitive graphical representation similar to the time-lines typically used by system developers. We show how automata derived from the regular expressions and the GIL specifications [5], [6] can be combined with hybrid automata constructed from the Ada code [7] to construct models of the partially-implemented systems that account for such properties as run-time overhead and scheduling, yet support tractable analysis of nontrivial programs. Our method can be fully automated. We illustrate the approach with analysis of a small example.

In the next section, we briefly discuss some related work. In the third section, we explain our approach and apply it to a small example. Some additional details of the approach are given in the fourth section, and the final section presents some conclusions and directions for further research.

2 Related Work

The main contribution of this paper is an approach to the analysis of real-time concurrent systems that allows some parts of a system to be specified in a temporal logic while making use of detailed implementation information about other components. The analysis thus involves combining information from very different sorts of formal models of real-time systems.

Many formal models have been proposed for general real-time concurrent systems. These include timed Petri nets, communicating finite state machines, timed automata, timed process algebras, and real-time logics. In this paper, we rely on hybrid automata, which are produced from Ada code and from specifications written using GIL and regular expressions. We believe that GIL's graphical representation, discussed and illustrated below, and the simple and familiar semantics of regular expressions make them especially suitable as high-level specification formalisms for use by developers of real-time systems. We convert GIL specifications into automata using algorithms that were originally developed for producing oracles to monitor executions of concurrent systems [5], [6] and that evolved from the GIL decision procedure described in [8].

For the most part, the work on formal models of real-time concurrent systems has been intended to represent specifications, not implementations. As such, it does not address some of the difficult issues that arise in representing implementation details of real software. For example, resource constraints are absent in most of these models and are awkward to represent within them. Also, the effects of run-time overhead, which can be significant, are not considered. Thus, although many of these models may have the expressive power needed to represent the detailed timing properties of fully-implemented components of a system, researchers generally have not addressed the problem of constructing such representations from real software. Corbett [7], [9], [10] has developed models for concurrent Ada programs that represent these detailed timing properties.

A number of authors (e.g., [11], [12], [13]) have proposed methods for doing compositional analysis by decomposing a concurrent system into subsystems with simple interfaces and replacing some of the subsystems by simpler processes that have the same interfaces to their environments. In most of this work, however, the simpler processes that replace subsystems are specified in the same formalism and notation used to describe the original system. The approach we propose here composes systems whose components are described in two very different notations, a graphical real-time logic and the Ada programming language.

Several researchers have considered the problem of integrating different types of notations for representing the components of a system. For example, Zave and Jackson [14] discuss the integration of different specification formalisms by

translating each formalism into predicate logic. This work does not address the problem of analyzing the systems so specified. Pezzè and Young [15], [16] present an approach to building state-space analysis tools that accept system descriptions involving several formalisms, but that work is chiefly concerned with the generation of tools rather than analysis of systems described in specific notations and does not discuss the representation of real-time systems. The work on Cabernet [17] involves the construction of an environment for the specification and analysis of real-time systems that uses a class of high-level Petri nets as the formal kernel but provides features for customization that could support specifications written in a variety of other formalisms.

Perhaps the work closest in spirit to this paper is that of Bagrodia and Shen [18], [19]. They do stochastic performance evaluation of real-time systems in which some components are fully implemented and others are represented by discrete-event simulation models. The main difference between this work and ours is that their analysis is dynamic, executing the models and assessing a particular set of executions, while ours considers all possible executions.

3 Approach

We illustrate our approach using the following example. A signal processing system consists of a sensor that produces data sporadically and an Ada program that processes this data as quickly and as accurately as possible. Fig. 1 shows the structure of the program, which contains three Ada tasks. The `Sensor` task is awakened by the sensor, reads the sensor, and offers this reading to the `Control` task. The `Control` task accepts a reading from the `Sensor` task and gives it to the `Tracking` task for processing. The `Tracking` task processes the sensor reading and returns a trace to the `Control` task for display. Only the `Sensor` and `Control` tasks are currently implemented; the specification for the `Tracking` task consists of a list of its possible interactions with the other tasks, a regular expression specifying the orders in which these interactions can occur, and GIL specifications for some of the timing properties of those interactions. The processing of the sensor reading is not described in the specification. Further details on the program are given below along with a description of the Ada and GIL constructs used to express them.

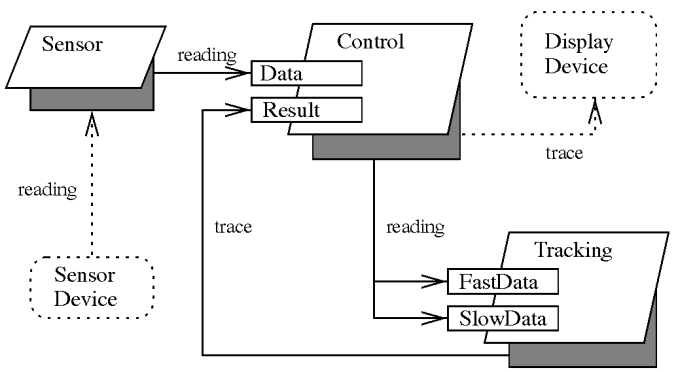


Fig. 1. Structure of example.

3.1 Ada

The Ada source code for the two implemented tasks and a specification of the interface for the unimplemented `Tracking` task are shown in Fig. 2. The program runs on a uniprocessor, so the three tasks must share a single CPU. Ada uses a preemptive priority scheduling policy for tasks. The priority ordering of the tasks, from highest to lowest, is: `Control`, `Sensor`, `Tracking`.

In Ada, two tasks may interact via a *rendezvous*, a synchronous communication in which one task, the *caller*, calls an *entry* of another task, the *acceptor*. The caller is blocked until the acceptor accepts the entry call with an `accept` statement naming the corresponding entry. After the rendezvous completes, both tasks may continue executing independently. For example, a task may call entry `E` of task `T` with the statement `T.E`, and task `T` may accept this call with the statement `accept E`. Rendezvous may be nested: if the body of an accept statement contains an entry call or an accept (e.g., the accept statements for entry `Data` of task `Control` in Fig. 2), then the caller of this entry (e.g., task `Sensor`) remains blocked while the inner rendezvous completes. Data may be exchanged during the rendezvous through parameters, as in a procedure call.

The sporadic nature of the sensor complicates the program. The sensor device awakens the `Sensor` task with an interrupt that causes the entry call on `SensorDevice.Interrupt` to complete. We do not model `SensorDevice` explicitly, but simply assume that this entry call can complete at any time. The `Sensor` task then constructs a reading and offers it to the `Control` task. If the sensor reading is not accepted within a certain time (`ReadingExpire`), then it is discarded. This timeout is implemented in Ada using a timed entry call, a version of the `select` statement that bounds the amount of time a task will wait for an entry call to be accepted.

The `Control` task adapts to the rate at which the sensor is producing data by switching between two modes: fast and slow. If a sensor reading is ready when the `Control` task reaches its `select` statement (i.e., if the `Sensor` task is blocked calling the entry `Control.Data`), the `Control` task switches to fast mode. The `Control` task indicates that it is in fast mode by calling the `FastData` entry of the `Tracking` task, which then performs a faster but less precise calculation to produce the trace. If a sensor reading is not ready when the `Control` task reaches its `select` statement, the task switches to slow mode and communicates the reading to the `Tracking` task via the `SlowData` entry. The mode is set on each iteration of the `Control` task's loop by the conditional select statement, which will choose the `else` alternative if a rendezvous at entry `Data` cannot begin immediately.

The behavior of the unimplemented `Tracking` task is specified by the stub in Fig. 2 and by the regular expression and GIL formulas discussed below. The stub of the `Tracking` task lists its possible interactions with the other tasks; in particular, it lists all of the communication statements the task will contain (e.g., entry calls, accepts). The `Tracking` task has two possible interactions with the other tasks: it can use a `select` statement to block waiting for an entry call on entry `FastData` or `SlowData`, or it can call the `Result` entry of the `Control` task. Note that the order in

```
task Sensor is           -- reads sensor
  pragma Priority(10);   -- middle priority
end Sensor;

task body Sensor is
  R : Reading;
begin
  loop
    SensorDevice.Interrupt; -- accepted on interrupt
    ReadSensor(R);          -- read the sensor
    select
      Control.Data(R);      -- offer data to control task
    or                      -- but after ReadingExpire
      delay ReadingExpire;  -- discard the reading
    end select;
  end loop;
end Sensor;

task Control is                -- controls other tasks
  pragma Priority(15);         -- highest priority task
  entry Data(R : in Reading);  -- data from sensor task
  entry Result(R : in Trace);  -- trace from tracking task
end Control;

task body Control is
begin
  loop
    select
      accept Data(R : in Reading) do  -- if Data ready
        Tracking.FastData(R);          -- use fast mode
      end Data;
    else                               -- otherwise
      accept Data(R : in Reading) do   -- wait for Data
        Tracking.SlowData(R);          -- use slow mode
      end Data;
    end select;
    accept Result(T : in Trace) do  -- wait for result
      Update_Display(T);             -- update display
    end Result;
  end loop;
end Control;

task Tracking is                   -- computes trace
  pragma Priority(5);              -- lowest priority task
  entry FastData(R : in Reading); -- data to process fast
  entry SlowData(R : in Reading); -- data to process slow
end Tracking;

task stub Tracking is   -- not real Ada
  T : Trace;
begin              -- stub is just list of interactions
  -- Interaction 1: accept data at SlowData or FastData
  interaction
    event "Ready";
    select       -- accept call on SlowData or FastData
      accept SlowData(R : in Reading);
      event "Slow";  -- accepted call on SlowData
    or
      accept FastData(R : in Reading);
      event "Fast";  -- accepted call on FastData
    end select;
  end interaction;
  -- Interaction 2: call entry Result of task Control
  interaction
    event "Result";
    Control.Result(T);
  end interaction;
end Tracking;
```

Fig. 2. Source code for example.

which these interactions are listed in the stub does not constrain the order in which they might occur in an execution of the system—the stub simply lists the possible interactions. We label the two interactions with the events *Ready* and *Result*, and also label the two communication statements in the first interaction with the events *Fast* and *Slow*. These labels are used to specify the order and timing of the events, as illustrated below. (Additional event labels could be inserted into a stub for any events whose order or timing is to be specified.) The stub does not specify any of the complex signal processing actually carried out by the **`Tracking`** task.

The hybrid automaton constructed to model the behavior of the partially-implemented Ada program must represent the unimplemented tasks, but it should not constrain their behavior. Thus, when constructing the hybrid automaton, we assume only that an unimplemented task alternates between performing an arbitrary amount of internal computation and nondeterministically selecting an interaction (from its stub) in which to engage.

Although this approach provides a conservative abstraction of any full implementation of the task, the resulting model is unlikely to be accurate enough to allow verification of interesting properties. Therefore, we specify additional constraints on the order and timing of the events in the task in order to improve the accuracy of the model. When the task is eventually implemented, we must verify that the implementation satisfies these constraints.

3.2 Regular Expressions

We restrict the order of stub task interactions using regular expressions. Each expression specifies a language over the event labels annotating the stub task interactions; such a language constrains the legal orderings of the event labels it contains. For example, the **`Tracking`** task alternates between accepting sensor readings and producing results, which we specify with the expression

(*Ready Result*)*

The order of interactions could also be specified by providing a skeleton of the task's control flow or a GIL formula. In our example, we could simply place a loop statement around the two interactions. In general, however, regular expressions may be preferable for specifying simple ordering properties due to their straightforward and familiar semantics. They can express regular patterns of interactions very concisely, and can be easier to read and write than code skeletons or GIL formulas.

3.3 GIL

We specify the timing constraints of stub tasks using a variant of GIL, a real-time temporal logic with an intuitive graphical representation. We use GIL because we find its visual formulas very natural and easy to understand. Other real-time logics or automata could be used equally well. The GIL specifications for a stub task constrain the order and timing of the events that label the stub's interactions. We, therefore, use an event-based interpretation for GIL, rather than the customary state-based interpretation. GIL formulas are given in a graphical form similar to the timelines frequently used by system developers. We explain the

features of GIL used in this example as we discuss the specifications shown in Fig. 3.

Recall that our example program runs on a uniprocessor. Processor sharing complicates the specification of a task's timing constraints—if a pre-emptible code region takes at most 3 sec to execute, then the task *runs* for at most 3 sec between the events representing the beginning and the end of the code region; however, the task may be preempted for part of this time, so the actual elapsed time may exceed 3 sec. When specifying the timing constraints of a task, we employ two different kinds of time: *local time* advances only when a task is running and is useful for specifying bounds on the execution time of code regions, while *global time* always advances and is useful for specifying timeouts and other intervals in absolute time. We illustrate the use of these different kinds of time below.

GIL formulas for the timing constraints of the **`Tracking`** task are shown in Fig. 3. We label individual formulas for convenience in referring to them below (**`Run1`**, **`Run2`**, etc.). The GIL formulas are read from top to bottom and left to right. The horizontal dimension shows the progression of time, which advances to the right. An outermost *interval* represents a legal sequence of (timed) events; the (sub)formula below it describes constraints on the ordering of events and on the times at which they occur. More generally, an interval represents the interval of time between two events, which designate its "endpoints"; the formula nested directly below an interval describes real-time properties of the events that occur within this time interval. A formula that must hold at the beginning of an interval is left justified below the start of the interval, a formula that must hold invariantly (at all times) in an interval is left justified below a henceforth operator (□) and a formula that must eventually hold (at some time) in an interval is left justified below an eventually operator (◇). Special *duration* predicates place bounds on the amount of time, either global or local, that can elapse between the endpoints of intervals.

GIL provides *search* operators for specifying the endpoints of intervals. A dotted arrow denotes a search to one of the *target* events that appear (left justified) immediately below the right arrowhead. For example, the nested interval in the formula **`Run1`** consists of all events in a legal event sequence up to (but excluding) either the earliest *Fast* event or the earliest *Slow* event, depending on which occurs first. A search is said to *succeed* if its target event is located; otherwise, it *fails*. If a search fails, evaluation of the remaining searches is abandoned, and the formula being evaluated is true if the failed search is *weak* (indicated by a single arrowhead) or false if the failed search is *strong* (indicated by a double arrowhead). Thus, **`Run1`** specifies that the tracking task eventually reaches either *Fast* or *Slow* and it does so within L_1 to U_1 μsec after the start of the program. The duration predicate in **`Run1`** uses local time—it bounds the time that the **`Tracking`** task can run before accepting a call on entry **`FastData`** or **`SlowData`**. There is no bound on the actual (global) time from the start of the program until the **`Tracking`** task accepts a call at **`FastData`** or **`SlowData`** since the sensor device might never fire.

A bidirectional search, indicated by arrowheads at both ends of the dotted search line, must be preceded by another

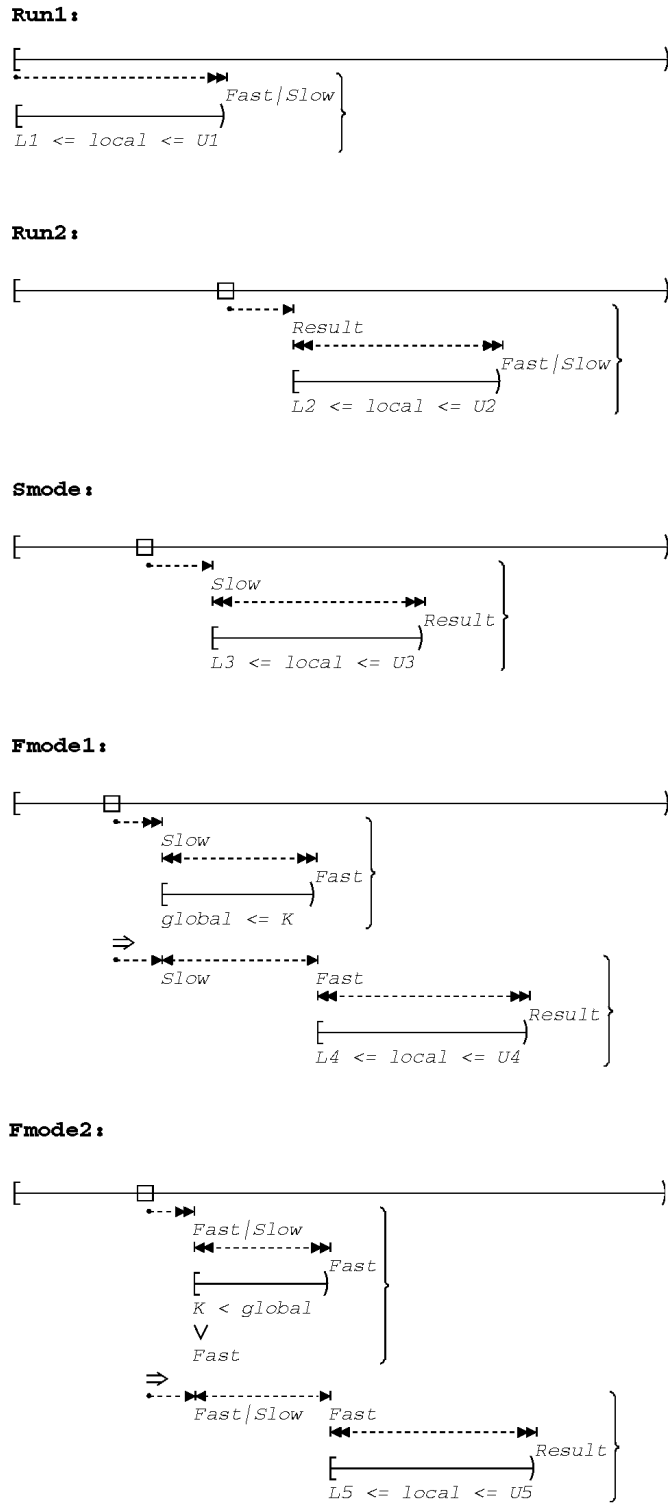


Fig. 3. GIL Specifications for **`Tracking`** task.

(regular or bidirectional) search, as in the remaining specifications in Fig. 3. A bidirectional search succeeds if its target event is located and if the event located by the prior search is the last such event to come before the target, and fails otherwise. In **`Run2`**, for example, if the next *Result* is the last *Result* before a subsequent *Fast* or *Slow*, the search succeeds; it fails, however, if a second *Result* occurs between the next *Result* and the subsequent *Fast* or *Slow*. As above, a

weak search may fail, but a strong search may not (unless an earlier weak search fails). For example, the formula below the henceforth operator in **Run2** requires the next *Result*, if one occurs, to cause an eventual *Fast* or *Slow* before any subsequent *Result* and also bounds the time that can elapse between the *Result* and the *Fast* or *Slow* that it causes. **Run2** asserts that this formula is an invariant. Thus, it specifies that the tracking task always reaches either *Fast* or *Slow* within L_2 to U_2 μsec (local time) after reaching *Result* and that it does so before reaching another *Result*. This specification thus bounds the time that the task can run between calling **Control.Result** and receiving the next reading at entry **FastData** or **SlowData** and also requires that each *Result* is followed by a *Fast* or *Slow* before the next occurrence, if any, of *Result*. Similarly, the formula **SMode** bounds the (local) time between receiving a reading at entry **SlowData** and calling the **Result** entry of the control task to within L_3 to U_3 μsec.

The last two formulas, **FMode1** and **FMode2**, specify the time to produce a result when the system is in fast mode. The task priorities ensure that the first set of data is processed in slow mode. Thus, there are two cases: 1) if a reading is received in fast mode and the previous reading was processed in slow mode less than K (global) μsec ago, then some of the information cached from that previous computation can be used to speed the processing of the current reading, so the time to produce the result is from L_4 to U_4 μsec; 2) if the previous reading was processed in fast mode or was processed more than K μsec ago, then the time to produce the result is from L_5 to U_5 μsec ($U_4 < U_5 < U_3$). In graphical formulas, we use a vertical layout with the classical boolean operators, e.g., implication ($\Rightarrow$) in **FMode1** and **FMode2**, and disjunction ($\vee$) in **FMode2**. The searches in the antecedent of the implication in **FMode1** are strong, since the time from a *Fast* to the next *Result* must lie in $[L_4, U_4]$ only if the *Slow* preceding this *Fast* actually occurred no more than K μsec ago. The antecedent of **FMode2** asserts that either the actual time of the interval exceeds K or the event that starts the interval is a *Fast*.

In addition to specifying the timing properties of unimplemented tasks, we can also use GIL to specify constraints on the environment in which the program executes. For example, the sensor device in our example can generate interrupts at most every F μsec. We express this constraint with the GIL formula **IntFreq** in Fig. 4.

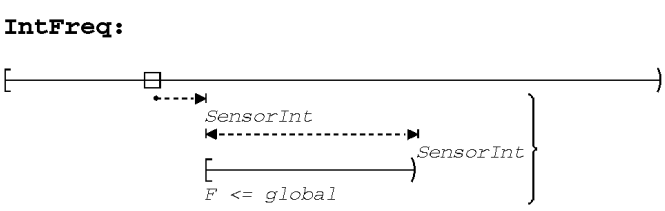


Fig. 4. GIL specification of constraint on environment.

3.4 Hybrid Automata

In order to perform analysis, we translate the various specifications of the program into a common abstract model: constant slope linear hybrid automata [20], [21]. Hybrid automata combine a finite-state control with a set of real-valued variables. The values of these variables change continuously while the automaton remains at a control location, and may change discretely with an instantaneous transition from one control location to another. We use the real-valued variables of the hybrid automaton to enforce timing constraints on its transitions.

We first construct the *program automaton*, a hybrid automaton representing the program, using the method of [7]. Each control location in the program automaton is an abstraction of the program's state, and each transition represents the execution of a code region transforming that state. The execution time of a code region is modeled with an appropriate delay before its transition; the transition occurs instantaneously when execution of the code region completes. The length of this delay must fall in the interval $[L, U]$ given by the bounds on the region's execution time and is measured using a stopwatch-like mechanism. A real-valued variable, x, which advances continuously as time passes, is reset to zero when the location representing the beginning of the code region is reached. We attach the *guard* condition $x \geq L$ to the transition that prevents its occurrence until at least L time units have passed, and we attach the *invariant* condition $x \leq U$ to the location that requires it to be exited before more than U time units have passed.

Fig. 5 shows a slightly simplified[1] part of the program automaton constructed from the (partial) program in Fig. 2. The full automaton has 70 locations. Each location is labeled with the task that is running there, an invariant, and the rate at which each real-valued variable is changing. The rate of variable x is denoted $\dot{x}$. Each transition is labeled with a guard, a synchronization label, and a set of assignments to the variables.

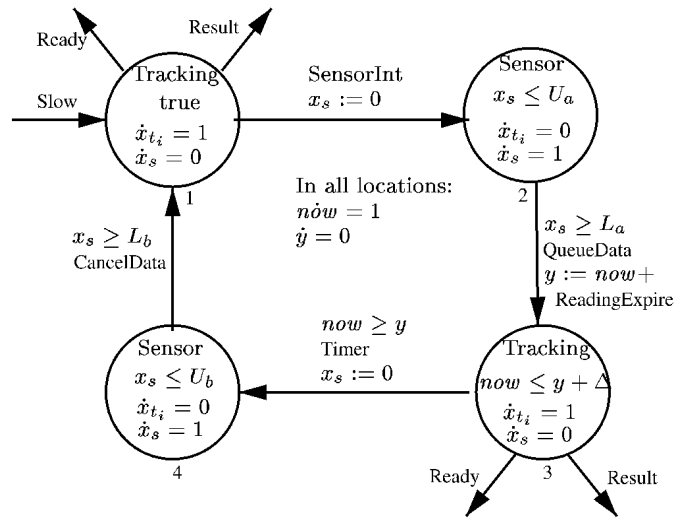


Fig. 5. Part of program automaton.

The stopwatch-like mechanism used to specify the bounds of code regions is illustrated in location 2 of Fig. 5. Location 2 represents the program state where the **Sensor**

1. Certain sequences of transitions have been collapsed into single transitions and certain variables not relevant to this part of the automaton have been omitted.

task has just been awakened by the sensor device's interrupt and has preempted the **Tracking** task, which is still processing the previous reading (which the **Control** task is blocked waiting to receive). The *QueueData* transition from location 2 to 3 represents the code region that reads the sensor, queues a call on entry **Control.Data**, sets a timer to expire after **ReadingExpire** seconds, and then blocks the task. We denote the bounds of this interval with $[L_a, U_a]$ and use variable x_s to record the CPU time allocated to the **Sensor** task. Variable x_s is reset by the transition into location 2, where it advances at rate 1 (and hence records the amount of time spent in the location). After L_a μsec, x_s is at least L_a, so the *QueueData* transition may be taken; the transition must be taken before more than U_a μsec have elapsed. Similar constraints are generated for the *CancelData* transition from location 4, which represents the code region that cancels the entry call on **Control.Data** and blocks waiting for the sensor's interrupt. The bounds L_i and U_j would be derived from the code represented by the transitions, as discussed below.

The *SensorInt* and *Timer* transitions represent the execution of code in interrupt handlers rather than in specific tasks. The *Timer* transition represents the timer interrupt that awakens the **Sensor** task **ReadingExpire** μsec after the call on **Control.Data** is queued. Its timing constraints are specified using the variable now, which records the current time, and the variable *y*, which records the time of the pending timer signal. Note that $\dot{y} = 0$ and $\dot{now} = 1$ in all locations. The constant Δ accounts for the inaccuracy of the timer mechanism. The *SensorInt* transition represents the interrupt caused by the sensor device that awakens the **Sensor** task. This transition has no timing constraints and thus may occur at any time.

In location 1, the **Tracking** task has just received a sensor reading to process in slow mode (event *Slow*) after computing a result. After a stub completes an interaction, it then performs some amount of computation and chooses its next interaction. The *Ready* and *Result* transitions represent this computation and the two possible interactions that might follow. Note that there are no timing constraints on these transitions since a stub does not specify any timing properties.

The program automaton generated from Fig. 2 is a conservative abstraction of (any full implementation of) the program, in the sense that its behaviors are a superset of those of any implementation, but is not accurate enough to verify many interesting properties. For example, without constraining the order of the stub task's interactions, the model contains a deadlock; in the location reached by taking the *Ready* transition from location 1 or 3 in Fig. 5, the **Control** and **Tracking** tasks are in a deadly embrace (the **Control** task is waiting for a call on entry **Result** while the **Tracking** task waits for a call on entries **FastData** or **SlowData**). To filter out these spurious locations, we convert the regular expression constraining the order of the stub task interactions that was given earlier into an automaton and intersect this automaton with the program automaton. (Intersection is performed using the standard product operator for hybrid automata [20] in which transitions sharing the same event label must be taken together.) The resulting intersection eliminates the transitions on *Ready* from locations 1 and 3.

Even without deadlocks, the resulting model is still not accurate enough for timing analysis without incorporating the timing constraints of the stub task specified with the GIL formulas (e.g., the **Tracking** task may run arbitrarily long before reaching the *Ready* interaction). To filter out behaviors that violate these timing constraints, we convert each GIL formula into a hybrid automaton that accepts only (timed) event sequences satisfying the constraint. We then intersect these automata with the automaton obtained by intersecting the program automaton and the hybrid automaton derived from the regular expression. The resulting hybrid automaton accepts only timed event sequences that satisfy the timing and sequence constraints of the implemented Ada tasks, the sequence constraints of the regular expression, and the timing constraints of the GIL formulas.

For example, consider the GIL formula **SMode** in Fig. 3, which specifies the CPU time to produce a trace in slow mode. Using the method outlined in the next section, we convert this formula into the hybrid automaton in Fig. 6. The GIL formula specifies a bidirectional search; it says that the local time between a *Result* event, if any occurs, and the most recent *Slow* event is in the interval $[L_3, U_3]$. Since the formula specifies a local timing constraint for the **Tracking** task, we use a variable x_{t_1} that advances in locations where the **Tracking** task is running. Each local timing constraint for the **Tracking** task uses its own variable (e.g., x_{t_1}, x_{t_2}, ...). If a *Slow* event occurs in the initial location, there is a transition to the second location and x_{t_1} is reset to 0. The invariant in that location allows the automaton to remain there only while $x_{t_1} \leq U_3$ If a *Result* occurs when x_{t_1} is at least L_3, there is a transition back to the first location.

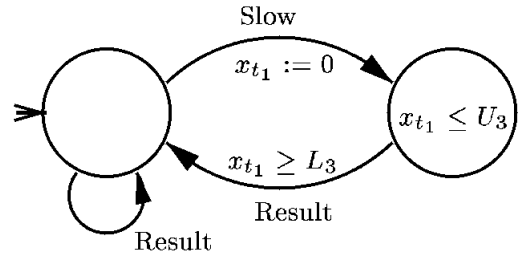


Fig. 6. Hybrid automaton from formula **SMode** in Fig. 3.

Intersecting the automaton in Fig. 6 with the program automaton effectively combines their timing constraints: the transition on *Slow* shown in Fig. 5 now resets x_{t_1}, the condition $x_{t_1} \leq U_3$ is conjoined to the invariants of locations 1–4 (which appear between events *Slow* and *Result*), and the condition $x_{t_1} \geq L_3$ is conjoined to the guards of the *Result* transitions.

The ability to stop clocks allows a simple representation of timing constraints in the presence of pre-emption. For example, if code region *Result* is interrupted in location 1 by the pre-emption of the **Sensor** task, then when the **Tracking** task resumes the execution of *Result* in location 3, the variable x_{t_1} will still contain the CPU time expended on *Result*

in location 1, but not the time spent in location 2 while the **Sensor** task was running. In fact, the **Tracking** task may be pre-empted many times (represented by the cycle through locations 1–4) before it accumulates enough CPU time (recorded in x_{t_1}) to produce a result.

Fig. 7 shows the automaton for the formula **FMode1**. The variables y_1 and x_{t_2} represent a global clock and another local clock that advances in locations where the **Tracking** task is running, respectively. We believe this formula illustrates that GIL specifications of nontrivial timing properties are generally easier to write and understand than their equivalent automata.

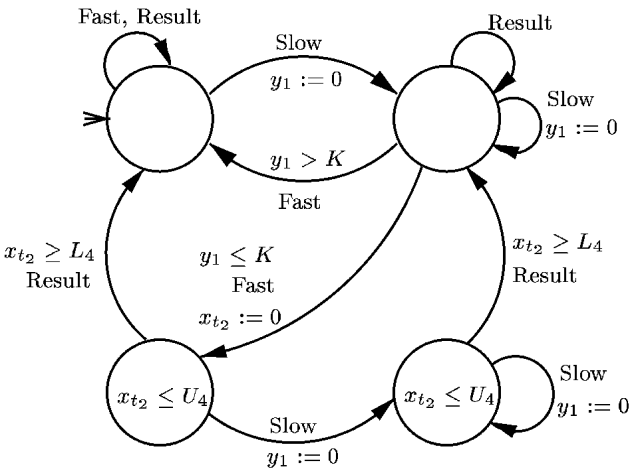


Fig. 7. Hybrid automaton from formula **FMode1** in Fig. 3.

Table 1 gives the bounds we used for the (implemented and unimplemented) code regions, as well as the values of several constants used in the specification and implementation. Since we have not integrated our model building tool for Ada with a sequential timing analysis tool, we estimated plausible durations for the implemented code regions, as well as for the overhead of various run-time operations.

We note that a single transition might represent a sequence of code regions and that the time bounds for each transition are *polished* to be multiples of a constant *m*. We polish constraints in such a way that they are weakened—the set of runs of the automaton with the polished constraints is a superset of the set of runs of the original automaton. In particular, all lower bounds are rounded down to the nearest multiple of *m* and all upper bounds are rounded up to the nearest multiple of *m*. The constant *m* controls the accuracy of the analysis, but also affects the cost. In general, larger values of *m* will degrade the accuracy (i.e., tightness) of the bounds, but speed up the analysis. For our example, we set $m = 25$.

3.5 Analysis

Once we have combined the hybrid automata constructed from the Ada code, the regular expressions, and the GIL formulas (including the formula **IntFreq** of Fig. 4 that constrains the behavior of the environment) into a single hybrid automaton M_S capturing the behavior of the system, we can use standard techniques for analyzing hybrid automata to verify that the system has certain properties. To verify that the system has a property *P*, we construct a hybrid automaton M_P that accepts timed event sequences violating *P*. We then compose M_P with M_S and check to see whether M_P is in an accepting location for any states of the composition reachable from the initial state (the reachable states of a hybrid automaton are computed using a fixpoint calculation; see [20] for details). The property automaton M_P does not constrain the behavior of M_S; it simply observes the behavior (via synchronizing transitions) and accepts violations of *P*.

TABLE 1
DURATIONS USED IN EXAMPLE (μSEC)

Duration/Delay	Range/Value
ReadSensor	[100, 200]
ReadingExpire	1500
Update_Display	[75, 100]
Code to queue/pend call in caller (before blocking)	[60, 80]
Code to complete call in caller (after being signaled)	[10, 20]
Code to begin rendezvous in acceptor (before body)	[25, 50]
Code to complete rendezvous in acceptor (after body)	[25, 50]
Code to set timer request	[25, 50]
Code to process timer expiration	[25, 50]
Timer interrupt period (Δ)	100
Cache life (*K*)	2500
Max sensor interval (*F*)	2000
Start to first data ($[L_1, U_1]$)	[60, 120]
Result to next data ($[L_2, U_2]$)	[60, 120]
Process slow data ($[L_3, U_3]$)	[1000, 1500]
Process fast data (with cache) ($[L_4, U_4]$)	[200, 400]
Process fast data (without cache) ($[L_5, U_5]$)	[400, 600]

The system designer would specify the property *P* with a GIL formula, which is then negated and converted into a hybrid automaton. We use the HyTech verifier for hybrid systems [22] to analyze the hybrid automata we construct. We illustrate our technique by verifying several properties of our example.

First, we verify that the **Sensor** task will never time out waiting to deliver the data to the **Control** task. We label all transitions in M_S representing this timeout with the event label *Expire* and construct the automaton **Expire** in Fig. 9 from the negation of the GIL formula **NeverExpire** in Fig. 8. Using HyTech, we determine that there are no reachable states of the composition of M_S and automaton **Expire** in which **Expire** is in an accepting location. This result proves that no data can be dropped by the **Sensor** task. Note that, although this property does not explicitly mention time, its proof depends on the timing of events.

An analyst evaluating the performance of the system may be concerned about the possible degradation of the quality of tracking if two or more consecutive sensor readings are processed in fast mode. To determine whether this can occur, we compose M_S with the automaton **TwoFast** in Fig. 9, which was constructed from the negation of the formula **NeverTwoFast** in Fig. 8. (The $\Diamond$ is the standard "eventually" operator of temporal logic, so this specification asserts that a *Slow* event occurs between any two occurrences

Fig. 8. GIL Specifications of properties.

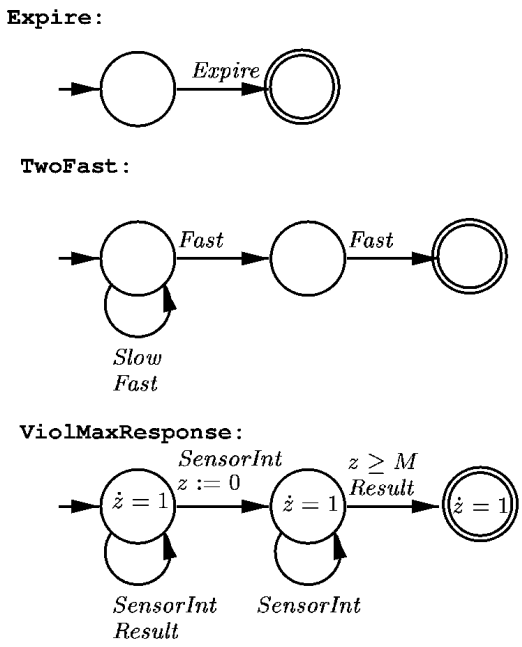


Fig. 9. Automata for properties.

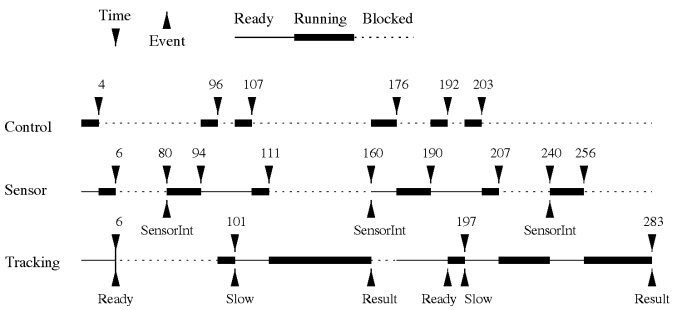


Fig. 10. Behavior illustrating bound on time from sensor interrupt to result (time in units of 25 μsec).

of *Fast*.) This time, HyTech reports that there are reachable states of the composition in which automaton **`TwoFast`** is accepting and displays a time-stamped sequence of transitions containing two *Fast* events without an intervening *Slow* event. This sequence can help the analyst understand how such degraded tracking can occur, and perhaps suggest ways to modify the program to avoid it. As in the case of **`NeverExpire`**, the proof of **`NeverTwoFast`** depends on the real-time behavior of the program.

We can also verify explicit timing requirements. One common timing requirement is that the time from a stimulus to a response is bounded by a given constant. In our example, we verify that the time from a sensor interrupt (event *SensorInt*) to a result being produced (event *Result*) is less than M μsec. We specify this property using the GIL formula **`MaxResponse`** in Fig. 8, whose negation can be converted to the hybrid automaton **`ViolMaxResponse`** in Fig. 9. Although we could fix M at a specific value and proceed with the analysis as before, we instead instruct HyTech to perform a parametric analysis to determine the longest time from a *SensorInt* event to the following *Result* event. In particular, HyTech can solve for the (weakest) constraints on M that are required for the composition of **`ViolMaxResponse`** and M_S to reach an accepting location of **`ViolMaxResponse`**. HyTech reports that this constraint is $M \leq 3{,}075$, thus an upper bound on the time between these events is 3,075 μsec. HyTech also produces a time-stamped sequence of transitions illustrating this bound, which we diagram in Fig. 10. Note that the implementation of the rendezvous mechanism and the priority scheduling of the tasks on a single CPU produces a fairly complex behavior.

In the analyses described above, we used the model building tool for Ada described in [7] to construct the hybrid automaton representing the program from an Ada-like specification[2] of the code in Fig. 2. The time bounds of the code regions are specified using special *computation events* embedded in this specification; in a real timing analysis tool, these durations would be derived from the sequential code comprising these regions using techniques like [23]. The automata for the GIL formulas were constructed by hand using a set of reduction rules and the tableau method of [6]. These automata are fed to HyTech, which takes their product and computes the set of reachable states. The performance of HyTech (version 1.04) on the analyses described above is given in Table 2. Times are in seconds on a Sun SPARCstation 10 with 96 MB of memory and include both user and system time.

TABLE 2
ANALYSIS TIMES FOR EXAMPLE (SEC)

Property	Time (sec)
`Expire`	106
`TwoFast`	139
`MaxResponse`	207

4 DETAILS

The approach presented here combines two separate bodies of work: Corbett's method for constructing timed models of

2. Our specification language contains (a subset of) Ada's constructs, but in a Lisp-like syntax to facilitate parsing. For example, we write `(call T E)` rather than `T.E`.

Ada tasking programs [7] and Dillon et al.'s method for using Graphical Interval Logic to construct test oracles [5], [6]. In this section we briefly sketch the two methods and outline the modifications and extensions that were necessary to combine them.

4.1 Ada to Hybrid Automata

The program automaton is generated using the method of Corbett [7], with some slight modifications to allow unimplemented tasks. Here, we sketch the basic method and describe the necessary modifications for analysis of partially-implemented programs.

An Ada tasking program is transformed into a hybrid automaton in two stages. In the first stage, we derive a state-transition system from the program representing the program's untimed behavior. Each state in the transition system represents an abstraction of the program's state, and each transition represents the execution of program code transforming that state. The program's state comprises the control locations of the tasks, the values of any critical program variables (identified by the analyst), and the contents of the run-time systems structures used to implement the tasking constructs (e.g., the ready queue, the entry queues, task priorities). Each tasking statement is expanded via source-level transformations into sequential code that checks/updates run-time structures and invokes thread primitives to block/signal the appropriate tasks. This process results in a very accurate model, which is reduced using a virtual coarsening method described in [7].

In the second stage, we capture the timing constraints of the program by adding timing constraints to the state-transition system, thus transforming it into a hybrid automaton. The state of the hybrid automaton consists of the state of the transition system (i.e., the location of the hybrid automaton) and the values of continuous variables used to measure the current time, the time of pending interrupts, and the amount of CPU time allocated to each task. The timing constraints for a transition are derived from the time bounds of the code it represents and are expressed in terms of the continuous variable representing the CPU time allocated to the task executing the code.

In order to construct models of partially implemented programs, we modified this basic method to allow stub tasks. A stub task is defined by a set of interactions (communication statements) and has no timing constraints from the Ada code. For this paper, we use a simple source-level transformation to rewrite a stub task as a normal task that nondeterministically selects a sequence of interactions. For example, the stub on the left below would be translated into the task body on the right:

```
task stub T is
begin
  interaction
    A;
  end interaction;
  interaction    ->
    B;
  end interaction;
end T;
```

```
task body T is
begin
loop
    if... then
      A;
    elsif... then
      B;
    end if;
  end loop;
end T;
```

The ... is an additional language primitive recognized by the tool that generates the hybrid automata and represents a nondeterministic choice. In addition to this transformation, we suppress generation of timing constraints for stub tasks. Instead, we specify a set of variables for each stub task that are to be advanced when the stub task is running; these variables can be used by the automata generated from GIL formulas to specify the local timing constraints of the stub task.

In the future, we may be able to generate some or all of the Ada stubs automatically from the implemented tasks. For instance, in the example of Fig. 2, we can infer from the bodies of the implemented tasks that the unimplemented `Tracking` task accepts `FastData` and `SlowData`, and must call `Control.Result`. We cannot tell, however, whether the `FastData` and `SlowData` entries should be accepted together in one interaction, or separately in two interactions. Also, any of these interactions might have a time limit implemented with a timed entry call or select statement. The regular expression(s) and GIL formulas can sometimes provide clues to the intended interaction, but we are still exploring how much information must be provided in a task stub to yield a useful model.

The scalability of modeling/analysis techniques for concurrent systems is limited by the state explosion problem, and our technique is no exception. Using a deterministic task scheduling policy, however, significantly mitigates the state explosion since it greatly reduces the number of possible interleavings of task actions (most concurrency analysis techniques assume scheduling is arbitrary). We believe that the severity of the state explosion is largely dependent on the number of `delay` statements/alternatives in the Ada code. Since task scheduling is deterministic, most of the nondeterminism in the transition system is caused by transitions representing timeouts (although nondeterminism may also be introduced by unmodeled program variables). Currently, we believe our technique is limited to programs with a small number of tasks and a very modest amount of modeled data. Nevertheless, we have shown the technique can still be applied to programs that are far too difficult to analyze by hand.

4.2 GIL to Hybrid Automata

The standard interpretation for GIL formulas is state-based: A GIL formula is evaluated at a state within a linear sequence of states, where a state maps primitive propositions to boolean values and has an associated duration (the amount of time spent in the state). To model timed event sequences by timed state sequences, we introduce a primitive proposition for each event, which is true in "states" immediately following the event. The durations associated with states indicate the amount of time that elapses between event occurrences.

Hybrid automata are produced from GIL formulas by a tableau procedure like that described in [5], [6], but extended to handle duration predicates and bidirectional searches. A tableau procedure uses rules encoding the semantics of a logic to generate a graph, called a tableau, that encodes all alternative ways to satisfy a formula. For example, the formula in node 1 of Fig. 11 yields two alternatives:

one where the leading target *a* is not satisfied immediately (i.e., is not the next event to occur), so the formula must be satisfied at the next time (i.e., after the next event occurs); and one where the leading target *a* is satisfied immediately, in which case the interval starts at the current time, so the formula shown in node 2 must hold. Similarly, the formula in node 2 may be satisfied by satisfying *c* immediately and finding a *b* event in the future (node 3), or by satisfying *a* immediately and satisfying the formula at the next time. As shown in Fig. 11, temporal succession is represented as succession in the graph, with alternatives producing multiple successors. Node 1 is the start node of the tableau. Nodes 1 and 4, which do not require that any events occur, are final nodes. (Node 1 only requires *c* and *b* if *a* occurs.) The tableau is thus seen to accept all event traces in which the first *a* event results in a future *b* event, but only after an intervening *c* event. The nodes become locations in the hybrid automaton. Note that we do not use the final nodes in our analysis, thus the hybrid automata we generate from the GIL formulas actually accept any prefix of a trace satisfying the formula. This allows us to find a prefix of a system behavior that violates a property, but does not satisfy the strong searches of one or more GIL formulas. See [24], [25] for more complete descriptions of tableau procedures for propositional temporal logics.

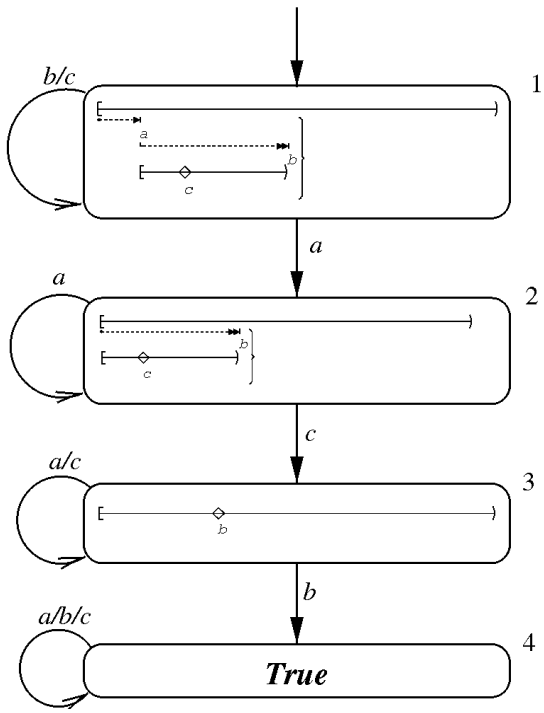


Fig. 11. Tableau.

The tableau algorithm for the untimed fragment of GIL is thus driven by a table of rules that specify the alternatives for the various operators—"□," "◇," intervals, etc. Duration predicates, such as "$L2 \leq local \leq U2$," are handled by special rules, which describe clock activities. These rules introduce various nonlogical terms to indicate the clocks that a transition must activate, inspect, and/or deactivate.

Bidirectional searches are not expressible in standard GIL, which was designed to be insensitive to finite repetition of states in order to facilitate refinement of specifications. However, when specifying properties of event sequences, we often want to express properties relating to the frequency of event occurrences, e.g., `IntFreq`. To express such properties, a logic must be able to detect repeated events. Bidirectional searches are defined by translation to formulas expressed using the standard GIL operators and the next state operator of propositional temporal logic. Our current implementation of the tableau algorithm for GIL uses semantic rules for bidirectional searches that perform this translation; however, we found that this approach explodes the size of the automaton produced from a GIL formula that uses bidirectional searches. Therefore, we are currently modifying our tool to use rules that directly encode the semantics of bidirectional searches. The automata in the previous examples were produced by hand using these new rules for bidirectional searches.

We determinized the hybrid automata produced from the GIL specifications in Fig. 3 to permit analysis with HyTech. (It is not necessary to determinize property automata.) While it is not possible, in general, to represent an arbitrary GIL formula by a deterministic hybrid automaton, we are able to determinize automata for formulas in which all intervals to be timed are specified using a bidirectional search. For the experiment, we determinized the automata by hand using a subset algorithm that we adapted to use the semantics of nodes in simplifying and pruning the subsets. Automation of this method is a topic of ongoing research.

5 Concluding Remarks

Existing static analysis methods for real-time systems assume that all the components of the system can be expressed in the same notation, typically a specification or programming language. For many reasons, however, analysts would benefit from static analysis techniques that can be applied to partially-implemented systems. These reasons include the different rates of development of different components, evolutionary modification or maintenance in which new or modified components are added to an existing system, the use of compositional analysis methods, and the need to model the environment of the system.

In this paper, we have presented a method for analyzing partially-implemented systems for which some components are written in Ada and others are specified using the real-time temporal logic GIL. Our method combines Corbett's method for constructing timed models of Ada tasking programs and the work of Dillon et al. on producing automata from specifications in GIL. We believe that the basic idea of our method is quite general and that it could be extended to apply to systems in which the fully implemented components were written in a programming language other than Ada and the high-level specifications were given in a specification formalism other than GIL. To carry out such an extension, it would be necessary to develop a method for building hybrid automata reflecting the concurrency and timing constructs of the programming language and to produce an algorithm for producing hybrid automata from the specifications. Such a project would certainly involve some significant

challenges (though the work of Dillon and Ramakrishna [6] might simplify the generation of automata from temporal logic specifications), but there is nothing in our basic approach that limits its applicability to Ada and GIL.

We are currently pursuing a number of research directions aimed at improving the performance and scalability of our analysis tools. We are currently completing the automation of our algorithm for producing hybrid automata from GIL formulas. As noted above, we are experimenting with rules that directly encode the semantics of higher-level operators, such as the bidirectional searches, which the current implementation translates into more primitive operators. Past experience has shown that this direct approach can often produce simpler automata more quickly. We are also looking at other kinds of optimizations for the GIL-to-hybrid automata translation process that might reduce the number of clock variables needed. As mentioned in Section 4.1, we are exploring ways to reduce the amount of information that must be provided in the task stubs of the unimplemented tasks. Finally, we are exploring ways to make the automata we generate more amenable to analysis by reducing the number of clocks used and deactivating clocks as soon as possible.

ACKNOWLEDGMENTS

This research was partially supported by the National Science Foundation under Grant Nos. CCR-9308067, CCR-9407182, CCR-9505392, and CCR-9708184; by the Air Force Materiel Command, Rome Laboratory, and the Defense Advanced Research Projects Agency under Contract No. F30602-94-C-0137; and by the Regents of the University of California and Hughes Electronics Corporation under MICRO Grant UCM-20880. This is a revised and expanded version of a paper that appeared in the *Proceedings of the 19th International Conference on Software Engineering*.

REFERENCES

[1] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[2] M.G. Härbour, M.H. Klein, and J.P. Lehoczky, "Timing Analysis for Fixed-Priority Scheduling of Hard Real-Time Systems," *IEEE Trans. Software Eng.*, vol. 20, no. 1, pp. 13–28, 1994.

[3] L. Sha and J.B. Goodenough, "Real-Time Scheduling Theory and Ada," *Computer*, vol. 23, no. 4, pp. 53–62, Apr. 1990.

[4] L.K. Dillon, G. Kutty, L.E. Moser, P.M. Melliar-Smith, and Y.S. Ramakrishna, "A Graphical Interval Logic for Specifying Concurrent Systems," *ACM Trans. on Software Eng. and Methodology*, vol. 3, no. 2, pp. 131–165, Apr. 1994.

[5] L.K. Dillon and Q. Yu, "Oracles for Checking Temporal Properties of Concurrent Systems," *Proc. Second ACM SIGSOFT Symp. Foundations of Software Eng.*, D. Wile, ed., pp. 140–153, New Orleans, ACM Press, Dec. 1994. (Proceedings appeared in *Software Engineering Notes*, vol. 19, no. 5.)

[6] L.K. Dillon and Y.S. Ramakrishna, "Generating Oracles from Your Favorite Temporal Logic Specifications," *Proc. Fourth ACM SIGSOFT Symp. Foundations Software Eng.*, D. Garlan, ed., San Francisco, ACM, Oct. 1996. (Proceedings appeared in *Software Eng. Notes*, vol. 21, no. 6, pp. 106–117.)

[7] J.C. Corbett, "Timing Analysis of Ada Tasking Programs," *IEEE Trans. Software Eng.*, vol. 22, no. 7, July 1996.

[8] Y.S. Ramakrishna, P.M. Melliar-Smith, L.E. Moser, L.K. Dillon, and G. Kutty, "Interval Logics and Their Decision Procedures, Part II: A Real-Time Interval Logic," *Theoretical Computer Science*, vol. 170, nos. 1/2, pp. 1–46, Dec. 1996.

[9] J.C. Corbett, "Modeling and Analysis of Real-Time Ada Tasking Programs," *Proc. Real-Time Systems Symp.*, pp. 132–141, San Juan, Puerto Rico, IEEE CS Press, Dec. 1994.

[10] J.C. Corbett, "Constructing Abstract Models of Concurrent Real-Time Software," Zeil [26], pp. 250–260.

[11] E.M. Clarke, D.E. Long, and K.L. McMillan, "Compositional Model Checking," *Proc. Fourth Ann. IEEE Symp. Logic in Computer Science*, pp. 353–362, 1989.

[12] W.J. Yeh and M. Young, "Compositional Reachability Analysis Using Process Algebra," *Proc. Symp. Testing, Analysis, and Verification (TAV4)*, ACM SIGSOFT, pp. 178–187, ACM, New York, Oct. 1991.

[13] S.C. Cheung and J. Kramer, "Enhancing Compositional Reachability Analysis with Context Constraints," *Proc. First ACM SIGSOFT Symp. Foundations of Software Eng.*, D. Notkin, ed., pp. 115–125, Dec. 1993,

[14] P. Zave and M. Jackson, "Conjunction as Composition," *ACM Trans. on Software Eng. and Methodology*, vol. 2, no. 4, pp. 379–411, 1993.

[15] M. Pezzè and M. Young, "Generation of Multi-formalism State-Space Analysis Tools," Zeil [26], pp. 172–179.

[16] M. Pezzè and M. Young, "Constructing Multi-Formalism State-Space Analysis Tools: Using Rules to Specify Dynamic Semantics of Models," *Proc. 19th Int'l Conf. Software Eng.*, Boston, pp. 239–249, May 1997.

[17] M. Pezzè, "A Formal Approach to the Development of High Integrity Programmable Electronic Systems," *High Integrity Systems*, vol. 1, no. 6, pp. 531–540, 1996.

[18] R.L. Bagrodia and C.-C. Shen, "MIDAS: Integrated Design and Simulation of Distributed Systems," *IEEE Trans. Software Eng.*, vol. 17, no. 10, pp. 1,042–1,058, Oct. 1991.

[19] C.-C. Shen and R.L. Bagrodia, "Parallel Hybrid Models in System Design," *Proc. Winter Simulation Conf.—WSC '93*, G.W. Evens, M. Mollaghasemi, E.C. Russell, and W.E. Biles, eds., pp. 589–594, Dec. 1993.

[20] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The Algorithmic Analysis of Hybrid Systems," *Theoretical Computer Science*, vol. 138, pp. 3–34, 1995.

[21] R. Alur, T.A. Henzinger, and P.-H. Ho, "Automatic Symbolic Verification of Embedded Systems," *IEEE Trans. Software Eng.*, vol. 22, no. 3, pp. 181–201, Mar. 1996.

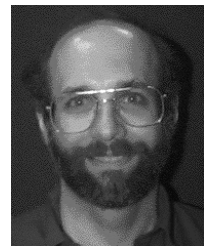
[22] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi, "HyTech: The Next Generation," *Proc. Real-Time Systems Symp.*, pp. 56–65, IEEE CS Press, 1995.

[23] C. Yun Park and A.C. Shaw, "Experiments with a Program Timing Tool Based on Source-Level Timing Schema," *Computer*, pp. 48–57, May 1991.

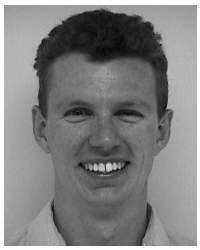
[24] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety*. New York: Springer-Verlag, 1995.

[25] P. Wolper, "The Tableau Method for Temporal Logic: An Overview," *Logique et Analyse*, vols. 110–111, pp. 119–136, June–Sept. 1985.

[26] *Proc. 1996 Int'l Symp. Software Testing and Analysis (ISSTA)*, S.J. Zeil, ed., San Diego: ACM Press, Jan. 1996.



George S. Avrunin received the PhD degree in mathematics from the University of Michigan, Ann Arbor. He is a professor in the Department of Mathematics and Statistics and an adjunct professor in the Department of Computer Science at the University of Massachusetts at Amherst. In addition to formal methods and tools for the analysis of concurrent and real-time software systems, his research interests include the cohomology and representation theory of finite groups. Dr. Avrunin is a member of the IEEE Computer Society, the Association for Computing Machinery, and the American Mathematical Society.



James C. Corbett received the BS degree in computer science from Rensselaer Polytechnic Institute, and the MS and PhD degrees in computer science from the University of Massachusetts at Amherst. He is currently an associate professor in the Department of Information and Computer Science at the University of Hawaii at Manoa. His research interests include methods and tools for analysis of concurrent and real-time software. Dr. Corbett is a member of the Association for Computing Machinery.

Laura K. Dillon received the BA and MS degrees in mathematics from the University of Michigan, Ann Arbor, and the MS and PhD degrees in computer science from the University of Massachusetts at Amherst. She is an associate professor in the Department of Computer Science at Michigan State University. Her research interests include formal methods for modeling and analysis of real-time software systems, formal specification and verification of software, and specification-based software testing. Dr. Dillon is a member of the Association for Computing Machinery, the IEEE Computer Society, and Computer Professionals for Social Responsibility.