Analyzing Partially-Implemented Real-Time Systems

George S. Avrunin\*  
Department of Mathematics and Statistics  
University of Massachusetts  
Amherst, MA 01003-4515 USA  
+1 413 545 4251  
avrulin@math.umass.edu

James C. Corbett†  
Department of Information and Computer Science  
University of Hawaii  
Honolulu, HI 96822 USA  
+1 808 956 6107  
corbett@hawaii.edu

Laura K. Dillon‡  
Computer Science Department  
University of California  
Santa Barbara, CA 93106 USA  
+1 805 893 3411  
dillon@cs.ucsb.edu

ABSTRACT
We propose a method for analyzing 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.

Keywords
Real-time, concurrency, static analysis, Ada, temporal logic, hybrid systems, Graphical Interval Logic

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 non-deterministic 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 [14], to guarantee that a system’s timing requirements are satisfied [12, 19].

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 de-
  velopment of the various components of large systems
seldom proceeds at a uniform rate. Some compo-
  nents 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 im-
plemented components to verify that the system will
meet its requirements if the unimplemented compo-
  nents 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 com-
ponents 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 im-
plementation, it may be possible to abstract much of the
implementation detail away without affecting the anal-
ysis 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
be another computer system, it may involve compo-
nents 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 con-
current systems for which some components are imple-
mented in Ada and some are partially specified using regular
expressions and Graphical Interval Logic (GIL) [9], a real-
time temporal logic with an intuitive graphical representa-

similar to the time-lines typically used by system develop-
  ers. We show how automata derived from the regular expres-
sions and the GIL specifications [11] can be combined with
hybrid automata constructed from the Ada code [8] to con-
struct models of the partially-implemented systems that ac-
count for such properties as run-time overhead and schedul-
ing, yet support tractable analysis of nontrivial programs.
Our method can be fully automated. We illustrate the ap-
proach 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.

**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 in-
formation 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 algorithms for converting GIL specifications into au-
tomata that were originally developed for producing oracles
to monitor executions of concurrent systems [11]. We be-
lieve that GIL's graphical representation, discussed and il-
ustrated below, makes it especially suitable as a high-level
specification formalism for use by developers of real-time
systems.

For the most part, the work on formal models of real-time
concurrent systems has been intended to represent specifi-
cations, not implementations. As such, it does not address
some of the difficult issues that arise in representing im-
plementation details of real software. For example, resource
constraints are absent in most of these models and are awk-
ward 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 expres-
sive 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 [6-8] has
developed models for concurrent Ada programs that repre-
sent these detailed timing properties.

A number of authors (e.g., [4, 5, 22]) have proposed methods
for doing compositional analysis by decomposing a concur-
rent system into subsystems with simple interfaces and re-
placing 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 interval 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 [23] 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 [18] 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 [3, 20]. 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.

**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. Figure 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.

---

**Figure 1: Structure of Example**

---

**Ada**

The Ada source code for the two implemented tasks and a specification of the interface for the unimplemented Tracking task are shown in Figure 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 Figure 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
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
    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
and 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;
      or
        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
and 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
      accepted call on FastData
      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;

Figure 2: Source Code for Example

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 Figure 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 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.

We model an unimplemented task (whose possible interactions are listed in a stub) with a normal task that alternates between performing an arbitrary amount of internal computation and nondeterministically selecting an interaction in which to engage. Although this provides a conservative abstraction of the fully implemented 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.

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.

**GIL**

We specify the timing constraints of stub tasks using a variant of GIL, a real-time temporal logic with an intuitive graphical representation. 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 time-lines frequently used by system developers. We explain the features of GIL used in this example as we discuss the specifications shown in Figure 3.

Recall that our example program runs on a uniprocessor. Processor sharing complicates the specification of a task’s timing constraints—if a preemptible code region takes at most 3 seconds to execute, then the task runs for at most 3 seconds 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 seconds. 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 Figure 3. We label individual formulas for convenience in referring to them below. 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 and a formula that must hold invariantly (at all times in an interval) is left justified below a henceforth operator (\( \Box \)). 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, if any, 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, if the tracking task eventually reaches either Fast or Slow, then it must do...
so within $L_1$ to $U_1$ microseconds (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.

The formula Run2 describes an invariant property. It specifies that the tracking task always reaches either Fast or Slow within $L_2$ to $U_2$ μsec (local time) after reaching Result, unless one or both of the searches fail. The weak searches in Run1 and Run2 allow for the possibility of deadlock. 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. 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 FModel1 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 not processed in slow 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 (V) in FMode2. The searches in the antecedents of the implications are strong, since the time from a Fast to the next Result depends on the events in the antecedent actually occurring within the specified time bounds. The bi-directional search arrow in FMode1 further restricts the event located by the first search to be the last Slow that precedes a Fast, and in FMode2 it restricts the event located by the first search to be the last Fast or Slow that precedes a Fast. 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 Figure 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 [1, 2]. 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 a hybrid automaton representing the program using the method of [8]. Each control location in this 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.

Figure 5 shows a slightly simplified part of the hybrid automaton constructed from the (partial) program in Figure 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.

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

---

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.
The discussed below.

transition must be taken before more than transition from location 4, which represents the code region that be derived from the code represented by the transitions, as cancels the entry call on La, and

Variable of time spent in the location). After

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 \(L_a\) \(\mu\)sec, \(x_s\) is at least \(L_a\), so the QueueData transition may be taken; the transition must be taken before more than \(U_a\) \(\mu\)sec 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 \(\mu\)sec 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 \(y = 0\) and \(now = 1\) in all locations. The constant \(\Delta\) 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.

Figure 5: Part of Hybrid Automaton for Program

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 \(L_a\) \(\mu\)sec, \(x_s\) is at least \(L_a\), so the QueueData transition may be taken; the transition must be taken before more than \(U_a\) \(\mu\)sec 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 \(\mu\)sec 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 \(y = 0\) and \(now = 1\) in all locations. The constant \(\Delta\) 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.

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 automata generated from the program and 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 Figure 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 a hybrid automaton in Figure 6. Since the formula specifies a local timing constraint for the Tracking task, we use variables \(x_{t_1}\) and \(x_{t_2}\) that advance in locations where the Tracking task is running (\(x_{t_1}\) is used to enforce the upper bound, \(x_{t_2}\) is used to enforce the lower bound). Each local timing constraint for the

Figure 6: Hybrid Automaton from Formula SMode in Figure 3

The hybrid automaton generated from Figure 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 Figure 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 which was given earlier into a hybrid automaton and intersect this automaton with the automaton in Figure 5. (Intersection is performed using the standard product operator for hybrid automata [1] 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.

The hybrid automaton generated from Figure 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 Figure 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 which was given earlier into a hybrid automaton and intersect this automaton with the automaton in Figure 5. (Intersection is performed using the standard product operator for hybrid automata [1] 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 automata generated from the program and 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 Figure 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 Figure 6. Since the formula specifies a local timing constraint for the Tracking task, we use variables \(x_{t_1}\) and \(x_{t_2}\) that advance in locations where the Tracking task is running (\(x_{t_1}\) is used to enforce the upper bound, \(x_{t_2}\) is used to enforce the lower bound). Each local timing constraint for the
Tracking task uses its own variable (e.g., $x_{t_1}, x_{t_2}, \ldots$). Intersecting the automaton in Figure 6 with the automaton in Figure 5 effectively combines their timing constraints: the transition on Slow now resets $x_{t_1}$ and $x_{t_2}$, 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 $x_{t_2} \geq L_3$ is conjoined to the guard of the Result transitions.

The ability to stop clocks allows a simple representation of timing constraints in the presence of preemption. For example, if code region Result is interrupted in location 1 by the preemption of the Sensor task, then when the Tracking task resumes the execution of Result in location 3, the variables $x_{t_1}$ and $x_{t_2}$ 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 preempted many times (represented by the cycle through locations 1–4) before it accumulates enough CPU time (recorded in $x_{t_2}$, which is compared with the lower bound) to produce a result.

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 $m = 25$.

**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 Figure 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 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 [1] 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 $P$.

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 [13] 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 Figure 8 from the negation of the GIL formula NeverExpire in Figure 7. 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.

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 Figure 8, which was constructed from the negation of the formula NeverTwoFast in Figure 7. (The $\Diamond$ is the standard "eventually" operator of temporal logic, so this specification asserts that a Slow event occurs between any two occurrences 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

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

**Table 1: Durations Used in Example (microseconds)**
to modify the program to avoid it.

We can also verify 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 Figure 7, whose negation can be converted to the hybrid automaton ViolMaxResponse in Figure 8. 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.

In the analyses described above, we used the model building tool for Ada described in [8] to construct the hybrid automaton representing the program from an Ada-like specification 2 of the code in Figure 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 [16]. The automata for the GIL formulas are constructed by hand using an extension of the algorithm in [11]. These automata are fed to HyTech, which takes their product and computes the set of reachable states. The performance of HyTech (version 1.02b) 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.

Figure 7: GIL Specifications of Properties

Expire:

TwoFast:

ViolMaxResponse:

Figure 8: Automata for Properties

Table 2: Analysis Times for Example (seconds)

<table>
<thead>
<tr>
<th>Property</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expire</td>
<td>55</td>
</tr>
<tr>
<td>TwoFast</td>
<td>70</td>
</tr>
<tr>
<td>MaxResponse</td>
<td>94</td>
</tr>
</tbody>
</table>

Figure 9: Behavior Illustrating Bound on Time From Sensor Interrupt to Result (time in units of 25 μsec)

DETAILS

The approach presented here combines two separate bodies of work: Corbett’s method for constructing timed models of Ada's constructs, but in a Lisp syntax to facilitate parsing. For example, we write (call T E) rather than T.E.

2Our specification language contains (a subset of) Ada's constructs, but in a Lisp syntax to facilitate parsing. For example, we write (call T E) rather than T.E.
Ada to Hybrid Automata
In order to construct models of partially implemented programs, we modified the translation of Ada to hybrid automata described in [8] to allow stub tasks. A stub task is generated from GIL formulas to specify the local timing constraints of the stub task.

Ada tasking programs [8] and Dillon et al.’s method for using Graphical Interval Logic to construct test oracles [10,11]. Although space limitations preclude presentation of the complete details of our approach, in this section we sketch the modifications and extensions that were necessary to combine the two methods. A full presentation of our approach is in preparation and will be published separately.

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 Dillon and Yu’s method for producing test oracles 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 [10] 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 have illustrated our method by analyzing a small, but
nontrivial, example. Our method for analyzing partially-implemented systems is itself only partially-implemented at this time; we are still in the process of automating our algorithm for generating hybrid automata from GIL specifications. When that is completed, we will be able to apply the method to a variety of larger examples and get a better idea of its practical performance. A paper describing the full details of our approach and its application to some of these larger examples is in preparation.

REFERENCES


