Run-Time Monitoring of Real-Time Systems

Sarah E. Chodrow

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712

Farnam Jahanian
Marc Donner

IBM T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Abstract

This paper presents a model and an implementation of a run-time environment for specifying and monitoring properties of real-time systems. The proposed approach supports annotating real-time programs with events that are recorded and examined at run-time. It provides two general methods for synchronous or asynchronous monitoring of real-time constraints. In the synchronous case, a system constraint is embedded inside a program; thus the constraint is examined at a particular point in the execution of a real-time task. In the asynchronous case, a constraint is monitored by a separate task during the entire execution of real-time tasks.

1 Introduction

In designing real-time systems, we often make assumptions about the behavior of the system and its environment. These assumptions take many forms: upper bounds on interprocess communication delay, deadlines on the execution of tasks, or minimum separations between occurrences of two events. They are often made to deal with the unpredictability of the external environment or to simplify a problem that is otherwise intractable or very hard to solve. Such assumptions may be expressed as part of the formal specification of the system or as scheduling requirements on the real-time tasks. Despite the contribution of formal verification methods and recent real-time scheduling results, the need to perform run-time monitoring of these systems is not diminished for several reasons: the execution environment of most systems is imperfect and the interaction with the external world introduces additional unpredictability; design assumptions can be violated at run-time due to unexpected conditions such as transient overload; application of formal techniques or scheduling algorithms in turn requires assumptions about the underlying system; and it may be infeasible (or impossible) to verify formally some properties at design time, thus further necessitating run-time checks.

This paper presents a model, based on [6] [1], for formal specification and monitoring of run-time constraints for time-critical systems. The objective is to specify complex system constraints including timing requirements and to provide a general frame work for monitoring time-critical systems at run-time. One can envision a system in which run-time monitoring can provide feedback so that the system can adapt to a changing environment or an exception condition. In particular, the information collected by the monitoring facility can be used to detect a violation of system constraints or to manage resources at run-time. The paper also describes an implementation of a run-time monitoring toolkit as a set of library function calls in C. The current implementation on the IBM RS/6000 workstations running AIXv.3 [1] takes advantage of several real-time features of the operating system. The underlying model views a system computation as a sequence of event occurrences. The observable events in a real-time system are specified by annotating real-time programs with those events that are to be monitored at run-time. Examples of these events include start or completion of a program segment, and an assignment to a state variable. A system constraint can be viewed as an assertion on the relationship between the occurrences of these observable events. The proposed approach distinguishes between a system constraint that is embedded in a real-time program and a constraint that is monitored asynchronously by a separate task. Our prototype implementation allows the specification and monitoring of both types of system constraints. In particular, the implementation supports annotat-

[1] RS/6000 and AIX are trademarks of IBM Corporation.

0-8186-2450-7/91 $1.00©1991 IEEE

ing C programs with events and specifying system constraints. It also provides the mechanisms for asynchronous or synchronous monitoring of those constraints at run-time.

Despite extensive work on monitoring and debugging facilities for parallel and distributed systems, run-time monitoring of real-time systems has received little attention with a few exceptions. Special hardware support for collecting run-time data in real-time applications has been considered in a number of recent works [4] [10]. These approaches introduce specialized co-processors for the collection and analysis of run-time information. A related work studies the use of monitoring information to aid in scheduling task in a real-time environment[3]. The underutilization of a CPU due to the use of scheduling methods based on the worst-case execution times of tasks is addressed by the use of a hardware real-time *monitor* which measures the task execution times and delays due to resource sharing. The monitored information is fed back to the operating system for achieving an adaptive behavior. A work closer to the approach in this paper is a system for collection and analysis of distributed/parallel (real-time) programs [8]. The work is based on an earlier system for exploring the use of an extended E-R model for the specification and the access to monitoring information at run-time [9]. The assumption is that the relational model is an appropriate formalism for structuring the information generated by a distributed system. [11] presents a real-time monitor that is developed for the ARTS distributed operating system. The proposed monitor requires certain support from the kernel. In particular, the ARTS kernel records certain events that are seen by the operating system as the state changes of a process, e.g., waking-up, being scheduled. These events are sent periodically by the local host to a remote host for displaying the execution history. The invasiveness of the monitoring facility is included in the schedulability analysis.

The organization of this paper is as follows. Section 2 describes two main classes of events and an annotation system for specifying observable events in a system execution. Section 3 presents two general methods for specifying assertions that describe the system constraints to be monitored at run-time. The underlying model and the corresponding implementation is described for each method. Section 4 describes the implementation of a satisfiability checker which is invoked by the monitoring facility. The satisfiability checker detects the violations of a certain class of system constraints as specified by the programmer. Section 5 discusses some of the operating system dependent issues of the implementation. The last section contains the concluding remarks and the future direction of this work.

2 Events

The run-time monitor model in this paper is based on the model proposed in [1]. A computation of a real-time system can be viewed as a sequence of *event occurrences*. Informally, events represent things that happen in a system. An event occurrence defines a point in time in a computation at which a particular instance of an event happens. Thus, timing properties can be expressed as relationships among event occurrences in a computation.

We distinguish between two classes of observable events in this model: *label events* and *transition events*. Label events are used to denote the initiation and completion of a sequence of program statements. (They correspond to start/stop events in RTL.) They are defined by inserting labels in appropriate places in the code. In figure 1, two

```
..
E1 ->
S;
<-E2
..
```

Figure 1: Code fragment with event labels.

event labels are defined, E1 and E2. The right-pointing-arrow is a syntactic marker that specifies E1 as the event that denotes the start of statement S. The left-pointing-arrow associates event E2 with the end of statement S. Two events may be placed between a pair of statements, one bound to the end of the first statement and the other bound to the beginning of the second. This can be used to make preemption between two consecutive statements observable in an execution.

Transition events capture assignments of values to a particular type of variable referred to as a *watchable variable*. The term "watchable variable" is borrowed from the ORE language [2]. An assignment to a watchable variable denotes a potential change in a system state and it is observable as an event occurrence when monitoring a system. The state of a program execution can be characterized by the values of of its program counter and its state variables. The state changes in a computation that must be monitored can be captured by the two types of events in the proposed model, label and transition events. Hence, run-time monitoring of a real-time program is achieved by examining the observable events at run-time.

Run-time monitoring of a system requires recording of earlier event occurrence in a system computation. If it is sufficient to remember only the last occurrence of each event, then a bound can be imposed a priori on the the number of event occurrences that must be kept at any given time. Furthermore, the algorithms for recording and discarding the event occurrences in a system computation are very simple. However, since examining a property at run-time may involve multiple occurrences of the same event, it may be necessary to remember more than one occurrence of an event to detect the violation of a timing property. We provide *event histories* that store the times (and values, for transition events) of a number of previous occurrences. Although one can attempt to keep the entire history for each event, it is impractical to do so in most nontrivial systems. As described in the next section, the size of the event history for each event is either specified by the system designer or is determined by examining the assertion to be monitored at run-time.

We also provide two RTL-like [7] functions for accessing the event histories: the occurrence function $@(e, i)$ which returns the time of the ith occurrence of event e, and $@val(v, i)$, which returns the value of the ith occurrence of watchable variable v's transition event. A positive occurrence index is absolute with respect to the beginning of the computation sequence. $@(e, 5)$ refers to the 5th occurrence of event e. When the index i is negative, it refers to the ith most recent occurrence of the event in a computation. For example, $@(e, -1)$ denotes the time of the most recent occurrence of e. An occurrence index of 0 is undefined. An additional function, $@index(e, i)$, returns the absolute index of an occurrence of event e, given an index i relative to the beginning or end of the sequence.

3 Embedded and Monitored Timing Constraints

We have constructed a run-time environment that supports two methods of expressing timing properties: *embedded* constraints and *monitored* constraints. With embedded constraints, a programmer can actively check for the satisfiability of a timing property at particular points in the execution of a program, and modify the computation accordingly. An example of an embedded constraint is an acceptance test in a recovery block. The primary advantage of this approach is that it permits the programmer to manipulate the @ functions and to access event histories directly, through C constructs. The constraint applies only when it is checked, and at no other time in the execution sequence. For example, suppose `temp` is a watchable variable denoting the temperature reading from a sensor. Each assignment to `temp` is a new instance of the corresponding event. The code segment in figure 2 specifies an assertion which requires consecutive readings of new temperature values to be within a specific tolerance.

```
temp = read_sensor();
if(@val(temp,-1) - @val(temp,-2) > 200) {
  shutdown_reactor();
}
else {
  raise_rods();
}
```

Figure 2: An embedded constraint.

A complementary approach to embedded constraints is to make the timing specification independent of the program. A separate monitoring process runs concurrently with the real-time application tasks and checks the satisfiability of the constraints. The constraints are enforced at all times in the execution of the program. This approach serves two purposes: it separates the timing concerns from the functional specification of the program; and it allows the expression of deadline and delay properties that cannot be checked at a specific point in the execution sequence. An exception that must be raised when a task misses a deadline is an example of a monitored constraint. The formula in figure 3 illustrates a deadline constraint which requires every send message to be acknowledged by a corresponding message within 5ms.

$$@(send, i) \leq @(ack, i) \wedge$$
$$@(ack, i) \leq @(send, i) + 5$$

Figure 3: A deadline constraint.

Below we elaborate on the two approaches to handling system constraints. The first section discusses the model and implementation of embedded constraints. The second section discusses monitored constraints. In both models, the following assumptions hold: (1) Two occurrences of the same event cannot happen simultaneously; (2) Event names are unique across the system of tasks. The same name in two different tasks refers to the same event; and (3) There exists a single monotonically increasing clock, accessible by all tasks.

3.1 Embedded Constraints

Under the model of embedded constraints, timing properties are enforced at a particular point in the execution sequence, providing *synchronous* monitoring of constraints. This is done by allowing direct manipulation of the @ functions and occurrence indices through C constructs.

3.1.1 Model

For embedded constraints, our model consists of a set of application tasks and shared event histories (see figure 4). The model provides for communicating the event occurrences among tasks through the histories. Event occurrences are recorded by tasks in the shared history. The satisfiability of an embedded constraint is tested by retrieving the appropriate values from the history for the corresponding event occurrences.

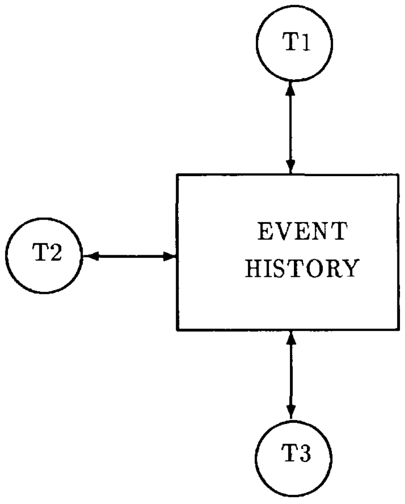

Figure 4: Embedded model.

Each event data structure consists of a finite length circular queue of times (and values), with a relative index pointing to the most recent event. An absolute index counts the number of occurrences of the event. The name field identifies each event uniquely in the system of tasks. (See figure 5.)

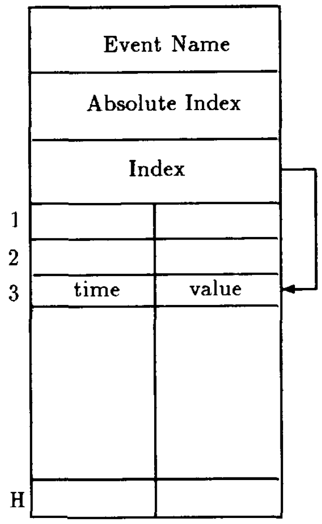

Figure 5: Event history data structure.

During program execution, application tasks write the event histories by noting event occurrences (both label events and watchable variable assignments), and read event histories by using the @ and *@val* functions to retrieve the time and value of an event occurrence. @ and *@val* map relative and absolute indices into a location in the event history, and return the requested time or value, or an error, since an expired or non-existent event can be detected easily.

3.1.2 Implementation

Event occurrence annotations are expanded to a code fragment that locks the event history, obtains a timestamp for the event, inserts the time (and value) of the occurrence into the history, and then unlocks the event history. Appropriate operating system support for atomically obtaining a timestamp with an acceptable granularity is explored in section 5. The locking of an event history during an event occurrence forces additional synchrony on tasks, as it may cause other tasks to block while attempting to write to the same history. However, the recording of event occurrences is very quick, so a contention for the lock on an event history is very unlikely. Furthermore, the programmer can redefine the lock and unlock utilities if he chooses to trade the guaranteed accuracy of the history for speed. This will be discussed in more detail in section 5. There are no locks encoded into the read accesses to the histories: the decision of whether to use read locks is left to the programmer.

The application tasks are AIX processes. Event histories are kept in file-mapped shared memory. An initialization process allocates the shared memory and a semaphore to protect it, then divides it into history structures for each event and exits. The type and history length for each event in the system are read at compile time from a file provided by the programmer.

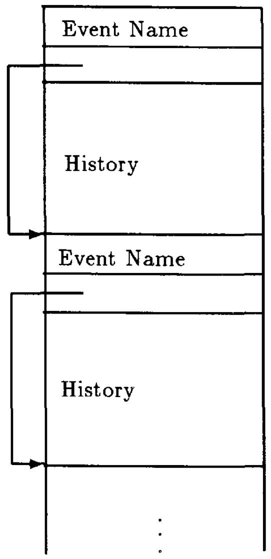

Figure 6: Shared history data structure.

Each history structure in shared memory contains the string corresponding to the event name; a circular queue to store the time and value data; a pointer to the most recent event in the queue (the relative index); the absolute index within the computation of the most recent event; and a pointer to the beginning of the next history. This last pointer is used as part of the task initialization process. (Figure 6 illustrates how histories for different events are chained together.)

A long constraint embedded in a task can include multiple references to the same event. However, the meaning of a relative index may change during the evaluation of the constraint, due to the occurrence of the same event in a concurrent tasks. For this reason the *@index* function is provided, so that the programmer may fix the absolute indices to which she refers before checking the constraint.

3.2 Monitored Constraints

Under this model, timing properties may be enforced during the entire execution, providing *asynchronous* monitoring of the constraints. Thus the timing information can be separated from the function specification of the program. Asynchronous monitoring is required to correctly implement deadline and delay constraints. One may attempt to enforce the property in figure 3 by inserting the condition to be checked at a specific point in the execution of a task, after an acknowledgement is received, for example. However, there are two potential problems. If the acknowledgement is not received within the required deadline, the violation is not detected until after the acknowledgement. If the acknowledgement is never received, due to a failure, the violation of the property may not be detected at all. Hence, the constraint may be viewed as a property to be enforced when a send occurs, until either an acknowledgement is received, or 5 time units have passed. The latter test can be triggered by a timer interrupt associated with the send event.

3.2.1 Model

For monitored constraints, our model consists of a set of application tasks (*tasks*) generating events and a monitoring task (the *monitor*). A queue provides interprocess communication between each task and the monitor. Event histories are local to the monitor rather than shared among all tasks. (See figure 7.) During task execution, as events occur, they are sent to the monitor. The monitor processes each event occurrence and records it in a local repository. As events are recorded, the monitor checks the satisfiability of the corresponding formulas. Unlike the embedded case, there is no need to lock event histories, because there is a separate queue between each task and the monitor. The monitor maintains the event histories for the tasks, and arbitrates simultaneous occurrences.

3.2.2 Implementation

Initializing tasks that utilize monitored constraints is somewhat more involved than in the embedded

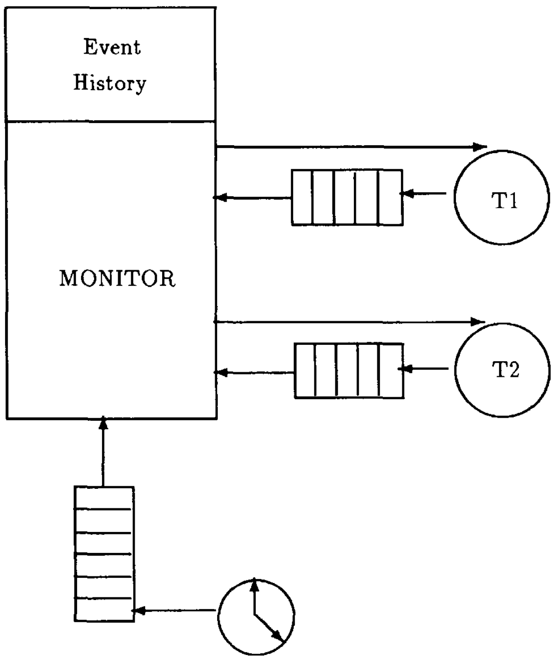

Figure 7: Monitored model.

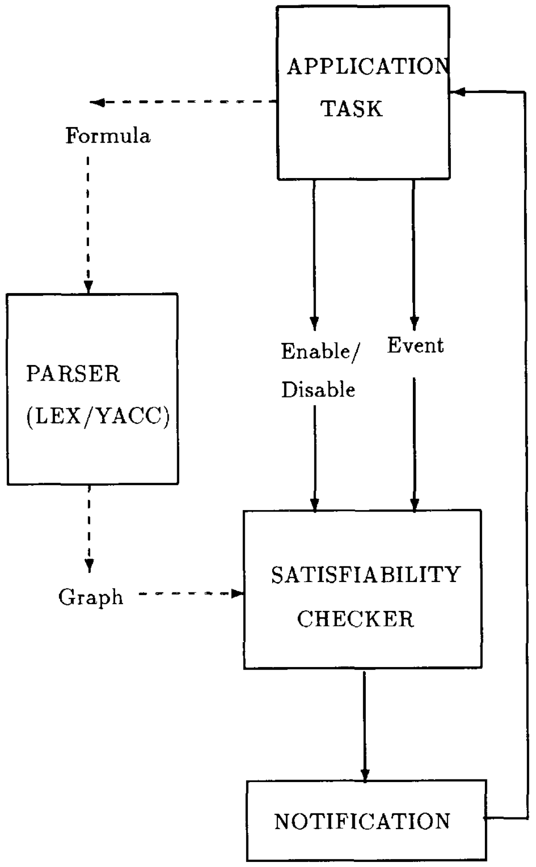

Figure 8: Monitored implementation.

case. (See figure 8.) Each task registers with the monitor to set up an interprocess communication queue. During execution, the task sends the monitor event occurrences and requests to initiate or cancel the monitoring of given constraints. If a constraint monitored for the task is violated, the task is notified by a software signal and a message in its queue from the monitor. The programmer can modify the signal handler to respond to failures as she wishes; the default behavior is to terminate the process on failure.

Under the monitored model, a shared count field for each event determines whether an occurrence is enqueued on the process queue to the monitor. The value of the field corresponds to the number of formulas being monitored that involve the event.

The monitor can be divided into two phases. In the initialization phase, the monitor uses the event declaration file to create event histories in its own memory. Pointers to the histories are stored in a hash table, keyed by the event name. The timing constraints to be monitored, also specified in a file, are parsed into an internal representation that is also hashed, keyed by the string corresponding to the formula. After the formulas have been parsed, it is possible to determine which formulas are associated with which events, and to infer the lengths of the event histories, which are then allocated. Before entering the second, operational phase, the monitor sets up an area of shared memory through which the application tasks can request a process queue, and creates a "timer queue" to store timer interrupt events.

The second phase consists of an infinite loop. If there is a queue allocation request from a *new* task, then the monitor allocates a piece of shared memory for the task queue, returns a pointer to the new queue to the requesting task, and adds the queue to its list. The monitor then cycles through its list of queues (including the timer queue), and selects the entry with the earliest timestamp. If the entry is an event occurrence, then the time and data values are added to the corresponding history. The event occurrence may cause a constraint to be checked at some future time. If so, then a timer is set. All the formulas associated with that event are then checked for satisfiability. In case of a constraint violation, the monitor notifies each task that monitored the constraint by a signal and a message in its process queue. If the entry is a request to monitor or unmonitor a formula, then the list of processes interested in the formula is modified accordingly, as are the count fields of the events in the formula. The entry may be a timer interrupt associated with an event. In this case, the corresponding formulas are sent to the satisfiability checker, but no further information is added to the event history.

It is possible that a task may write to its process queue (which is of finite length) faster than the monitor is able to remove entries, resulting in a queue overflow. The overflow is detected by the runtime library, and the task is notified. The method of handling the error is left up to the programmer.

4 A Satisfiability Checker

In the previous section, we described a monitoring task that examines events received from application tasks, and determines whether the given constraints have been violated. However, we did not discuss how the constraints are specified. [1] defines a specification language based on RTL. In particular, it introduces three expressive classes of (RTL) properties that meet the requirements of the system as stated in section 2: bounded history lengths can be derived from constraint formulas, and satisfiability can be checked in polynomial time. In our prototype, we implemented the first class of properties.

This class consists of properties (expressed as RTL-like formulas) that explicitly identify the occurrence index of each event in the formula relative to the first or last occurrence of the event in a computation prefix. The RTL formulas in this class allow @ functions of the form:

$$@(e, a), \text{ or} \\ @(e, -a)$$

where e is an event and a is a positive integer constant. In particular, if the occurrence index of an @ function is a positive integer constant, the @ function is interpreted as defined in section 3. For example, $@(SIGNAL, 1)$ refers to the time of the first occurrence of event $SIGNAL$. Alternatively, the occurrence index can refer to a particular most recent occurrence of an event. We use the notation $@(e, -a)$ to denote the a^{th} most recent occurrence of event e. For example, $@(SIGNAL, -1)$ is the time of the last occurrence of event $SIGNAL$.

Example: Consider a system that executes the action $RESPONSE$ when a $SIGNAL$ event occurs. An exception condition must be raised if another $SIGNAL$ occurs during the execution of a $RESPONSE$ action:

$$\begin{aligned} &@(\uparrow RESPONSE, -1) \leq @(SIGNAL, -1) \Rightarrow \\ &\quad @(\uparrow RESPONSE, -1) \leq @(\downarrow RESPONSE, -1) \wedge \\ &\quad @(\downarrow RESPONSE, -1) < @(SIGNAL, -1) \end{aligned}$$

The above formula specifies that if the last occurrence of action $RESPONSE$ starts before the last occurrence of a $SIGNAL$ event, the action must also complete before that $SIGNAL$ event.

To simplify the task of parsing the language, the prototype accepts input constraint formulas in disjunctive normal form. This does not limit the expressibility of the language, although it places a larger burden on the programmer. The formulas of this class consist of a disjunct of conjuncts of predicates, where each predicate is an inequality of the form $@(e, \pm i) \leq @(f, \pm j) + C$, where i and j are positive integer constants, and C is an integer constant. C corresponds to an offset time value–a delay or deadline. The history length of each event is bounded by the absolute value of the largest negative index associated with the event in all constraints, plus the number of explicitly defined positive indices. The disjunctive normal form of the formula in the preceding example is shown below:

$$\begin{aligned} &@(SIGNAL, -1) + 1 \leq @(\uparrow RESPONSE, -1) \vee \\ &\quad [@(\uparrow RESPONSE, -1) \leq @(\downarrow RESPONSE, -1) \wedge \\ &\quad @(\downarrow RESPONSE, -1) + 1 \leq @(SIGNAL, -1)] \end{aligned}$$

The scanner and parser for the language were constructed using the UNIX facilities LEX and YACC. A formula string is translated by YACC into a parse tree whose leaves are the inequalities, whose root is $\vee$, and whose internal nodes are $\wedge$'s. The parse tree for the example above is given in figure 9. Each conjunct subtree can be transformed into an equivalent weighted directed graph. (See figure 10.) Each unique occurrence function call corresponds to a node on the graph. The predicate $@(SIGNAL, -1) \leq @(\uparrow RESPONSE, -1) - 1$ becomes an edge $@(\uparrow RESPONSE, -1) \xrightarrow{-1} @(SIGNAL, -1)$. A predicate of the form $@(SIGNAL, 1) \leq C$ where C is an absolute time value is translated to an edge $\mathbf{0} \xrightarrow{C} @(SIGNAL, 1)$ where $\mathbf{0}$ is a special "zero node" designed to take care of constants. Similarly, a

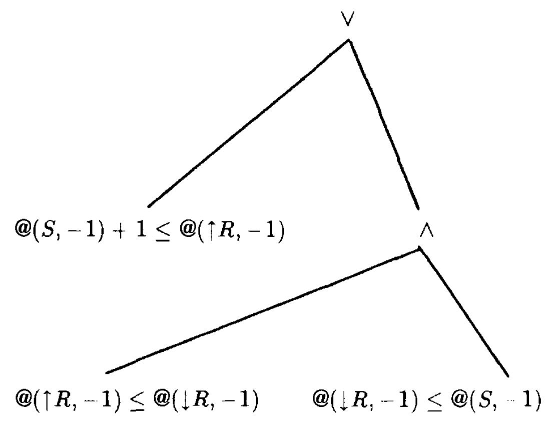
∨
@$(S,-1)+1 \leq$ @$(\uparrow R,-1)$
∧
@$(\uparrow R,-1) \leq$ @$(\downarrow R,-1)$
@$(\downarrow R,-1) \leq$ @$(S,-1)$

Figure 9: Parse tree for example.

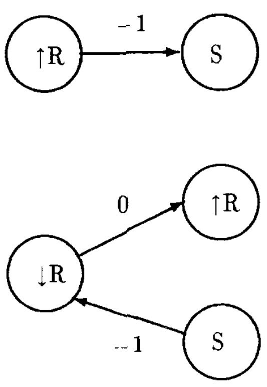

Figure 10: Uninstantiated constraint graph.

predicate of the form $C \leq$ @$(SIGNAL,1)$ is represented by an edge @$(SIGNAL,1) \xrightarrow{-C} \mathbf{0}$.

When the satisfiability checker is invoked on a constraint, each conjunct graph is instantiated from the current event histories. If a negative cycle is found in the instantiated graph, then the conjunct is unsatisfiable. The (DNF) constraint has been violated if all its graphs are not satisfied.

Graph instantiation is based on the event histories. Every edge with weight w incoming to a node, to which a time value t has been assigned, is replaced with an edge with weight $w-t$ incident on the "zero node." Conversely, every edge with weight w outgoing from a node, to which a time value t has been assigned, is replaced with an edge with weight $w+t$ outgoing from the "zero node." Furthermore, for each node that has no time value associated with it, i.e., the corresponding event has not happened yet, an edge from that node to the "zero node" is added with the weight $-NOW$ where NOW is the current time. This edge denotes that the corresponding event may happen at or after time NOW.

In figures 11-14, we present an example of an erroneous event sequence leading to constraint violation. Figure 11 shows an execution prefix over which we test the constraint formula given above. At time 1 (figure 12), no cycles exist in either graph, so the constraint is satisfiable. At time 2 (figure 13), a negative cycle is found in the second conjunct. Since the first conjunct has a valid instantiation, the formula remains satisfiable. At time 3 (figure 14), a $SIGNAL$ event occurs before the end of the $RESPONSE$ action. This violates both conjuncts and causes a negative cycle to appear in both graphs. Observe that since $\downarrow RESPONSE$ has not occurred before time 3, the corresponding node in the graph has no value associated with it. Hence, a dashed edge from the node $\downarrow RESPONSE$ to "node zero" with the value -3 is added during instantiation, as shown in figure 14. (The constraint remains unsatisfiable at time 4, after the $\downarrow RESPONSE$ event occurs.)

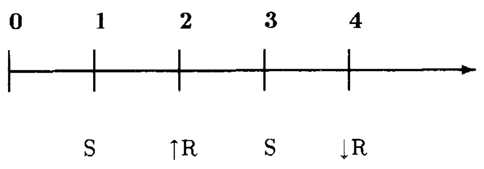

Figure 11: Execution prefix.

The Floyd-Warshall shortest path algorithm is used to search for negative cycles on the instantiated graph: if such a cycle is found, the conjunct is unsatisfiable. The complexity of the negative cycle detection algorithm is $O(n^3)$ in the number of nodes in the graph. In many cases, however, it is not necessary to call the algorithm because a one-node negative cycle is detected on the "zero node" during instantiation.

5 OS Dependent Features

The implementation of our system hides the operating system dependent features in a layer which sits between the operating system and the monitoring facility. The current implementation on RS/6000 running AIX v.3 imposes certain conditions on the run-time environment. The most intrusive of these was the lack of support for threads in the native operating system. As a result, we were forced to use heavyweight processes to implement tasks, and file-mapped shared memory to maintain event histories. This added certain complications which lightweight processes with shared address space would have avoided. We defined two special calls, lock and unlock, to hide the operating system dependencies in providing mutual exclusive access to shared data structures. We guarantee the atomicity of history updates by using semaphores in implementing the lock function. However, an alternative solution is to use the *compare and swap* function in AIX v3. This second solution is more efficient because it does not require a system call.

Since run-time monitoring of a system involves examining sequences of time stamped events, granularity and atomicity of event timestamps may require special support from the run-time system. The granularity of event timestamps directly affects the satisfiability checks as a new event or a timer interrupt occurs. Suppose reading the system clock takes 30 microseconds (not an unreasonable assumption if a system call is required), then the ordering of events in two processes may be indeterminate if the timestamps are closer than 30 microseconds. A similar problem arises if the clock being read is updated infrequently, e.g., updates to the clock may occur at 60Hz intervals. In many operating systems, the system clock is accessible to a program through system calls. Fortunately, the RS/6000 provides very fine granularity, and using the POSIX system calls, can differentiate between events that occur tens of microseconds apart [5]. An alternative solution is to obtain the timestamps by examing the hardware registers directly. Certain processors support special hardware clocks that can be examined by a process without the overhead of a system call. This allows a much finer granularity. The hardware clock registers of the RS/6000 can be read directly providing a granularity of 100s of nanoseconds.

6 Concluding Remarks

This paper presented a model and an implementation of a run-time environment for specifying and monitoring timing properties of real-time systems. The run-time environment supports two general methods for synchronous or asynchronous monitoring of real-time constraints. In the synchronous case, a system constraint is embedded inside a program; the constraint is examined at a particular point in the execution of the program. In the asynchronous case, a constraint is expected to be monitored by a separate task during the entire execution of the program.

There are several directions for extending this work. In the short term, several enhancements to the toolkit can be made. We would like to add a front-end to the library to allow programmers to code with the RTL-like notation for the @ functions and label events. [6] identifies two additional classes of RTL formulas with polynomial satisfiability algorithms and inferrable finite history lengths. Implementing the parsers and satisfiability checkers for these classes would allow us to experiment further with the system. It is also desirable to incorporate the proposed run-time monitoring system into the ORE run-time environment. This provides a powerful mechanism for expressing temporal assertions in ORE.

For the long term, several research directions require further investigation. The first is that of adaptive scheduling. It should be possible to use the timing information generated by the tasks to determine whether a task will meet its deadline or other constraints. Given such timing information, the scheduler can choose to give an at-risk task higher priority, or terminate it gracefully. A related problem involves integrating run-time monitoring with scheduling real-time tasks. The requirement for monitoring system constraints and taking appropriate actions when a violation is detected can intrude upon normal activities of a system. Quantifying the intrusiveness of run-time monitoring on other tasks requires further exploration.

The second research direction is that of monitoring a distributed set of application tasks running on distinct processors. Even when clocks on different processors are synchronized within a specific bound, the model must take into account deviations between clocks and message delays in exchanging timing information on events on different nodes. Another possible direction is to investigate provisions for an extensible monitor. One can envision an environment in which an application task specifies a user-defined function that is invoked by the run-time monitor in response to an event occurrence. Arbitrary constraints can be specified as user-defined functions that are invoked at run-time to examine event histories and to detect violations of these constraints as new events occur.

References

[1] M. Donner and F. Jahanian. Rtl meets ore. In *7th IEEE Workshop on Real-Time Operating Systems and Software*, pages 55–61, May 1990.

[2] Marc D. Donner and David H. Jameson. Language and operating system features for real-time programming. *Computing Systems*, 1(1):33–62, 1988.

[3] D. Haban and K. Shin. Application of real-time monitoring to scheduling tasks with random execution times. In *Proceedings of Real-Time Systems Symposium*, pages 172–181, December 1989.

[4] D. Haban and D. Wybranietz. A hybrid monitor for behavior and performance analysis of distributed systems. *IEEE TSE*, 16(2):197–211, Feb. 1990.

[5] IBM Corporation. IBM AIX version 3.1 RISC system/6000 as a real-time system. Technical report, International Technical Support Center, 1991.

[6] F. Jahanian and A. Goyal. A formalism for monitoring real-time constraints at run-time. In *Proc. of Fault-Tolerant Computing Symposium (FTCS-20)*, June 1990.

[7] Farnam Jahanian and Aloysius Ka-Lau Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9), 1986.

[8] C. Kilpatrick, K. Schwan, and D. Ogle. Using languages for capture, analysis and display of performace information for parallel or distributed application. In *International Conference on Comp. Language '90*, March 1990.

[9] R. Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems*, 6(2):157–196, May 1988.

[10] J.P. Tasi, K-Y Fang, and H-Y Chen. A noninvasive architecture to monitor real-time distributed systems. *Computer*, 23(3):11–23, March 1990.

[11] H. Tokuda, M. Koreta, and C.W. Mercer. A real-time monitor for a distributed real-time o.s. *ACM SIGPlan Notices*, 24(1):68–77, January 1989.

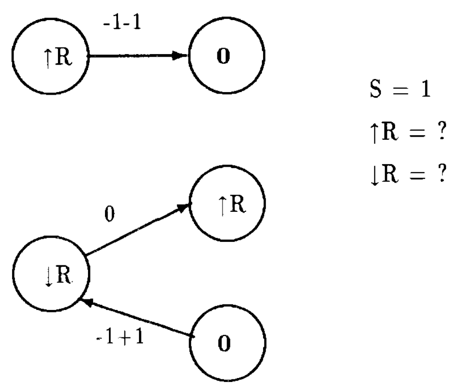

Figure 12: Graph instantiation, time = 1.

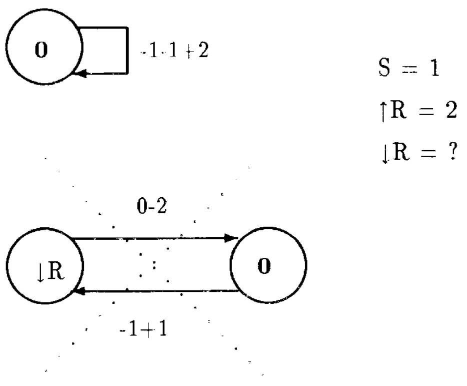

Figure 13: Graph instantiation, time = 2.

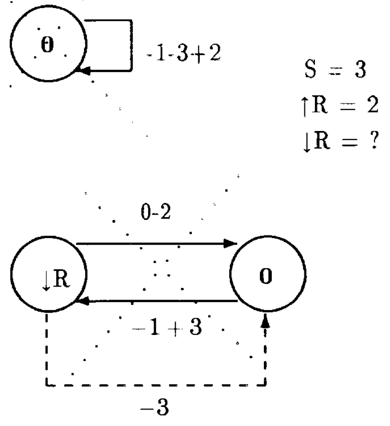

Figure 14: Graph instantiation, time = 3.