Efficient Run-Time Monitoring of Timing Constraints *

Aloysius K. Mok and Guangtian Liu

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712
E-mail: {mok,liugt}@cs.utexas.edu

Abstract

A real-time system operates under timing constraints which it may be unable to meet under some circumstances. The criticality of a timing constraint determines how a system is to react when a timing failure happens. For critical timing constraints, a timing failure should be detected as soon as possible. However, early detection of timing failures requires more resource usage which may be deemed excessive. While work in real-time system monitoring has progressed in recent years, the issue of tradeoff between detection latency and resource overhead has not been adequately considered. This paper presents an approach for monitoring timing constraints in real-time systems which is based on a simple and expressive specification method for defining the timing constraints to be monitored. Efficient algorithms are developed to catch violations of timing constraints at the earliest possible time. These algorithms have been implemented in a tool called JRTM (Java Run-time Timing-constraint Monitor) in the language Java. This tool can be used to specify and monitor timing constraints of Java applications.

1. Introduction

Real-time systems are systems that operate under timing constraints, such as responding to an external signal within a specified deadline or performing some tasks repeatedly at specific rates. While extensive research efforts have been devoted to scheduling system resources to guarantee the satisfaction of timing constraints, violations may still occur owing to unexpected behavior of the external environment or errors in system specification or implementation. Hence, it is essential to have a monitoring facility to catch timing constraint violations at run time and perform appropriate recovery actions when possible. Another use of a timing constraint monitor is in debugging the timing behavior of real-time systems during development.

Typically a timing constraint monitor collects system timing information at run time and checks them against the specified timing constraints to be monitored. If a violation is found, certain predefined actions may be triggered such as notifying the user of the violation. In general, three goals need to to be achieved for such a monitoring system:

- Transparent monitoring: To reduce the impact of monitoring on the target system (i.e., the system to be monitored), it is usually desirable to separate the monitor from the target system so that users only need to specify what to monitor rather than having the system programmers to insert extra code into the target system programs.

- Bounded violation detection latency: In many real-time systems, especially hard real-time systems, it is critical to take proper actions within a certain time once a violation has occurred. To achieve this goal, the monitor need to guarantee bounded delays on collecting timing information and checking them against the constraints.

- Minimum monitoring overhead: The monitoring overhead, if too high, may affect the timing behavior of the target system if it shares resources with the monitoring process, and thus may cause timing constraint violations. To avoid this, the computation overhead of the monitoring process should be as low as possible, and the monitoring process should be properly scheduled or placed so that it will not affect the timing of the target system processes.

In this paper we shall present an approach for monitoring timing constraints of real-time systems with explicit consideration of detection latency. Our approach starts with a specification language based on Real Time Logic (RTL) to

*This work is supported by a grant from the Office of Naval Research under grant number N00014-94-1-0582.

define the timing constraints to be monitored. Efficient algorithms are developed to detect any constraint violations at run time. We shall show that the violation detection latency is bounded and the monitoring overhead is low.

The practicality of our approach is demonstrated by the implementation of a timing constraint monitor for the Java language. With the increasing popularity of Java in Internet applications, we can expect more and more real-time, multimedia applications to be developed in Java in the near future. Besides the built-in multi-threading feature, work is going on to provide real-time support for Java [18, 19, 20]. As pointed out in [18, 19, 20], many potential Java applications have timing constraints associated with them. As such, we see Java as a good candidate for timing constraint monitoring. We have implemented our monitoring approach in Java in a package called JRTM (Java Run-time Timing-constraint Monitor).

1.1. Related Work

Significant work has been done in recent years on monitoring real-time systems [27]. Hardware monitoring approaches are proposed in [1, 11, 13, 21, 26]. These approaches use dedicated hardware to detect event occurrences by snooping and matching bus signals of target systems and storing event data for post-processing. These methods are especially suitable for monitoring hard real-time systems because they are non-intrusive. In [3, 8, 25], software monitoring approaches are proposed which insert event-detection and event-data-collecting code into application programs, operating system kernels or monitoring systems. In [25], a real-time monitor (ARM) for a distributed real-time operating system (ARTS) is described. Events, which are generated whenever a process changes state, are captured by the ARTS kernel and sent to a visualizer on another machine for displaying and analysis. In [3], a monitoring system is described that monitors events in a distributed environment. In this system, code for generating events are inserted into the kernel, system call library, interrupt handlers, shared variable access methods as well as application programs. These software monitoring approaches are intrusive because they all to various degrees interfere with the run-time behavior of the target systems. Less interference to the target systems can be achieved by systems which use special hardware for event detection and event data collection but nevertheless instrument target programs to trigger events. Examples of this type of hybrid monitoring approach can be seen in [4, 5, 14].

Most of the forementioned research mainly addresses various problems in event detection and event data collection. The event data collected during monitoring is usually used for postmortem analysis for violation of timing constraints. Work on run-time detection of timing constraint violation has received less attention. In [16], an annotation method was introduced which marks the events of interests in Ada programs and uses Real Time Logic (RTL) formulas to specify the timing constraints to be enforced. In FLEX [9], certain predefined timing constraints can be monitored for violation. The work closest to our research is [2] which presents an event-based model for specifying timing constraints to be monitored and proposes two methods for synchronous or asynchronous monitoring of real-time constraints. A timing constraint satisfiability checking algorithm is also described in that paper. In [22], the model of [2] is extended to distributed systems and the problem of detecting timing constraint violations in a distributed environment is discussed. It shows that the problem of minimizing the number of messages among processors in order to detect a violation as early as possible is NP-hard. But for a sub-class of timing constraints, this problem is in PTime.

Our approach is heavily influenced by the work in [2, 16, 22]. We extend previous work by providing a more expressive language for timing constraint specification which allows using future relative event occurrences in the definition of timing constraints. We have also developed a timing constraint compilation algorithm which enables more efficient run-time violation detection at the earliest time possible. We show that the memory cost for monitoring has an upper bound which can be determined at compile time.

The rest of this paper is organized as follows: Section 2 defines the event model and describes the approach we use for specifying the timing constraints. Algorithms for timing constraint compilation and violation detection are discussed in Section 3 and Section 4, respectively. Section 5 introduces the Java timing constraint monitor that we have built and discusses several implementation issues. Section 6 is the conclusion.

2. Specifying Timing Constraints

2.1. Event Model

Most real-time monitoring systems work by capturing the events generated from the target systems. Informally, *events* represent state changes of interests that may occur in a system. For example, “received a message from process A”, “start the execution of function foo()” can be defined as events. We adopt the event model first proposed in [6, 15]. Events are a finite set of names specified by the user and are generally *recurrent*, i.e., an event may occur multiple times during a computation. We shall not concern ourselves with the specific syntactic rules in application programs for defining the occurrence of events. For our purposes, an *event occurrence* defines a point in time at which an instance of the event happens (i.e., an event occurrence is a pair consisting of an event name and a time value). The computation of a real-time system can be viewed as a sequence of sets of *event occurrences*.

To capture the relationship between event instances and their occurrence time during a computation, we introduce below an uninterpreted function symbol: the @ function which we call the event occurrence function [6, 15].

Definition 1 *For event e and integer $i \in N^+$, we define @-function as*

$$@(e, i) = occurrence\, time\, of\, the\, ith\, instance\, of\, event\, e$$

i is called the occurrence parameter of the @-function.

Example 1 $@(e, 4)$ *represents the occurrence time of the $4th$ instance of event e.*

We assume that the $@$-function is monotonic with respect to the occurrence parameter, i.e.

$$\forall event\ e,\ \forall i \in N^+,\ @(e, i+1) > @(e, i)$$

We also define another function, denoted by $\#(e, t)$, to represent the occurrence index of the most recent instance of event e at time t during a computation.

Definition 2 *For event e and time t during a computation, we define #-function as*

$$\#(e, t) = \begin{cases} 0 & if\ \ t < @(e, 1) \\ i & if\ i \geq 1\ \wedge\ @(e, i) \leq t\ \wedge\ @(e, i+1) > t \end{cases}$$

[2, 22] extends the @-function to represent occurrence times of past relative instances. We further extend it to represent future relative instances.

Definition 3 *For event e, integer $i \in N$ and time t during a computation, we define relative @-function as*

$$@_r(e, t, i) = @(e, \#(e, t) + i) \quad when\ \#(e, t) + i > 0$$

t is called the reference time and i is called the occurrence parameter of the relative @-function.

Whenever there is no ambiguity, we shall overload the @ operator by using it in place of the $@_r$ symbol for the relative @-function.

Example 2 $@(e, 4, 1)$ *represents the next occurrence time of event e at time 4.* $@(e_1, @(e_2, 4), 0)$ *denotes the occurrence time of the most recent instance of event e_1 at time* $@(e_2, 4)$.

2.2. Simple Constraints

With the event occurrences representing points of time during the execution of real-time systems, we can express timing constraints as a set of assertions that relate the time of occurrences of different events to one another. With the help of the @-function and the relative @-function, the timing constraints between two event instances can be expressed as inequalities relating the corresponding @-functions or relative @-functions. We call such an inequality as a *simple constraint*.

Definition 4 *A **simple constraint** is an expression of the form: $T_1 + D \geq T_2$, where D is an integer constant, and T_1, T_2 are two time terms which may be an @-function, relative @-function or 0, subject to the following restrictions:*

- *At least one of T_1 and T_2 must be @-function.*
- *If @-function is used to represent T_1 and/or T_2, the occurrence parameter of the @-function is an arithmetic expression in the form of $a * i + b$, where i is a variable and a, b are integer constants with $a \geq 0$. The domain of the variable i is N^+.*
- *If T_1 is a relative @-function, its reference time must be T_2 and its occurrence parameter must be an integer constant, and vice versa.*

Example 3 *The simple constraint $@(e_1, i) + d \geq @(e_2, i)$ with $d > 0$ specifies the deadline for the ith occurrence of event e_2 to be d time units after the ith occurrence of event e_1. The simple constraint $(e_1, i) - d \geq @(e_2, i)$ with $d > 0$ requires the ith occurrence of event e_1 be at least d time units later than the ith occurrence of event e_2.*

When relative @-functions are involved, we shall use a shorthand. Consider a typical simple constraint: $@(e_1, i) + 10 \geq @(e_2, @(e_1, i), 2)$. For abbreviation, we shall also write this constraint as $@(e_1, i) + 10 \geq @(e_2, \%2)$.

2.3. Timing Constraint Specification

By expressing the timing relationship between two event instances with simple constraints, we can specify timing constraints of a real-time system as universally quantified formulas of simple constraints in disjunctive normal forms. Existential quantifiers are disallowed for now and their consideration is left for future work. Conjuncts of simple constraints in such formulas will be called *constraint conjunctions*.

Example 4 *In a database system, a timing rule like*

$$@(Start\_T, i) + 100 > @(Commit\_T, \%1)\ \vee$$
$$@(Start\_T, i) + 100 > @(Abort\_T, \%1)$$

asserts that each execution of transaction T, should be finished, either aborted or committed, within 100 time units.

3. Compiling Timing Constraints

3.1. Implicit Constraints

In general, timers are needed to detect violations for simple constraints, especially if we want to catch violations as early as possible. For instance, the simple constraint $@(e_1, 1) + 5 > @(e_2, 1)$ will be violated 5 time units after the first occurrence of event e_1 if the first instance of event e_2 has not occurred by then. But event e_2 may never occur, so the detection of the violation cannot be triggered by the occurrence of event e_2 alone. Furthermore, the detection problem is complicated by the existence of *implicit* constraints, which are derived from constraint conjunctions.

Definition 5 *Given a constraint conjunction C, an **explicit** constraint of C is a simple constraint that appears in C. A simple constraint that is not an explicit constraint but is logically implied by C is called an **implicit** constraint of C.*

Example 5 *Given constraint conjunction*

$$@(e_1, i) + 1000 \geq @(e_2, i) \ \wedge \ @(e_2, i) - 999 \geq @(e_3, i)$$

we can get an implicit constraint $@(e_1, i) + 1 \geq @(e_3, i)$, whose related explicit constraints are $@(e_1, i) + 1000 \geq @(e_2, i)$ and $@(e_2, i) - 999 \geq @(e_3, i)$. Suppose the ith instance of event e_1 happens at time 1. If none of the ith instances of event e_2 and e_3 have occurred by time 2, then at time 2 this implicit constraint is violated but none of the explicit ones are. Hence, checking this implicit constraint will enable us to catch a violation early. Note, however, not all implicit constraints are violated before the related explicit ones. For instance, the implicit constraint $@(e_1, i) + 1999 \geq @(e_3, i)$, which is derived from the explicit constraint conjunction

$$@(e_1, i) + 1000 \geq @(e_2, i) \ \wedge \ @(e_2, i) + 999 \geq @(e_3, i)$$

cannot be violated before both of the two related explicit constraints.

Definition 6 *Given a set of simple constraints S, a constraint c in S is said to be **unnecessary** if there exists a subset S_c of S, $S_c = \{c_i \mid c_i \in S,\ c_i \neq c,\ i = 1, \ldots, k, k \geq 1\}$ such that at any time t during a computation, if conjunct $\wedge_{i=1}^{k} c_i$ is satisfied at time t, then so is c, and if c is violated at time t, then so is $\wedge_{i=1}^{k} c_i$. Any simple constraint in S that is not unnecessary is called a **necessary** constraint of S.*

Definition 7 *Given constraint conjunction C, let SC be the set consisting of all explicit and implicit constraints of C. Suppose S is a subset of SC and C' is the conjuction of all the simple constraints in S. We call S a **useful constraint set** of C if and only if at any time t during a computation, if C' is satisfied, then so is C, and if C is violated, then so is C'.*

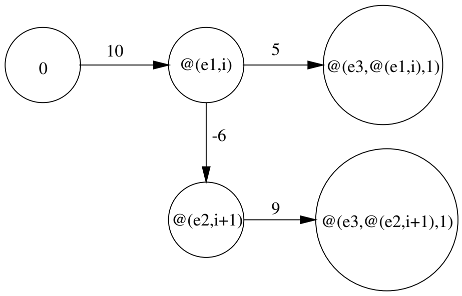

Figure 1. A Constraint Graph

3.2. Constraint Graph

As shown in Figure 1 where vertices denote the terms in simple constraints, we can represent a simple constraint as a weighted, directed edge and a constraint conjunction as a weighted directed graph, to be called the *constraint graph* of the constraint conjunction. The constraint graph in Figure 1 represents the following constraint conjunction:

$$@(e_1, i) - 6 \geq @(e_2, i+1) \ \wedge \ @(e_1, i) + 5 \geq @(e_3, \%1)$$
$$\wedge \ @(e_2, i+1) + 9 \geq @(e_3, \%1) \ \wedge \ @(e_1, i) \leq 10$$

Notice a special *zero* vertex is introduced to represent constraints like $@(e_1, i) \leq 10$. We shall call those vertices representing @-function terms, such as $@(e, i)$, *absolute* vertices and we call those vertices representing relative @-function terms, such as $@(e, \%4)$, *relative* vertices. Furthermore, those *relative* vertices whose corresponding relative @-function terms represent future event occurrences (e.g. $@(e, \%2)$) are called *future relative* vertices. Likewise, the vertices for $@(e, \%0)$ and $@(e, \%-3)$ are *past relative* vertices because they refer to past event instances. Notice that according to our constraint definition, each *relative* vertex should have an *absolute* vertex as its *reference* vertex. Obviously, each path in a constraint graph with at least one intermediate vertex represents an implicit constraint of the corresponding constraint conjunction. We call a cycle in a constraint graph with negative weight a *negative cycle*. The following theorem regarding negative cycles is immediate:

Theorem 1 [1] *If a negative cycle exists in a constraint graph, then the corresponding constraint conjunction is unsatisfiable.*

3.3. Compiling Constraint Graphs

In [2] Chodrow et al described a satisfiability checking algorithm which instantiates a constraint graph at each check point and searches for negative cycles on the instantiated graph by using the Floyd-Warshall algorithm. If a

[1] See [17] for proof of all theorems and corollaries.

negative cycle is found, then the corresponding instantiated constraint conjunction is unsatisfiable and a violation is detected. This means at each check point $O(n^3)$ time is needed in worst case for each instantiated constraint graph, where n is the number of vertices in the instantiated constraint graph. However, if we can resolve most, if not all, of the shortest paths of an instantiated constraint graph at compile time, the computation needed at each check point can be greatly reduced. In fact, in our algorithm only $O(n)$ time is needed in the worst case at each check point.

Since a path between any two vertices in a constraint graph G represents either an explicit or implicit constraint, we define a path in G to be *necessary* if its corresponding constraint c_p is necessary for the constraint set $S = \{c_p\} \cup \{c \mid c \text{ is the constraint corresponding to edge } e \text{ in } G, \forall e \text{ in } G\}$. Similarly, a path is *unnecessary* if its corresponding constraint is unnecessary in S_c.

Theorem 2 *Given a constraint graph $G = (V, E)$, $\forall u, v \in V$, let l be the length of the shortest path from u to v. If edge $u \xrightarrow{l} v \in E$, then any longer paths from u to v are unnecessary.*

Corollary 2.1 *Positive cycles are unnecessary paths.*

Theorem 3 *Given a constraint conjunction C, let $G = (V, E)$ be its corresponding constraint graph and $G' = (V, E')$ be the constraint graph where $\forall u, v \in V, u \neq v$, edge $u \xrightarrow{w} v \in E'$ if and only if w is the length of the shortest path from u to v in G. Then the constraint set S, which consists of all the constraints represented by edges in G', is a useful constraint set of C.*

According to Theorem 3, the shortest paths between vertex pairs in G consist of a useful constraint set of C. However, not all of these shortest paths represent necessary constraints. Theorem 4 and its corollaries will be used to help decide whether such a shortest path corresponds to a necessary constraint.

Theorem 4 *Consider constraint graph $G = (V, E)$. $\forall u, v, w \in V$, $(u, v), (v, w) \in E$, let t_u, t_v, t_w be the time points denoted by vertices u, v, w in a computation and let d_1, d_2 be the weights on edges (u, v) and (v, w). If t_v is always no later than the earliest violation detection time of the implicit constraint corresponding to the path $p = u \to v \to w$, then p is unnecessary.*

Corollary 4.1 *Consider a constraint graph $G = (V, E)$. $\forall v \in V$, if v is a relative vertex, then any path starting from v or ending at v that has at least one intermediate vertex is unnecessary.*

Corollary 4.2 *Consider a constraint graph $G = (V, E)$. $\forall$ $path$ p of G such that p has at least one intermediate vertex, if the weight on the first edge along the path p is negative, then p is unnecessary.*

Corollary 4.3 *Consider a constraint graph $G = (V, E)$. $\forall$ path p of G such that p has at least one intermediate vertex, if the weight on the first edge along the path p is non-negative but not bigger than the length of p, then p is unnecessary.*

With Theorem 3 and Theorem 4, we can compile a constraint graph by first searching the shortest paths between all pairs of vertices in the graph to eliminate those unnecessary paths identified in Theorem 3, and then checking the remaining shortest paths to eliminate those unnecessary constraints which can be identified by Theorem 4 and its corollaries. Moreover, if a negative cycle is found during the compilation, we can conclude that the constraint conjunction is unsatisfiable from Theorem 1 . Algorithm 1 describes this compilation procedure. The Floyd-Warshall algorithm is used here.

This compilation algorithm can eliminate most of the unnecessary paths in the compiled constraint graph. However, some vertices in the compiled constraint graph may become the same vertex in some of its instantiated graphs, i.e., they all represent the occurrence of the same event instance in an instantiated constraint graph. We call these vertices *equivalent* vertices. In some instantiated constraint graphs, these *equivalent* vertices may give rise to additional necessary paths that are not discovered by our compilation algorithm. As discussed in Section 4, some of these additional necessary paths can be resolved at compile time while others can be determined at run time at a cost of $O(n)$ time, where n is the number of vertices in the instantiated constraint graph.

4. Monitoring Timing Constraints

4.1. Detecting Violation of a Constraint

We make the following observations regarding the violation time of a simple constraint:

Theorem 5 *Consider a constraint graph $G = (V, E)$. $\forall u, v \in V$ such that $(u, v) \in E$, let t_u, t_v be the time points denoted by vertices u, v in a computation and let d be the weight on edge (u, v). Suppose t is the earliest violation detection time for the constraint represented by (u, v), then*

a. $t \geq t_u + d$ *if* $d \geq 0$*;*

b. $t = t_u$ *if* $d < 0$.

By exploiting Theorem 5, we given a violation detection algorithm for a simple constraint $c = (T_1 + d \geq T_2)$ in Algorithm 2, where we use $c.T_1, c.d, c.T_2$ to denote T_1, d and T_2, respectively.

Algorithm 1: Compile Constraint Graph

Input: V : the vertex vector of a constraint graph G;
$Dist$: distance metrics of the constraint graph G;

Output: $Dist$ is updated such that $Dist(u, v) = \infty$ if there is no necessary path from u to v; otherwise $Dist(u, v)$ is the length of the necessary path from u to v.

COMPILE(V, $Dist$)

for $i = 0$ **to** $V.size() - 1$
  $Dist(i, i) = 0$;
for $k = 0$ **to** $V.size() - 1$
  if $d(k, k) < 0$ **then**
    NEGATIVECYCLEHANDLER();
    return
  else
    for $i = 0$ **to** $V.size() - 1$
      for $j = 0$ **to** $V.size() - 1$
        $Dist(i, j) \leftarrow min(Dist(i, j), Dist(i, k) + Dist(k, j))$;
for $i = 0$ **to** $V.size() - 1$
  if $V(i)$ is a relative vertex **then**
    for $j = 0$ **to** $V.size() - 1$
      if $V(j)$ is not the reference vertex of $V(i)$ **then**
        $Dist(i, j) \leftarrow \infty$;
  else
    for $j = 0$ **to** $V.size() - 1$
      if $V(j)$ is a relative vertex **then**
        if $V(i)$ is not the reference vertex of $V(j)$ **then**
          $Dist(i, j) \leftarrow \infty$;
      else
        for $k = 0$ **to** $V.size() - 1$
          if $k \neq i$ **and** $k \neq j$ **and** $Dist(i, j) \neq \infty$ **and** $Dist(i, k) \neq \infty$ **and** $Dist(k, j) \neq \infty$ **and** $Dist(i, j) = Dist(i, k) + Dist(k, j)$ **and** $(Dist(i, k) < 0$ **or** $Dist(i, k) \leq Dist(i, j))$ **then**
            $Dist(i, j) \leftarrow \infty$;
            break
  return

Algorithm 2: Check the Satisfiability of a Constraint

Input: c: the constraint to be checked

Output: none

CHECK(c)

if $c.T_1$ is known
  if $c.T_2$ is known
    if $c.T_1 + c.d < c.T_2$
      CONSTRAINTVIOLATIONHANDLER(c)
  else if $c.d \geq 0$
      Set deadline timer for the event instance corresponding to $c.T_2$ with deadline at $c.T_1 + c.d$
    else
      CONSTRAINTVIOLATIONHANDLER(c)
else if $c.T_2$ is known **and** $c.d < 0$
    Set delay timer for the event instance corresponding to $c.T_1$ with timeout at $c.T_2 - c.d$
return

4.2. Detecting Violation of a Constraint Conjunction

According to our discussion in Section 3, all necessary constraints in a useful constraint set of a constraint conjunction need to be checked in order to catch any violation as early as possible. As we mentioned in Section 3.3, we expect most of the necessary paths in the compiled constraint graph to be identified after the compilation. However, some implicit necessary constraints may remain unidentified for some instantiated constraint conjunctions owing to the presence of *equivalent vertices*. Two vertices are *equivalent* if the time terms they represent are instantiated to the same value. There are two cases regarding equivalent vertices:

a. Some vertices may be merged when the time terms they represent are unified by the instantiation of the occurrence parameter to certain values. Figure 2(a) and 2(b) show such an example. Vertices v_1 and v_2 in the constraint graph of Figure 2(a) become one vertex v in the instantiated constraint graph of Figure 2(b) when $i = 1$. This reduces the length of the shortest path length from u to w to 2 from the previous value 5.

b. Some *relative* vertices may merge with other vertices into one vertex at run time. For example, vertices v_1 and v_3 in Figure 2(a) become vertex v' in the instantiated constraint graph of Figure 2(c) when $i = 2$ and $\#(e_2, @(e_1, 2)) = 1$. This reduces the length of the shortest path from u to w to 3 from the previous 5.

The equivalent vertices of case (a) can be identified at compile time by solving for the variable values that unify the time terms of the vertices. For a constraint graph of n vertices, there can be at most $\frac{n(n-1)}{2}$ such variable values, each of which corresponds to an instantiated constraint graph. We can run our compilation algorithm on these instantiated constraint graphs to identify the necessary paths. We call this approach *static instantiation*. Using static instantiation, we do not have to worry about the equivalence of case (a) during the run-time monitoring. The trade-off is increased space complexity, from the previous worst case of $O(n^2)$ to $O(n^4)$. An alternative is to defer this second compilation to run time when we check the satisfiability of the instantiated constraint graphs. This means that we incur an extra $O(n^3)$ time complexity in the worst case when checking certain instantiated constraint graphs. There are at most $\frac{n(n-1)}{2}$ such instantiated graphs for a constraint conjunction. We call this alternative approach *dynamic instantiation.*

When relative vertices are involved, however, there is no way we can determine the equivalence of vertices in case (b) until the reference event instances occur at run time. This means we need to search for additional necessary paths at run time when a relative vertex becomes equivalent to some

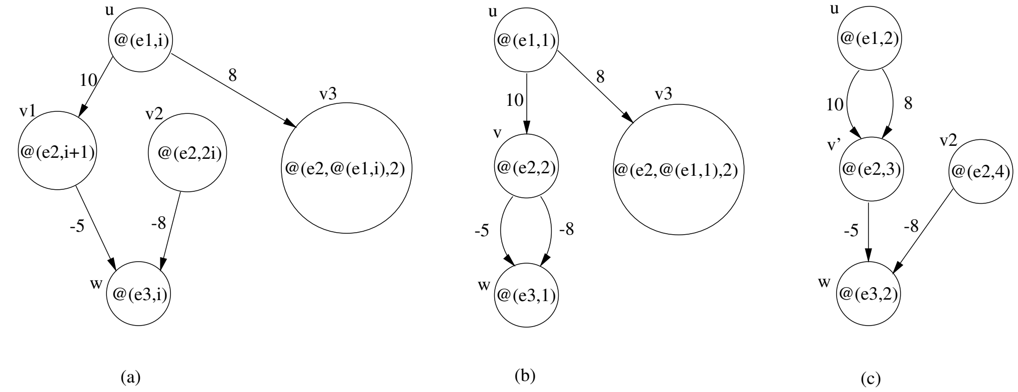

Figure 2. Equivalent vertices

other vertices in an instantiated constraint graph. The following theorem helps us identify the new necessary paths in such cases.

Theorem 6 *Consider a compiled constraint graph $G = (V, E)$, $v, v_r, v_e \in V$, where v_r is a relative vertex, v is v_r's reference vertex. Suppose at time t in a computation, v_r and v_e become equivalent, i.e., they both represent the occurrence of the same event instance in the instantiated constraint graph G'. Then any new shortest path p in G' resulting from this equivalence is unnecessary unless v_r is a future relative vertex and p's source or destination is v.*

When a relative vertex becomes equivalent with other vertices in an instantiated constraint graph, Theorem 6 tells us to search for new necessary paths only when a future relative vertex is involved. In updating those paths to and from the reference vertex, at most $O(n)$ time is needed. This update algorithm is given below:

Algorithm 3: Update Paths
Description: Update necessary paths in an instantiated constraint graph when a future relative vertex becomes equivalent with another vertex
Input: G : the compiled constraint graph;
G_i : the instantiated constraint graph of G;
u : the future relative vertex in G;
v : the reference vertex of u in G_i;
w : the equivalent vertex of u in G
Output: updated instantiated constraint graph
UPDATEPATH(G, G_i, u, v, w)
(1) $y \leftarrow$ the vertex representing u and w in G_i;
(2) **if** edge (u, v) exists in G
(3) **foreach** vertex x in G_i
(4) **if** $G_i.dist(x, y) + G.dist(u, v) < G_i.dist(x, v)$
(5) $G_i.dist(x, v) \leftarrow G_i.dist(x, y) + G.dist(u, v)$
(6) **if** edge (v, u) exists in G
(7) **foreach** vertex x in G_i
(8) **if** $G.dist(v, u) + G_i.dist(y, x) < G_i.dist(v, x)$
(9) $G_i.dist(v, x) \leftarrow G.dist(v, u) + G_i.dist(y, x)$
(10) **return**

Now we can check satisfiability of a constraint conjunction at run time by checking the constraints corresponding to edges in the updated instantiated constraint graph at their corresponding check points, using Algorithm 2. If a constraint is violated, the corresponding instantiated constraint conjunction is violated. If all these constraints are satisfied, then the corresponding instantiated constraint conjunction is satisfied. The skeleton of the satisfiability checking algorithm for a constraint graph is described in Algorithm 4.

Algorithm 4: Check Satisfiability of a Constraint Graph
Description: Check the satisfiability of an instantiated constraint graph at the occurrence of an event instance (e, k)
Input: G : the compiled constraint graph;
G_i : the instantiated constraint graph of G;
v : an absolute vertex in G corresponding to (e, k);
v_i : the corresponding vertex of v in G_i
Output: none
CHECKCONSTRAINTGRAPH(G, G_i, v, v_i)
(1) **foreach** future relative vertex u referencing v in G
(2) **if** vertex w of G is equivalent with u in G_i
(3) UPDATEPATH(G, G_i, u, v_i, w)
(4) **foreach** adjacent vertex x of v_i in G_i
(5) **if** edge (x, v_i) exists in G_i **and** (x corresponds a relative vertex in G **or** $G_i.distance(x, v_i)$ newly updated)
(6) CHECKCONSTRAINT(constraint (x, v_i))
(7) **if** constraint (x, v_i) is violated
(8) **return**
(9) **if** edge (v_i, x) exists in G_i
(10) CHECKCONSTRAINT(constraint (v_i, x))
(11) **if** constraint (v_i, x) is violated
(12) **return**
(13) **return**

If the static instantiation approach is adopted, the worst case computation complexity of this checking algorithm is $O(n)$. If we use dynamic instantiation, then the worst-case time complexity of our checking algorithm becomes $O(n^3)$ which is incurred by the need to search for additional necessary paths as a result of equivalent vertices (case (a)).

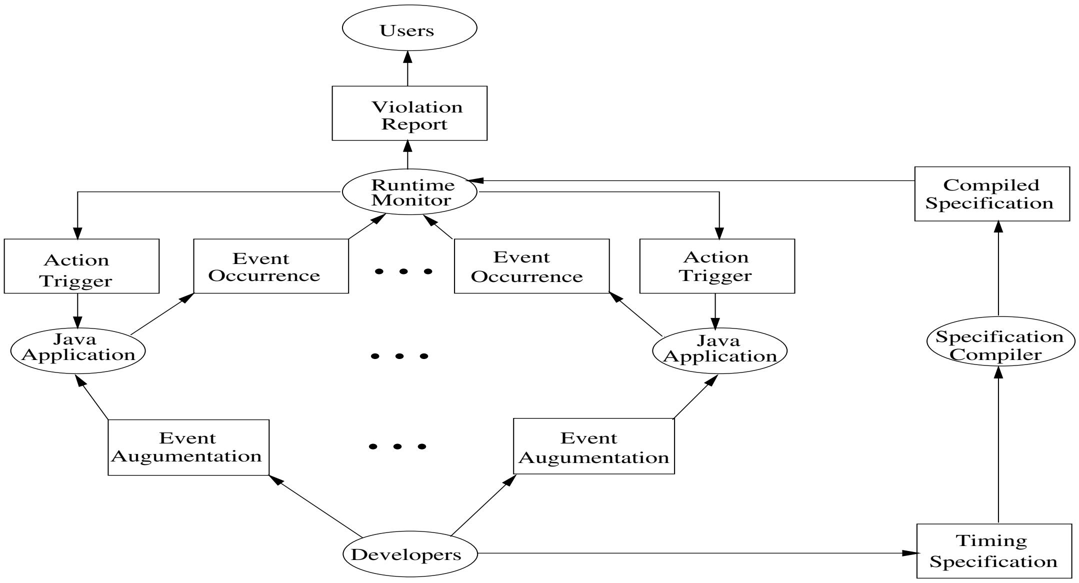

Figure 3. The interaction between JRTM and Java applications

4.3. Memory Requirements

We need to maintain an event occurrence log for each event involved in the checking algorithm and yet we should not allow the log to grow unbounded so as to render monitoring impractical to implement. Hence, it is important for us to know the *necessary length* of an event log, i.e., the minimum number of occurrences we need to record in such a log so that no violation will be missed. As we have proven in [17], an upper bound for the necessary length can be determined for each event log at compile time.

Since timers are used in our constraint checking algorithm in Section 4.1, the careful reader may also be concerned about the number of outstanding timers during the monitoring. Similar to the necessary event log length, it turns out that we can also determine at compile time an upper bound for the number of outstanding timers [17].

5. JRTM — A Java Run-time Timing-constraint Monitor

Java is a popular object-oriented programming language introduced by Sun Microsystems in May, 1995. Java has been claimed to be architecture independent and portable over the Internet because any Java program can be compiled into a standard byte-code format which can run on any platform with a Java interpreter that implements the Java Virtual Machine. The built-in multi-threading support in the language makes it a candidate for implementing real-time and/or multimedia applications. Unfortunately, the current Java Virtual Machine does not lend itself to the implementation of deterministic behavior very well. It is thus important to incorporate timing violation detection in the Java run-time environment.

Since many anticipated Java applications are distributed systems and have either soft or hard real-time requirements, we have implemented a Java run-time timing constraint monitor (JRTM) based on the monitoring approach presented in this paper. JRTM provides a simple but expressive language for specifying timing constraints to be monitored. The run-time monitor can catch any violations of the specified timing constraints with a low overhead.

5.1. System Model

The interaction between the JRTM and Java applications is shown in Figure 3. In this model, we assume that Java applications may run on different processors, whose clocks are synchronized within a maximum deviation ϵ. All occurrences of events of interests are sent to the monitor for violation detection. In addition, we assume that the communication between target processes and the monitoring process is reliable and the communication delay has an upper bound.

The timing constraints of the Java application are expressed by the specification method as described in Section 2. These specifications are compiled by the Timing Constraint Specification Compiler, using the compilation algorithm in Section 3. Necessary constraints and event log length are automatically derived by the compiler. The compiled specification is loaded into the monitor at run time. Because there is no general event triggering mechanism implemented in Java, we provide an event class that has a triggering method. Java programmers can insert the event triggering method calls in their Java programs where event instances are supposed to occur. At run time, when these event triggering methods are executed, the applications send the event instance occurrence timestamps along with the event names to the monitor. The monitor keeps these event occurrence messages in a sorted queue with the earliest event message at the head of the queue. The event message at the head is processed at an appropriate time to check the related constraints. Once a violation of the specification is found, users are notified and some predefined exception triggering messages may be sent to applications to invoke recovery actions.

The monitor can run on either the same machine with a target process or on a stand-alone monitoring machine. Since the run-time monitor and the specification compiler are both written in Java, this toolset can be used on all machines which support Java. The monitor can be run as an applet inside a web page. A standalone version is also available to run outside of web browsers.

5.2. Event Class

To monitor the timing constraints on event occurrences, the monitor needs to know the occurrence time of event instances on the target systems. Since Java does not have a mechanism that generates and reports event occurrence information to the monitor, we provide a simple event class for this purpose. The interface of our event class is:

```
class RTEvent {
  public void static init();
  public RTEvent(String ename);
  public void trigger();
}
```

The static method *init()* should be called at the beginning of a Java program to initialize the connection with the monitor. The constructor *RTEvent()* is used to create an object for each event. The method *trigger()* is to be inserted at locations in Java programs where events of interest are supposed to occur. Whenever an event $trigger()$ method is executed, the current time is recorded as the event occurrence time and this timestamp is sent to the monitor along with the event name.

The following is an example of augmented Java application programs:

```
...
// Initialize connection with the monitor
RTEvent.init();
...
// Create event 'BeginTransaction'
RTEvent e1=new RTEvent(``BeginTransaction'');
// Create event `EndTransaction'
RTEvent e2=new RTEvent(``EndTransaction'');
...
// Trigger the 'BeginTransaction' event
e1.trigger();
// Transaction code
...
// Trigger the 'EndTransaction' event
e2.trigger();
...
```

To understand the overhead incurred by this instrumentation, we measured the execution time of these three methods of $RTEvent$ class. The experiment is done on Pentium133 PCs with 48M memory, running Linux2.0.0 kernel and XFree86 3.1.2, using JDK 1.0.2. Our results show the $init()$ method takes about 72 milliseconds to execute, where 60 milliseconds was due to setting up a socket connection with the monitor. Since this overhead is a one-time fixed cost, it should be acceptable for most applications. The overhead of the constructor method is quite low, taking only 0.17 millisecond. The execution time of $trigger()$ method is about 3.3 milliseconds, where 2.1 milliseconds comes from sending the event occurrence message over the socket connection, and a $long$ to $string$ conversion takes about 0.8 millisecond. This overhead seems a little bit high considering $trigger()$ may be called by the applications many times. An alternative implementation of the $trigger()$ method uses a background thread to send queued event occurrence messages to the monitor so that each call of the $trigger()$ only involves attaching the event occurrence time and the event name to the queue. For this implemenation, $trigger()$ takes only 0.03 millisecond to finish. However, the trade-off is the longer delay in event occurrence message transmission since the background thread has lower priority than application threads.

5.3. Run-time Monitoring

Whenever an event is triggered at run time, the application sends a message to the monitor reporting the occurrence time of the event instance. Upon receiving such an event occurrence message, the monitor inserts the message into a queue sorted according to event occurrence time. Whenever the messages of all event instances that occur before or at the occurrence time of the message at the head of the queue (i.e., the message with the earliest occurrence time in the queue) have been received by the monitor, the message at the head of the queue is processed. The monitor first updates the occurrence log for the event corresponding to the head message and then cancels all related timers set for this event instance. After that, the constraints related to this event instance is checked for violation by the checking algorithms described in Section 4. Timers may be set for the occurrences of some event instances during this check. Synchronization is enforced by delaying processing the message at the head until the time by which all earlier messages have been received. This delay should be at least $d_{max} - d_{min} + \epsilon$, where d_{max}, d_{min} are the upper and lower bound on the message communication delay from target processes to the monitor process, and ϵ is the maximum clock deviation between target system clocks. In this case, the maximum violation detection latency is $2d_{max} + \epsilon - d_{min} + ct_1 + ct_2$, where ct_1 is the message queuing delay at the monitor, and ct_2 is the worst case computation time of the checking procedure. This monitoring procedure is described in Algorithm 5.

Algorithm 5: Monitoring Process
Input: Specs: compiled timing constraint specification
Output: none
MONITORING(Specs)
while true
  if received a message from applications
    $(evtName, occTime) \leftarrow$ received message;
    $recvTime \leftarrow currentTime()$;
    insert $(evtName, occTime, recvTime)$ into $mqueue$;
  if $mqueue$ is not empty **and** $currentTime() \leq mqueue.headMsg.recvTime + Delay$
    update the occurrence log for the event with $mqueue.headMsg.evtName$;
    $occ \leftarrow$ the occurrence no. of this event instance;
    cancel timers set for this event instance;
    check constraints using $mqueue.headMsg$;
    remove head message from $mqueue$;

We also did some performance measurement under the same environment as described in Section 5.2. Monitoring a timing constraint of five *simple constraint* conjunctions, we determined the violation detection latency to be about 34 milliseconds on the average and 55 milliseconds in the worst case. Measurements indicated that 21 milliseconds was due to message passing delay from the application to the monitor, about 2 milliseconds was caused by message queuing delay at the monitor, and violation detection took about 11 milliseconds in average and 32 milliseconds in worst case. However, a careful study of the measured data found that 90% of the violation detection time were under 12 milliseconds. It is possible that the longer detection latency may be caused by process context switching which happened during detection.

6. Conclusions and Future Work

In this paper we have presented an approach for monitoring timing constraint violations in real-time systems. Our approach includes a language for specifying timing constraints and efficient monitoring algorithms which can catch timing constraint violation as early as possible. We have shown that by compiling the timing constraint specifications, run-time monitoring can be performed efficiently with low overhead. We have also shown that the memory requirement for monitoring is bounded for our specification language and an upper bound can be determined at compile time. An implementation of this approach has been done in Java. JRTM, our Java run-time timing constraint monitor can be used in a distributed environment to help Java developers specify and monitor the timing constraints of their applications.

In view of the results reported in this paper, other research issues should be pursued to extend the utility of our monitoring approach. Some of these issues are:

- Although our results provide the shortest latency for violation detection, sometimes an application can tolerate a longer delay. In this case, we might be able to optimize our monitoring process so that even lower monitoring overhead can be achieved. Optimization algorithms need to be explored for this purpose.
- When relative @-functions are used in a constraint, we restrict its reference time to be the other @-function in the constraint. What will be the impact to our monitoring algorithms and memory requirements if we relax this restriction?

References

[1] W.C. Brantley, K.P. McAuliffe, and T.A. Ngo, "RP3 Performance Monitoring Hardware", in *Instrumentation for Future Parallel Computing Systems*, M. Simmons, R. Koskela, and I. Bucher, eds., ACM Press, New York, 1989, pp.35-47.

[2] S.E. Chodrow, F. Jahanian, and M. Donner, "Run-Time Monitoring of Real-Time Systems", *Proc. Real-Time Systems Symp.*, 1991, pp.74-83.

[3] P.S. Dodd and C.V. Ravishankar, "Monitoring and Debugging Distributed Real-Time Programs", *Software–Practice and Experience*, Vol.22, No.10, Oct. 1992, pp.863-877.

[4] M.M. Gorlick, "The Flight Recorder: An Architectural Aid for System Monitoring", *Proc. ACM/ONR Workshop Parallel and Distributed Debugging*, 1991, pp.175-183.

[5] D. Haban and D. Wybranietz, "A Hybrid Monitor for Behavior and Performance Analysis of Distributed Systems", *IEEE Trans. Software Eng.*, Vol.16, No.2, Feb. 1990, pp.197-211.

[6] F. Jahanian and A. K. Mok, "Safety Analysis of Timing Properties in Real-Time Systems", *IEEE Trans. Software Eng.*, Vol.SE-12, No.9, Sept. 1986, pp.890-904.

[7] F. Jahanian and A. Goyal, "A Formalism for Monitoring Real-Time Constraints at Runtime", *Proc. IEEE Fault-Tolerant Computing Symp.*, June 1990, pp.148-155.

[8] J. Joyce, G. Lomow, K. Slind, and B. Unger, "Monitoring Distributed Systems", *ACM Trans. Computer Systems*, Vol.5, No.2, May 1987, pp.121-150.

[9] K.B. Kenny and K.-J. Lin, "Building Flexible Real-Time Systems using the Flex Language", *Computer*, Vol.24, No.5, May 1991, pp.70-78.

[10] R.J. LeBlanc and A.D. Robbins, "Event-Driven Monitoring of Distributed Programs", *Proc. 5th Int'l Conf. Distributed Computing Systems*, 1985, pp.515-522.

[11] A.-C. Liu and R. Parthasarathi, "Hardware Monitoring of a Multiprocessor System", *IEEE Micro*, Vol.9, No.5, Oct. 1989, pp.44-51.

[12] J.E. Lumpp et al., "Specification and Identification of Events for Debugging and Performance Monitoring of Distributed Multiprocessor Systems", *Proc. 10th Int'l Conf. Distributed Computing Systems*, 1990, pp.477-483.

[13] D.C. Marinescu, J.E. Lumpp, Jr., T.L. Casavant, and H.J. Siegel, "Models for Monitoring and Debugging Tools for Parallel and Distributed Software", *J. Parallel and Distributed Computing*, Vol.9, June 1990, pp.171-183.

[14] A. Mink, R. Carpenter, G. Nacht, and J. Roberts, "Multiprocessor Performance-Measurement Instrumentation", *Computer*, Vol.23, No.9, Sept. 1990, pp.63-75.

[15] A.K. Mok, "A Graph-Based Computation Model for Real-Time Systems", *Proc. IEEE Parallel Processing*, August 1985, pp.619-623, .

[16] A.K. Mok, "Annotating Ada for Real-Time Program Synthesis", *Proc. Computer Assurance*, 1987

[17] A.K. Mok and G. Liu, "Early Detection of Timing Constraint Violations", *Technical Report*, Real-Time System Lab, Department of Computer Sciences, The University of Texas at Austin, 1997

[18] K. Nilsen, "Issues in the Design and Implementation of Real-Time Java", Iowa State University: Ames, Iowa, 1995.

[19] K. Nilsen, "Real-Time Java", Iowa State University: Ames, Iowa, 1996.

[20] K. Nilsen, "Embedded Real-Time Development in the Java Language", Iowa State University: Ames, Iowa, 1996.

[21] B. Plattner, "Real-Time Execution Monitoring", *IEEE Trans. Software Eng.*, Vol.SE-10, No.6, Nov. 1984, pp.756-764.

[22] S.C.V. Raju, R. Rajkumar, and F. Jahanian, "Monitoring Timing Constraints in Distributed Real-Time Systems", *Proc. Real-Time Systems Symp.*, 1992, pp.57-67.

[23] J.D. Schoeffler, "A Real-Time Programming Event Monitor", *IEEE Trans. Education*, Vol.31, No.4, Nov. 1988, pp.245-250.

[24] R. Snodgrass, "A Relational Approach to Monitoring Complex Systems", *ACM Trans. Computer Systems*, Vol.6, No.2, May 1988, pp.157-196.

[25] H. Tokuda, M. Kotera, and C.W. Mercer, "A Real-Time Monitor for a Distributed Real-Time Operating System", *Proc. ACM Workshop Parallel and Distributed Debugging*, 1988, pp.68-77.

[26] J.J.P. Tsai, K.-Y. Fang, and H.-Y. Chen, "A Noninvasive Architecture to Monitor Real-Time Distributed Systems", *Computer*, Vol.23, No.3, Mar. 1990, pp.11-23.

[27] J.J.P. Tsai and S.J.H. Yang, eds, "Monitoring and Debugging of Distributed Real-Time Systems", IEEE CS Press, Los Alamitos, CA, 1995.