Early Detection of Timing Constraint Violation at Runtime *

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

As real-time applications become more complex and distributed, monitoring for timing constraint compliance becomes more important in facilitating the enforcement of conditional guarantees and for recovery purposes. In [1] Chodrow et al described a $O(n^3)$ satisfiability checking algorithm for timing constraint monitoring at each check point, where n is the number of time terms in the timing constraint specification. In this paper, we show that a timing violation can be caught as early as possible by deriving and monitoring a minimum set of timing constraints from the timing constraint specification. We show that only $O(n)$ time is needed in the worst case for checking at each check point. An implementation based on the results reported herein appears in a companion paper[8].

1 Introduction

Real-time systems are systems for which the violation of a timing constraint is a significant system event which must be reported so that recovery actions may be taken. In this paper we shall investigate the algorithmic problems involved in detecting a timing constraint violation as early as possible at run time. Our approach starts with a specification language based on Real Time Logic (RTL) to define the timing constraints to be monitored. In general, the direct monitoring of the explicitly specified timing constraints may not detect a violation at the earliest possible time, and there is the question of how much information can be inferred from the timing constraint specification so as to facilitate early detection of a violation. By exploiting the "hidden" timing constraints, i.e., timing constraints inferred from the explicitly specified ones, we show that timing constraint violations can be caught as early as possible by monitoring the hidden constraints. Most hidden timing constraints can be determined before run time while others have to be derived at run time. We shall show that a smallest set of timing constraints exists for a given timing constraint specification such that any timing constraint violation will be caught as soon as it happens by checking the constraints in such a set. Furthermore, we show that timing violation detection takes in the worst case $O(n)$ time, where n is the number of time terms appearing in the timing constraint specification. We also show that timers are generally necessary for early detection of timing constraint violations. We prove that the memory requirement for our monitoring algorithm, including the number of outstanding timers is bounded and an upper bound can be determined at compile time. The implementation of our monitoring algorithm in the Java environment was reported in [8].

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

1.1 Related Work

Significant work has been done in recent years on monitoring real-time systems [8, 12]. However, most of these real-time monitoring work mainly addresses various problems in real-time 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 been relatively scarce. In [7], 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 [4], certain predefined timing constraints can be monitored for violation. The work closest to our research is [1] 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 [9], the model of [1] is extended to distributed systems to study the problem of minimizing the number of messages need to be exchanged between the processors in the target distributed system in order to detect any timing constraint violation as early as possible. Our approach is heavily influenced by the work in [1, 7, 9]. 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.

The rest of this paper is organized as follows: Section 2 defines the event model and the target system model in our monitoring approach and describes our timing constraint specification approach. Algorithms for timing constraint compilation and violation detection are discussed in Section 3 and Section 4, respectively. Section 6 is the conclusion.

2 Timing Constraint Specification Approach

2.1 Events

The objects to be monitored are event occurrences in the execution of a real-time system. Informally, *events* represent state changes of interests that may occur in an application. For example, "received a message from process A", "start the execution of function foo()" may be events deemed to be important by the system designer. We adopt the event model first proposed in [2].Events are a finite set of names specified by the users. We shall not concern ourselves with the specific syntactic rules in application programs for defining the occurrence of events which are generally *recurrent*, i.e., an event may occur multiple times during a computation. 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*.

An event occurrence function, denoted by @, is defined to represent the relationship between event instances and their occurrence time during a computation[2].

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.

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*

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

[1, 9] extends the @-function to represent occurrence time 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*

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

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 1 $@(e, 4, 1)$ *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 Timing Constraint Specification

A timing constraint is an assertion which relates the time of occurrences of different event pairs. With the help of @-function and relative @-function, the timing constraints between any two event instances can be expressed as inequalities between 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 of the @-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 2 *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. $@(e, 3) \geq 4$ means the third instance of event e should not occur earlier than time 4.*

As we will see in Section 3, the reason for the restriction on the reference time of relative @-function in a simple constraint is to keep down the constraint violation detection overhead.

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 $@(e_1, i) + 10 \geq @(e_2, \%2)$.

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. We disallow existential quantifiers for now but may consider them in future work. Conjuncts of simple constraints in such formulas will be called *constraint conjunctions*.

Example 3 *Consider a system where a server broadcasts messages to two clients. The following constraint conjunction*

$$@(ClientARecv, i) + \delta \geq @(ClientBRecv, i) \wedge$$
$$@(ClientBRecv, i) + \delta \geq @(ClientARecv, i)$$

asserts that the two clients, A and B, should receive the same messages at approximately the same time, i.e., their receiving time deviation should be within δ time units. The server may be subject to another constraint

$$@(ServerSend, i) + \Delta \leq @(ServerSend, i + 1)$$

which restricts the server's maximum message broadcast rate to be $\frac{1}{\Delta}$.

3 Compiling Timing Constraint Specification

3.1 Catching Violation Early is NOT Trivial

Now that we have defined the class of timing constraints to be monitored, the next question is how to detect violations. A naive solution is to evaluate each of the formulas with respect to the event occurrence history at every event occurrence. As we have mentioned in Section 1, one of our goals is to detect violation as soon as possible, or at least within certain latency requirements, say, D time units. The naive approach of simply checking at every event occurrence does not in general guarantee detection latency bounds. For example, the 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! Waiting on its occurrence to do the checking may miss the violation altogether. Intuitively, a timer is needed in this case to catch the violation as early as possible.

Performing the checking periodically is also not a good idea. To meet the violation detection delay bound D, the checking period should be less than or equal to D, which is usually very short. This may lead to high monitoring overhead.

Furthermore, the existence of *implicit* constraints, which are derived from constraint conjunctions, makes the detection problem even more complicated.

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 4 *Given constraint conjunction*

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

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 is necessary for 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

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

cannot be violated before both of the two related explicit constraints. Implicit constraints of this type are unnecessary to check because they will not help us catch violations earlier.

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

Definition 7 *Given constraint conjunction C, let SC be the set consisting of all explicit and implicit constraints of C. $\forall S \subseteq SC$, let $C' = \wedge_{c_i \in S} c_i$. We call S a* ***useful constraint set*** *of C if and only if at any time t during a computation, if C' is satisfied, so is C, and if C is violated, so is C'. A useful constraint set S_e of C is called* ***minimum*** *if $\forall$ useful constraint set S, $|S_e| \leq |S|$.*

Theorem 1 *Given constraint conjunction C, let S_1 be a useful constraint set of C and $S_2 = S_1 - \{c\}$, where c is an unnecessary constraint in S_1. Then S_2 is a useful constraint set of C.*

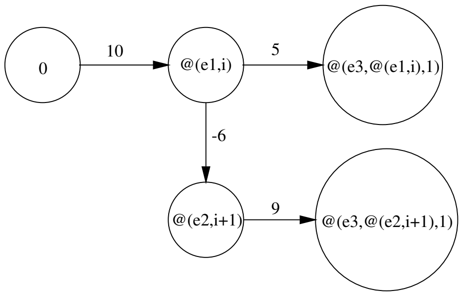

Figure 1. A Constraint Graph

Proof: Let $C_1 = \wedge_{c_i \in S_1} c_i$ and $C_2 = \wedge_{c_j \in S_2} c_j$. Obviously $C_1 = C_2 \wedge c$. At any time t, if C_2 is satisfied, according to Definition 6, we know c is satisfied. So C_1 is satisfied at time t, which leads to the satisfaction of C. On the other hand, if C is violated at time t, we know C_1 must have been violated, which means either C_2 or c have been violated by t. If c is violated at time t, so is C_2, according to the Definition 6. So in either case, C_2 is violated at time t. According to Definition 7, we know S_2 is a useful constraint set of C. □

3.2 Constraint Graph

By denoting the terms in simple constraints by vertices, we can represent a simple constraint as a weighted, directed edge and a constraint conjunction as a directed graph, called *constraint graph*, as shown in Figure 1. Therefore, 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 time 0. Also note that the two $@(e_3, \%1)$ in the above conjunction are shorthands for $(e_3, @(e_1, i), 1)$ and $@(e_3, @(e_2, i+1), 1)$, respectively. So they are represented as two different vertices in the constraint graph.

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 $@(e, \%1)$, *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.

Definition 8 *Given a constraint graph G, we define the constraint set of G,* $CS(G) = \{c \mid c$ *is the constraint represented by edge e,* $\forall e$ *in* $G\}$.

Given a constraint graph $G = (V, E)$, we say that there is a *path* from vertex u to v if there exists a set of vertices $X = \{x_i | x_i \in V, x_1 = u, x_n = v, i = 1, \ldots, n, n \geq 2\}$ such that for $i = 1, \ldots, n-1$, edge $(x_i, x_{i+1}) \in E$. Vertex u is called the *source* of the path and v is called the *destination*. The *length* of the path is the sum of the weights on all edges along the path. Obviously, each path in a constraint graph corresponds to a constraint, either explicit or implicit, of the constraint conjunction. A path with at least one intermediate vertex represents an implicit constraint. For instance, in Figure 1 the path from the *zero* vertex to vertex $@(e_2, i+1)$ with length 4 actually represents the implicit constraint $@(e_2, i+1) \leq 4$, derived from $@(e_1, i) - 6 \geq @(e_2, i+1)$ and $@(e_1, i) \leq 10$. If we add edges in G for all these paths, e.g. adding edge $u \xrightarrow{l} v$ in G if such an edge does not exist in G and there is a path from u to v with length l, the resulting constraint graph G' is called a *full* constraint graph of C.

A path whose source and destination are the same vertex is a *cycle*. A cycle with a negative length is called a *negative cycle*. Since a negative cycle with length L essentially represents an unsatisfiable timing constraint $t + L \geq t$, where t is the time corresponding to the source vertex and $L < 0$, we have the following theorem:

Theorem 2 *If a negative cycle exists in a constraint graph, then the corresponding constraint conjunction is unsatisfiable.*

Proof is trivial and therefore is omitted here.

Corresponding to variables that appear within constraint conjunctions, we define *instantiated constraint conjunctions* as follows:

Definition 9 *Given constraint conjunction C with a variable v, if we instantiate the variable v and/or some of the relative @-functions in C, the resulting constraint conjunction C′ is called an* ***instantiated*** *constraint conjunction of C. If G is the constraint graph corresponding to C, then the constraint graph representing C′ is called an* ***instantiated*** *constraint graph of G.*

Example 5 $@(e_1, 1) + 5 \geq @(e_2, 1) \ \wedge \ @(e_2, 1) + 6 \geq @(e_3, 7)$ *is an instantiated constraint conjunction of* $@(e_1, i) + 5 \geq @(e_2, i) \wedge @(e_2, i) + 6 \geq @(e_3, \%5)$ *when* $i = 1$ *and* $\#(e_3, @(e_2, 1)) = 2$.

3.3 Compiling Constraint Graphs

In [1] 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 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 need in the worst case at each check point.

Since a path between 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 $S = \{c_p\} \cup CS(G)$. And similarly, a path is *unnecessary* if its corresponding constraint is unnecessary in S. We make the following observations about unnecessary paths:

Theorem 3 *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.*

Proof: Assume there is another path p' from u to v with a longer length l', i.e. $l' > l$. Let t_u and t_v be the time points represented by u and v in a computation. The constraints represented by p and p' are $t_u + l \geq t_v$ and $t_u + l' \geq t_v$, respectively. Since $l' > l$, we know at any time

$$\begin{array}{lll} t_u + l \geq t_v & \Longrightarrow & t_u + l' \geq t_v \\ t_u + l' \leq t_v & \Longrightarrow & t_u + l \leq t_v \end{array}$$

According to Definition 6, $t_u + l' \geq t_v$ is an unnecessary constraint. p' is an unnecessary path. □

Corollary 3.1 *Positive cycles are unnecessary paths.*

Proof is trivial and therefore is omitted here.

Theorem 4 *Given a constraint conjunction C, let $G = (V, E)$ be its corresponding full constraint graph and $G' = (V, E')$ be the constraint graph where $\forall u, v \in V, u \neq v$, edge $u \xrightarrow{l} v \in E'$ if and only if l 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.*

Proof: Let S_0 be the constraint set which consists of all the constraints of C, either explicit or implicit. Obviously each constraint in S_0 corresponds to an edge in G and $S \subseteq S_0$. Assume $S' = \{c_i \mid c_i \in S_0 \wedge c_i \notin S,\ i = 1, \ldots, k\}$, and $S_i = S_{i-1} - \{c_i\}$, for $i = 1, \ldots, k$. Then $S = S_k$. According to Theorem 3, c_i is an unnecessary constraint in S_{i-1}. Since S_0 is a useful constraint of C, from Theorem 1 we know S_i is a useful constraint set of C, for $i = 1, \ldots, k$. Therefore, $S = S_k$ is a useful constraint set of C. □

According to Theorem 4, the shortest paths between vertex pairs in a constraint graph consist of a useful constraint set of C. However, not all of these shortest paths represent necessary constraints. Theorem 5, Theorem 6 and its corollaries will be used to help decide whether such a shortest path corresponds to a necessary constraint.

Lemma 1 [1] *Consider constraint graph $G = (V, E)$. $\forall u, v \in V$ such that $(u, v) \in E$, let t_u, t_v be time points in a computation denoted by vertices u, v, and let d be the weight on edge (u, v). If u is not a relative vertex, then for the constraint represented by edge (u, v), which is $t_u + d \geq t_v$, the earliest time at which this constraint can be violated is:*

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

b. *t_u if $d < 0$;*

As an example, for constraint $@(e_1, i) + 10 \geq @(e_2, i)$, the earliest time it can be violated is 10 time units after the ith occurrence of event e_1. And the earliest time at which the constraint $@(e_1, i) - 5 \geq @(e_2, i)$ can be violated is the occurrence time of the ith instance of event e_1.

Lemma 2 *Consider a constraint graph $G = (V, E)$. $\forall u, v \in V$ such that $(u, v) \in E$, let t_u, t_v be time points denoted by vertices u, v in a computation and d be the weight on edge (u, v). If u is a relative vertex, then the earliest violation detection time for the constraint represented by edge (u, v) is*

a. *t_v if $d \geq 0$;*

b. *t_u is $d < 0$.*

Proof: Since t_u is unknown until the time=t_v, there is no way to tell if the constraint is violated or not before t_v. Consider the following two cases:

a. $d \geq 0$: If u is a future relative vertex, then the constraint is always true. Assume u is a past relative vertex. Then by time=t_v, t_u is already known. We can tell whether the constraint is violated or not by comparing $t_u + d$ and t_v at time t_v.

b. $d < 0$: If u is a past relative vertex, then the constraint is unsatisfiable. Let us assume u is a future relative vertex. Then the constraint is violated if the event instance corresponding to vertex u occurs before the time $t_v - d$. So the earliest possible violation time is the time at which this event instance occurs, i.e., at t_u. □

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*

[1] [9] has a similar observation. See [8, 9] for the proof.

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

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

Proof: From Lemma 1 we know this is true when u is not a relative vertex. If u is a relative vertex, then from Lemma 2 we know its earliest violation detection time is t_v ($t_v > t_u + d$) when $d \geq 0$ and t_u when $d < 0$. So the theorem also holds in this case. □

Theorem 6 *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* $p = u \rightarrow v \rightarrow w$*, then* p *is unnecessary.*

Proof: Let t be the earliest possible violation detection time for the implicit constraint corresponding to the path $u \rightarrow v \rightarrow w$. Then according to the condition, we have $t_v \leq t$. If the implicit constraint, which can be expressed as $t_u + d_1 + d_2 \leq t_w$, is violated at time $t' \geq t$, then according to Theorem 5, we have $t_u \leq t \leq t'$. Since t_u, t_v are known at time t', we can decide whether the constraint $t_u + d_1 \geq t_v$ is violated. If it is violated, then for the implicit constraint, one of its explicit constraints has been violated before its own violation. Otherwise, we know $t_u + d_1 \geq t_v$. Since the implicit constraint is violated at time t', we have $t_w > t_u + d_1 + d_2 \geq t_v + d_2$, i.e., the constraint $t_v + d_2 \geq t_w$ has been violated by time t'.

So by the time the implicit constraint is violated, at least one of its explicit constraints has already been violated. According to Definition 6, this implicit constraint is unnecessary. □

Corollary 6.1 *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.*

Proof: Since a relative vertex is only connected to its reference vertex in a constraint graph, any path to or from a relative vertex must also pass its reference vertex. Let vertex u be v's reference vertex and t_u be the time represented by vertex u. Consider any vertex $w \in V$ such that there is a path from w to v via u or from v to w via u. Since the event instance corresponding to vertex v will not be determined until t_u, any violation of the implicit constraint between w and v cannot be detected before time t_u. From Theorem 6, we know the path is unnecessary. □

Corollary 6.2 *Consider 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.*

Proof: Without loss of generality, assume $p = u \overset{d_1}{\rightarrow} v \overset{d_2}{\rightarrow} w$, where $u, v, w \in V$, $(u, v), (v, w) \in E$. Let t_u, t_v, t_w be the time points represented by vertices u, v, w, respectively in a computation. We have the following constraints:

$$\begin{aligned} \textit{edge } (u, v) &: \quad t_u + d_1 \geq t_v & (1) \\ \textit{edge } (v, w) &: \quad t_v + d_2 \geq t_w & (2) \\ \textit{path } p &: \quad t_u + d_1 + d_2 \geq t_w & (3) \end{aligned}$$

Suppose the earliest violation detection time for constraint (1) is t_1, for constraint (2) is t_2 and for constraint(3) is t. Since $d_1 < 0$, from Theorem 5 we know $t_1 = t_u$ and $t \geq t_1$. If t_v is unknown at time t, then the constraint (1) has been violated by time t. If t_v is known at time t and constraint (3) is violated at time t, then from the proof of Theorem 6 we know at least one of the constraints (1) and (2) has been violated by t. So in either case, at least one of the constraints (1) and (2) is violated before constraint (3) is. Path p must be unnecessary. □

Corollary 6.3 *Consider 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 no bigger than the length of* p*, then* p *is unnecessary.*

The proof is similar to that of Corollary 6.2 and therefore is omitted here.

For a *full* constraint graph G of constraint conjunction C, Theorem 4 tells us only those edges whose lengths equal to the shortest path lengths from their sources to destinations may be *necessary*. A subgraph G_s of G whose edge set consists of all such edges in G is called *the shortest path* constraint graph of C. Obviously the edge set of G_s corresponds to a *useful* constraint set of C. Furthermore, Corollaries 6.1∼6.3 can be used to identify the unnecessary paths in G_s. In fact, as long as G_s has no *zero cycles*, i.e. cycles with zero lengths, the following theorem enables us to get a *minimum useful* constraint set of C by removing all those unnecessary paths in G_s.

Theorem 7 *Given the shortest path constraint graph* G_s *of constraint conjunction* C*, let* G_m *be the remaining subgraph of* G_s *after all those unnecessary edges identified by Corollaries 6.1∼6.3 are removed. If* G_s *has no zero-cycle in it, then the constraint set* $S_m = CS(G_m)$ *is a minimum useful constraint set of* C*.*

Proof: We first prove that S_m is a *useful* constraint set of C. Assume $G_0 = G_s = (V, E_0)$, $G_i = (V, E_i)$, where $E_i = E_{i-1} - \{e\}$, e is an unnecessary edge in G_{i-1} identified by Corollaries 6.1∼6.3, for $i = 1, \ldots, k$, and G_k has no further unnecessary edges that can be identified by Corollaries 6.1∼6.3. Let $G_m = (V, E_m)$. Obviously $E_m \subseteq E_k$.

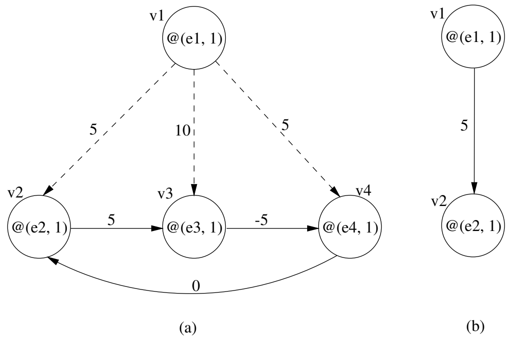

Figure 2. Constraint Graph with Zero-Cycles

Suppose $\exists$ edge $e : u \xrightarrow{l} v$, where $u, v, \in V, e \in E_k$ and $e \notin E_m$. Then $\exists$ edge $e' : u \xrightarrow{l'} w$ such that $w \in V, w \neq v$, e' is on one of the shortest path from u to v in G_0 and $l' < 0$ or $l' \leq l$. Since there is no zero-cycle in G_s, e cannot be on any of the shortest paths from u to w in G_0. Therefore, either $e' \in E_k$, or $e' \notin E_k$ but $\exists$ edge $e'' : u \xrightarrow{l''} x$ such that $x \neq w$, e'' is on one of the shortest paths from u to w in G_k and $l'' < 0$ or $l'' < l'$. Either case makes e unnecessary in G_k, which means $e \notin E_k$, a contradiction. So $E_k = E_m$, in other wards, $G_k = G_m$. According to Theorem 1, $CS(G_i)$s are useful constraint sets of C. Hence, $S_m = CS(G_m) = CS(G_k)$ is a useful constraint set of C.

Next we prove that S_m is minimum. $\forall$ minimum useful constraint set S of C, let $G' = (V, E')$ be the constraint graph corresponding to S. According to Theorem 3, G' is a subgraph of G_s. Suppose $\exists$ edge $e \in E_m$ but $e \notin E'$. Since e is unnecessary in G' and G' is a subgraph in G_s, e can be identified as unnecessary in G_s, i.e. $e \notin E_m$, a contradiction. $\square$

When zero-cycles exist in G_s, as shown in Figure 2(a), removing all *unnecessary* paths in G_s (e.g., dashed edges in Figure 2(a)) does not necessarily produce a *useful* constraint set of C. However, since a zero-cycle represents a set of constraints which assert the exact distances between event occurrences, we can replace the zero-cycle with the vertex representing the earliest event occurrence in the cycle and update edges to and from other vertices in the cycle accordingly. For example, the zero-cycle in Figure 2(a), $v_2 \xrightarrow{5} v_3 \xrightarrow{-5} v_4 \xrightarrow{0} v_2$, represents constraint set $S_0 = \{@(e_2, i)+5 = @(e_3, i), (e_2, i) = @(e_4, i)\}$. v_2 or v_4 represent the earliest occurrence in this cycle. We call them *starting vertices* of the zero-cycle. Figure 2(b) shows the new graph after replacing the zero-cycle in Figure 2(a) with v_2, one of its starting vertices, and updating edges $v_1 \xrightarrow{10} v_3$ and $v_1 \xrightarrow{5} v_4$ in Figure 2(a) to edge $v_1 \xrightarrow{5} v_2$. In fact, the constraints represented by those two edges are unnecessary for the constraint set $S_0 \cup \{c\}$, where c is the constraint for edge $v_1 \xrightarrow{5} v_2$.

With these observations, we can get a minimum useful constraint set for constraint conjunction C by performing the following three steps:

a. Search the shortest paths between all pairs of vertices in the constraint graph of C to eliminate those unnecessary paths identified in Theorem 3; if a negative cycle is found, then the constraint conjunction is unsatisfiable, exit;

b. Search for zero-cycles, replace them with their *starting vertices* and update the graph accordingly;

c. Check the shortest paths in the updated graph to eliminate those unnecessary implicit constraints identified by Corollaries 6.1$\sim$6.3.

Assume G_m is the remaining constraint graph after step (c), Z_is are zero cycles replaced in step (b), $i = 1, \ldots, k$. Let $CZ_i = \cup_k c_{ik}$, where c_{ik} is the constraint represented by edge (s_i, v), $\forall v$ in Z_i and s_i is the starting vertex of Z_i. With Theorem 7, we can conclude that the constraint set $CS(G_m) \cup \cup_{i=1}^{k} CZ_i$ is a minimum constraint set of C.

4 Monitoring Timing Constraints

4.1 Detecting Violation of Simple Constraints

Consider a simple constraint $T_1 + d \geq T_2$. If T_1 is not the occurrence time of a relative event instance, then according to Lemma 1, we need to check satisfiability at time T_1 to ensure that a violation will be caught as early as possible. If T_1 is the occurrence time of a relative event instance, then according to Lemma 2, satisfiability needs to be checked at time T_2. From our discussion in Section 3.1, we know that timers are in general necessary for detecting violations at the earliest possible time. Specifically, if we check constraint $T_1 + d \geq T_2$ at time T_1 where $d \geq 0$ and T_2 is undefined at this time, then a watchdog timer that expires at $T_1 + d$ is needed to set a deadline for monitoring the event instance et_2 that defines T_2. If et_2 occurs before the timeout, then the constraint is satisfied and the deadline timer can be canceled. Otherwise the constraint is violated when the timer expires. On the other hand, if we check the constraint at time T_2 for the case $d < 0$ and T_1 is the occurrence time of an event instance et_1 that has not occurred yet, then a delay timer to expire at $T_2 - d$ is needed to set a delay for monitoring the occurrence of et_1 . If et_1 occurs before the timeout, then the constraint is violated at the occurrence of et_1. Otherwise the constraint is satisfied when this delay timer expires. The detailed algorithm for checking a simple constraint $c = (T_1 + d \geq T_2)$ is described [8].

4.2 Detecting Violation of Constraint Conjunctions

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 violations

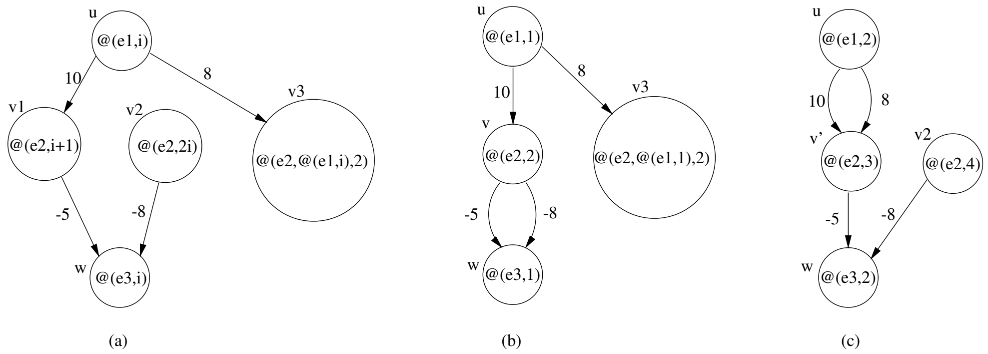

Figure 3. Equivalent vertices

as early as possible. Although our compilation procedure in Section 3.3 produces a minimum useful constraint set for constraint conjunction C, some implicit necessary constraints may remain unidentified for some *instantiated* constraint conjunctions of C, owing to the fact that some vertices in the compiled constraint graph may become the same vertex in some of its instantiated graphs and hence may give rise to additional necessary implicit constraints. We call these vertices *equivalent* because they represent the occurrence of the same event instance in an instantiated constraint graph. 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 3(a) and 3(b) show such an example. Vertices v_1 and v_2 in the constraint graph of Figure 3(a) become one vertex v in the instantiated constraint graph of Figure 3(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 3(a) become vertex v' in the instantiated constraint graph of Figure 3(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 other vertices in an instantiated constraint graph. The following theorem helps us identify the new necessary paths in such cases.

Theorem 8 *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_v in a computation, the vertices 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' that is caused by this equivalence is unnecessary unless v_r is a future relative vertex and p's source or destination is v.*

Proof: Suppose p's source is u and its destination is w. Since p is the new shortest path from u to w as a result of the merging of v_r and v_e, and v is the only vertex in G that connects with v_r, we can conclude that $v, v_r(v_e)$ must be on path p to result in a shorter distance from u to w. There are three cases.

- v is an intermediate vertex on path p: Because p has been unknown until time t_v, any violation of its corresponding implicit constraint cannot be detected before t_v. According to Theorem 6, this implicit constraint is unnecessary, i.e., p is unnecessary.

- $u = v$: In this case, $(v, v_r) \in E$. Without loss of generality, assume $(v_e, w) \in E$. If v_r is a past relative vertex, which means that we already know the time represented by $v_r(v_e)$ at time t_v, then the path

p is unnecessary according to Theorem 6. If v_r is a future relative vertex, then p may be necessary (e.g., $p = v \overset{10}{\rightarrow} v_r(v_e) \overset{-9}{\rightarrow} w$) because it is possible that the implicit constraint corresponding to path p is violated before the event instance of $v_r(v_e)$ occurs.

- $w = v$: Similar arguement as in case $u = v$ can be applied here to prove that p may be necessary only if v_r is a future relative vertex. □

When a relative vertex becomes equivalent with other vertices in an instantiated constraint graph, Theorem 8 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. The update algorithm is described in [8].

Now that we can identify all the necessary paths in the instantiated constraint graphs, we can check satisfiability by checking the necessary constraints at their corresponding check points, using the checking method described in Section 4.1. If a necessary constraint is violated, the corresponding instantiated constraint conjunction is violated. If all these constraints are satisfied, then the corresponding instantiated constraint conjunction is satisfied. A detailed description of this satisfiability checking algorithm is given in [8].

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

4.3 Event History Length

We need to maintain an event occurrence log for each event involved in the checking algorithm and we do not want to 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 our checking will never miss any violations. We can indeed determine an upper bound for the necessary length of each event log at compile time.

Lemma 3 *Consider a constraint* $c = @(e_1, f(i)) + d \geq @(e_2, g(i))$*, where* $f(i)$ *and* $g(i)$ *are functions of variable* i*. The event logs for* e_1 *and* e_2 *must satisfy the following properties at any time during the violation detection:*

- *at least the most recent occurrences must be kept in the event logs of events* e_1 *and* e_2*;*
- *when* $d > 0$ *: remembering the event occurrences of* e_1 *in the past* d *time units is sufficient for checking constraint c;*
- *when* $d < 0$ *: remembering the event occurrences of* e_2 *in the past* $|d|$ *time units is sufficient for checking constraint c.*

Proof: According to our checking algorithm, we know this constraint is checked at either the $f(i)th$ occurrence of event e_1 or the $g(i)th$ occurrence of e_2. Consider the following two cases:

- $d > 0$: If the constraint is checked at time $@(e_1, f(i))$, then no previous instance of e_2 except the most recent one is needed because we only need to know whether the $g(i)th$ instance of e_2 has occurred by this time. If the constraint is checked at time $@(e_2, g(i))$ and the $f(i)th$ instance of e_1 has occurred, then we need to know whether the occurrence of the $f(i)th$ instance of e_1 is before time=$@(e_2, g(i)) - d$. By the monotonic property of event occurrences, we can decide whether the constraint is violated by comparing $\#(e_1, @(e_2, g(i)) - d)$ and $f(i)$. So keeping those event instances of e_1 which occur during the previous d time units is sufficient.
- $d < 0$: Similar arguement as those in the case $d > 0$ can be applied here and prove that the log of e_2 only needs to cover the previous $|d|$ time units. □

Definition 10 *Given a constraint graph G, we say that an event e is in G if there exists at least one vertex* $v_e \in V$ *which denotes the occurrence time of e.*

Theorem 9 *Given a constraint graph G, the necessary log length of any event in G is bounded.*

Proof: Suppose e is an event in G and the maximum occurrence rate of e is r. Since we can get an upper bound ub_e at compile time on the lengths of all the necessary paths that end at any absolute vertices of e in G and a lower bound lb_e on the lengths of all the necessary paths that start from any absolute vertices of e in G. Then according to Lemma 3, an upper bound $d \cdot r$ on the necessary log length of event e for G can be determined at compile time, where $d = \max(ub_e, -lb_e)$. □

4.4 Outstanding Timers

Since timers may be set during runtime monitoring of the timing constraints, as is described in Section 4.1, we need to make sure that the number of outstanding timers does not grow unbounded during run time. Similar to the case of event history length, Theorem 10 tells us that we can also determine an upper bound for this number at compile time.

Lemma 4 *Consider a constraint c in the form:* $T_1 + d \geq T_2$ *where* T_1 *denotes an occurrence time term of event* e_1*,* T_2 *denotes an occurrence time term of event* e_2*. Suppose that* *the maximum occurrence rate of event e_1 and e_2 are r_1 and r_2. At time t, let $tnum(c,t,e_1), tnum(c,t,e_2)$ be the numbers of outstanding timers that have been set during the checking of the constraint c before t owing to the occurrences of event e_1 and event e_2, respectively. Then the following properties must hold:*

- *If $d \geq 0$: $tnum(c,t,e_1) \leq d \cdot r_1$ and $tnum(c,t,e_2) = 0$;*
- *If $d < 0$: $tnum(c,t,e_1) = 0$ and $tnum(c,t,e_2) \leq |d| \cdot r_2$.*

Proof: If $d \geq 0$, then whenever an instance of event e_1 occurs, a deadline timer need to be set for the corresponding instance of e_2 if it has not yet occurred. In the worst case, during the period $[t-d,t)$ event e_1 occurs at its maximum rate but none of the corresponding instances of e_2 has occurred. So in the worst case $tnum(c,t,e_1)$ is $d \cdot r_1$. But whenever an instance of e_2 occurs, no timer need to be set for e_1. Therefore $tnum(c,t,e_2) = 0$.

Similar argument can be applied when $d < 0$. □

Theorem 10 *Consider a constraint graph G. At any time during the runtime monitoring of G, the number of outstanding timers set for all events in G is upper bounded. Such an upper bound can be determined at the compile time of G.*

Proof: $\forall$ event e in G, at the compile time we can decide a set of those constraints that may be checked at the occurrence of e. According to Lemma 4, at any time t, we can get an upper bound of the outstanding timers that are set because of the occurrence of e. Therefore at time t, an upperbound for all outstanding timers set during the violation detection of G can be decided at the compile time. □

5 Conclusions and Future Work

In this paper we have investigated the algorithmic problems involved in the detection of a timing constraint violation at the earliest possible time. 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 our monitoring algorithm is bounded and an upper bound can be determined at compile time.

There are many other research issues that need to be pursued to extend the utility and efficiency of our real-time monitoring algorithms. 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 simple constraint, we restrict its reference time to be the other @-function in the constraint. What is the impact to our monitoring algorithms and memory requirements if we relax this restriction?

References

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

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

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

[4] 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.

[5] 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.

[6] 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.

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

[8] A.K. Mok and G. Liu, "Efficient Run-Time Monitoring of Timing Constraints", *Technical Report*, Real-Time System Lab, Department of Computer Sciences, The University of Texas at Austin, 1997, an abridged version to appear in *Proc. Third IEEE Real-Time Technology and Applications Symposium*, June, 1997.

[9] 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.

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

[11] 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.

[12] 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.