21 Oct 1993

A Run-Time Decision Procedure for Responsive Computing Systems

Grace Tsai

Matt Insall
Missouri University of Science and Technology, insall@mst.edu

Bruce M. McMillin
Missouri University of Science and Technology, ff@mst.edu

A RUN-TIME DECISION PROCEDURE FOR RESPONSIVE COMPUTING SYSTEMS[1]

Grace Tsai, Matt Insall[2] and Bruce McMillin[3]

CSC-93-029

[1]This work was supported by UM research board, the National Science Foundation under Grant Numbers MSS-9216479 and CDA-9222827, and from the Air Force Office of Scientific Research under contract number F49620-92-J-0546.

[2]Matt Insall is with the Department of Mathematics and Statistics at the University of Missouri-Rolla, Rolla, MO 65401.

[3]Grace Tsai and Bruce McMillin are with the Department of Computer Science at the University of Missouri-Rolla, Rolla, MO 65401.

Abstract

A responsive computing system is a hybrid of real-time, distributed and fault-tolerant systems. In such a system, severe consequences will occur if the logical and physical specifications of the system are not met. In this paper, we present a logic, Interval Temporal Logic (ITL), to specify responsive systems and give decision procedures to verify properties of the system at run-time as follows. First, we collect, during execution, events occurring in the system to represent a distributed computation. Next, we specify properties of the system using ITL formulas. Finally, we apply the decision procedures to determine satisfaction of the formulas. Thus, we can verify properties of the system at run-time using these decision procedures.

ABSTRACT

A responsive computing system is a hybrid of real-time, distributed and fault-tolerant systems. In such a system, severe consequences will occur if the logical and physical specifications of the system are not met. In this paper, we present a logic, Interval Temporal Logic (ITL), to specify responsive computing systems and give a decision procedure to verify properties of the systems at run-time as follows. First, we specify properties of the system using ITL formulas. Next, we collect, at run-time, events and maintain equivalent event histories to represent system execution. Finally, we apply a decision procedure to determine satisfaction of the formulas. The proposed decision procedure is essentially a run-time procedure which makes use of event histories computed during execution to verify properties of the system in the actual operational environment.

1 INTRODUCTION

A responsive computing system [1] is one which responds to internal programs or external inputs in a timely, dependable and predictable manner. This system is a hybrid of real-time, distributed and fault-tolerant systems. In such a system, any failure can cause a catastrophe, and hence, it is very important to ensure that run-time behavior of the system conforms to its expected behavior (specification). Thus, a logic, Interval Temporal Logic (ITL), is developed to specify the behavior of a responsive computing system, and a decision procedure is given to verify properties of the system at run-time.

With the logical system of ITL, we use *interval formulas* and *responsiveness assertions* to denote the logical specification of a responsive computing system. In the operational environment, we expect that the run-time behavior of the system (the execution history) conforms to its logical specification. Otherwise, an error has occurred. Thus, to verify properties of the system in the operational environment, the following steps are performed:

1. given a responsive computing system R, specify the property that R must hold using an ITL formula S_R,
2. collect events and maintains an equivalent history V_h during execution, and
3. apply the decision procedure $\Pi(V_h, S_R)$ to determine satisfaction of S_R.

A distributed computation can be considered as a set of partially ordered events[4], so an execution history V_h can be obtained by collecting and partially ordering events occurring in the system. This execution history V_h denotes run-time behavior of a system, and hence it is expected to conform to the logical specification S_R of the system. Thus, at step 3, we apply the procedure Π to check whether the specification S_R is satisfied by the history V_h. Since the history V_h is built at run-time, the decision procedure Π is essentially a run-time procedure which checks satisfaction of ITL formulas in the operational environment.

In our previous work, we used a decision procedure to check satisfaction of liveness assertions in the operational environment [2]. A liveness assertion $(\phi \rightarrow \mathrm{EF}\psi)$ denotes

[4] An event can be modeled as execution of one statement or a set of statements.

that when a program starts from a state satisfying assertion ϕ, *eventually* it will get to a state satisfying assertion ψ. This kind of assertions can not describe properties that must hold within bounded intervals and hence is not suitable for responsive computing systems. Thus, this paper focuses on constructing a decision procedure Π to check, at run-time, satisfaction of responsiveness assertions and interval formulas which can assert properties of the systems within bounded intervals of time.

For the determination of satisfaction of formulas, [3] translates temporal logic formulas into finite automata. In contrast, we establish correspondence between states and events in the history and examine the history for the determination of satisfaction of formulas. The work of [4] embeds system constraints into programs and examines them at run-time. However, they use a centralized monitor to obtain an execution history, while our method does not require monitors to compute execution histories.

The organization of this paper is as follows. In Section 2, we introduce the logic ITL. Next, we provide an approach to building execution histories, and present the decision procedure Π to determine satisfaction of ITL formulas at run-time. In Section 4, we give an example on constructing execution histories and applying the decision procedure for the determination of satisfaction of ITL formulas at run-time. Section 5 concludes this paper.

2 INTERVAL TEMPORAL LOGIC

This section presents a logic, Interval Temporal Logic(ITL), for the specification of responsive computing systems. The logic ITL is an extension of Interleaving Set Temporal Logic ([5], [6]). This logic ITL adopts a partial order semantics which considers a distributed computation as a set of partially ordered events. Hence, it can capture temporal and distributed aspects of the responsive systems that we are modeling.

Syntax

The logic ITL is built upon a classical predicate logic L as follows. The symbols of ITL are those of classical predicate logic along with operators, U, E, F, X, X^n, and $\hat{\mathrm{X}}$. Let Φ be the smallest set of words over the symbols of ITL such that

- If $p \in L$ then $p \in \Phi$.
- If $p, q \in \Phi$, then $p \vee q, \neg p \in \Phi$.
- If $p, q \in \Phi$, then $p\mathrm{U}q, \mathrm{E}p, \mathrm{F}p, \mathrm{X}p, \mathrm{X}^n p, \hat{\mathrm{X}}p \in \Phi$.
- If $p, q, r \in \Phi$, then $[p]r, [p, q]r \in \Phi$.

We call a member of Φ a *path formula*, and a formula of the language L a *state formula.* A path formula can be used to characterize a sequence of states, while a state formula is interpretable only on a single state.

Definition 2.1 *Let $|\sigma|$ denote the length (possible infinite) of a state sequence σ, and let $\sigma(i)$ denote the i^{th} state of a state sequence σ. We call i a* ***time index*** *of σ .*

Definition 2.2 *A state sequence* $\xi = (\xi(0), ..., \xi(|\xi|))$ *refines a given state sequence,* $\sigma = (\sigma(0), \cdots, \sigma(|\sigma|))$, *iff* $(\exists j \in [0, |\sigma|])$ $((\sigma(j), \sigma(j+1), \cdots \sigma(j+|\xi|)) = \xi)$. *Let* $R(\sigma) = \{\xi | \xi \leq \sigma\}$.

From the above, $R(\sigma)$ is a collection of subsequences of the state sequence σ.

Semantics

In this logic system, formulas are quantified by fundamental operators E, F, U, X, X^n , $\hat{X}$, and X^n, which are defined in the following semantics.

Definition 2.3 (Satisfaction) *Let* σ *be a state sequence, let* i *be an integer with* $0 \leq i < |\sigma|$, *let* f *be a state formula, and let* p, q, ϕ, ψ *be any formulas of ITL. Then, we write*

$(\sigma, i) \models f$	if $\sigma(i) \models f$,
$(\sigma, i) \models p \vee q$	if $(\sigma, i) \models p$ or $(\sigma, i) \models q$,
$(\sigma, i) \models \neg p$	if not $(\sigma, i) \models p$,
$(\sigma, i) \models p\mathrm{U}q$	if there exists $i \leq j < \|\sigma\|$, such that both $(\sigma, j) \models q$ and for every k such that $i \leq k < j$, $(\sigma, k) \models p$,
$(\sigma, i) \models \mathrm{X}\phi$	if $\|\sigma\| \geq i+1$ and $(\sigma, i+1) \models \phi$,
$(\sigma, i) \models \mathrm{X}^n\phi$	if $(\sigma, i+n) \models \phi$,
$(\sigma, i) \models \hat{\mathrm{X}}\phi$	if $(\exists n \geq i)((\sigma, i) \models \mathrm{X}^n\phi)$,
$(\sigma, i) \models \mathrm{F}\phi$	if $\exists(k \geq i)(\forall j \geq k)(\sigma, j) \models \phi$,
$(\sigma, i) \models [p]\phi$	if we have that if $(\sigma, i) \models p$ then $(\sigma, i) \models \phi$,
$(\sigma, i) \models [p, q]\phi$	if for every $\xi \in R(\sigma)$ such that $(\xi, 0) \models p$ and $(\xi, \|\xi\|) \models q$, if there exists $k_i \in \{0, 1, ..., \|\xi\|\})(\xi(k_i) = \sigma(i))$, then $(\xi, k_i) \models \phi$,
$\sigma \models \phi$	if for all $i \in 0, ..., \|\sigma\|$, $(\sigma, i) \models \phi$.

In each case, the symbol $\models$ is read "satisfies". We abbreviate $(\neg\phi \wedge \hat{\mathrm{X}}\phi)$ by $\mathrm{E}\phi$.

Informally, an *interval* is of the form $[p]$ or $[p, q]$ and an *interval formula* is of the form $[p]\phi$ or $[p, q]\phi$ where p, q and ϕ are any formulas of ITL. An interval formula $[p]\phi$ ($[p, q]\phi$) is true over a state sequence σ, iff the interval $[p]$ ($[p, q]$) cannot be found or the formula ϕ holds on every interval $[p]$ ($[p, q]$). Thus, there are two ways to conclude that an interval formula holds. This would cause a problem in the composition of interval formulas. Thus, the following *responsiveness assertions* are proposed.

Definition 2.4 *A responsiveness assertion is a path formula of the form* $([p]\phi \rightarrow [p, q]EF\psi)$, *where* p, q, ϕ, *and* ψ *are formulas of ITL.*

A responsiveness assertion $([p]\phi \rightarrow [p, q]\mathrm{EF}\psi)$ is true over a state sequence σ, iff the following holds: if ϕ holds whenever p holds, then ψ will occur at or before any q following p. The assertion ensures bounded response of ψ to $[p]\phi$ within the intervals $[p, q]$. The following Progress Rule can be applied to reason about responsiveness properties. Let $p, q, r, \phi_0, \phi_1, \phi_2$ be any formulas of ITL,

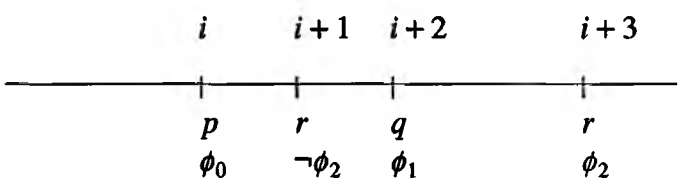

Figure 1: A computation sequence σ fails to satisfy $[p]\phi_0 \rightarrow [p,q]\mathrm{EF}\phi_2$.

Progress Rule

$$\begin{array}{ll} (1) & [p]\phi_0 \rightarrow [p,q]\mathrm{EF}\phi_1 \\ (2) & [q]\phi_1 \rightarrow [q,r]\mathrm{EF}\phi_2 \\ (3) & [p,r]\mathrm{E}q \\ \hline & [p]\phi_0 \rightarrow [p,r]\mathrm{EF}\phi_2 \end{array}$$

From the above, we can conclude that if ϕ_0 holds whenever p holds, then ϕ_2 will occur in intervals $[p,q]$, if the following premises hold:

(1) if ϕ_0 holds whenever p holds, then ϕ_1 will occur at or before any q following p,

(2) if ϕ_1 holds whenever q holds, then ϕ_2 will occur at or before any r following q,

(3) q will occur within the intervals $[p,r]$,

Notice that the premise (3) is necessary as follows. Consider a state sequence σ in Figure 1 with an index i such that $\sigma(i) \models (p \wedge \phi_0)$, $\sigma(i+1) \models (r \wedge \neg\phi_2)$, $\sigma(i+2) \models (q \wedge \phi_1)$, and $\sigma(i+3) \models (r \wedge \phi_2)$ where $p, q, r, \phi_0, \phi_1, \phi_2$ are state formulas. Clearly, $\sigma \models [p]\phi_0 \rightarrow [p,q]\mathrm{EF}\phi_1$ and $\sigma \models [q]\phi_1 \rightarrow [q,r]\mathrm{EF}\phi_2$, but we can not derive $\sigma \models [p]\phi_0 \rightarrow [p,r]\mathrm{EF}\phi_2$, because σ does not satisfy premise (3).

Note that the soundness and relative completeness of the logic system was shown in [7].

3 THE DECISION PROCEDURE

In this section, we present a technique to build execution histories, and provide the decision procedure Π to determine satisfaction of ITL formulas at run-time. First, we describe construction of execution histories. which is performed at run-time.

3.1 Constructing Execution Histories

Recall that system execution can be viewed as a set of partially ordered events with each process involved in the execution viewed as a sequence of events[5]. In other words, we

[5]Without loss of generality, we do not consider the concurrency between events on a process, intra-process concurrency. A process designer may choose any granularity of events such as execution of a statement or a set of statements.

can obtain system execution by collecting and ordering events occurring in the system. Without external monitors, the following describes the construction of execution histories by processes involved in a distributed computation.

Every process maintains an event history, which is a collection of events occurring within itself, together with events occurring within other processes and observed by the process through communications. An event in a process is either an internal event or the sending or receiving of a message. A process assigns a timestamp to each event as it occurs. To obtain events occurring in other processes, whenever there is a communication, processes exchange their event histories, i.e., they exchange their observations of event occurrences in the system. Note that the timestamps of events in a history are sent along with the history.

Through the exchanges of event histories, every process obtains its collection of events occurring within itself and within other processes. This collection of events can be ordered according to a causal relation $\rightarrow$, which is defined as follows.

Definition 3.1 *Event e precedes event f in an execution, i.e., $e \rightarrow f$, iff any one of the following conditions holds* [8]*:*

1. *e and f are events of the same process, and e occurs before f,*
2. *e is a send event, and f is the corresponding receive event, or*
3. *there exists an event g, such that $e \rightarrow g$, and $g \rightarrow f$.*

Definition 3.2 *Two events e, f are* ***causally related*** *if either $e \rightarrow f$ or $f \rightarrow e$ holds. If neither $e \rightarrow f$ nor $f \rightarrow e$ holds, then e and f are considered as* ***concurrent*** *or* ***independent*** *events.*

According to the casual relation $\rightarrow$, processes can partially order events in their own histories. These histories are equivalent in the sense that they only differ in the order of independent (not causally related) events. The relation $\rightarrow$ can be implemented by the existing vector clock scheme ([9], [10]) for the determination of causality between any two events.

3.2 Computing Histories for a Non-Faulty Environment

This subsection presents the construction of an event history of process P_i in a non-faulty environment. In Figure 2, processes P_i and P_j exchange their respective histories V_{h_i} and V_{h_j} during a communication. Then, process P_i computes its new history V_{h_i} by incorporating events in V_{h_j} into V_{h_i} (step 1). The function $\hbar$ incorporates every event in a received history V_{h_j} into V_{h_i} such that for every event e of V_{h_i}, e does not cause any event preceding e and e causes its next event. In other words, event e and its preceding events are not causally related, but e is causally related to its next event in history V_{h_i}. Notice that this is not the only way to incorporate events into a process's history, since events are timestamped by vector clocks and they form *a partial ordering* instead of *a total ordering.* At step 2, process P_i updates its clock (C^i) on the i^th component ($C^i[i]$).

A run-time history V_{h_i} of process P_i is computed as follows.
During a communication, processes P_i and P_j exchange their respective histories V_{h_i} and V_{h_j}. After the exchange, both processes incorporate the received history into their own histories. The following shows that process P_i incorporates V_{h_j} into its history V_{h_i}.

step 1 $V_{h_i} = \hbar(V_{h_j}, V_{h_i})$.

step 2 $C^i[i] = C^i[i] + 1$.

Figure 2: *Computing History* for process P_i

The examples of constructing event histories and the proof of the following theorem are shown in [2].

Theorem 3.1 *The history, V_{h_i}, built by the Computing History of Figure 2, is correct in a non-faulty environment.*

3.3 Computing Histories in a Faulty Environment

This subsection presents the construction of an event history of process P_i in a faulty environment. In such an environment, faulty processes may fool non-faulty ones by sending incorrect values. In this case, processes may build incorrect histories due to receipt of erroneous values. Thus, we introduce a *consistency check* to deal with faulty processes sending inconsistent values to different processes.

Consistency Checks

The idea of a consistency check is that if a value of a variable is sent from a non-faulty process to a set of processes on more than one path, then, under a bounded number of faults, non-faulty processes will receive the same value of the variable, or an inconsistency is detected. The following defines a consistency check.

Definition 3.3 *Let $\circ(V_{h_i}, V_{h_j})$ denote a consistency check by process P_i against its received history V_{h_j}, such that for a given event e in V_{h_i} and V_{h_j}, the message content of e in V_{h_i} is different from that in V_{h_j}.*

From the above, if a given message is received by processes P_i and P_j, then the contents of the message in their respective histories V_{h_i} and V_{h_j} should be the same, or else an error has occurred. Figure 3 shows *Computing History* for a faulty environment.

Theorem 3.2 *The history V_{h_i}, built according to the Computing History of Figure 3, is correct* [2].

An event history V_{h_i} of process P_i is computed as follows. During a communication, processes P_i and P_j exchange their respective histories V_{h_i} and V_{h_j}. After the exchange, both processes apply consistency checks and incorporate the received history into their own histories. The following shows that process P_i does consistency checks and incorporates V_{h_j} into its history V_{h_i}.

1. If $o(V_{h_i}, V_{h_j})$, then STOP.
2. if $\neg(o(V_{h_i}, V_{h_j}))$ then $V_{h_i} = \hbar(V_{h_j}, V_{h_i})$.
3. $C^i[i] = C^i[i] + 1$.

Figure 3: *Computing History* for process P_i

3.4 The Procedure

We have shown the construction of event histories, and the histories can represent the execution of a responsive system R. In this subsection, we present the decision procedure Π which makes use of event histories to verify properties of the system R at run-time. The formulas to which the decision procedure Π can apply include interval formulas and responsiveness assertions.

An interval formula $[p]\phi$ ($[p,q]\phi$) can be used to denote bounded response of ϕ within the intervals $[p]$ ($[p,q]$), where p, q, ϕ are formulas of ITL. A responsiveness assertion ($[p]\phi \rightarrow [p,q]\mathrm{EF}\psi$) ensures bounded response of ψ to $[p]\phi$ within the intervals $[p,q]$ where p, q, ϕ, ψ are formulas of ITL. Therefore, both interval formulas and responsiveness assertions denote a sequence of states. However, an event history is a collection of events occurring in the system. Thus, we need to relate states to events in a history.

To establish the correspondence between states and events, we consider computations to be event driven, i.e., a process receives a message, processes it, sends messages (possibly zero) to other processes, and waits for the next message. Thus, the externally observable events (send or receive events) are the actions that change the states of a process, and they can be regarded as intermediate states of a program. [2] shows the correspondence between states and externally observable events in a history.

After establishing the correspondence between states and events in a history, we define satisfaction of a formula by a state in terms of satisfaction by an event.

Definition 3.4 *Given a history V_{h_i}, an event e satisfies an assertion ϕ, denoted by $e \models \phi$, iff $s \models \phi$, where s is the corresponding state of event e in V_{h_i}.*

For an interval formula Ψ denoting behavior of one process, the following is the procedure Π for the determination of satisfaction of Ψ with respect to the history V_{h_i} of process P_i.

Definition 3.5 *Let Ψ be an interval formula, $[p]\phi$, which specifies behavior of one process. According to the event history V_{h_i}, the decision procedure $\Pi(V_{h_i}, \Psi)$ returns TRUE,*

if for every event α of P_j satisfying p, then α satisfies ϕ. Otherwise, $\Pi(V_{h_i}, \Psi)$ returns FALSE.

Definition 3.6 *Let Ψ be an interval formula, $[p,q]\phi$, which specifies behavior of one process. According to the event history V_{h_i}, the decision procedure $\Pi(V_{h_i}, \Psi)$ returns TRUE, if the following condition holds:*

> *for every event α_l of P_j satisfying p, if there exists an event α_m of P_j such that α_l happens before α_m and α_m satisfies q, then ϕ must hold on the intervals from α_l to α_m.*

Otherwise, $\Pi(V_{h_i}, \Psi)$ returns FALSE.

Note that the procedure $\Pi(V_{h_i}, \Psi)$ is executed by every process to check behavior of all the other processes according to its own history V_{h_i}. For a responsiveness assertion Ψ, the following is the procedure Π for the determination of satisfaction of Ψ with respect to the history V_{h_i} of process P_i.

Definition 3.7 *Let Ψ be a responsiveness assertion, $[p]\phi \rightarrow [p,q]EF\psi$, which denotes behavior of one process. According to the event history V_{h_i}, the decision procedure $\Pi(V_{h_i}, \Psi)$ returns TRUE, if the following condition holds:*

> *for every event α_l of P_j satisfying $p \rightarrow \phi$, if there exists an event α_m of P_j such that α_l happens before α_m and α_m satisfies q, then α_m must satisfy ψ.*

Otherwise, $\Pi(V_{h_i}, \Psi)$ returns FALSE.

The above interval formulas and responsiveness assertions involve behavior of one process, so we only have to examine events of the process in history V_{h_i}. Next, we consider formulas which specify behavior of more than one process. In this situation, we need to know which events in an event history are executed concurrently, or which events in the history correspond to a global state of a distributed system. This can be done by utilizing consistent cuts ([11], [12], [10]) as follows.

Definition 3.8 *A **cut** is an ordered tuple of events with one event from each process. A cut $< e_{1,k_1}, e_{2,k_2}, \ldots, e_{n,k_n} >$ is a **consistent cut**, if for any two events e_{i,k_i} and e_{j,k_j} $(i \neq j)$, e_{i,k_i} and e_{j,k_j} are concurrent.*

Definition 3.9 *A **snapshot**, $s =< s^1_{k_1}, \ldots, s^n_{k_n} >$, is an ordered tuple of states with one state from each process, and with any two states of s being concurrent. A snapshot is also a global state in a distributed computation.*

Theorem 3.3 *Given a history V_{h_i}, for every consistent cut $c =< e_{1,k_1}, e_{2,k_2}, \ldots, e_{n,k_n} >$, there exists a unique global state $s =< s^1_{k_1}, \ldots, s^n_{k_n} >$, and for every global state s, there also exists a unique cut c* [2].

Now, we define satisfaction of a formula by snapshots in terms of satisfaction by consistent cuts.

Definition 3.10 *Given a history V_{h_i}, a consistent cut c satisfies an assertion ϕ, denoted by $c \models \phi$, iff $s \models \phi$, where s is the corresponding global state of c in V_{h_i}.*

The following is the decision procedure $\Pi(V_{h_i}, \Psi)$ where V_{h_i} is the history of process P_i and Ψ is an interval formula denoting behavior of multiple processes. In this case, the satisfaction of the formula is determined by satisfaction of consistent cuts in the history V_{h_i}.

Definition 3.11 *Let Ψ be an interval formula, $[p]\phi$, which asserts behavior of more than one process. According to the event history V_{h_i}, the decision procedure $\Pi(V_{h_i}, \Psi)$ returns TRUE, if for every consistent cut c of P_j satisfying p, then c satisfies ϕ. Otherwise, $\Pi(V_{h_i}, \Psi)$ returns FALSE.*

Definition 3.12 *Let Ψ be an interval formula, $[p, q]\phi$, which asserts behavior of more than one process. According to the event history V_{h_i}, the decision procedure $\Pi(V_{h_i}, \Psi)$ returns TRUE, if the following condition holds:*

> *for every consistent cut c_l of P_j satisfying p, if there exists a consistent cut c_m of P_j such that c_l happens before c_m and c_m satisfies q, then ϕ must hold on the intervals from c_l to c_m.*

Otherwise, $\Pi(V_{h_i}, \Psi)$ returns FALSE.

For a responsiveness assertion Ψ denoting behavior of multiple processes, the following is the procedure Π for the determination of satisfaction of Ψ with respect to the history V_{h_i} of process P_i.

Definition 3.13 *Let Ψ be a responsiveness assertion, $[p]\phi \rightarrow [p, q]EF\psi$, which specifies behavior of multiple processes. According to the event history V_{h_i}, the decision procedure $\Pi(V_{h_i}, \Psi)$ returns TRUE, if the following condition holds:*

> *For every consistent cut c_l satisfying $p \rightarrow \phi$, if there exists a consistent cut c_m such that every event of c_l happens before that of c_m and c_m satisfies q, then c_m must satisfy ψ.*

Otherwise, $\Pi(V_{h_i}, \Psi)$ returns FALSE.

4 Railroad Crossing Example

In this section, we present a railroad crossing example to illustrate determination of satisfaction of ITL formulas at run-time. For simplicity, we assume a non-faulty environment. Thus, no consistency checks are applied to examine event histories. There are four transitions associated with a train, traveling, approaching, ingate and departure. Two transitions are associated with the gate, up and down. A warning signal is sent by a train to the gate controller at a distance from the crossing or after the departure.

In Figure 4, when communication m_1 occurs, process P_{gate} is notified that the train is approaching. Then, during communication m_2 process P_{train} is notified that the gate

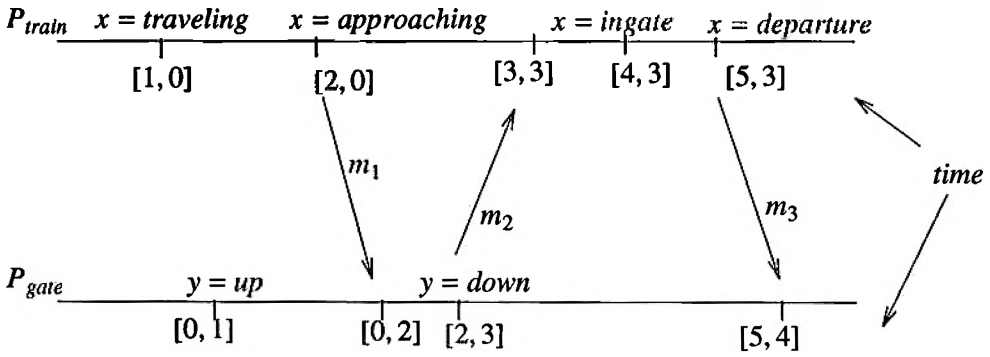

Figure 4: Railroad crossing example

is down. Likewise, the messages of departure from a train is sent when communication m_3 occurs.

First, we construct the event histories of processes P_{train} and P_{gate} according to the procedure *Computing History* of Figure 2. Then, we check satisfaction of an interval formula $\phi = [x = ingate]y = down$ against the event histories. In Figure 4, before communication m_1 occurs, the train is traveling and the gate is up. Thus, processes P_{train} and P_{gate} have the following state information in their histories, V_{h_t} and V_{h_g}.

$$V_{h_t} = < (P_{train}, \ x = traveling, [1,0]), (P_{train}, x = approaching, [2,0]) >$$
$$V_{h_g} = < (P_{gate}, \ y = up, [0,1]) >$$

The tuple $(P_{train}, x = traveling, [1,0])$ shows that at (vector) time $[1,0]$ the train is traveling, and $(P_{train}, x = approaching, [2,0])$ denotes that at time $[2,0]$ the train is approaching. Likewise, the tuple $(P_{gate}, y = up, [0,1])$ states that the gate is up at time $[0,1]$.

When communication m_1 occurs, processes P_{train} and P_{gate} exchange their respective histories V_{h_t} and V_{h_g}, and incorporate received histories into their own histories. Recall that the incorporation of histories is based on the causality on events. The histories of processes P_{train} and P_{gate} after the incorporation are as follows.

$$\begin{aligned} V_{h_t} =< & (P_{train}, x = traveling, [1,0]), (P_{train}, x = approaching, [2,0]), \\ & (P_{gate}, y = up, [0,1]), (P_{train}, P_{gate}, receive, [2,1]) > \\ V_{h_g} =< & (P_{gate}, y = up, [0,1]), (P_{train}, x = traveling, [1,0]), \\ & (P_{train}, x = approaching, [2,0]), (P_{train}, P_{gate}, receive, [2,1]) > \end{aligned}$$

The tuple $(P_{train}, P_{gate}, receive, [2,1])$ shows that at time $[2,1]$ process P_{gate} receives a message from P_{train}.

After communications m_2 and m_3, the histories V_{h_t} and V_{h_g} are as follows.

$$
\begin{aligned}
V_{h_t} \; =< \; & (P_{train}, x = traveling, [1,0]), (P_{train}, x = approaching, [2,0]), \\
& (P_{gate}, y = up, [0,1]), (P_{train}, P_{gate}, receive, [2,1]) > \\
& (P_{gate}, y = down, [2,3]), (P_{gate}, P_{train}, receive, [3,3]) > \\
& (P_{train}, x = ingate, [4,3]), (P_{train}, x = departure, [5,3]), \\
& (P_{train}, P_{gate}, receive, [5,4]) > \\
V_{h_g} \; =< \; & (P_{gate}, y = up, [0,1]), (P_{train}, x = traveling, [1,0]), \\
& (P_{train}, x = approaching, [2,0]), (P_{train}, P_{gate}, receive, [2,1]) > \\
& (P_{gate}, y = down, [2,3]), (P_{gate}, P_{train}, receive, [3,3]) > \\
& (P_{train}, x = ingate, [4,3]), (P_{train}, x = departure, [5,3]), \\
& (P_{train}, P_{gate}, receive, [5,4]) >
\end{aligned}
$$

Now, we apply the decision procedure Π to check validity of the formula $\phi = [x = ingate]y = down$. According to the procedure $\Pi(V_{h_t}, \phi)$ of Definition 3.11, $\Pi(V_{h_t}, \phi)$ returns TRUE, if the condition holds: whenever the train is ingate, the gate must be down. For the events in histories V_{h_t}, the tuple $(P_{train}, x = ingate, [4,3])$ denotes that the train is ingate, and the tuple $(P_{gate}, y = down, [2,3])$ denotes that the gate is down. Also, by comparison of (vector) timestamps of $[2,3]$ and $[4,3]$, $(P_{gate}, y = down, [2,3])$ happens before $(P_{train}, x = ingate, [4,3])$. Therefore, history V_{h_t} satisfies the formula ϕ at run-time. Likewise, we can conclude that history V_{h_g} satisfies the formula ϕ at run-time.

5 CONCLUSION

We have presented a logic, Interval Temporal Logic (ITL), to specify a responsive computing system and have given a decision procedure Π to verify properties of the systems as follows. First, properties of the system are specified using ITL formulas. Next, system execution is obtained by collecting events and maintaining equivalent event histories. Finally, the decision procedure Π is applied to determine satisfaction of the formulas at run-time. The procedure Π is essentially a run-time procedure, since it verifies properties of the system based on the event histories computed at run-time.

Currently, experiments on applying Π to verify, at run-time, properties of a railroad crossing system are under way. Future research will examine other types of assertions in addition to interval formulas and responsiveness assertions for the run-time analysis of the behavior of responsive computing systems.

References

[1] M. Malek, "Responsive systems: A challenge for the nineties," in *Proc. EUROMICRO'90, 16th Symposium Microprocessing and Microprogramming*, (Amersterdam, The Netherlands), pp. 9–16, 1990. Keynote Address.

[2] G. Tsai, M. Insall, and B. McMillin, "Ensuring the satisfaction of a temporal specification at run-time," *UMR Department of Computer Science Technical Report Number CSC 93-020*, 1993.

[3] C. Jard and T. Jeron, "On-line model-checking for finite linear temporal logic specifications," in *Automatic Verification Methods for Finite State Systems, Lecture Notes in Computer Science 407*, pp. 189–196, 1989.

[4] S. E. Chodrow, F. Jahanian, and M. Donner, "Run-time monitoring of real-time systems," in *IEEE Symposium on Real-Time Systems*, pp. 74–83, 1991.

[5] S. Katz and D. Peled, "An efficient verification method for parallel and distributed programs," in *Lecture Notes in Computer Science 354*, pp. 489–507, 1988.

[6] D. Peled and A. Pnueli, "Proving partial order liveness properties," *17th Colloquium on Automata, Language and Programming*, pp. 553–571, 1990.

[7] G. Tsai, M. Insall, and B. McMillin, "Constructing an interval temporal logic for real-time systems," *UMR Department of Computer Science Technical Report Number CSC 93-025*, 1993.

[8] L. Lamport, "Time, clocks and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[9] J. Fidge, "Timestamps in message passing systems that preserve the partial ordering," in *Proceeding of the Tenth International Conference of Software Engineering*, pp. 182–187, 1992.

[10] F. Mattern, "Virtual time and global states of distributed systems," in *Parallel and Distributed Algorithms: Proceedings of the International Workshops on Parallel and Distributed Algorithms* (M. Cosnard *et al.*, eds.), pp. 215–226, Ed. Elsevier Science Publishers B.V., 1989.

[11] L. Lamport, "Paradigms for distributed programs: Computing global states," in *Distributed Systems-Methods and Tools for Specification, Lecture Notes in Computer Science 190* (M. Paul and H. Siegert, eds.), pp. 454–468, 1985.

[12] S. Alagar and S. Venkatesan, "Hierarchy in testing distributed programs," in *Computer Science Technical Report UTDCS-8-92, The University of Texas at Dallas*, 1992.