01 Jan 1995

Ensuring the Satisfaction of a Temporal Specification at Run-Time

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

ENSURING THE SATISFACTION OF A TEMPORAL SPECIFICATION AT RUN-TIME

Grace Tsai
Mathematics and Computer Science
Fairleigh Dickinson University
Teaneck, NJ 07666, USA

Matt Insall
Department of Mathematics and Statistics
University of Missouri-Rolla
Rolla, MO 65401 USA

Bruce McMillin
Department of Computer Science
University of Missouri-Rolla
Rolla, MO 65401 USA

Abstract

A responsive computing system is a hybrid of real-time, distributed and fault-tolerant systems. In such a system, severe consequences can occur if the run-time behavior does not conform to the expected behavior or specifications. In this paper, we present a formal approach to ensure satisfaction of the specifications in the operational environment as follows. First we specify behavior of the systems using Interval Temporal Logic (ITL). Next we give algorithms for trace checking of programs in such systems. Finally, we present a fully distributed run-time evaluation system which causally orders the events of the system during its execution and checks this run-time behavior against its ITL specification. The approach is illustrated using a train-set example.

1 Introduction

A responsive computing system [1] is one which is required to respond to internal programs or external inputs in a timely, dependable and predictable manner. These systems are 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).

The specification of such critical systems can be rigorously represented using formal methods of logic. Formal methods are the use of mathematical techniques in the design and analysis of computer hardware and software. One of the many advantages of using formal methods is that when a property is obtained, it comes from certainty and not from doubtful or approximate inferences.

Conceptualization. The goal of our work is to find ways to execute program specifications along with the actual program's execution for purposes of run-time assurance – namely for error detection within the scope of fault tolerance. If the execution of the program does not satisfy the specification at run time, then an error has occurred. Since error detection is conceptually the most difficult problem in fault tolerance, this quantification of error detection has proved quite powerful - a system need not rely on hardware or software confidence to avoid or detect errors; the specification provides the absolute truth of correctness.

The notion of "the program satisfies the specification" is a powerful abstraction as it immediately draws the researcher into the area of formal logic to express the specification. This, coupled with an existing set of axioms and inference rules for a particular (programming) language provides the appropriate level of representation for run-time error checking. Essentially, the same tools used in program verification are immediately applicable to run-time assurance, namely execution of the specification in either a predicate or temporal framework.

Our work provides the run-time semantics to carry out such execution of specifications, possibly in the presense of failed hardware and/or software. Thus, the approach taken here adopts a formalized specification language together with a mechanized support tool to allow detection of certain types of errors and faults.

Methodology. There are three steps involved in this approach of ensuring a system's specification at run-time:

1. Specify properties of a responsive computing system using Interval Temporal Logic (ITL) formulas,
2. build a run-time event history, the causal structure of the execution, and
3. evaluate properties of the system according to the event histories.

At step 1, the logic ITL was developed to specify behavior of a responsive computing system. In particular, we use *interval formulas* and *responsiveness assertions* to denote properties of a system. These formulas denoting system specifications are expected to hold within bounded intervals. Thus, an error has

0-8186-7123-8/95 $4.00

occurred if the system does not satisfy these formulas within bounded intervals.

At step 2, we give algorithms to obtain a run-time history or to build the causal structure of the execution. The run-time history of a (distributed) system can be obtained by collecting and partially ordering events[1] occurring in the system[2].

At step 3, we apply a decision procedure, Π, to check whether the collected event history satisfies the specification of a system derived at step 1. Since an event history is a sequence of events occurring in a system, it represents a process' observation of all the processes during execution. This history can be utilized to do evaluation of assertions at run-time. The evaluation is a simple matter, then, to break down the temporal assertions into predicate calculus expressions quantified over this history sequence[3]. Thus, this approach uses the ITL formulas to detect errors. If the run-time behavior violates its specification denoted by ITL formulas, then appropriate actions should be taken.

For example, one way to avoid cars and trains occupying a crossing at the same time is to lower a gate before a train arrives on the crossing [2]. An ITL formula (step 1) for this situation can be used to represent the timing constraints of the system. Knowing how long it takes to lower the gate, and the minimum time that can elapse between a train passing a sensor and reaching the crossing, we can deduce timing constraints on the gate controller. At run-time, then, we collect the events of the train passing the sensor (step 2) and can check if the gate controller violates the ITL specifications denoting these constraints (step 3). If yes, actions should be taken to react to the error.

In our previous work, we used a decision procedure to check satisfaction of liveness assertions in the operational environment [3]. A liveness assertion ($\phi \rightarrow \mathrm{EF}\psi$) denotes that when a program starts from a state satisfying assertion ϕ, *eventually* it will get to a state satisfying assertion ψ. This kind of assertion 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 system specifications within bounded intervals of time.

In related work, for the determination of satisfaction of formulas, [4] translates temporal logic formulas into finite automata. In contrast, we establish a correspondence between states and events in the collected event history and examine the history against the specification for the determination of satisfaction of formulas. The work of [5] 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 histories.

The organization of this paper is as follows. In Section 2, we introduce the logic ITL. Section 3 describes the notion of an event, our model of distributed computation, and other definitions to be used in this paper. In Section 4, we describe a timestamping scheme, vector clocks, to order events in a collected history. Next, we present an algorithm, *Compute History*, to construct event histories and then give a decision procedure for the determination of satisfaction of ITL formulas at run-time. Section 5 presents a train set example to illustrate the application of operational evaluation. Section 6 concludes this paper.

2 Interval Temporal Logic

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

With the logic ISTL, one can not reason about a property within a bounded interval of time, which motivates the development of the ITL for the responsive computing systems. There are other temporal logics for real-time systems for example [7, 8, 9, 10]. The logic [8] does not have a proof system for reasoning about a system. [9] is designed for reasoning about hardware, while we aim at a logic which can reason about a distributed real-time system. In our approach, the specification of a system will be tested to detect errors at run-time. So we need to build a mechanized support tool for the logic. For efficiency consideration, we built the logic ITL which has a small set of syntactic forms and includes only the inference rules necessary for the run-time evaluation. Hence we do not adopt the logic [9, 10].

Due to the page limits, we only present two types of formulas, *interval formulas* and *responsive assertions*. The reader may refer to [11] for the syntax and semantics of the logic. 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 to create a "leads-to" property. Thus, the following *responsiveness assertion* is proposed.

Definition 2.1 *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

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

[2] Note that *we consider neither the underlying scheduling strategies nor predict which branches or statements will be executed at run-time.*

[3] for example, the temporal expression "now and always p, in predicate calculus becomes $\forall i$ in history H (of events H_i indexed by i) such that $H_i \models p$. A program to check this temporal expression over a collected history, H, loops through H checking that each event H_i satisfies p

ϕ holds whenever p holds, then ψ will occur at any q following p. This assertion ensures bounded response of ψ to $[p]\phi$ within the intervals $[p, q]$.

The above formulas are to be applied to a run-time system to check if a system does what it is supposed. The following notation and background are necessary to understand the proposed algorithms for evaluating the ITL formulas at run-time.

3 Background

A distributed program consists of n processes, $P_1, P_2, \cdots, P_n$, which cooperate to perform a computation. Each process resides on a unique processor. The mapping between processors and processes is one-to-one. There are no global clocks, and processes must communicate via message-passing to exchange information. Thus, the events occurring within a process can be totally ordered according to its processor's clock while events occurring within other processors cannot.

There are three types of operations, internal (local) operations, send operations, and receive operations. A send (receive) event of a processor includes execution of a sequence of local operations followed by a send (receive) operation in that processor. Send or receive events are referred to as *externally observable* events. The externally observable events can be partially-ordered by any processor whereas internal events of a processor are non-observable by other processors.

The following definitions are necessary before the presentation of the algorithms.

Notation 3.1 *Event execution in a distributed program is represented by a diagram, where each horizontal line describes one process behavior, and the horizontal direction of each line denotes time which increases from left to right. Message exchanges are shown by directed lines.*

Definition 3.1 *Event execution in a program forms an irreflexive partial order(denoted by $\rightarrow$) on the events which occur in the program.*

Definition 3.2 *Event e precedes event f in an execution, i.e., $e \rightarrow f$, if and only if any one of the following conditions holds* [12]*:*

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 $g \rightarrow f$.*

Note that we assume that a system has been verified to be deadlock-free.

Definition 3.3 *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.*

Definition 3.4 *A* **history** *of a program P is a pair $h =< J, v >$ where J is the initial interpretation and $v = \alpha_1, \alpha_2, \ldots, \alpha_n$ is a sequence of events of the program in the causal relation $\rightarrow$ order.*

Throughout the paper, we will use the letter J in a history (e.g., $h =< J, v >$) to denote an initial state.

Definition 3.5 *Let A be a collection of events of a program. Given an initial state J and two sequences of events v and w ($v, w \in A^*$), two histories $h =< J, v >$ and $h' =< J, w >$ are* **equivalent**, *if there exist histories $< J, v_1 >, < J, v_2 >, \ldots, < J, v_n >$ with $v_1 = v$ and $v_n = w$ and for each $1 \leq i < n$, there exist $\alpha \not\rightarrow \beta$ and $x, y \in A^*$, such that $v_i = x\alpha\beta y$, $v_{i+1} = x\beta\alpha y$. In other words, v_i and v_{i+1} only differ by the order of adjacent symbols which are independent according to the causal relation $\rightarrow$ of Definition 3.2* [6].

Definition 3.6 *A* **trace** *is an equivalence class of histories, denoted by $[J, w]$ where J is an initial state and $< J, w >$ is some member of the equivalence class $[J, w]$*([13], [6]).

Definition 3.7 *Let a set V_{h_k} be a processor P_k's collection of events including processor P_k's (local) events, and those it* **observes** *(processor P_x communicates with processor P_k about its local events). This set V_{h_k} denotes processor P_k's* **history**. *Also, processor P_k's* **knowledge** *or* **view** *about system execution is based on the events in its history V_{h_k}.*

Notice that the history $h =< J, v >$ is a complete history of a program, which contains events of the whole execution of the program, while the history V_{h_k} of processor P_k is a partial history or a collection of events observed or executed by processor P_k during execution.

Definition 3.8 *Let $e_{k,1}, e_{k,2}, \cdots, e_{k,n}$ be the first, second, $\cdots$, n^{th} observable (send or receive) events of processor P_k. An event $e_{k,m}$ is the m^{th} event of processor P_k.*

4 Event Histories

In this section, we present algorithms for the construction of event histories to represent system execution. These histories will be used to evaluate system specifications written in ITL formulas (described in the following section). First, we gives a brief introduction to events, our model of distributed computation and the notation and definitions used in this chapter.

4.1 Ordering Events

Recall that a history V_{h_k} is a collection of events observed or performed by processor P_k (process) during execution. Among these events, some are causally related, while some are not causally related (independent). Thus, a time-stamping scheme is necessary to decide causality of any two events in a history. What we would like is a clock scheme that imposes no arbitrary orderings on any two events which are

not originally causally related. Thus, vector clocks ([14, 15, 16]) are chosen to determine causality between any two events.

Vector Clock Scheme. Let C^i be the vector clock of process P_i, and let C^i_k denote the clock value after the execution of event $e_{i,k}$. On sending a message, a process P_i timestamps the message by appending the clock value to it. When process P_i executes an event e, the following operations are performed on its clock C^i.

Operation 1: for each event e, P_i increments its clock C^i on the i^{th} component of the vector, i.e., $C^i[i] = C^i[i]+1$, where $C^i[i]$ denotes the i^{th} component of vector C^i.

Operation 2: for a receive event e with a vector timestamp T, $\forall m, C^i[m] = max(C^i[m], T[m])$, where $C^i[m]$ and $T[m]$ denote the m^{th} components of vectors C^i and T, respectively. In other words, the value of each component of vector C^i is obtained by taking the maximal value from the corresponding components of vectors C^i and T.

The following definition describes the mechanism of deciding causality of any two time-stamped events.

Definition 4.1 *Given two timestamps (vectors) C^i_k, C^j_l for events $e_{i,k}$ and $e_{j,l}$, respectively, the relation $(e_{i,k} \rightarrow e_{j,l})$ holds, iff*

$$(\forall r, C^i_k[r] \leq C^j_l[r]) \wedge (\exists s, C^i_k[s] < C^j_l[s]).$$

In other words, event $e_{i,k}$ occurs before event $e_{j,l}$, if and only if all the components of C^i_k are less than or equal to the corresponding components of C^j_l and there exists a component of C^i_k which is strictly less than that of C^j_l.

4.2 Correct Histories and Algorithms

In Section 3.2, we described vector clock timestamping which can be used by a processor in a distributed system to order events from different processors. The purpose of this subsection is to show that a processor P_k can construct an execution history V_{h_k} *without a global clock and without any monitors.* We begin with the definition of correct histories and then present an algorithm *Compute History*, which allows a processor P_k to construct an execution history V_{h_k} by collecting events occurring in the system. Finally, the history V_{h_k} constructed according to the algorithm is shown to be correct.

Correct Histories. Recall that a history V_{h_k} is a collection of events executed or observed by processor P_k during execution, and it is processor P_k's view of all the externally-observable events performed by other processors involved in a computation. The objective is to utilize this history to check for a violation of a program's specification at run-time. Thus, it is very important to have correct histories. The following definitions show that a history is correct with respect to the history $h =< J, v >$ iff its continuation (see Definition 4.2) is a member of the equivalence class of history h.

Definition 4.2 *A history $h =< J, w >$ is a* **continuation** *of a history $h' =< J, w' >$, if* (1) *for every event e in h', e is also in h, and* (2) *causality in h' implies causality in h.*

Definition 4.3 *For a processor P_k, its (run-time) history $V_{h_k} =< J, w >$ is* **correct** *with respect to the history $h =< J, v >$, if and only if the following condition holds:*

> *there exists a history $(V_{h_k})'$, a continuation of V_{h_k}, such that $(V_{h_k})'$ is a member of the equivalence class of $< J, v >$, i.e., $(V_{h_k})' =< J, w > \in [J, v]$.*

Notice that, in Definition 4.3, histories h and $(V_{h_k})'$ are equivalent, where h is a correct and complete history of a program and $(V_{h_k})'$ is a history resulting from extending the history V_{h_k}. The following proposition shows that *a history V_{h_k} is correct, if and only if causality among events in V_{h_k} is preserved during execution.*

Proposition 4.1 *For a processor P_k, its history $V_{h_k} =< J, w >$ is correct with respect to a history $h =< J, v >$ if and only if for events in V_{h_k}, causality in $V_{h_k} \Leftrightarrow$ causality in h, i.e., causality in V_{h_k} is preserved during execution.*

Proof: (if part) If the history V_{h_k} is correct, then there exists a continuation $(V_{h_k})' =< J, w' >$ of V_{h_k}, and $< J, w' > \in [J, v]$. In other words, $< J, w' >$ and $< J, v >$ only differ by the order of independent operations, which implies v and w' have the same orderings for those causally related events, i.e., causality in $h \Leftrightarrow$ causality in $(V_{h_k})'$. Since $(V_{h_k})'$ is a continuation of V_{h_k}, for the events in V_{h_k}, causality in $V_{h_k} \Leftrightarrow$ causality in $(V_{h_k})'$. Therefore, for the events in V_{h_k}, causality in $h \Leftrightarrow$ causality in V_{h_k}. This implies that causality in V_{h_k} is preserved during execution.

(only-if part) From assumption for events in V_{h_k}, causality in $V_{h_k} \Leftrightarrow$ causality in h, we know that h is a continuation of V_{h_k}. Then, there exists a continuation $(V_{h_k})' = h$ of V_{h_k}, such that $(V_{h_k})'$ and h are equivalent. By Definition 4.3, the history V_{h_k} is correct with respect to the history h. □

4.3 Computing Histories in a Non-Faulty Environment

In this subsection, we present an algorithm which allows a processor P_k in a distributed system to collect events into a history V_{h_k}, limiting ourselves, for now, to a non-faulty environment [17]. Later, it will be shown that these histories can be utilized to detect a violation of processors' run-time behaviors.

Main idea. A processor P_k relies on communications to find out events that have occurred in other

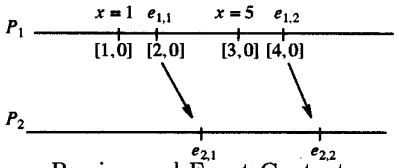

Figure 1: Message Passing and Event Contents

processes, and to collect these events into its history V_{h_k}. Thus, *whenever there is a communication, processes exchange their latest observations (histories) of event occurrences in the system*[4]. After the exchange, processors incorporate the received histories into their own histories. Through the exchanges of histories, every processor can obtain a view of the execution of all the other processors in the system.

Now, we describe the contents of events, the relevant information for processors to compute their histories. Then, examples are given to illustrate how processors exchange their histories (observations), followed by the algorithm *Compute History*.

Definition 4.4 *Let a tuple* $t_1 = (processor, var = val, time)$ *denote a timestamped local operation. Let a tuple* $t_2 = (processor_1, processor_2, send/receive, time)$ *represent a timestamped send/receive operation of* $processor_1$, *where* $processor_2$ *is the corresponding communicating processor. A send/receive event e is denoted by* $(t_1, t_1, \cdots, t_1, t_2)$.

From the above, a tuple can represent a timestamped local operation or a time-stamped send/receive operation, and a send/receive event e contains information of local operations followed by an send/receive operation, i.e., $(t_1, t_1, \cdots, t_1, t_2)$. For example, in Figure 1, there are two send events, $e_{1,1}$ and $e_{1,2}$. The contents of these two events are as follows.

$$e_{1,1} = ((P_1, x = 1, [1,0]), (P_1, P_2, send, [2,0]))$$
$$e_{1,2} = ((P_1, x = 5, [3,0]), (P_1, P_2, send, [4,0]))$$

Events $e_{1,1}$ shows that at time $[1,0]$ the value of x in processor P_1 is 1, and P_1 sent a message to P_2 at time $[2,0]$. Likewise, event $e_{1,2}$ shows that at time $[3,0]$ the value of x in processor P_1 is 5, and P_1 sent a message to P_2 at time $[4,0]$. Here, the result of a local operation, $x = 1$, is considered as part of the next (observable) event $e_{1,1}$. Similarly, the result of a local operation, $x = 5$, is considered as part of the next (observable) event $e_{1,2}$.

Based on this example, then, how does processor P_2 incorporate the received events into its history V_{h_2}? The following describes the incorporation of an event into a history.

[4]Note that, for efficiency considerations, only those events which have not been communicated previously need to be sent.

A run-time history V_{h_i} of processor P_i is computed as follows. During a communication, processors P_i and P_j exchange their respective histories V_{h_i} and V_{h_j}. After the exchange, both processors incorporate the received history into their own histories. The following shows that processor 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: Algorithm *Compute History* for processor P_i

Definition 4.5 *Given an event e, the incorporation of e into a history* $V_{h_i} =< J, \alpha_1\alpha_2\cdots\alpha_n >$ *of processor* P_i *is to insert e into the history* V_{h_i}, *such that the new history* $V_{h_i} =< J, \alpha_1\cdots\alpha_{k-1}e\alpha_k\cdots\alpha_n >$ *satisfies the following conditions:*

1. *for each* $m < k$, $e \not\rightarrow \alpha_m$, *i.e., e does not cause any* $\alpha_m (1 \leq m < k)$.
2. $e \rightarrow \alpha_k$, *i.e., e causes* α_k.

Therefore, in the new sequence $(\alpha_1\cdots\alpha_{k-1}e\alpha_k\cdots\alpha_n)$, e does not cause any event preceding e (i.e., events $\alpha_1\cdots\alpha_{k-1}$), and e causes its next event (i.e., event α_k). Notice that there are many ways of incorporating events into a processor's history, since events are timestamped by vector clocks and they form *a partial ordering instead of a total ordering*. However, it is important that during execution, causality in a collected history V_{h_i} is preserved, and at termination, history V_{h_i} is a member of the equivalence class of the history $h =< J, v >$. The following describes an incorporation of one history into another.

Definition 4.6 *Given two histories* V_{h_1} *and* V_{h_2}, *a function* $\hbar(V_{h_1}, V_{h_2})$ *returns a history* V_{h_2}, *such that for each event e of* V_{h_1}, *if e is not in* V_{h_2}, *then, using Definition 4.5, incorporate e into* V_{h_2}.

The following example illustrates exchanges of observations (histories) and incorporations of histories. The examples of communicating histories are in [18].

Algorithm for a Non-Faulty Environment. Assume that there are n isolated processors which can communicate only by two-party messages. Figure 2 presents Algorithm *Compute History*, which computes a history of processor P_i in a non-faulty environment. In this algorithm, processors P_i and P_j exchange their respective histories V_{h_i} and V_{h_j} during a communication. Then, processors P_i computes its new history V_{h_i} by incorporating events in V_{h_j} into V_{h_i} (step 1). Finally, processor P_i updates its clock (step 2).

Theorem 4.1 *The history,* V_{h_i}, *built by the algorithm Compute History of Figure 2, is correct in a non-faulty environment.*

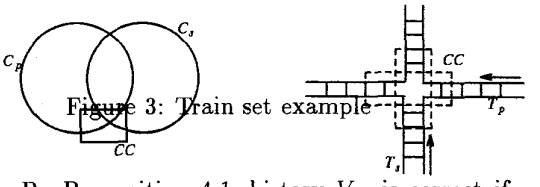

Figure 3: Train set example

Proof: By Proposition 4.1, history V_{h_i} is correct if causality in V_{h_i} is preserved during execution. Recall that V_{h_i} is processor $P_i's$ collection of events during execution. In the algorithm, upon the receipt of history V_{h_j}, processor P_i computes its history V_{h_i} from function $\hbar(V_{h_j}, V_{h_i})$, which incorporates events of V_{h_j} into V_{h_i} according to Definition 4.5. Therefore, events in V_{h_i} are in a linear order compatible with the causal relations $\rightarrow$. Also, history V_{h_i} is built in a non-faulty environment. Thus, causality in V_{h_i} is preserved. □

The reader may refer to [18] for the algorithms for the faulty environments. Next we use a train set example to illustrate the application of operational evaluation of temporal interval formulas.

5 Train Set Example

In this section, we illustrate the proposed approach using a train set example ([19], [20]). This is an example of a safety-critical system which involves interactions between controllers and physical processes. In Figure 3, the physical process consists of circuits, C_p and C_s, and trains, T_p and T_s. The circuits are divided into sections and the crossing section (CC) is where the circuits intersect. Each section has a sensor, while for each train there is an actuator that can stop the train within any section. For such a safety-critical system, accidents will occur if the physical specification and the logical specification are not met. We can then check whether the physical specification denoted by a run-time history V_{h_i} satisfies the logical specification denoted by ITL formulas.

5.1 Safety Constraints

In this subsection, we describes the safety constraints of the system. These constraints are to be embedded into the train set program to check if the run-time system behaves as what we expect. Figure 4 describes the state variables to be used.

Definition 5.1 *Let T_l^x denote the current time when train x enters section l, and let $T_{\bar{l}}^x$ denote the point of time immediately before $T_{l\oplus 1}^x$.*

Notice that addition $\oplus$ and subtraction $\ominus$ on section numbers are performed modulo the number of sections of the circuit.

SC1 (Reservation constraints): for any train, the current occupied section ($Ptrain(x)$) and the following $mcsf \oplus 1$ sections must always be reserved.

$$(\forall x \in Tr)$$
$$[T_i^x, T_{\bar{l}}^x]\{Ptrain(x), Ptrain(x)\oplus 1, \cdots, Ptrain(x)\oplus (mcsf \oplus 1)\} \subseteq Rtrain(x),$$

$mcsf$ denotes the maximal number of consecutive sensor failure.

SC2 (Exclusion constraints): mutual exclusion must be achieved for reserved sections. In other words, if $Rtrain(x)$ and $Rtrain(y)$ are the sets of sections reserved, respectively, by trains x and y ($x \neq y$), then $Rtrain(x) \cap Rtrain(y) = \emptyset$.

$$\forall x, y \in Tr$$
$$[T_i^x, T_{\bar{l}}^x](x \neq y \rightarrow Rtrain(x) \cap Rtrain(y) = \emptyset)$$

SC3: if the number of consecutive sensor failures is greater than $mcsf$, then the system must be shut down.

$$\forall c \in L, \forall x, y \in Tr$$
$$[T_i^x, T_{i\oplus mcsf}^x](\neg Sens(c,i) \wedge \cdots \wedge \neg Sens(c, i\oplus mcsf)) \rightarrow [T_i^x, T_{i\oplus mcsf}^x]\mathrm{E}F(Shut\_Down)$$

The above formula can be derived from Progress Rule. First, we apply Progress Rule to derive a formula Ψ, which describes sensor failure on two consecutive sections as follows. To derive $\Psi = [T_i^x]Ptrain(x) \rightarrow [T_i^x, T_{\overline{i\oplus 1}}^x]\mathrm{E}F(on(c,x) \wedge \neg Sens(c, Ptrain(x)))$, the following premises must hold:

(1) $[T_i^x]Ptrain(x) \rightarrow [T_i^x, T_{\bar{i}}^x]\mathrm{E}F(on(c,x) \wedge \neg Sens(c, Ptrain(x)))$

(2) $[T_{\bar{i}}^x]Ptrain(x) \rightarrow [T_i^x, T_{\overline{i\oplus 1}}^x]\mathrm{E}F(on(c,x) \wedge \neg Sens(c, Ptrain(x)))$

(3) $[T_i^x, T_{\overline{i\oplus 1}}^x]\mathrm{E}(on(c,x) \wedge \neg Sens(c, Ptrain(x)))$

Premise (1) states that the sensor of section i fails within the interval $[T_i^x, T_{\bar{i}}^x]$. Likewise, Premise (2) states that the sensor of section $i \oplus 1$ fails within the interval $[T_{\bar{i}}^x, T_{\overline{i\oplus 1}}^x]$. Premise (3) states that the formula $(on(c,x) \wedge \neg Sens(c, Ptrain(x)))$ will hold in the interval $[T_i^x, T_{\overline{i\oplus 1}}^x]$. If these three premises hold, then according to Progress Rule, we can conclude Ψ, the sensor failure on two consecutive sections. Similarly, we can conclude sensor failure on $(mcsf \oplus 1)$ sections by applying Progress Rule.

SC4: if an actuator ever fails, the system must be shut down, i.e., if an actuator is set to stop a train x on section i ($[T_i^x]Act(x,i)$) and the train is moving beyond section i ($[T_i^x, T_{i\oplus 1}^x](Ptrain(x) > i)$), then the system must be shut down.

$$(\forall x \in Tr)(\exists i \in Sc)$$
$$[T_i^x]Act(x,i) \wedge [T_i^x, T_{i\oplus 1}^x](Ptrain(x) > i) \rightarrow [T_i^x, T_{i\oplus 1}^x]\mathrm{E}F(Shut\_Down)$$

Variable	Comments
$On(x, c)$	train x is on circuit c
$Ptrain(x)$	the position(section) contains the front of a train x
$Rtrain(x)$	the set of sections that are reserved by a train x on Circuit c
$Sens(c, i)$	sensor of section i detects a train on circuit c.
$Act(x, j)$	$Act(x, j)$ is set to stop train x on section j.
Shut_Down	*Shut_Down* holds when all trains must be stopped, i.e., all actuators are set.

Figure 4: The state variables

These safety constraints govern the operation of the system - if they are ever violated, then there is an error. In other words, for a safety constraint SC, if the evaluation $\Pi(V_{h_k}, SC)$ returns FALSE, then an error has occurred.

5.2 Results

In this subsection, we examine results of performance measurement experiments on the train set example. Let P_{tr}, P_c, P_s denote processes train, circuit and section, respectively. Also let $V_{h_{tr}}$, V_{h_c} V_{h_s} denote respectively the histories of processes P_{tr}, P_c and P_s. For this experiment, the trains T_p and T_s of Figure 3 have the same speed. Define a round to be the time a train takes to get back to section i when starting from section i.

Let the overhead time be the time a process takes to generate a run-time history V_{h_i}, parse ITL formulas and evaluate the formulas. In particular, this time includes the exchanges of histories, consistency check and incorporation of the received histories. Thus, the percentage of overhead incurred is (overhead time / total time).

The train-set program is implemented on a SUN 10/40 under Solaris. Consider the cases where it takes 1, 5 and 10 minutes for a train to traverse a section. Figure 5 lists the percentage of overhead for processes P_{tr}, P_c and P_s in 3 hours. If the section traversal time is 1 minute, then the percentages of overhead are respectively 25.83%, 9.46% and 12.09% for processes P_{tr}, P_c and P_s. The overhead of process P_{tr} is larger than that of processes P_c and P_s because in each round the process P_{tr} performs more communications (and hence more auxiliary communications) than processes P_c and P_s and the communications are synchronized. From Figure 5, the percentage of overhead decreases as the section traversal time increases since most of the time is spent on traversing sections instead of doing auxiliary communications.

In this implementation, the histories to be sent are reset to empty after the auxiliary communications or the exchanges of histories. A process maintains its own history V_{h_i} together with a collection of histories with respect to every other process in the system. Whenever there is a communication between processes P_i and P_j, P_i and P_j exchange the histories that contain new events that they have observed since the last communication between P_i and P_j. So processes only exchange the events which are never sent before. After the exchange, P_i and P_j clean the respective histories. Thus, the histories to be sent do not grow as the execution proceeds, nor does the overhead.

process	P_{tr}	P_c	P_s
1 min/section	25.83%	9.46%	12.09%
5 mins/section	6.67%	2.43%	3.20%
10 mins/section	3.46%	1.26%	1.67%

Figure 5: The amount of overhead.

6 Summary and Conclusions

This paper presents a formal approach which incorporates formal methods to detect errors and to guide fault-finding process for the development of the critical systems. In this approach, system behavior is specified using a formal logic, ITL. The system behavior or specification can then be used as a metric to verify if the run-time behavior of the system satisfies its expected behavior (specification).

In summary, this approach includes the following steps.

1. Specify a system using ITL formulas.
2. Collects events and obtains an event history for each process in the system. This allows a process to have a (global) view of the execution.
3. Perform evaluation of the ITL formulas according to their collected event histories. This is to detect an violation of an ITL specification at run-time.

We built a run-time evaluation system using the formalisms of this paper. The simulation of scaling-up is underway - check if the overhead of communicating histories grows proportionally as the number of processes increases. This work will be extended to enable evaluation of specifications written in other, suitable, temporal logic languages. Also, we will consider other formal languages such as I/O automata, process algebras or timed CSP for the run-time evaluation system.

Acknowledgements

The authors are grateful to the members of Engineering Computer Laboratory of The University of Missouri-Rolla, who provide many helpful suggestions

and comments to this paper. The authors would also want to thank the anonymous referees for their constructive comments. This work is supported in part by the National Science Foundation under Grant Numbers MSS-9216479 and CDA-9222827, and, in part, from the Air Force Office of Scientific Research under contract number F49620-92-J-0546 and F49620-93-1-0409, and, in part by a grant from the University of Missouri Research Board.

References

[1] M. Malek, "Responsive systems: A challenge for the nineties," in *Proceeding EUROMICRO'90, 16th Symp. in Microprocessing and Microprogramming*, (Amersterdam, The Netherlands), pp. 622–628, North Holland, 1990.

[2] J. Rushby, "Formal methods and the certification of critical systems," in *Technical Report CSL-93-7*, 1993.

[3] G. Tsai, M. Insall, and B. McMillin, "Ensuring value liveness of distributed software through Changeling," *UMR Department of Computer Science Technical Report Number CSC 93-03*, 1993.

[4] 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.

[5] 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.

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

[7] B. Moszkowski and Z. Manna, "Reasoning in interval temporal logic," in *Lecture Notes in Computer Science #164, Logic of Programs*, pp. 371–382, 1983.

[8] K. T. Narayana and A. A. Aaby, "Specification of real-time systems in real-time temporal interval logic," in *Proceeding of Real-Time Systems Symposium*, pp. 86–95, IEEE Computer Society, Dec. 1988.

[9] B. Moszkowski, "Temporal logic for multilevel reasoning about hardware," *The Computer Journal*, pp. 10–18, 1985.

[10] B. Moszkowski, "Some very compositional temporal properties," Tech. Rep. TR-466, Department of Computer Science, University of Newcastle upon Tyne, Newcastle NE1 7RU, Great Britain, 1993.

[11] G. Tsai, M. Insall, and B. McMillin, "Constructing an interval temporal logic for real-time systems," in *20th IFAC/IFIC Workshop in Real-Time Programming*, 1995. UMR Department of Computer Science Technical Report Number CSC 93-25.

[12] 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.

[13] A. Mazurkiewicz, "Trace semantics," in *Lecture Notes in Computer Science 255*, 1986.

[14] 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.

[15] 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.

[16] D. Jefferson, "Virtual time," *ACM Transactions on Programming Language and Systems*, vol. 7, no. 3, pp. 404–425, 1985.

[17] H. Lutfiyya, M. Schollmeyer, and B. McMillin, "Formal generation of executable assertions for Application-Oriented Fault Tolerance," *UMR Department of Computer Science Technical Report Number CSC 92-15*, 1992.

[18] G. Tsai, *Providing Run-Time Assurance for Responsive Computing Systems*. PhD thesis, University of Missouri-Rolla, Department of Computer Science, Rolla, MO 65401, 1994. Technical Report Number CSC 94-030.

[19] R. de Lemos, A. Saeed, and A. Waterworth, "Exception handling in real-time software from specification to design," in *The Second International Workshop on Responsive Computing Systems*, pp. 108–121, 1992.

[20] R. de Lemos, A. Saeed, and T. Anderson, "Analysis of timeliness requirements in safety-critical systems," in *Lecture Notes in Computer Science 571, Formal Techniques in Real-Time and Fault-Tolerant Systems*, pp. 171–192, 1992.