30 Jun 1993

Operational Evaluation of Responsiveness Properties

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

OPERATIONAL EVALUATION OF RESPONSIVENESS PROPERTIES†

Grace Tsai, Matt Insall and Bruce McMillin

CSC-93-013

June 30, 1993

† This work was 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.

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

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

ABSTRACT

In this paper, a new technique for ensuring run-time satisfaction of properties-specifically responsiveness property, a subset of liveness property, in responsive systems, is presented. Since whether the run-time behavior of a system is satisfied depends on the execution (operational) environment, we develop a translation which takes into account the constraints in the operational environment, and generates histories for each process in the system. Thus, every process can utilize its history to operationally evaluate the system behavior and signal errors if its history is violated. Therefore, this technique provides software safety, handles error-detection, and ensures run-time satisfaction of responsiveness property in the operational environment. To illustrate this approach a train set example is presented.

1. INTRODUCTION

A responsive system [Male90] is one which responds to internal programs or external inputs in a timely, dependable and predictable manner. In such life-critical system, any failure can cause catastrophe, and ensuring run-time satisfaction of assertions–expected behavior, is a necessity. This motivates our work of developing a technique for ensuring run-time satisfaction of system behavior.

This technique of ensuring run-time satisfaction of expected behavior, characterized by assertions, is as follows. First, a sound mathematical basis-Interval Temporal Logic(ITL) is provided to model responsiveness property, a subset of liveness property, of the system. Then, in order to apply responsiveness property in the operational environment, a transformation taking into account the constraints of the operational environment is developed. The transformation helps processes create histories, with a consistent view, for all the (local) operations in the system. Therefore, processes can make use of their histories to operationally evaluate, at intermediary or observable stages, the behavior of other processes-this avoids self-evaluation. If a history is violated, which means the expected behavior is violated, then an error is signaled. Therefore, the operational evaluation of responsiveness property-expected behavior, provides not only error-detection but software safety.

This approach–operational evaluation of assertions provides software safety through implicit redundancy, since safety constraints or run-time satisfaction are examined upon the communications. There are other approaches to cope with dependencies in the execution environment. In [Mok91], implementation dependencies, e.g., architecture or resource constraints, are explicitly identified and are brought in as need. In other words, the implementation dependencies are isolated first and then they check if system specification and the required implementation dependencies can be enforced by control structures that meet the required timing constraints.

The proposed approach is based on *Changeling* [LuMc91b, LuSM92, LuSM92a], which chooses axiomatic proof system as the mathematical model and uses assertions generated from the proof outline to detect errors. However, this paper focuses on establishing run-time satisfaction of system behavior-characterized by responsiveness property, for the life-critical system.

The responsiveness property shows progress of the system behavior, and can be established by applying the proof rules of ITL. At run time, the property is evaluated against a time-index computation history collected by processes of the system. If the history is not violated, then it is ensured run-time satisfaction of responsiveness property, i.e., the system does what it supposes to do.

This paper is organized as follows. Section 2 describes the logic ITL which can be used to model and reason about responsiveness property in life-critical systems. Then, a translation procedure allowing operational evaluation of responsiveness property is presented in Section 3. In Section 4, the complexity of the translation and operational evaluation of assertions is addressed in terms of computation cost and communication cost. Section 5 illustrates, on a train set example, the technique-operationally ensuring run-time satisfaction, and analyizes the overhead incurred by this technique. Section 6 concludes this paper.

2. Interval Temporal Logic

In this section, we developed a logic for the design and analysis of safety-critical systems. In such systems, bounded response is crucial and one failure can cause a catastrophe. The logic is called Interval Temporal Logic(ITL), which supports the analysis of responsiveness properties-a subset of the liveness properties. This ITL is an extension of Interleaving Set Temporal Logic($ISTL^*$) [PePn90], which adopts a partial order semantics. Hence, the logic ITL captures the temporal and distributed aspects of responsive systems.

The interval formulas of ITL have the form $[I]\alpha$: the formula α holds on the interval $[I]$; if the interval $[I]$ can not be found then the interval formula is vacuously true. An interval $[I]$ is bounded by assertions. For instance, let s_i be the assertion which characterizes the state at position i in a behavior σ; $[s_i, s_j]$ denotes the interval $s_i, \ldots, s_j$. Interval formulas are used to specify properties within "bounded" intervals of time.

Definition 2.1: Any formula in the language ITL is a *path formula*, while a *state formula* is one containing no intervals or temporal operators, and is interpretable on a single state.

From the above definition, we know that interval formulas are path formulas. The following definitions are needed before the satisfaction definitions.

Definition 2.2: If a state s satisfies a state formula ϕ, then we say that s is a *ϕ. state*.

Definition 2.3: Let ξ be a sequence of states and $\xi(i)$ be the i^{th} state of ξ. An *interval* $[p, q]$, bounded by state formulas p and q, is given by $[p, q] = \{\xi | \ \xi(0) \models p, \xi(|\xi|) \models q\}$. Here, $|\xi|$ denotes the length of the state sequence ξ, and $\models$ denotes the satisfaction.

Definition 2.4: Let $R(\sigma) = \{\xi | \xi \le \sigma\}$, where the state sequence $\xi = (\xi(0), \ldots, \xi(|\xi|))$ refines $\sigma = (\sigma(0), \ldots, \sigma(|\sigma|))$. In symbols, $\xi \le \sigma$, iff $(\exists j \in [0, |\sigma|])\ ((\sigma(j), \sigma(j+1), \cdots \sigma(j+|\xi|)) = \xi)$. In other words, $R(\sigma)$ is the collection of subsequences of the state sequence σ.

Definition 2.5: The following satisfaction definitions are to be added to the semantics of the logic ISTL*. Let ϕ be any formula, and let p, q be state formulas.

$(\sigma, i) \models \psi$	$\equiv \sigma(i) \models \psi$, where ψ is a state formula.
$(\sigma, i) \models [p, q]\phi$	$\equiv$ for every $\xi \in R(\sigma)$ such that $(\xi, 0) \models p$, and $(\xi, |\xi|) \models q$, if $(\exists k_i \in \{0, 1, \ldots, |\xi|\})(\xi(k_i) = \sigma(i))$, then $(\xi, k_i) \models \phi$.
$\sigma \models \phi$	$\equiv \forall i \in \{0, \ldots, |\sigma|\}, (\sigma, i) \models \phi$.
$\sigma \models [p]\phi$	$\equiv (\forall i)$ (if $(\sigma, i) \models p$ then $(\sigma, i) \models \phi$).
$\sigma \models [p, q]\mathrm{EF}\phi$	$\equiv$ For all $\xi \in R(\sigma)$ such that $(\xi, 0) \models p$, and $(\xi, |\xi|) \models q$, we have $(\xi, |\xi|) \models \phi$.

Definition 2.6: A *responsiveness* assertion is a path formula of the form $([p]\phi \rightarrow [p, q]\mathrm{EF}\psi)$, where p, q, ϕ, and ψ are state formulas.

A responsiveness assertion $([p]\phi \rightarrow [p, q]\mathrm{EF}\psi)$ is true over a state sequence σ, iff the following holds: ϕ holds at $p.state$ of σ, then a $\psi.state$ will occur within the interval $[p, q]$. The assertion ensures the requirement of a timed response ψ to ϕ within the time interval $[p, q]$. The following Progress Rule can be applied to reason about responsiveness properties.

Let $p, q, r, \phi_0, \phi_1, \phi_2$ be state formulas.

Progress Rule:

$$\frac{\begin{array}{l}[p]\phi_0 \rightarrow [p, q]\mathrm{EF}\phi_1 \\ [q]\phi_1 \rightarrow [q, r]\mathrm{EF}\phi_2\end{array}}{[p]\phi_0 \rightarrow [p, r]\mathrm{EF}\phi_2}$$

According to the premises, if ϕ_0 holds at $p.state$, then there exists a path or state sequence such that ϕ_1 will occur within the interval $[p, q]$, and if ϕ_1 holds at $q.state$, then there exists a path ϕ_1 will occur within the interval $[q, r]$. Therefore, we can conclude that if ϕ_0 holds at $p.state$, then there exists a state sequence such that ϕ_2 will occur within the time interval $[p, r]$.

The following remark and definitions are needed for the proofs of soundness and completeness of the Progress Rule.

Remark: A program P can be identified with any collection Σ_p of formulas, such that a state sequence σ is generated by P iff $\sigma \models \Sigma_P$.

Definition 2.7: If P is a program, then $\Sigma_P \models \phi$ (read "ϕ is a *consequence of* Σ_P") means every state sequence σ generated by P satisfies ϕ.

Definition 2.8: Given a program P and a formula(property) ϕ, we say *P satisfies the specification* ϕ iff $\Sigma_P \models \phi$

Theorem 1 (Soundness): The Progress Rule is sound.

Proof: Assume that all the premises hold, i.e.,

(1) $\Sigma_P \models [p]\phi_0 \rightarrow [p,q]\mathrm{EF}\phi_1$, and

(2) $\Sigma_P \models [q]\phi_1 \rightarrow [q,r]\mathrm{EF}\phi_2$.

Let σ be an arbitrary state sequence generated by P,i.e., $\sigma \models \Sigma_P$. Then,

(3) For all i, $(\sigma,i) \models [p]\phi_0 \rightarrow [p,q]\mathrm{EF}\phi_1$, and

(4) For all i, $(\sigma,i) \models [q]\phi_1 \rightarrow [q,r]\mathrm{EF}\phi_2$.

The implication($\rightarrow$) of the equation (3) can be removed, and (3) is equivalent to "if $(\sigma,i) \models [p]\phi_0$, then $(\sigma,i) \models [p,q]\mathrm{EF}\phi_1$." Thus, ϕ_1 holds at t_q, where t_q is a time index of any *q.state* in state sequence σ. That is, $(\sigma,t_q) \models \phi_1$, i.e.,

(5) $\sigma \models [q]\phi_1$.

Likewise, the implication($\rightarrow$) of the equation (4) can be removed, and we can rewrite (4) as "if $(\sigma,i) \models [q]\phi_1$, then $(\sigma,i) \models [q,r]\mathrm{EF}\phi_2$." From (4) and (5), we can derive $(\sigma,i) \models [q,r]\mathrm{EF}\phi_2$. Symmetrically, we can conclude $(\sigma,t_r) \models \phi_2$, and $(\sigma,i) \models [p,r]\mathrm{EF}\phi_2$.

Hence, for all i, $(\sigma,i) \models [p]\phi_0 \rightarrow [p,r]\mathrm{EF}\phi_2$. Thus $\sigma \models [p]\phi_0 \rightarrow [p,r]\mathrm{EF}\phi_2$. So $\Sigma_P \models [p]\phi_0 \rightarrow [p,r]\mathrm{EF}\phi_2$, which means that every state sequence generated by P satisfies $([p]\phi_0 \rightarrow [p,r]\mathrm{EF}\phi_2)$.□

Theorem 2 (Relative Completeness): Suppose P is a program, $([p]\phi_0 \rightarrow [p,q]\mathrm{EF}\phi_2)$ is a responsiveness assertion, and for each state sequence σ of P, if $\sigma \models ([p]\phi_0 \rightarrow [p,q]\mathrm{EF}\phi_2)$. Then $\Sigma_P \vdash ([p]\phi_0 \rightarrow [p,q]\mathrm{EF}\phi_2)$.

Proof: From the assumption $\Sigma_P \models [p]\phi_0 \rightarrow [p,r]\mathrm{EF}\phi_2$, we need to show that there exists a proof for $[p]\phi_0 \rightarrow [p,r]\mathrm{EF}\phi_2$. Let σ be an arbitrary state sequence generated by P,i.e., $\sigma \models \Sigma_P$. Then, from the assumption, we know that $\sigma \models [p]\phi_0 \rightarrow [p,r]\mathrm{EF}\phi_2$. Now $[p]\phi_0 \rightarrow [p,r]\mathrm{EF}\phi_2$ holds on a sequence $\sigma = (s_0, s_1, \cdots)$, if whenever there exist indexes t_0, t_2, and j, such that $(\sigma,t_0) \models p$, $(\sigma,t_2) \models r$, ϕ_0 holds on s_{t_0}, and $t_0 \le j \le t_2$, ϕ_2 holds on $\sigma|_{[j,t_2]} = (s_j, s_{j+1}, \ldots, s_{t_2})$.

Let $t_1 = \max\{t | t_0 \le t < t_2$ and $\sigma \models [t_0,t]\neg\phi_2\}$, i.e., t_1 is the last point where $\neg\phi_2$ holds. Let $\hat{t}_1 = \min\{t | t_0 \le t < t_2$ and $\sigma \models [t,t_2]\phi_2\}$,i.e., $\hat{t}_1$ is the first point where ϕ_2 holds.

Let $\phi_1 = at(t_1)$ and let $\hat{\phi}_1 = at(\hat{t}_1)$; let $q = at(t_1)$ and let $\hat{q} = at(\hat{t}_1)$ Then, $\Sigma_P \vdash [p]\phi_0 \rightarrow [p, q]\mathrm{EF}\phi_1$, $\Sigma_P \vdash [q]\phi_1 \rightarrow [q, \hat{q}]\mathrm{EF}\hat{\phi}_1$, and $\Sigma_P \vdash [\hat{q}]\hat{\phi}_1 \rightarrow [\hat{q}, r]\mathrm{EF}\phi_2$. So, $\Sigma_P \vdash ([p]\phi_0 \rightarrow [p, q]\mathrm{EF}\phi_2)$.□

3. THE TRANSLATION SCHEME

In Section 2, we developed a formal tool ITL for modeling the behaviors of reactive systems within finite intervals of time. The behavior is represented by responsiveness assertions. To show that responsiveness is guaranteed in the execution environment, a translation procedure which takes into account the constraints in the operational environment is developed. This translation helps every process create a global schedule(a history) for all the (local) operations of the responsive(distributed) system. Meanwhile, processes can utilize their histories to evaluate, at observable stages, the behavior of the system characterized by responsiveness assertions. If any of the assertions are violated, then an error has occurred. Thus, the incorporations of the translation and operational evaluation of responsiveness assertions define an error-detecting algorithm. The following steps outline the generation of error-detecting algorithms.

(1) obtain responsiveness assertions from ITL specifications: the logic ITL adopts partial order semantics and provides a more suitable representation of concurrency than interleaving semantics does.

(2) develop a translation procedure: in the translation, every process maintains a history-a global view of the system through a set of auxiliary variables. These variables are used to keep track of the operations performed and observed in the system.

(3) derive error-detecting algorithms: run-time correctness is established by evaluating responsiveness assertions-obtained in the verification environment, against the history. This step is called operational evaluation of responsiveness assertion, which allows distributed processes to monitor, within finite intervals of time, whether their behavior is satisfied or not. Hence, an error-detecting algorithm is generated.

For a process P_i, the evaluation of responsiveness on process P_j requires the information about P_j. Thus, auxiliary variables are used to communicate variables in the assertions, which allows processes to evaluate responsiveness assertions or the behaviors of other processes. Hence, we can avoid having a process test itself. Now, we formally describe the operations for the translation. The following definition describes the actions with respect to auxiliary variables in the translation.

Definition 3.1: Let t_i and t_j denote the local counters of processes P_i and P_j, respectively. The actions performed in the translation include updates of auxiliary variables, sending and receiving messages, which are described below.

$(P_j, !, t_i, v)$: $(P_j!, v, t_i)$ in CSP [Hoar69] notation, which denotes that a message with content v is sent to process P_j at time t_i with respect to the clock of process P_i.

$(P_j, ?, t_i, v)$: $(P_j?v, t_i)$ in CSP notation, which denotes that a message with content v is received from process P_j at time t_i with respect to the clock of process P_i.

$(t_j, v_1, \ldots, v_n)$: at time t_j, the variables $v_1, \ldots, v_n$ are instantiated in process P_j.

The counter t_i of a process P_i is incremented by one after every execution of an operation and is updated after the receipt of a message. The incorporation of logical clocks into the translation is to obtain a total ordering of all causally related events of the system, which is based on the concept of a "happened before" relation [Lamp78].

To keep track of operations, each process must maintain a history that records all the operations performed and observed so far, which is defined below.

Definition 3.2: Let V_{h_j} be the collection of operations observed by process P_j, where the operations are described in Definition 3.1. V_{h_j} is used to keep track of the operations involving state variables in the assertions to be evaluated. These state variables in assertions are referred to as auxiliary variables.

Each process keeps a collection of sets of auxiliary variables with respect to the other processes in the system, so that every process has state information of other processes and can evaluate whether an assertion about the behavior of other processes is satisfied. The following definition describes the auxiliary variables maintained by process P_j with respect to n processes in the system.

Definition 3.3: Let G_j be a collection of subsets $g_{j0}, g_{j1}, \ldots, g_{j(n-1)}$, where each subset is a set of operations from Definition 3.1. The set $g_{ji}(j{\neq}i)$ represents the changes made to the auxiliary variables in P_j since the last communication with P_i; g_{jj} describes the auxiliary variables updated by process P_j since the last communication with any process.

Notice that each set $g_{ji}(j{\neq}i)$ can be considered as a queue of process P_j to be sent on next communication with process P_i. Before the communication with P_i, P_j updates its queues $(g_{j0}, \ldots, g_{j(i-1)}, g_{j(i+1)}, \ldots, g_{j(n-1)})$ with respect to the operations in g_{jj}, which records all the operations since last communication with any process. Also, the global view or the history V_{h_j} is updated with respect to g_{jj}. The following definition describes these operations to be performed before two processes communicate.

Definition 3.4: The function $update_{bc}(g_{ji}, g_{jj}, V_{h_j}, t_j)$ describes the operations performed by P_j before the communication with P_i.

$update_{bc}(g_{ji}, g_{jj}, V_{h_j}, t_j)$
(1) apply each operation in g_{jj} to $(G_j \backslash \{g_{jj}\})$;
(2) $V_{h_j} \leftarrow V_{h_j} \| g_{jj}$;
(3) $g_{jj} \leftarrow \varnothing$;
(4) $t_j := t_j + 1$;

The above equations are explained below.

(1) update $(G_j \backslash \{g_{jj}\})$ with respect to g_{jj}, where "\" denotes set difference.

(2) record operations in g_{jj} , where "||" denotes concatenation.

(3) g_{jj} is set to empty, since $(G_j \backslash \{g_{jj}\})$ are updated.

(4) increment the counter t_j.

Before the communication, processes P_i and P_j perform their respective updates of $update_{bc}(g_{ij}, g_{ii}, V_{h_i}, t_i)$ and $update_{bc}(g_{ji}, g_{jj}, V_{h_j}, t_j)$; g_{ij} and g_{ji} are exchanged when P_i and P_j communicate. The operations in $update_{bc}$ are performed before the communication. The following definition formally defines the operations following the communication of non-auxiliary variables.

Definition 3.5: The function $update_{ac}$ describes the communications of auxiliary variables and the updates following the exchanges of auxiliary variables. let V_{h_i} be the collection of operations observed by P_i and let g_{recv} denote the auxiliary variables received by P_i from P_j. The following function describes the operations performed by P_i.

$update_{ac}(g_{ij}, g_{recv}, V_{h_i}, t_i)$
(1) $(P_j, ?, t, g_{recv})$; $t_i := \max(t_i, t) + 1$;
(2) $V_{h_i} \leftarrow V_{h_i} \| (P_j, ?, t, g_{recv})$;
(3) $(P_j, !, t_i, g_{ij})$; $t_i := t_i + 1$;
(4) $V_{h_i} \leftarrow V_{h_i} \| (P_j, !, t_i, g_{ij})$;
(5) $V_{h_i} \leftarrow V_{h_i} \| g_{recv}$;
(6) $t_i := t_i + 1$;

The above equations are explained as follows.

(1) receive auxiliary variables of P_j in g_{recv}, and increment the counter t_i.

(2) record the operation of (1) in V_{h_i}.

(3) send g_{ij} to process P_j and increment the clock t_i.

(4) record the operation of (3) in V_{h_i}.

(5) record the operations of g_{recv} in history V_{h_i}.

(6) increment the counter t_i.

Observe that, in (5), there is no need to consider duplicate tuples and we can append g_{recv} to the history V_{h_i} of process P_i directly, since all the operations in g_{recv} are instantiated by process P_j. The function $update_{ac}(g_{ji}, g_{recv}, V_{h_j}, t_j)$, which describes the operations performed by $P_j (j < i)$, has the same operations as in $update_{ac}(g_{ij}, g_{recv}, V_{h_i}, t_i)$ except that process P_j sends its queue g_{ji} before it waits to receive the auxiliary variables of P_i, g_{ij}. Notice that $(j < i)$ is used to introduce an arbitrary order to the communications of auxiliary variables, which avoids the occurrence of deadlock. The interchange of auxiliary variables is described in Figure 3.1 for one matching communication pair between P_i and P_j.

3.1 Soundness and Completeness of the Translation

This section shows that the assertions, derived from the verification proof or the verification environment, can be preserved after the transformation. The transformation is a process which considers operational constraints and generates a consistent view for distributed processes of the system. In [TsIM93], we have shown soundness and relative completeness of the translation with respect to all formulas(properties) except interval formulas of the form $[p, q]\phi$. Now we have to show that the translation preserves soundness and relative completeness with respect to interval formulas. Then, responsiveness assertions ($[p]\phi \rightarrow [p, q]\mathrm{EF}\psi$) can be obtained from interval formulas and implication. Therefore, responsiveness assertions are preserved after the translation, i.e., they are preserved in the operational environment.

Let Σ_P identify the collection of formulas for program P. Let $TR(\Sigma_P)$ identify the collection of formulas for the corresponding program of P after the translation. Then $TR(\Sigma_P) = \Sigma_P \cup S$, where S is the collection of formulas for the operations of *update* and the communications of auxiliary variables in the translation. $\Sigma_P \subseteq TR(\Sigma_P)$, since the translation helps processes to communicate their views of the system, and affects neither program variables nor the control of flow. Thus, $TR(\Sigma_P)$ includes Σ_P, in addition to the formulas(assertions) which describe update operations and the communications of auxiliary variables. Notice that if a formula $\phi \in S$, then ϕ can be derived from Σ_P using the same proof rules

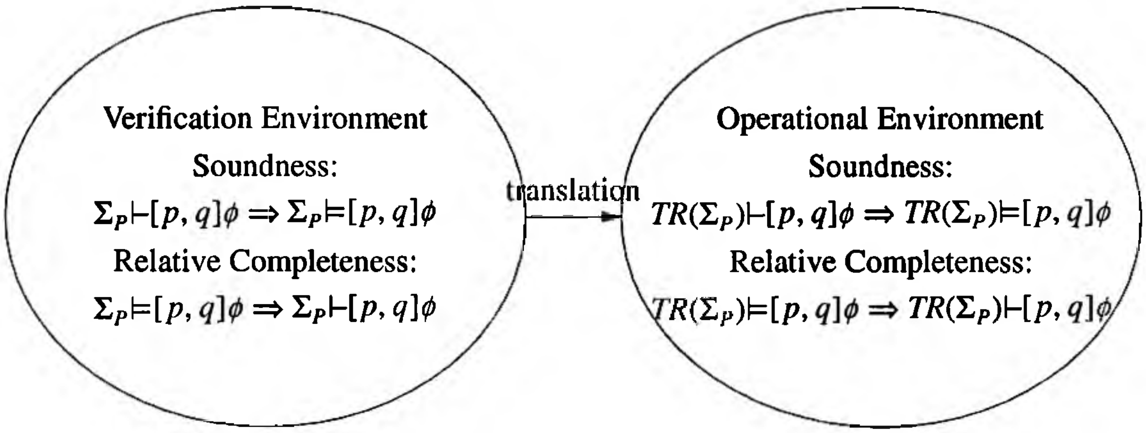
```
For process Pi:
/* execute arbitrary set of statements excluding communication */
/*but including assignments to auxiliary variables */
Si1; ti:=1
Si2; ti:=ti + 1
...;
Sik; ti:=ti + 1
/* update the auxiliary variables */
update_bc(gij, gii, V_hi, ti);
/* perform communications with process Pj */
(Pj?VAR, t); ti:=max(ti, t) + 1
/* and update the auxiliary variables */
update_ac(gij, grecv, V_hi, ti);

For process Pj:
/* execute arbitrary set of statements excluding communication */
/*but including assignments to auxiliary variables */
Sj1; tj:=1
Sj2; tj:=tj + 1
...;
Sjk; tj:=tj + 1
/* update the auxiliary variables */
update_bc(gji, gjj, V_hj, tj);
/* perform communications with process Pi */
(Pi!VAR, t); tj:=tj + 1
/* and update the auxiliary variables */
update_ac(gij, grecv, V_hj, tj);
```

Figure 3.1. communications of auxiliary variables for one matching communication pair.

(i.e., $\Sigma_P \vdash \phi$), since both environments have the same proof rules and we can inductively show that the weakest predicate of ϕ is true in the verification environment($\Sigma_P \models \phi$).

The interval formula $[p, q]\phi$ describes a sequence of operations and contains no auxiliary variables, where p and q are state formulas and ϕ is a path formula. The following two



theorems show that the translation preserves soundness and relative completeness in the operational environment.

Theorem 3 (Soundness): If $TR(\Sigma_P) \vdash [p,q]\phi$, then $TR(\Sigma_P) \models [p,q]\phi$. That is, if there is a proof of $[p,q]\phi$ in the operational environment, then $[p,q]\phi$ is true in the operational environment.

Proof: First we need to show that there is a proof of $[p,q]\phi$ in the verification environment. By assumption, we know that there is a proof of $[p,q]\phi$ in the operational environment($TR(\Sigma_P) \vdash [p,q]\phi$). $TR(\Sigma_P) \vdash [p,q]\phi$ says that there exists a sequence $TR(\phi_0), TR(\phi_1), \ldots, TR(\phi_{n-1})$ of formulas such that the consequence $TR(\phi_{n-1})$ is $[p,q]\phi$, and each formula can be obtained from the previous formulas by the proof rules from the proof system.

For each formula $TR(\phi_k)$ $(0 \le k < n)$, if $TR(\phi_k)$ contains auxiliary variables $(TR(\phi_k) \in S)$ results from the execution of *update* or communication of auxiliary variables, then we can replace $TR(\phi_k)$ by a sequence $TR(\phi_{k_0}), TR(\phi_{k_1}), \ldots, TR(\phi_{k_m})$ of formulas having no auxiliary variables, because $\Sigma_P \models TR(\phi_k)$. Thus, we can construct a corresponding sequence $\phi_0, \phi_1, \ldots, \phi_{n-1}$ of formulas, such that each $TR(\phi_k)$ has a corresponding formula ϕ_k in the verification environment.

Since we have shown soundness in the verification environment, $[p,q]\phi$ is true for all executions in the verification environment, i.e. $\Sigma_P \models [p,q]\phi$. Then $[p,q]\phi$ is true in the operational environment $(TR(\Sigma_P) \models [p,q]\phi)$, because $\Sigma_P \subseteq TR(\Sigma_P) = (\Sigma_P \cup S)$, and $\Sigma_P \models S$.□

Theorem 4 (Relative Completeness): If $TR(\Sigma_P) \models [p,q]\phi$ then $TR(\Sigma_P) \vdash [p,q]\phi$. That is, if $[p,q]\phi$ is true in the operational environment, then there is a proof of $[p,q]\phi$ in the operational environment.

Proof: If $[p,q]\phi$ is true in the operational environment $(TR(\Sigma_P) \models [p,q]\phi)$, first we need to show that $[p,q]\phi$ is true in the verification environment. $TR(\Sigma_P) \models [p,q]\phi$ says that given an arbitrary computation σ, $\sigma \models [p,q]\phi$, i.e., $[p,q]\phi$ is true in the operational environment.

Let σ' be an arbitrary computation in the verification ($\sigma' \models \Sigma_P$). Let σ be the computation obtained from σ by introducting the transitions of *update* and communications of auxiliary variables. In other words, σ' is a projection of σ ($\sigma' = \Pi(\sigma)$). Since σ is a computation in the operational environment ($\sigma \models TR(\Sigma_P)$), $\sigma' = \Pi(\sigma)$, and $\sigma \models [p,q]\phi$, we have $\sigma' \models [p,q]\phi$. Then $\Sigma_P \models [p,q]\phi$, i.e., $[p,q]\phi$ is true in the verification environment. Thus, $\Sigma_P \vdash [p,q]\phi$, since we have shown relative completeness in the verification environment. $\Sigma_P \vdash [p,q]\phi$ says that there exists a sequence($\phi_0, \phi_1, \ldots, \phi_{n-1}$) of formulas such that the consequence $[p,q]\phi$ is ϕ_{n-1}, and each formula can be obtained from the previous formulas by proof rules in the proof system.

Now we need to show that there is a proof of $[p,q]\phi$ in the operational environment. That is, in the operational environment there exists a sequence $TR(\phi_0), TR(\phi_1), \ldots, TR(\phi_{n-1})$, which correspond to the sequence $\phi_0, \phi_1, \cdots, \phi_{n-1}$ of formulas in the verification environment. Since both environments have the same proof rules, each ϕ_i is $TR(\phi_i)$ in the operational environment. Thus, we can construct a corresponding sequence $TR(\phi_0), TR(\phi_1), \ldots, TR(\phi_{n-1})$ of formulas in the operational environment. This concludes that in the operational environment there is a proof of $[p,q]\phi$, i.e., $TR(\Sigma_P) \vdash [p,q]\phi$.□

3.2. OPERATIONAL EVALUATION OF RESPONSIVENESS ASSERTIONS

We have shown that the translation process defines a global schedule (a history) for all the local operations of processes in the distributed system. This history is defined in terms of multiple clock readings through the incorporation of a "happened before" relation[Lamp78]. This section describes how processes can utilize their own histories to check the satisfaction of responsiveness assertions in the execution environment.

The idea is that the run-time satisfaction of responsiveness assertions is guaranteed if there exist no tuples in a history that *violate* the assertion to be evaluated. Notice that in the verification we check all possible behavior or execution sequences to conclude that a responsiveness assertion is satisfied. However, in the operational environment, we examine if a history is ever violated, i.e., if there exists a tuple in the history V_h which satisfies the negation of the assertion to be evaluated. If so, an error has occurred. The evaluation of responsiveness assertions is formally defined below.

Definition 3.6: Let V_h be the collection of tuples, which records the operations observed by a process in a distributed system. The tuples of V_h are of the form $(t, v_1, v_2, \ldots, v_n)$, which denotes the instances of $v_1, v_2, \ldots, v_n$ at time t.

Definition 3.7: Point violation-a formula ϕ is violated in a history V_h, iff there exists a tuple

$Q = (t, v_1, v_2, \ldots, v_n)$ such that $Q \not\models \phi$. In symbols, there exists a mapping Π on the history V_h and a formula ϕ, $\Pi(V_h, \phi) = \{Q \mid Q \not\models \phi\}$. In particular, $[T]\phi$ is violated in the history V_h, iff the set $\Pi(V_h, [T]\phi) = \{Q = (t, v_1, \ldots, v_n) \mid Q \not\models \phi$ and $Q \models T\}$ is nonempty.

Definition 3.8: Interval violation-an interval formula $[T_1, T_2]\phi$ is violated in a history V_h iff the set $\Pi(V_h, [T_1, T_2]\phi) = \{Q = (t, v_1, \ldots, v_n) \mid Q \not\models \phi$ and $Q \in [T_1, T_2]\}$ is nonempty.

Definition 3.9: An responsiveness formula $([T_1]\phi_1 \rightarrow [T_1, T_2]\mathrm{EF}\phi_2)$ is satisfied for a history V_h iff the set $\Pi(V_h, [T_1]\phi_1)$ is empty, and the set $\Pi(V_h, [T_1, T_2]\mathrm{EF}\phi_2) = \{Q = (t, v_1, \ldots, v_n) \mid Q \models \phi_2$ and $Q \in [T_1, T_2]\}$ is empty as well.

4. COMPLEXITY

This section examines the overhead of operational evaluations of assertions from two aspects–computation cost and communication cost. First, we consider the computation cost incurred by the translation and operational evaluations of assertions. Upon communications, assertions are examined against a history V_h. If there is a violation in the history V_h, then an error is signaled. The overhead of evaluating an assertion, i.e., $\Pi(V_h, \textit{assertion})$, is described below.

- For a process with m operations, there are at most m tuples created, one tuple for each operation involving the modifications of auxiliary variables. Notice that auxiliary variables are state variables in the assertions to be evaluated, and no tuples are generated for those operations involving no auxiliary variables. In other words, it is sufficient to maintain relevant state information for the evaluation of assertions.
- The overhead of maintaining a consistent view(V_h) for a process is $O(mn)$–every process has to maintain a copy of auxiliary variables for n processes in the system. Thus, the length of V_h or the number of tuples in V_h is at most mn.

In summary, the cost of operationally evaluating an assertion is $O(mn)$ which accounts for the examination of mn tuples in V_h after a communication. Since there are C communications(i.e., C evaluations of assertions), $O(mn \times C)$ is the computation cost induced by the translation and the operational evaluations of assertions. However, the length of history V_h is problem-dependent. The maximum length of V_h is mn, which can be reduced as follows. Let's say, the evaluation $\Pi(V_h, \textit{assertion}_1)$ is ahead of the evaluation $\Pi(V_h, \textit{assertion}_2)$, and there are no evaluations in between. If the time indexes of $\Pi(V_h, \textit{assertion}_2)$ are later or greater than those of $\Pi(V_h, \textit{assertion}_1)$, then the history V_h can be chopped off, i.e., the tuples in V_h can be removed up to the one with the largest time index in $\Pi(V_h, \textit{assertion}_1)$. The reason is that the time indexes of tuples in V_h are generated according to a "happened before" [Lamp78] relation, and there are no overlapped time indexes between the sets $\Pi(V_h, assertion_1)$ and $\Pi(V_h, assertion_2)$, so there is no need to keep the tuples for $assertion_1$.

Then, we consider the overhead of communication. In the translation, exchanges of auxiliary variables are needed to maintain a consistent view among processes in the system. The exchanges of auxiliary variables occur after each communication of non-auxiliary variables. For the sake of efficiency, auxiliary variables are piggybacked in the communication. Therefore, the number of communications of auxiliary variables incurred by the translation is C, where C is the number of communications in the underlying algorithm.

Notice that the piggybacking of auxiliary variables does not incur much cost. This can be understood as follows. The time to transfer data between two processes is $(S + RL)$, where S is the setup time, R is the transfer rate(in secs/byte), and L is the length of the message. Typically numbers, for a Sun 4/20 using TCP/IP on an IEEE CSMA/CD, yield, S = 16msec, and R is $(10Mb/sec)^{-1}$. The additional message will affect the transfer time, when $RL \geq S$, this happens when $L \geq 10^4$, and for a $10bytes$ message, $mn \geq 1000$.

To sum up, the communications introduced by the translation is linearly proportional to the communications in the underlying algorithm, while the computation cost can be greatly reduced when the time indexes in the assertins to be evaluated are not overlapped.

5. TRAIN SET EXAMPLE

Safety-critical systems usually involve the interactions between the controller and the physical process. In the train set example [LeSW92, LeSA92], the physical process consists of primary track circuit C_p and secondary track circuit C_s, and two types of train-T_p and T_s. The circuits are divided into sections and there are two crossing sections where the two circuits intersect. Each section has a sensor, while for each train, there is an actuator that can stop the train within any section. To avoid accidents, a train has to get permission from Circuit to enter the next section, and get permission from DANGER_ZONE to enter the danger zone(crossing section). The circuit C_p, C_s, and the crossing section CC are illustrated below.

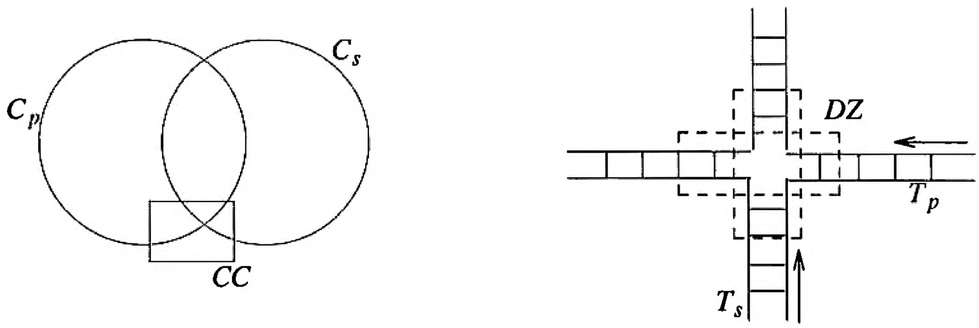


Figure 5.1. The train set circuits and the crossing section

Behavior of Sensors and Actuators

In this example, we are concerned with those failures of sensor or actuator that can affect the safety of the system. For example, the sensor failure that misses a train or the actuator failure that fails to stop a train. The behavior and failure behavior of the sensors and actuators are defined after the following description of the variables to be used.

- c denotes type of circuit, $c \in L = \{p, s\}$.
- x, y denote trains, $x, y \in Tr = \{1, \ldots, N_{tc}\}$.
- i, j are sections, $i, j \in Sc = \{0, \ldots, N_{sc}\}$.
- Addition $\oplus$ and subtraction $\ominus$ on section numbers are performed modulo the number of sections of the circuit.

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

Definition 5.2: The behavior of a sensor is denoted by $SS(x, i)$-a sensor detects train x in section i. The failure behavior of a sensor is denoted by $FS(x, i)$-a sensor fails to detect a train x in section i. Thus, $SS(x, i) = on(x, c) \wedge Sens(c, Ptrain(x))$; $FS(x, i) = on(x, c) \wedge \neg Sens(c, Ptrain(x))$.

Definition 5.3: The behavior of an actuator is denoted by $SS(x, j)$-when the actuator is set, train x cannot enter a new section. The failure behavior of an actuator is denoted by $FA(x, j)$-when the actuator of train x is set, train x enters a new section. Thus, $SS(x, j) = ([T_{xj}](Act(x, j) \wedge Ptrain(x) = j)) \wedge ([T_{xj}, T_{xj+1}]Ptrain(x) = j)$; $FA(x, j) = ([T_{xi}](Act(x, i) \wedge Ptrain(x) = i) \wedge ([T_{xi}, T_{xi+1}]Ptrain(x) > i)$.

Appendix A shows the algorithm of train, circuit and section, where P_{tr}, P_c, and P_s denote the processes train, ciruit, and sections, respectively. Process P_s informs processes P_c and P_{tr} upon a detection of a train entering a section. Process circuit-P_c localizes sensor and

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 5.2. The state variables

actuator failures. P_c signals minor failure if the number of consecutive sensor failures is less than or equal to a constant *mcsf*. Major failure occurs if an actuator fails to stop a train, or the number of consecutive sensor failures is greater than *mcsf*.

Figure 5.3 shows the safety constraints that must be guaranteed by the system. These safety constraints are represented by ITL formulas and can be operationally evaluated according to the mapping defined in Section 3.1. The reservation constraint(SC1) states that for any train, the current occupied section and the following mcsf⊕1 sections must always be reserved. The exclusion constraint(SC2) asserts that mutual exclusion must be achieved for reserved sections, i.e., no section can be reserved by more than one train. SC3 is a responsiveness property, which says that if the number of consecutive sensor failures is greater than *mcsf*, then the system must be shut down. SC3 can be formalized and proved by Progress Rule as follows.

Let T_{xl} denote the time when train x enters section l. Let $T_{x\bar{l}}$ denote the point of time immediately before $T_{xl\oplus 1}$. The sensor failures on section i can be represented by the responsiveness assertion: $[T_{xl}, T_{x\bar{l}}]EF(On(c,x) \wedge \neg Sens(c, Ptrain(x)))$. Likewise, we can derive the corresponding formulas for the sensor failures of the following *mcsf* sections. By Progress Rule and the following premises, we can conclude $[T_{xl}, T_{x\overline{l \oplus mcsf}}]EF(Shut\_Down)$. In other words, if the following premises hold,

Name	Safety Constraints	ITL formulas
$SC1$	Reservation constraint: for any train, the current occupied section and the following mcsf⊕1 sections must always be reserved. $\forall c \in L, \forall x \in Tr$: $on(x, c) \wedge \{Ptrain(x) \ominus (mcsf \oplus 1), \ldots, Ptrain(x)\} \subseteq Rtrain(x)$	$SC1(T_{xl}, T_{x\bar{l}}) = (\forall x \in Tr)$ $([T_{xl}, T_{x\bar{l}}] Ptrain(x) \notin Rtrain(x)) \Rightarrow ERROR.$
$SC2$	Exclusion constraint: mutual exclusion must be achieved for reserved sections, i.e., no section can be reserved by more than one train: $\forall c \in L, \forall x, y \in Tr$: $x \neq y \Rightarrow Rtrain(x) \cap Rtrain(y) = \emptyset$	$SC2(T_{xl}, T_{x\bar{l}}, y) = (\forall x, y \in Tr)$ $([T_{xl}, T_{x\bar{l}}](On(x, c) \wedge On(y, c) \wedge (x \neq y) \wedge$ $Rtrain(x) \cap Rtrain(y) \neq \emptyset) \Rightarrow Error.$
$SC3$	If the number of consecutive sensor failures is greater than mcsf, then the system must be $Shut\_Down$. $\forall c \in L, \forall x, y \in Tr$, $[T_{xi}](Ptrain(x) = i) \wedge [T_{x\overline{i \oplus mcsf}}](Ptrain(x) > i \oplus mcsf) \wedge [T_{xi}, T_{xi \oplus mcsf}] (\neg Sens(c, i) \wedge \cdots \wedge \neg Sens(c, i \oplus mcsf))$ then $Shut\_Down$.	Let $SC3.1(T_{xl}, T_{x\bar{l}}, c) = (\forall x \in Tr)$ $([T_{xl}, T_{x\bar{l}}](On(c, x) \wedge \neg Sens(c, Ptrain(x))).$ Let $SC3.2(T_{xl}, T_{x\overline{l \oplus mcsf}}, Shut\_Down) = (\forall x \in Tr)$ $[T_{xl}, T_{x\overline{l \oplus mcsf}}] EF(Shut\_Down)).$ $SC3(T_{xl}, T_{x\overline{l \oplus mcsf}}) =$ $SC3.1(T_{xl}, T_{x\bar{l}}, c) \wedge$ $SC3.1(T_{xl}, T_{x\overline{l \oplus 1}}, c) \wedge$ $\ldots$ $\wedge SC3.1(T_{xl}, T_{x\overline{l \oplus mcsf}}, c) \wedge$ $\neg SC3.2(T_{xl}, T_{x\overline{l \oplus mcsf}}, Shut\_Down)$ $\Rightarrow Error.$
$SC4$	If an actuator ever fails, the system must be Shut_Down. $[T_{xi}](Act(x, i) \wedge Ptrain(x) = i)$ and $[T_{xi}, T_{xi \oplus 1}](Ptrain(x) > i)$ then $Shut\_Down$.	$SC4(T_{xi}, T_{xi \oplus 1}) = (\forall x \in Tr)(\exists i \in Sc)$ $([T_{xi}](Act(x, i) \wedge Ptrain(x) = i) \wedge$ $[T_{xi}, T_{xl \oplus 1}](Ptrain(x) > i) \wedge$ $\neg([T_{xi}, T_{xi \oplus 1}] EF(Shut\_Down)))$ $\Rightarrow Error.$

Figure 5.3. Safety constraints of the system

$$[T_{xl}, T_{x\bar{l}}]EF(On(c, x) \wedge \neg Sens(c, Ptrain(x)))$$
$$[T_{xl+1}, T_{x\overline{l+1}}]EF(On(c, x) \wedge \neg Sens(c, Ptrain(x)))$$
$$\cdots$$
$$[T_{xl \oplus mcsf}, T_{x\overline{l \oplus mcsf}}]EF(On(c, x) \wedge \neg Sens(c, Ptrain(x)))$$

then $[T_{xl}, T_{x\overline{l \oplus mcsf}}]EF(\neg Sens(c, Ptrain(x)) \wedge \cdots \wedge \neg Sens(c, Ptrain(x)))$. Since *mcsf* consecutive sensor failures occur, we derive $[T_{xl}, T_{x\overline{l \oplus mcsf}}]EF(Shut\_Down)$.

SC4 is also a responsiveness assertion. It asserts that if an actuator ever fails, the system must be shut down. Whenever an actuator is set and the position of train x is in section i (i.e. $[T_{xl}](Act(x, i) \wedge Ptrain(x) = i)$, and the train moves (i.e., $[T_{xi}, T_{xi+1}](Ptrain(x) > i)$ then the system must be shut down within the interval $[T_{xi}, T_{xi+1}]$, i.e., $[T_{xi}, T_{xi+1}]EF(Shut\_Down)$.

The safety constraints of the controller involving sensors and actuators can be checked as follows. The translation process creates a history for each process, where the history consists of a collection of tuples. In this example, each tuple contains the information of $On(x, c)$, $Ptrain(x)$, $Rtrain(x)$, $Sens(c, i)$, $Act(x, j)$, $Shut\_Down$, and so on. The operational evaluation of safety constraints against a history is performed at communication points. For example, process circuit-P_c checks the satisfaction of safety constraint SC1 by examining the tuples in its history V_h to see if the set $\Pi(V_h, SC1)$ is not empty. Appendix A shows operational evaluations of assertions for processes train, circuit, and section, denoted by P_{tr}, P_c, and P_s, respectively.

5.1 A Model of Performance

To describe the overhead of the proposed technique-operational evaluation, a theoretical model for the train set example is presented. For comparison purpose, the translated algorithm or the algorithm after the translation can be rewritten in terms of the underlying algorithm. Let t_{orig} denote the execution of the underlying algorithm, where t_{orig} consists of computation time t_{comp}, and communication time t_{comm}, i.e.,

$$t_{orig} = t_{comp} + t_{comm}.$$

Let t_{assert} be the overhead of operational evaluation of an assertion, i.e., t_{assert} denotes the time of finding tuples for *assertion* in V_h by performing $\Pi(V_h, assertion)$. The overhead of the translated algorithm includes t_{orig} and the cost introduced by the translation and operational evaluation, described in Section 4. Then,

$$t_{trans} \cong 2 \times t_{comp} + 2 \times t_{comm} + C \times t_{assert},$$

where t_{comp} can be considered as the maximum time to create tuples for the history V_h, t_{comm} can also denote the exchanges of auxiliary variables, and $C \times t_{assert}$ is the evaluation of assertions on C communications. Notice that t_{assert}, the examination of tuples in history V_h, is less than or equal to t_{comp}, since only operations involving relevant information in assertions, e.g. the state variables in Figure 5.2 for the train set example, generate tuples. Moreover, the tuples for the previous evaluation can be removed, if we do not operationally evaluate overlapped intervals. Thus, t_{assert} can be rather small when the time indexes in assertions are not overlapped. Therefore, t_{extra}–the cost introduced by the translation and operational evaluation is

$$t_{comp} + t_{comm} + C \times t_{assert} = t_{orig} + C \times t_{assert}.$$

However, most of the assertions to be evaluated, in this example, have non-overlapped time indexes except the assertion $SC3$: if the number of consecutive sensor failures is greater than $mcsf$, then the system must be shut down. Thus, the tuples in V_h start to accumulate only when consecutive sensor failures occur. Let's say, the average length of the history V_h is $1/\delta$ of the maximum length of V_h, and it takes $1/\delta t_{assert}$ to evaluate assertions on each communication. Then,

$$t_{extra} \cong t_{orig} + C/\delta t_{assert}.$$

For the train set example, the translation process and operational evaluation denoted by boxed statements are shown in Appendix, where $C = 7$. Therefore, in the worst case, there is $(t_{orig} + 7/\delta t_{comp})$ overhead than the underlying algorithm, since the parameters, t_{assert} and t_{comp}, of t_{extra} denote the worst case scenario.

6. CONCLUSION

This paper extends the Changeling methodology to operationally ensure responsiveness-a crucial attribute of the responsive system. Since the correctness of system behavior established from the verification usually depends on the operational environment, a transformation procedure is developed to cope with this dependency. Also, a history or a time-indexed computation history is generated in the translation; this history can be applied to operational evaluation of responsiveness properties. If a history is violated, then a singal is raised. Therefore, the operational evaluation of assertions checks run-time satisfaction of expected behavior, and hence provides error detection-a step toward safeware safety.

REFERENCES

[LeSA92] de Lemos, R., Saeed, A. and Anderson, T., "Analysis of Timeliness Requirements in Safety-Critical Systems,", *Lecture Notes in Computer Science 571,* Formal Techniques in Real-Time and Fault-Tolerant Systems, 1992. pp. 171-192.

[LeSW92] de Lemos, R., Saeed, A. and Waterworth, A., "Exception Handling in Real-Time Software from Specification to Design,", *The Second International Workshop on Responsive Computing Systems,* 1992, pp. 108-121.

[Hoar69] Hoare, C. "An Axiomatic Basis for Computer Programming", *Communications of the ACM,* 12, 10, 1969, pp. 576-583.

[Lamp78] Lamport, L., "Time, Clocks and the Ordering of Events in a Distributed System," *Communications of the ACM,* Vol. 21, No. 7, 1978, pp. 558-565.

[LuMc91b] Lutfiyya, H. and McMillin, B. "Formal Generation Of Executable Assertions For A Fault-Tolerant Parallel Matrix Relaxation," UMR Department of Computer Science Technical Report CSC-91-13, October, 1991.

[LuSM92] Lutfiyya, H., Schollmeyer, M., and McMillin, B., "Fault-Tolerant Distributed Sort Generated from a Verification Proof Outline," *2nd Responsive Systems Symposium,* Springer-Verlag (to appear).

[LuSM92a] Lutfiyya, H., Schollmeyer, M., and McMillin, B., "Formal Generation of Executable Assertions for Application-Oriented Fault Tolerance," UMR Department of Computer Science Technical Report CSC-92-15, October, 1992.

[LuSM92b] Lutfiyya, H., Sun, A., and McMillin, B., "Fault-Tolerant Concurrent Branch and Bound Algorithm Derived from Program Verification," *The Sixteen Annual Internal Computer Software and Applications Conference,* 1992, pp. 182-187.

[Male90] Malek, M., "Responsive Systems: A Challenge for the Nineties," Keynote Address, *Proc. EUROMICRO'90, 16th Symp. in Microprocessing and Microprogramming,* Amersterdam, The Netherlands, North Holland, 1990, pp. 9-16, pp. 622-628.

[Mok91] Mok, Aloysium K. "Coping with Implementation Dependencies in Real-Time Systme Verification," Lecture Notes in Computer Science 600, *Real-Time: Theory in Practice,* 1991 pp. 485-501.

[PePn90] Peled, D. and Pnueli, A. "Proving Partial Order Liveness Properties," *17th Colloquium on Automata, Language and Programming,* edited by M.S. Peterson, 1990, pp. 553-571.

Appendix A

P_s:: /* **Process Section IS** */
Sn:Section_Number;
Cn:Circuit_Number;
Tn:Train_Number;
/*Distances from danger zones */
Dist_DZ: integer;
Sens:boolean; /*Signal of the sensor.*/

begin

while(true) {
$\boxed{update_{bc}(g_{ssn}, g, V_{h_s}, t_s)}$
/* Sensor detects a train entering a section */
[<Signal from sensor> → Sens:=TRUE;]
$\boxed{update_{ac}(g_{ssn}, g_{recv}, V_{h_s}, t_s)}$
/* evaluate state information of sensor */
/* no tuples satisfy the assertion: T_{xSn}–train x on section Sn */
$\boxed{\text{if } \Pi(V_h, T_{xS_n}) = \varnothing \text{ then Error}}$
[(Sens) →
/*set the signal Sens to false after detection of train entering a section.*/
Sens:= false;
$\boxed{update_{bc}(g_{sc}, g, V_{h_s}, t_s)}$
/*inform the process circuit-P_c train x in Section Sn.*/
$(P_c!x, Sn)$;
$\boxed{update_{ac}(g_{sc}, g_{recv}, V_{h_s}, t_s)}$
/* evaluate state information about P_c-circuit */
/* SC1–train x is on section Sn which is not reserved, then error */
/* SC2–Section Sn are reserved by trains x and y, then error */
$\boxed{\text{if } \Pi(V_h, SC1(T_{xSn}, T_{x\overline{Sn}}) \vee SC2(T_{xSn}, T_{x\overline{Sn}}), y)) = \varnothing \text{ then Error}}$
/*inform the process danger zone.*/
$\boxed{update_{bc}(g_{sDZ}, g, V_{h_s}, t_s)}$
$(P_{DZ}, x, Sn, Dist\_DZ)$;]
$\boxed{update_{ac}(g_{sDZ}, g_{recv}, V_{h_s}, t_s)}$
/* evaluate state information about P_{DZ}-danger zone*/
$\boxed{\text{if } \Pi(V_h, SC1(T_{xSn}, T_{x\overline{Sn}}) \vee SC2(T_{xSn}, T_{x\overline{Sn}}), y)) = \varnothing \text{ then Error}}$

}

```
        end while;
    END.

Pc:: /*Process Circuit IS */
    NOS: Set_of_Section_Numbers;
    /*max. # of consecutive sensors failure.*/
    mcsf:integer;
    /*NS:the set of sections that are reserved */
    NS: Reservation_Flag_Table;
    /*B1:Record about NS */
    /*B2:Record about failure sensors */
    B1,B2: Reservation_Flag_Record;
    No_Failure:boolean;
    Cn:Circuit_Number;
    Sn:Section_Number;
    Tn:Train_Number;
    ti: integer; //Table index.
    Curr:integer;
    Current: B2_Tuple;
begin
    while(true) {
        /*read sensor- Train Tn in Section Sn from process section-Ps */
        /*adds Sn to the set of section numbers-NOS.*/
        update_bc(g_csn, g, V_hc, t_c)
        (Ps?x, Sn)→ NOS := NOS + Sn;
        update_ac(g_csn, g_recv, V_hc, t_c)
        /* evaluate state information of Ps, section */
        /* no tuples satisfy the assertion: T_xSn–train x occupies section Sn */
        if Π(V_h, T_xSn) = ∅ then Error
        /* Localize_Sensor_Failure RSS 1,1,1 (rssc); */
        ti := 0;
        No_Failure := false;
        Curr := NOS.Get;

        while (ti≠NS.NO_of_Entries) and ¬No_Failure {
        /*check if the section ti is reserved.*/
```

ti := ti +1;
[Curr = NS[ti].sn → No_Failure := true;
B1:= B1+New_Entry(NS[ti].Tn,Curr,true);
NS:= NS-NS[ti];]
}
end while;
[(ti=NS.NO_of_Entries) ∧ ¬No_Failure →
B2:= B2+New_Entry(NS[ti].Tn,Curr,false);]

/*Minor_failure: section ti is not reserved and # of consecutive sensor failures is less than mcsf.*/
[¬No_Failure ∧ (B2.NO_of_Entries≤mcsf) → Minor_Failure
□
[¬No_Failure ∧ (B2.NO_of_Entries>mcsf) → Major_Failure;]

Test_Reserve_Section;
NS := NS+Curr; /*section curr is reserved */
$\boxed{update_{bc}(g_{ctr}, g, V_{h_c}, t_c)}$
(No_Failure) $\wedge(P_{tn}!Sn, Cn)\rightarrow true$;
$\boxed{update_{ac}(g_{ctr}, g_{recv}, V_{h_c}, t_c)}$
/* evaluate process train -P_{tr} */
$\boxed{\text{if } \Pi(V_h, SC1(T_{rSn}, T_{r\overline{Sn}}) \vee SC2(T_{rSn}, T_{r\overline{Sn}}), y)) = \varnothing \text{ then Error}}$
}
end while;
END.

P_{tr}:: /* **Process Train IS** */
Cn:Circuit_Number;
Sn, DZ:Section_Number;
/*AES:train is allowed to enter section */
/*AEDZ:train is allowed to enter danger zone(DZ) */
AES, AEDZ: CTS_Record;
begin
while(true) {
/*get permission from Circuit, move to next section */
$\boxed{update_{bc}(g_{trc}, g, V_{h_tr}, t_{tr})}$

$(P_c?Sn, Cn)$;

$\boxed{update_{ac}(g_{trc}, g_{recv}, V_{h_{tr}}, t_{tr})}$

/* evaluate process circuit-P_c */

/* SC3: if the number of consecutive sensor failures > *mcsf* and the system is not shut down, then error */

$\boxed{\text{if } \Pi(V_h, SC3(T_{xSn}, T_{x\overline{Sn}}, c)) = \varnothing \text{ then Error}}$

AES := NEW_Entry(Cn,Sn);

[AES.Sn ≠ DZ → AES := EMPTY;

□

AES.Sn = DZ →

/*get permission from Danger Zone, move to next section-danger zone */

$\boxed{update_{bc}(g_{trDZ}, g, V_{h_{tr}}, t_{tr})}$

[$(P_{DZ}?Sn, Cn) \wedge$(AEDZ=AES) →

$\boxed{update_{ac}(g_{trDZ}, g_{recv}, V_{h_{tr}}, t_{tr})}$

/* evaluate process danger zone-P_{DZ} */

$\boxed{\text{if } \Pi(V_h, SC1(T_{xSn}, T_{x\overline{Sn}}) \vee SC2(T_{xSn}, T_{x\overline{Sn}}), y)) = \varnothing \text{ then Error}}$

AEDZ := NEW_Entry(Cn,Sn);

AES := EMPTY;

AEDZ := EMPTY;]

]

}

end while;

END.