Specification of Timed EFSM Fault Models in SDL

S. S. Batth, E. R. Vieira, A. Cavalli, and M. Ü. Uyar

The City College and Graduate Center of the City University of New York,
New York, NY 10016, USA
{`batth,umit`}`@ee-mail.engr.ccny.cuny.edu`
Laboratoire SAMOVAR (CNRS) and GET/INT Evry Cedex, France
{`elisangela.rodrigues,ana.cavalli`}`@int-evry.fr`

Abstract. In this paper, we apply our timing fault modeling strategy to writing formal specifications for communication protocols. Using the formal language of Specification and Description Language (SDL), we specify the `Controller` process of *rail-road crossing system*, a popular benchmark for real-time systems. Our extended finite state machine (EFSM) model has the capability of representing a class of timing faults, which otherwise may not be detected in an IUT. *Hit-or-Jump* algorithm is applied to the SDL specification based on our EFSM model to generate a test sequence that can detect these timing faults. This application of fault modeling into SDL specification ensures the synchronization among the timing constraints of different processes, and enables generation of portable test sequences since they can be easily represented in other formal notations such as TTCN or MSC.

Key words: Extended Finite State Machines, Timing Fault Models, SDL, Hit-or-Jump.

1 Introduction

If the inherent timing constraints are not properly specified in a formal specification of a communication protocol, start and expiration of concurrent timers may lead to infeasible test sequences, which can generate false results by failing correct implementations, or worse, passing the faulty ones.

In this paper, we first introduce an extended finite-state machine (EFSM) model with timer variables based on our earlier work [FUDA03, UWBWF05, UBWF06a] for the `Controller` process of the so-called *rail-road crossing system* [ALUR98]. This system has been studied as a benchmark in many real-time systems [HJL93,HL96,AKLN99,XEN04,CRV05a] . We then augment this timed EFSM model such that the test sequences generated from the augmented model, when applied by a tester to an implementation under test (IUT), will detect the presence of a class of timing faults. In this augmentation, a set of new edges and states are created in the system model (i.e., the edge conditions and actions use timing variables as well as the external inputs) such that the resulting model is a timed EFSM. In addition, a set of special purpose tester timers are implemented inside the testing system (not in the IUT since the implementation is assumed to be a black box). Only a small number of new states and edges are introduced by our augmentation, and hence the overall length of the test sequences generated from the augmented model, compared to the original system model, does not increase significantly.

We focus on the *incorrect timer setting faults* [EDK02, EDKE98, EKD99], which represent the timers that are incorrectly implemented either too short or too long in `Controller`. We then provide a formal specification for this system in Specification and Description Language (SDL) [ITUZ1], which represents the fault detection capabilities of the augmented EFSM model. In this SDL specification of `Controller`, a transition of the EFSM fault model that can be triggered when its time constraint is satisfied is represented by one or more continuous signal operators. We specify these EFSM timing constraints by using two variable types in SDL, namely `time` and `duration`, which are also used to define the test purposes. To achieve the synchronization among the timing constraints of different processes, we introduce a process, called `Clock`, to represent the discrete passage of time and use the variable `now` to verify the global instantiation of the time. The SDL specification can also handle the cases where multiple trains try to cross at the same time.

A test sequence is generated for this SDL specification using the *Hit-or-Jump* [CLRZ99] algorithm. Using the test purposes (also called *stop conditions*), which represent the timing constraints of the EFSM timing fault model, *Hit-or-Jump* algorithm constructs efficient test sequences while avoiding the state explosion. In [CRV05a], *Hit-or-Jump* has been applied to *railroad crossing system* without any fault detection capabilities of our EFSM model. In this paper, we generate the test sequences that are capable of detecting *incorrect timer setting faults.*

Section 2 of this paper presents an English specification of *railroad crossing system.* Section 3 introduces the definitions, graph augmentation algorithms `GA-A`, `GA-B` and `GA-C`, and fault modeling for `Controller`. The SDL specification with timing constraints and test sequence generation using *Hit-or-Jump* algorithm are in Section 4. The concluding remarks are presented in Section 5.

2 English Specification for Railroad Crossing System

The railroad crossing system is one of the popular examples for studying timing constraints in timed FSMs [HJL93, HL94, HL96, AKLN99, ALUR98, XEN04]. It consists of three main processes: `Train`, `Gate` and `Controller`, all of which must communicate with one another within certain time constraints. `Train` process communicates with `Controller` by sending the messages called *approach, in, out* and *exit.* The output signal *approach* must be sent to `Controller` at least two minutes before a train is crossing the railroad. When a train is inside (or outside) the gate, the corresponding output signal *in* (or *out*) is generated. Between the signals *approach* and *exit*, there must be a delay of maximum five minutes. When

`Controller` receives the input signal *approach*, it must send the output signal *lower* to `Gate` at most one minute after the receipt of *approach.* If `Controller` receives *exit*, it must send the output signal *raise* to `Gate` with a maximum delay of one minute.

`Gate` and `Controller` communicate through the signals *lower, raise, up* and *down.* The signals *lower* and *raise* are inputs to `Gate` process. If *lower* is received, `Gate` must respond with *down* output signal, indicating that the gate is closed and the crossing is safe. The interval between the reception of *lower* and the sending of *down* must be at most one minute. If the input signal *raise* is received by `Gate`, it must send the output signal *up* at least one minute and at most two minutes after the receipt of *raise.*

3 Modeling Timed Extended Finite State Machines

A communication protocol modeled as a finite state machine (FSM) can be represented by a directed graph $G(V, E)$. Vertex set V represents the nodes and edge set E represents the edges triggered by events of a system. A protocol specification may include timing variables and operations based their values. To represent these timing related variables, we extend FSMs with timing variables. Our model is complimentary to those presented in timed automata [ALUR98], and has the advantage that it is specifically designed for test generation without state explosion [FUDA03].

3.1 Definitions and Notations

Let $\mathbf{R}$ denote the set of real, $\mathbf{R}^{\circ+}$ the set of the non-negative real, and $\mathbf{R}^{\infty} = \mathbf{R}^{\circ+} \cup \{-\infty, +\infty\}$ is the set of non-negative real with elements $-\infty$ and $+\infty$. Let $\mathbf{Z}$ denote the set of integers and $\mathbf{Z}^{+}$ is the set of positive integers. Interval $[\alpha, \beta]$ is a subset of $\mathbf{R}^{\circ+}$, $[\alpha, \beta] \subset \mathbf{R}^{\circ+}$, and δ is an instant of $[\alpha, \beta]$, $\delta \in [\alpha, \beta]$. α is the lower bound of δ, $Inf(\delta) = \alpha$; β is the upper bound of δ , $Sup(\delta) = \beta$.

***Definition* 1** *A timed FSM is an FSM augmented to form an Extended Finite State Machine (EFSM), represented by directed graph G, denoted by* $M = (V, A, O, \mathcal{T}, E, v_0)$ *where* V *is a finite set of nodes,* $v_0 \in V$ *is the initial node,* A *is a finite set of inputs,* O *is a finite set of outputs,* $\mathcal{T}$ *is a finite set of variables, and* E *is a set of edges* $V \times A \times \mathcal{T} \longrightarrow V \times O \times \mathcal{T}$. *Edge* $e_i \in E$ *can be represented by a tuple* $e_i = (v_p, v_q, a_i, o_i, P_t(\mathcal{T}) = \langle e_i \rangle, Act_t\ (\mathcal{T}) = \{e_i\})$*, where* $v_p \in V$ *is a current node,* $v_q \in V$ *is a next node,* $a_i \in I$ *is the input that triggers the transition represented by* $v_p \overset{e_i}{\rightarrow} v_q$*,* $o_i \in O$ *is the output from current transition* $v_p \overset{e_i}{\rightarrow} v_q$*,* $P_t(\mathcal{T}) = \langle e_i \rangle$ *is the set of possible conditions of timing variables.* $Act_t(\mathcal{T}) = \{e_i\}$ *is the set of possible actions on timing variables.*

***Definition* 2** *A timer* $tm_j \in TM$ *can be defined with timing variables of* $(T_j, D_j, f_j) \subseteq \mathcal{T}$*, where* $TM = \{tm_1, \cdots, tm_j, \cdots\}$ *is a set of* N *timers,* $T_j \in \{0, 1\}$ *is a timer running status variable,* $D_j \in \mathbf{R}^{\circ+}$ *is a time-characteristic variable, and* $f_j \in \mathbf{R}^{\infty}$ *is a time-keeping variable.*

- *Time Keeping Variables* (D_j and f_j), where D_j indicates the length of timer tm_j, and f_j indicates the time elapsed since tm_j started. If tm_j has just started, $f_j := 0$; if tm_j is inactive, $f_j := -\infty$. Over an edge e_i, the value of f_j is increased by the amount of time $c_i \in \mathbf{R}^{\circ+}$ required to completely traverse the current transition e_i, $f_j := f_j + c_i$. The difference of $(D_j - f_j)$ represents the remaining time until tm_j's expiry.
- *Timer Status Variable* (T_j) is a boolean variable, where $T_j == 1$ (T_j) denotes timer tm_j is active and $T_j == 0$ ($\neg T_j$) denotes timer tm_j is passive (i.e., stopped, expired or not started yet).

***Definition* 3** *$TM_{active} \subseteq TM$ and $TM_{passive} \subseteq TM$ are sets of timers which are active and passive, respectively, such that $TM = TM_{active} \bigcup TM_{passive}$.*

- For a transition $e_i = (v_p, v_q, a_i, o_i, \langle e_i \rangle, \{e_i\})$, a set of passive timers $tm_j \in TM_{passive}$, $\forall j \in [1, N]$, can be activated by setting $T_j := 1$ and $f_j := 0$ in its edge actions. For all the other active timers $tm_k \in TM_{active}, \forall k \in [1, N], k \neq j$, f_k is updated by e_i's traversal time. Formally: $\langle e_i \rangle : \langle \neg T_j \wedge T_k \wedge (f_k < D_k) \rangle$ and $\{e_i\} : \{T_j := 1; f_j := 0; T_k := T_k; f_k := f_k + c_i\}$ $\forall k \in [1, N], \forall j \in [1, N], k \neq j$.
- For a transition $e_i = (v_p, v_q, a_i, o_i, \langle e_i \rangle, \{e_i\})$, an active timer $tm_j \in TM_{active}$, $j \in [1, N]$, can be stopped by setting $T_j := 0$ and $f_j := -\infty$ in its edge actions. For all the other active timers $tm_k \in TM_{active}, \forall k \in [1, N], k \neq j$, f_k is updated by e_i's traversal time. Formally: $\langle e_i \rangle : \langle T_j \wedge (f_j < D_j) \wedge T_k \wedge (f_k < D_k) \rangle$ and $\{e_i\} : \{T_j := 0;\ f_j := -\infty;\ T_k := T_k;\ f_k := f_k + c_i\}$ $\forall k \in [1, N], j \in [1, N], \forall k \neq j$.
- An active timer $tm_j \in TM_{active}$ is defined as expired or timed out *iff* f_j is equal or greater than the timer length D_j. Formally: $\langle T_j \wedge (f_j \geq D_j) \rangle$ and $\{T_j := 0; f_j := -\infty\}$.

***Definition* 4** *A transition which becomes feasible when one of the active timers, with the least remaining time, expires is defined as a timeout transition. In other words, $tm_j \in TM_{active}$ ($j \in [1, N]$), $tm_k \in TM_{active}$ ($\forall k \in [1, N], \forall k \neq j$), and tm_j's remaining time was the least, then it was tm_j that expired and triggers the timeout edge e_i. The edge actions set $T_j = 0$, $f_j = -\infty$, and f_k is updated by e_i's traversal time. Formally: $\langle e_i \rangle : \langle T_j \wedge (f_j \geqslant D_j) \wedge T_k \wedge (D_j - f_j < D_k - f_k) \rangle$ and $\{e_i\} : \{T_j := 0; f_j := -\infty; T_k := T_k; f_k := f_k + c_i\}$ $\forall k \in [1, N], \forall k \neq j$.*

***Definition* 5** *A non-timeout transition becomes feasible iff none of the active timers have expired, or all of the timers are passive. In other words, $tm_j \in TM_{active}, \forall j \in [1, N]$, and none of these active tm_j's have expired. f_j is updated by e_i's traversal time. Formally: $\langle e_i \rangle : \langle T_j \wedge (f_j < D_j) \rangle$ and $\{e_i\} : \{T_j := T_j;\ f_j := f_j + c_i)\}$ $\forall j \in [1, N]$.*

***Definition* 6** *Flow Enforcing Variable (L_p) is an exit condition to leave a state v_p. It is denoted by a boolean variable $L_p \in \{0, 1\}$ $\forall v_p \in V$, where $L_p == 0$ means none of the transitions is allowed to leave v_p, and $L_p == 1$ means transitions are allowed to leave v_p.*

***Definition* 7** *A transition whose action updates L_p from 0 to 1 is defined as an observer edge. Formally:* $\langle e_{p,obs} \rangle : \langle L_p == 0 \rangle$ *and* $\{e_{p,obs}\} : \{L_p := 1\}\ \forall v_p \in V$.

***Definition* 8** *For an active timer, a transition which consumes the pending timeout is defined as a wait edge. In other words,* $tm_j \in TM_{active}$ $(j \in [1, N])$, $tm_k \in TM_{active}$ $(\forall k \in [1, N], \forall k \neq j)$ *and* tm_j*'s remaining time is the least, then the wait edge updates* f_j *by* tm_j*'s remaining time* $D_j - f_j$. *Formally:* $\langle e_{p,wait} \rangle : \langle T_j \wedge (f_j < D_j) \wedge T_k \wedge (f_k < D_k) \wedge (D_j - f_j < D_k - f_k) \rangle$ *and* $\{e_{p,wait}\} : \{f_j := f_j + (D_j - f_j); f_k := f_k + (D_j - f_j)\}\ \forall k \neq j, k \in [1, N], \forall v_p \in V$.

***Definition* 9** *A return edge is an edge with no time constraints and actions:* $\left\langle e_p^{ret} \right\rangle : \langle 1 \rangle$ *and* $\left\{ e_p^{ret} \right\} : \{\ \}\ \forall v_p \in V$.

***Definition* 10** *During testing an edge* $e_i = (v_p, v_q, a_i, o_i, \langle \mathbf{e}_i \rangle, \{\mathbf{e}_i\})$, *after input* a_i *is applied to an IUT, the expected output* o_i *should be generated no later than a certain* θ *time units,* $\theta \in \mathbf{R}^{\circ+}$, *measured by a timer which is a part of the test harness rather than the IUT.*

3.2 Graph Augmentation Algorithm `GA-A`

To model the original system along with its timed behavior, we introduce a graph augmentation algorithm, called `GA-A` [UBWF06a], which is specifically designed for generating tests for the systems whose timer related variables are linear and their values implicitly increase with time. To ensure that the timing conditions and actions of the specification are correctly incorporated into the timed EFSM model, `GA-A` generates $G'(V', E')$ by converting self-loops in G to node-to-node edges, defining an exit condition for all the nodes, and creating a set of new nodes and edges:

Step (i): If there exists a self loop for $v_p \in V$ in G, an additional node called v'_p is created in G', to which all self-loops $e_{p,k} \in E$ defined in v_p are directed;

Step (ii): All self-loops $e_{p,k} \in E$ in G are converted to node-to-node edges in G' as $e_{p,k} = (v_p, v'_p)$.

Step (iii): For $v'_p \in V'$ in G', a return edge e_p^{ret} from v'_p to v_p is created in G' as $e_p^{ret} = (v'_p, v_p)$.

Step (iv): An *observer node* is created in G', namely $v_{p,wait}$, which is connected to v_p via newly created an observer edge as $e_{p,obs} = (v_p, v_{p,wait})$, a wait edge as $e_{p,wait} = (v_p, v_{p,wait})$, and a return edge from observer node as $e_{p,obs}^{ret} = (v_{p,wait}, v_p)$. The role of the observer node $v_{p,wait}$ is to *consume* pending timeouts on $e_{p,wait}$ and enable outgoing edges by setting the flow enforcing variable L_p to 1 on $e_{p,obs}$. Fig. 1 shows, for node v_p, the conversion of self-loops to node-to-node edges, the creation of the observer node, wait and observer edges.

The time condition and the action for the wait edge $e_{p,wait}$ are formulated as $\langle L_p == 0 \rangle$ and $\{f_j := f_j + (D_j - f_j)\}$, where $D_j - f_j$ is the remaining time of timer $tm_j \in TM_{active}$ to timeout. For the observer edge $e_{p,obs}$ from the original node v_p to the observer node $v_{p,wait}$ in G', the time condition and the action are formulated as $\langle L_p == 0 \rangle$ and $\{L_p := 1\}$, respectively. The return edges of e_p^{ret}

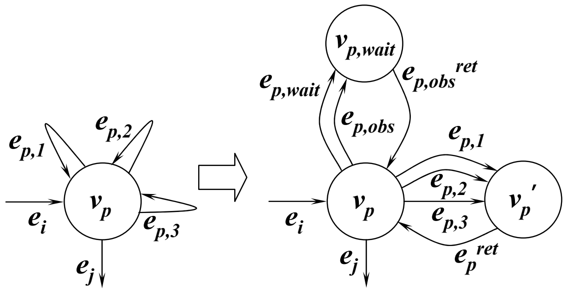

Fig. 1. Modeling self-loops for v_p in G into v_p, v'_p and $v_{p,wait}$ in G'.

and $e_{p,obs}^{ret}$ are added by `GA-A` to G' are no-cost edges with time condition as: $\langle 1 \rangle$ (i.e., always true with no time constraints imposed) with no actions: { }.

Step (v): The conditions and actions for a *timeout* edge in G' are:

- The condition for a timeout self-loop edge in G becomes: $\langle T_j \wedge (f_j \geqslant D_j) \wedge T_k \wedge (f_k < D_k) \wedge (D_j - f_j < D_k - f_k) \wedge (L_p == 0)\rangle\ \forall T_k \neq T_j$, where the remaining time for $tm_j \in TM_{active}$ is less than that of $tm_k \in TM_{active}$ (i.e., $D_j - f_j < D_k - f_k$) and the flow enforcing variable L_p is zero.
- The condition for a timeout node-to-node edge in G becomes: $\langle T_j \wedge (f_j \geqslant D_j) \wedge T_k \wedge (f_k < D_k) \wedge (D_j - f_j < D_k - f_k) \wedge (L_p == 1)\rangle\ \forall T_k \neq T_j$, where the remaining time for $tm_j \in TM_{active}$ is less than that of $tm_k \in TM_{active}$ (i.e., $D_j - f_j < D_k - f_k$) and L_p is 1.
- The actions for a timeout edge in G become: $\{T_j := 0; f_j := -\infty; T_k := T_k; f_k := f_k + c_i; L_p := 0\}\ \forall\, tm_k \neq tm_j$, where timer $tm_j \in TM_{passive}$ becomes passive and the time keeping variable for $tm_k \in TM_{active}$ is incremented by the edge cost of c_i.

These equations imply that before a timeout edge, tm_j should be still running, remaining time should be the least among all other running timers and the flow-enforcing variable is appropriately set for either a converted (i.e., self-loop edge in G) or an original (i.e., node to node edge in G) edge in G'.

Step (vi): The conditions and actions for a non-timeout edge in G' is formalized as follows:

- A non-timeout self-loop edge in G becomes: $\langle(\neg T_j \vee (T_j \wedge (f_j < D_j))) \wedge (L_p == 0)\rangle\ \forall\ tm_j \in TM_{active}$
- A non-timeout node-to-node edge in G becomes: $\langle(\neg T_j \vee (T_j \wedge (f_j < D_j))) \wedge (L_p == 1)\rangle\ \forall\ tm_j \in TM_{active}$
- The action for a non-timeout edge in G becomes:
  - $\{f_j := f_j + c_i; f_k := f_k + c_i; L_p := 0\}\ \forall\ tm_k \neq tm_j, tm_j \in TM_{active}, tm_k \in TM_{active}$ if edge starts no timers;
  - $\{T_j := 1; f_j := 0; T_k := T_k; f_k := f_k + c_i; L_p := 0\}\ \forall\ tm_k \neq tm_j$ if edge starts timer tm_j.

Since both timeout and non-timeout edges disable outgoing edges by setting $L_p := 0$ in Steps *(v)* and *(vi)* of `GA-A`, the only edges whose actions will enable the outgoing edges in G' are the artificially-created observer edges.

It is proven [UBWF06a] that `GA-A` terminates with a running time of $\mathbb{O}(E)$, and that the total number of the nodes and edges in $G'(V', E')$ and $G(V, E)$ have the same order of magnitude.

3.3 Classification of Timing Faults

A class of timing faults in an implementation of a timed system have been defined in [EDK02, EDKE98, EKD99] as 1-*clock timing faults* (including 1-*clock corner point* and 1-*clock interval faults*) and *incorrect timer length setting faults.*

Incorrect Timer Setting Faults occur in an IUT when a timer length is incorrectly implemented as either too short or too long (i.e., the timer expires either too early or too late). The definition of incorrect timer setting faults is based on the following timing requirement:

- ***Timing Requirement*:** In a test sequence, edge h_k starts timer tm_j and is traversed before e_i. Timeout transition $e_i = (v_p,\ v_q, \mathit{timeout\_tm}_j, o_i,\ \langle \mathbf{t}_j \rangle\ ,\ \{\mathbf{t}_j\})$ triggers exactly in D_j time units, where D_j is the timer length.
- ***Timing Fault B* (TF_B):** Timeout transition e_i triggers in D'_j time units and output o_i is observed and node v_q is verified in shorter than the expected time (i.e., $D'_j < D_j$).
- ***Timing Fault C* (TF_C):** Timeout transition e_i triggers in D'_j time units and output o_i is observed and node v_q is verified in longer than the expected time (i.e., $D'_j > D_j$).

In a specification, suppose a timer tm_j is defined to be of length D_j to be started by the actions of edge h_k and to expire at edge e_i (reachable from h_k). A special purpose timer tm_s with length $D_s = D_j$ is created in the test harness by `GA-2.B` to detect if tm_j is set too short as $D'_j < D_j$:

***Step (B.i)*:** Edge conditions and actions for h_k are modified such that it starts a *special purpose timer* tm_s.

***Step (B.ii)*:** e_i's condition is modified such that it traverses only when both tm_s and tm_j have expired.

***Step (B.iii)*:** All self-loops in v_p are represented as node-to-node edges by the creation of an additional node, called v'_p, to which they are directed. A return edge e_p^{ret} (with zero cost) is also created for their return to v_p.

***Step (B.iv)*:** An observer node $v_{p,wait}$ is appended to node v_p via a new observer edge $e_{p,obs}$, wait edge $e_{p,wait}$ (with cost $c_{p,wait}$) and return edge e_p^{ret} (with cost $c_p^{ret} := 0$). The edge condition of e_i is modified such that it triggers only when $f_s \geqslant D_s$ and tm_j expires.

As proven in [UBWF06a], `GA-2.B` terminates with a running time of $\mathbb{O}(E)$, and the order of magnitude of the nodes and edges in G' and G'' are the same. A test sequence generated from G'' will contain $\cdots$, h_k, $\cdots$, e_{i-1}, $e_{p,wait}$, e_p^{ret}, $e_{p,obs}$, e_p^{ret}, e_i which will not be feasible to traverse if timer tm_j expires earlier than expected. The condition for $e_{p,wait}$ requires that both the timers tm_j from the IUT and tm_s from the test harness are still running. If tm_j times out before

tm_s, it will create a deadlock at v_p (i.e., none of the conditions leaving v_p is valid), which in turn will flag the tester that a timing fault TF_B has occurred.

Algorithm `GA-2.C` [UBWF06a] for TF_C, is similar to `GA-2.B`, with the same run time complexity and the augmented graph size of G'.

3.4 Timed EFSM Model for Railroad Crossing System

Due to space constraints, we only consider timing fault TF_B in the edges of e_2 and e_4 in `Controller`, whose FSM model is given in Fig. 2. The steps for generating graph G'' is follows:

Step 1: Obtain graph G from the specification of `Controller` process. The directed graph representing `Controller` is in Fig. 2 with its actions and conditions given in Table 1. Timer tm_z can be started either in edge e_1 or in e_3 with the timer length of 1 min (i.e., $D_z = 1$ min).

Step 2: Generate G' for `Controller` by applying the graph augmentation algorithm `GA-A` to G. The new observer nodes and edges (i.e., $s_{0,wait}$, $e_{0,wait}$, $e_{0,obs}$, $e^{ret}_{0,obs}$, $s_{1,wait}$, $e_{1,wait}$, $e_{1,obs}$, $e^{ret}_{1,obs}$, $s_{2,wait}$, $e_{2,wait}$, $e_{2,obs}$, $e^{ret}_{2,obs}$, $s_{3,wait}$, $e_{3,wait}$, $e_{3,obs}$, $e^{ret}_{3,obs}$) are added to the original nodes of G. The self-loop edge of e_0 is converted to a node-to-node edge by introducing s'_0 and e^{ret}_0 in G'.

Step 3: Apply the graph augmentation algorithm `GA-B` to G' to generate G'' for `Controller`. A special purpose timer, namely tm_s (with $D_s = 1$), is introduced in the tester (not in the IUT) to model the timing constraints over the edges of e_2 and e_4. Note that, in G'', e_1 starts both the special purpose timer tm_s in the tester and the timer tm_z in the IUT; similarly, e_3 starts the same two timers in the tester and the IUT. Graph G'' is shown in Fig. 3 with its respective edge conditions and actions given in Table 2.

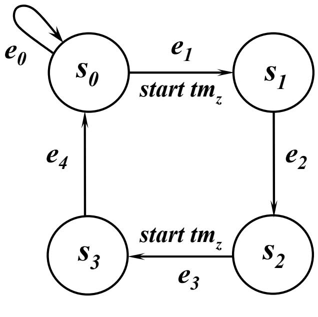

Fig. 2. Finite state machine for `Controller`.

Table 1. Original specification of `Controller` (Fig. 2) and its graph G

Edges	English Specification	Our EFSM Model G	
		Timing Conditions	Timing Actions
e_0	Idle	$\langle 1 \rangle$	$\{\ \}$
e_1	Input *approach* is received	$\langle (a_1 == approach) \rangle$	$\{T_z := 1; f_z := 0\}$
e_2	Output *lower* is generated at maximum delay of 1 mins after input *approach* is received	$\langle T_z \wedge (f_z \geq D_z) \rangle$	$\{o_2 := lower;$ $T_z := 0; f_z := -\infty\}$
e_3	Input *exit* is received	$\langle (a_3 == exit) \rangle$	$\{T_z := 1; f_z := 0\}$
e_4	Output *raise* is generated maximum delay of 1 mins after input *exit* is received	$\langle T_z \wedge (f_z \geq D_z) \rangle$	$\{o_4 := raise;$ $T_z := 0; f_z := -\infty\}$

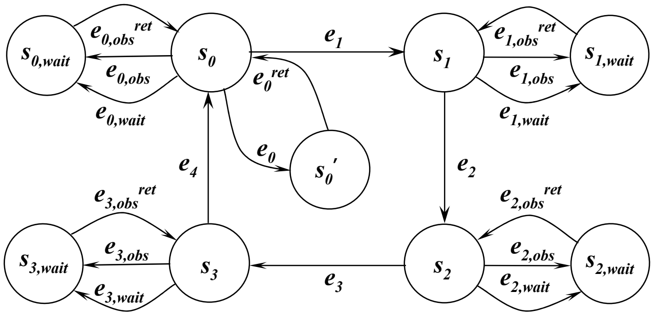

Fig. 3. Augmented Graph G'' for `Controller` (Fig. 2) after applying `GA-A` and `GA-B`

4 SDL Specification Based on Timed EFSM Model

To specify a set of timed EFSM models in SDL one may either *(i)* define each component (e.g., `Train, Gate` and `Controller`) as an independent system, where each exchange messages with the environment, or *(ii)* define each component as a process of the same system. Although both approaches are equivalent, in this paper we follow the latter approach. Our SDL specification is designed for testing purposes, where the evolution of time is modeled by the expiration of the clocks. We introduce a process, called `Clock`, as a part of the `Railroad` system to represent the passage of time. Therefore, our SDL specification for the *railroad crossing system* consists of a main `Railroad` system, which includes a `Railroad_Control` block (Fig. 4) with four processes, namely `Train`, `Gate`, `Controller` and `Clock`.

In our EFSM model, each edge e_i is associated with a timing cost c_i, representing the expected time that is required to traverse (or, realize) the edge in an implementation (see Section 3). The corresponding state transition in SDL

Table 2. Augmented edge conditions and actions of graph G'' (Fig. 3) of `Controller`.

Edges	$\langle$ **Edge Conditions** $\rangle$	**{ Edge Actions }**
e_0	$\langle \neg approach \rangle$	$\{\ \}$
e_0^{ret}	$\langle 1 \rangle$	$\{\ \}$
$e_{0,obs}$	$\langle L_p == 0 \rangle$	$\{L_p := 1\}$
$e_{0,wait}$	$\langle \neg approach \wedge L_p == 0 \rangle$	$\{f_i := f_i + c_{0,wait}\}$
$e_{0,obs}^{ret}$	$\langle 1 \rangle$	$\{\ \}$
e_1	$\langle approach \wedge L_p == 1 \rangle$	$\{T_s := 1; f_s := 0; L_p := 0\}$
$e_{1,wait}$	$\langle L_p == 0 \rangle$	$\{f_i := f_i + c_{1,wait}\}$
$e_{1,obs}$	$\langle L_p == 0 \rangle$	$\{L_p := 1\}$
$e_{1,obs}^{ret}$	$\langle 1 \rangle$	$\{\ \}$
e_2	$\langle T_s \wedge (f_s \geq D_s) \wedge (T_z \text{timeout}) \wedge L_p == 1 \rangle$	$\{lower; T_s := 0; f_s := -\infty; L_p := 0\}$
$e_{2,wait}$	$\langle \neg exit \wedge L_p == 0 \rangle$	$\{f_i := f_i + c_{2,wait}\}$
$e_{2,obs}$	$\langle L_p == 0 \rangle$	$\{L_p := 1\}$
$e_{2,obs}^{ret}$	$\langle 1 \rangle$	$\{\ \}$
e_3	$\langle exit \wedge L_p == 1 \rangle$	$T_s := 1; f_s := 0; L_p := 0\}$
$e_{3,wait}$	$\langle L_p == 0 \rangle$	$\{f_i := f_i + c_{3,wait}\}$
$e_{3,obs}$	$\langle L_p == 0 \rangle$	$\{L_p := 1\}$
$e_{3,obs}^{ret}$	$\langle 1 \rangle$	$\{\ \}$
e_4	$\langle T_s \wedge (f_s \geq D_s) \wedge (T_z \text{timeout}) \wedge L_p == 1 \rangle$	$\{raise; T_s := 0; f_s := -\infty; L_p := 0\}$

specification can be represented as the difference between two internal variables that are set at the instances of the beginning and end of the transition. This way, these two variables, one with the clock value at the beginning and the other one at the end, can be used to approximate the edge traversal time in SDL, Similarly, the following assumptions are considered to specify a real-time system in SDL [AKLN99, TMCB03]:

- All un-timed events will take a negligible time to realize;
- Time advances through the expiration of local clocks; if two clocks expire at the same moment, only one of them is taken into account first;
- As time progresses, time dependent transitions may trigger only if their conditions are satisfied;
- The global clock called `now` is the only clock which gives the current time.

In this approach, time constraints are represented as continuous signal operators. This construct allows to represent a transition that does not need an input signal to be fired, but is triggered when the time constraint is satisfied. In our SDL specification, two variable types are used: a `time` variable to register the moment when an event occurs, and a `duration` variable to represent the difference between two `time` variables. For example, in the timing condition of $(f_1 - f_2 > D_2)$, variables f_1 and f_2 are of type `time`, whereas D_2 is a `duration` variable. Both `time` and `duration` variables are also defined in our EFSM model in Section 3. For example, for the special purpose timer tm_s in G'' (Section 3.3),

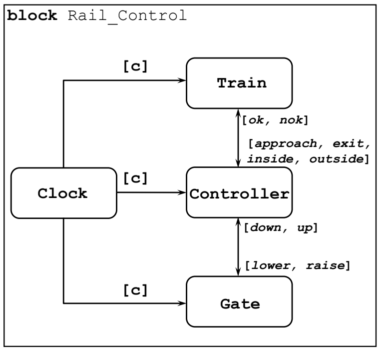

Fig. 4. Rail_Control block of SDL specification.

time keeping variable f_s and the timer length D_s are represented as the `time` and `duration` types of variables in our SDL specification, respectively.

Although we did not utilize the local timer construct in SDL to represent the timer tm_s, we have instead used the variable `now` and the process `Clock` to model the evolution of time in our SDL specification. Therefore, for `Controller`, f_s is represented by four `time` type variables, namely *zapproach*, *zexit*, *zlower* and *zraise*. The moment when *approach* and *exit* signals are received is represented by *zapproach* and *zexit*, respectively. Similarly, *zlower* and *zraise* are used to capture the moment when *lower* and *raise* are sent, respectively. Timer length D_s is modeled by two `duration` type variables, namely *sent_lower_delay* and *sent_raise_delay*, both equal to 1 min. Table 3 illustrates the relationship between our SDL specification and the EFSM model based on G''.

Our SDL specification also allows representation of more than one train trying to cross at the same time. To model multiple trains, additional variables such as (*ntrains* and *max_trains*), and signals (*ok* and *nok*) are introduced (in the SDL specification given in this paper, $max\_trains = 1$). Since there are a limited number of tracks available, variable *ntrains* counts the number of trains which have sent *approach* to `Controller`. Each *approach* received from a different train can be distinguished by `Controller` because an internal identifier with a distinct channel is created for each instantiation of the `Train` process. Therefore, if the condition of ($ntrains <= max\_trains$) is true, `Controller` sends *ok*; otherwise it sends *nok*. If `Train` receives *ok* from `Controller`, the train continues its approach to the railroad crossing. Similarly, if *nok* is received by `Controller`, the train waits until it receives a signal of *ok*. When one of the `Train` processes sends *exit*, `Controller` decrements the value of *ntrains* by one. If the updated value of *ntrains* is still greater than zero, `Controller` sends another *ok* to one of the `Train` processes waiting to approach the railroad crossing; otherwise, `Controller` sends *raise* signal to `Gate`.

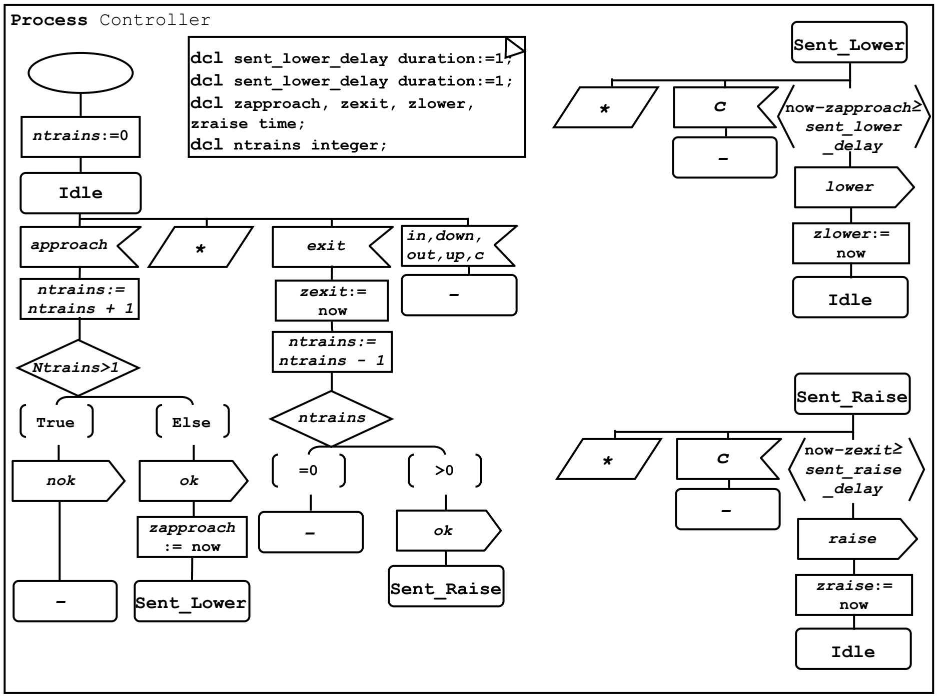

Fig. 5. SDL specification of `Controller`

4.1 Application of *Hit-or-Jump* Algorithm

Hit-or-Jump [CLRZ99] algorithm can be used for embedded testing of complex communication systems which are modeled as communicating EFSMs. It is a generalization and unification of exhaustive search and random walks; both of which are special cases of *Hit-or-Jump*. It efficiently constructs testing sequences with a high fault coverage, does not suffer from the drawback of state space explosion as encountered in exhaustive search, and quickly covers the system components under test without being *trapped*, as experienced by random walks. Furthermore, it has also been applied to embedded testing of telephone services [CLRZ99], conformance and interoperability testing of web services [CMZ04] and in the domain of real-time systems [CRV05a, CRV05b]. The strategy used to generate a partial accessibility graph in *Hit-or-Jump* is that if a visited node satisfies the test purposes, it is said that a *hit* is done; otherwise, the algorithm randomly choses another node from the neighborhood graph, and moves (*jump*) to it. Then from this new node, it continues its search. Parameters required to execute the *Hit-or-Jump* are:

(i) **SDL specification** of the IUT (Fig. 5);

(ii) **Test purposes** described in several *stop conditions*, which are the properties to be verified at each node. Each property can be defined in input signals, output signals, `time`, and `duration` variable types. In our case study, the test

Table 3. Relationship between SDL Specification (Fig 5) and EFSM Model (Fig 3) for `Controller`

Current State		**Next State**		**Edge Name**	**Constraint**	**Action**
SDL Spec.	**EFSM Model**	**SDL Spec.**	**EFSM Model**			
Start		Idle	s_0			ntrains:= 0
Idle	s_0	Idle	s_0		$(approach?)$ and $(ntrains > 1)$	$(nok!)$ and (zapproach:= `now`)
Idle	s_0	Sent_Lower	s_1	e_1	$(approach?)$ and $(ntrains \leqslant 1)$	$(ok!)$ and (zapproach:= `now`)
Sent_Lower	s_1	Sent_Lower	$s_{1,wait}$	$e_{1,wait}$	$(*?)$ or ($\texttt{now} - zapproach \leqslant$ sent_lower_delay)	
Sent_Lower	s_1	Idle	s_2	e_2	($\texttt{now} - zapproach \geqslant$ sent_lower_delay)	$(lower!)$ and (zlower := `now`)
Idle	s_2	Idle	s_2		$(exit?)$ and $(ntrains \geqslant 0)$	$(ok!)$ and (zexit := `now`)
Idle	s_2	Sent_Raise	s_3	e_3	$(exit?)$ and $(ntrains \leqslant 0)$	(zexit := `now`)
Sent_Raise	s_3	Sent_Raise	s_3	$e_{3,wait}$	$(*?)$ or ($\texttt{now} - zexit \leqslant$ sent_raise_delay)	
Sent_Raise	s_3	Idle	s_0	e_4	$(\texttt{now} - zexit \geq 1)$	$(raise!)$ and (zraise := `now`)
Legend: Input = ?, Output = !, `now` = Global Clock, $*$ = Any other signal; `Time` type variables = zapproach, zexit, zraise, zlower; `Duration` type variables = sent_lower_delay, sent_raise_delay						

purposes are defined according to the timing fault models of G'' graph. These are then modeled for SDL specification and used as stop conditions. Table 4 gives the details of test purposes for all the processes of the *railroad crossing system*;

(iii) A **preamble scenario** (optional) may be furnished in order to guide the algorithm to easily and quickly find a sequence which satisfies the stop conditions (test purposes). If no preamble scenario is given, the search starts from the initial state of all processes;

(iv) The **strategy of the search**, which can either be a breadth or a depth search, in order to generate an internal accessibility graph;

(v) A **local search parameter** (an integer), which defines the space required for the search before a *jump*.

The test sequence generated from SDL specification of `Controller` by applying *Hit-or-Jump* is given in Table 5. Note that all un-timed transitions have zero cost because of the assumption in SDL that these transitions take insignificant time to run. The cost of the wait edges is expressed in minutes.

Using our SDL specification, *Hit-or-Jump* generates test sequences with timing fault detection capabilities. Although, in our case study only timing fault

Table 4. Test purposes for SDL specification and EFSM model

Process Name	**Test Purposes for EFSM Model**	**Test Purposes for SDL Specification**
`Train`	Output *in* is generated in less than 2 minutes after *approach*	$xinside - xapproach < 2$
	Output *exit* is generated in more than 5 minutes after *approach*	$xexit - xapproach > 5$
`Controller`	Output *lower* is generated in less than 1 minutes after *approach*	$zlower - zapproach < 1$
	Output *raise* is generated in more than 1 minutes after *exit*	$zraise - zexit > 1$
`Gate`	Output *down* is generated in more than 1 minutes after *lower*	$ydown - ylower > 1$
	Output *up* is generated in more than 2 minutes after *raise*	$yup - yraise > 2$

TF_B is considered for `Controller`, other types of timing faults can also be modeled for `Controller`, `Train` and `Gate` processes [FUDA03,UWBWF05,UBWF06a]. *Hit-or-Jump* can then be used to generate a test sequence which takes into account all of the timing fault models for three processes. Therefore, the test sequences can be used both for unit testing of each process, and for verifying the communication among processes during the integration phase. Another advantage is the flexibility of representing the test sequences in Tree and Tabular Combined Notation (TTCN) [ETSI] or Message Sequence Chart (MSC) [ITUZ2] notation, facilitating the portability of the tests.

Table 5. Test sequence generated from SDL specification of `Controller`

Step No.	**Current State**	**Next State**	**Cost (Mins.)**	**Inputs**	**Outputs**
1	Idle	Sent_Lower	0	*approach*	
2	Sent_Lower	Sent_Lower	2		
3	Sent_Lower	Sent_Lower	0		
4	Sent_Lower	Idle	0		*lower*
5	Idle	Sent_Raise	0	*exit*	
6	Sent_Raise	Sent_Raise	2		
7	Sent_Raise	Sent_Raise	0		
8	Sent_Raise	Idle	0		*raise*

5 Conclusions and Future Work

In this paper, we apply our timing fault modeling strategy to writing formal specifications for communication protocols. As part of this approach, using the formal language of SDL, we specify the `Controller` process of *rail-road crossing* *system*, a popular benchmark for real-time systems. The EFSM model has the capability of representing a class of timing faults, which otherwise may not be detected in an IUT. We then apply *Hit-or-Jump* algorithm to the SDL specification based on our EFSM model to generate a test sequence that can detect these timing faults. In addition, including fault modeling into SDL specification ensures the synchronization among the timing constraints of different processes, and enables generation of portable test sequences since they can be easily represented in other formal languages such as TTCN or MSC.

As an extension of this work, we will consider the EFSM models with fault detection capabilities for other classes of timing faults, and multiple occurrences of these faults. This approach of modeling the timing faults of communicating processes into formal specifications will also applied to generate integration tests.

References

[AKLN99] M. Ashour, F. Khendek, and T. Le-Ngoc. Formal description of real-time systems using SDL. In *Proc. of the Sixth Int'l. Conf. on Real-Time Comp. Sys. and Appl. (RTCSA'99)*, Hong-Kong, December 1999.

[ALUR98] R. Alur, D. Dill A theory of timed automata. In *Theoretical Comput. Sci.*, vol. 126, pp. 183-235, 1994.

[CLRZ99] A. Cavalli and D. Lee and C. Rinderknecht and F. Zaidi. Hit-or-jump an algorithm for embedded testing with applications to in services. Proc. of IFIP Int'l. Conf. FORTE/PSTV'99, October 1999.

[CMZ04] A. Cavalli and A. Mederreg and F. Zaidi Application of a Formal Testing Methodology to Wireless Telephony Networks. Journal of the Brazilian Comp. Soc., Number 2, Vol. 10, pp 56-68, November 2004.

[CRV05a] A. Cavalli and E. Rodrigues Vieira. Test Case Generation based on Timed Constraints. In *IEEE ICESS 2005*, Xian, China, December 2005.

[CRV05b] A. Cavalli and E. Rodrigues Vieira. A Formal Approach of Interoperability Test Cases Generation Applied to Real Time Domain. In *IEEE I2TS 05*, Florianpolis, SC, Brazil, December 2005.

[DU04] A.Y. Duale and M.U. Uyar. A method enabling feasible conformance test sequence generation for EFSM models. *IEEE Trans. Commun.* **53**(5), pp. 614–627, 2004.

[EDK02] A. En-Nouaary, R. Dssouli, and F. Khendek. Timed Wp-method: Testing real-time systems. *IEEE Trans. Softw. Eng.* **28**(11), pp. 1023–1038, 2002.

[EDKE98] A. En-Nouaary, R. Dssouli, F. Khendek, and A. Elqortobi. Timed test cases generation based on state characterization technique. In *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, pp. 220–229, Madrid, Spain, 1998.

[EKD99] A. En-Nouaary, F. Khendek, and R. Dssouli. Fault coverage in testing real-time systems. In *Proc. IEEE Int'l Conf. Real-Time Comput. Syst. Appl. (RTCSA)*, Hong Kong, China, 1999.

[ETSI] ETSI. Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language.

[FAUD00] M.A. Fecko, P.D. Amer, M.U. Uyar, and A.Y. Duale. Test generation in the presence of conflicting timers. In *Proc. IFIP Int'l Conf. Test. Commun. Syst. (TestCom)*, pp. 301–320, Ottawa, Canada, 2000.

[FUDA03] M.A. Fecko, M.U. Uyar, A.Y. Duale, and P.D. Amer. A technique to generate feasible tests for communications systems with multiple timers. *IEEE/ACM Trans. Netw.* **11**(5), pp. 796–809, 2003.

[HJL93] C. L. Heitmeyer and R. D. Jeffords and B. G. Labaw. A Benchmark for Comparing Different Approaches for Specifying and Verifying Real-Time Systems. In *Proc. Tenth Int'l. Workshop on Real-Time Operating Sys. and Software*, May 1993.

[HL94] C. Heitmeyer and N. Lynch. The Generalized Railroad Crossing: A Case Study in Formal Verification of Real-Time System. In Proc. of the 15th IEEE Real-Time Sys. Symp., Puerto Rico, December 1994.

[HL96] C. Heitmeyer and N. Lynch. Formal Verification of Real-time Systems Using Timed Automata. *Trends in Formal Methods for Real-Time Computing.* John Wiley and Sons, Ltd., pages 83-106, April 1996.

[ITUZ1] ITU-T. Rec. Z.100 Specification and Description Language, 1980.

[ITUZ2] ITU-T. Rec. Z. 120 Message Sequence Charts, Geneva, 1996.

[LRS98] R. Lanphier, A. Rao and H. Schulzrinne. Real time streaming protocol (RTSP). RFC 2326, IETF, 1998.

[SCFJ96] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications. RFC 1889, IETF, 1996.

[TMCB03] C. Teyssie, Z. Mmammeri, F. Carcenac, and F. Buniol. Etude Comparative de SDL et UML pour la Modelisation de Systemes Temps Reel. In *11th Conf. on Real-Time and Embedded Systems*, pages 75–97, Paris, April 2003. Teknea.

[UWBWF05] M. U. Uyar, Y. Wang, S. S. Batth, A. Wise, M. A. Fecko Timing Fault Models for Systems with Multiple Timers, IFIP Int'l. Conf. on Testing of Comm. Systems (TESTCOM), Concordia, Canada, 2005.

[UBWF06a] M. U. Uyar, S. S. Batth, Y. Wang, and M. A. Fecko. EFSM graph augmentation algorithms for modeling a class of single timing faults. *IEEE Trans. Comput.* , 2006. (in review).

[UFDA01] M.U. Uyar, M.A. Fecko, A.Y. Duale, P.D. Amer, and A.S. Sethi. A formal approach to development of network protocols: Theory and application to a wireless standard. In *Proc. Concordia Prestigious Wksp Commun. Softw. Eng. (CP-WCSE)*, Montreal, Canada, 2001. **(invited paper)**.

[UZ93] H. Ural and K. Zhu. Optimal length test sequence generation using distinguishing sequences. *IEEE/ACM Trans. Netw.* **1**(3), pp. 358–371, 1993.

[XEN04] Z. Xiang and A. En-Nouaary. Test cases generation for embedded real-time systems based on test purposes. In *NOTERE'2004*, Saidia, Maroc, Juin 2004.