Towards Testing SDL Specifications: Models and Fault Coverage for Concurrent Timers[⋆]

Mariusz A. Fecko[1], M. Ümit Uyar[2], and Ali Y. Duale[3]

[1] Applied Research Area, Telcordia Technologies, Inc., Piscataway, NJ, USA, mfecko@research.telcordia.com
[2] Electrical Engineering Dept., CCNY, The City University of New York, USA, uyar@ccny.cuny.edu
[3] System Architecture Compliance, IBM Corp., Poughkeepsie, NY, USA, duale@us.ibm.com

Abstract. A recent model for testing systems with multiple timers is extended to compute proper input delays and timeout settings, and is applied to several types of timers required in a testing procedure. In the model, any transition in the specification can be made conditional on a set of running timers. Depending on the path taken to reach an edge, the values of the timer variables may render the traversal of the edge infeasible. The presented modeling technique, combined with the INconsistencies DEtection and ELimination (INDEEL) algorithms, allows the generation of feasible test sequences. The model also offers the flexibility to define timer lengths as variables, and have the INDEEL find the appropriate timer ranges. An approach to apply this new methodology to SDL timed extensions (guarding and delaying timers) is presented.

Keywords: conformance testing; timed EFSM; timed extensions; SDL

1 Introduction

Proper handling of timers is of special concern for computer-aided test generation from formal specifications [4, 8, 12, 13, 19, 22]. Consider sending a message that requires an acknowledgment, which implies that two timers may be started with different expiry times. If no acknowledgment is received before the first timer expires, the message is retransmitted and the timer restarted. When the second timer expires, the transmission attempts are abandoned. A feasible test sequence cannot proceed directly to the expiry of the second timer, bypassing that of the first timer. Such dependencies must be incorporated in the flow graph of the specification and resolved during test sequence generation [22].

⋆ M.A. Fecko initiated the timed EFSM model [9] at U Delaware. A.Y. Duale co-designed the INDEEL [6] at CUNY. They were then funded by the ATIRP Consortium sponsored by U.S. Army Research Lab (ARL) under the FedLab Program, Cooperative Agreement DAAL01-96-2-0002.

To address the above challenge, a new methodology has been introduced for test generation for timed systems modeled as Extended Finite State Machines (EFSMs) [14] with time-related variables [9, 10]. The methodology includes a novel model that uses simple linear expressions of time-related variables to represent complex timing dependencies. To ensure feasibility of test sequences, Duale and Uyar [6] designed INconsistencies DEtection and ELimination (INDEEL) algorithms to resolve inconsistencies among the condition or action variables of an EFSM. After these algorithms are applied to eliminate any time-related variable conflicts, all paths of the resulting EFSM are feasible.

This paper generalizes and extends the timing model introduced in Ref. [9], and shows a range of its capabilities:

- *Testing various classes of timers:* We illustrate a range of modeling that can be accomplished with our technique, including general models of various types of timers required during the testing procedure (i.e., global, guarding, delaying, and functional timers) (Section 2).
- *Fault models:* As a formal analysis of timing faults, we show that a test sequence, if it covers every feasible edge at least once [10], can detect 1- and n-clock timing faults [8] and incorrect timer settings (Section 3).
- *Application to SDL:* We illustrate how the above classes of timers can be defined and tested in SDL [4, 22]. Once the FSM is extracted [3], the time extensions for SDL [12] have a natural representation in our model (Section 4).
- *Delaying transitions:* We present a formal approach to identify transitions that have to be delayed. A step-by-step test-sequence derivation shows how delays are algorithmically computed by the augmented INDEEL (Section 4.3).
- *Flexible timeout and transition-execution settings:* We show how a tester can define a timer length as a constant or variable to automatically find proper timeout settings. This way, a test suite can cover service delivery and controllability features with multi-timer dependencies (Section 4.4).

2 Timing Model

The timed EFSM model for testing protocols with timers was introduced and formally described in Refs. [9, 10]. The model uses exclusively the paradigm of EFSM, which makes it easily applicable to the languages such as SDL [4, 22], VHDL [6], and Estelle [5], and thus enables testing timed systems with the numerous (E)FSM-based test generation methodologies [6, 14, 22].

To model concurrent timers, the original system along with its time-related behavior can be represented by an EFSM consisting of an FSM (the untimed model of the system) and a set of time-related variables. These variables represent features such as (1) a timer being *on* or *off*, (2) the amount of time elapsed since the last timeout, (3) the amount of time remaining until the next timeout, (4) the set of timers that must be *on* or *off* for a given transition to execute.

There are several classes of timers used in test sequences [12]:

- *global timers* to assure that the execution of a test case stops even if it is blocked due to unexpected behavior of the SUT;

- *guarding timers* to check constraints on the SUT response time;
- *delaying timers* to delay the sending of messages to the SUT in order to allow the SUT to get into a desired state.

The delaying timers help the SUT reach a state where it can receive the next signal when a tester is too fast. They also help check the reaction of the SUT if a signal is delayed too long to test invalid behavior, e.g., to check that the SUT does not send any signal for a given amount of time [12]. We also consider another type of delaying timers, whose purpose is to help the SUT reach certain states that otherwise would become unreachable (Sections 4.3 and 4.4).

Let us refer to these classes of timers, which focus on an SUT's behavior during testing, as *test-execution timers*. In addition, a protocol's specification may contain *functional timers*, such as retransmission and acknowledgement timers. The timed EFSM model [9] is capable of modeling each type of timers.

2.1 Model Overview

For an FSM represented by graph $G(V, E)$, consider a set of timers $K = \{tm_1, \ldots, tm_{|K|}\}$ that may be arbitrarily started and stopped. Each tm_j is associated with a boolean T_j whose value is *true* if tm_j is running, and *false* otherwise. Each transition can be guarded by ϕ—a time formula obtained from variables $T_1, \ldots, T_k$ by using logical operands $\wedge$, $\vee$, and $\neg$. We further define the following parameters [9, 10]:

- $T_j \in \{0, 1\}$—$T_j = 1$ if tm_j is running; $T_j = 0$ otherwise;
- $D_j \in \mathcal{R}^+$—tm_j's timeout value (i.e., timer length);
- $f_j \in \mathcal{R}^+$—time-keeping variable denoting the current time of tm_j;
- $c_i \in \mathcal{R}^+$—time needed to traverse transition $e_i \in E$, which is obtained from e_i's definition in the specification or assessed by a domain expert;
- time condition for e_i: $\langle \phi_i \rangle$—e_i can trigger only if its ϕ_i is satisfied;
- action list for e_i: $\{\varphi_{i,1}, \varphi_{i,2}, \ldots\}$—each one updates a variable's value;
- $c_p^s \in \mathcal{R}^+$—time needed to traverse a self-loop of node $v_p \in V$;

Based on the time-related variables described above, the model first defines a set of conditions and actions for four different types of transitions. The original G is then augmented as G', to which the INDEEL algorithms [6, 7] are applied. The algorithms, by analyzing the conflicts in a class of EFSMs, prevent inclusion of two or more conflicting edges in the same path in a test sequence. In the case of timed EFSMs, the time variables are treated as context variables. The conflicts are resolved by splitting the graph edges and nodes such that each conflicting pair of edges is placed in a different sub-graph. The resulting EFSM graph (represented by G'') does not contain any infeasible paths, and hence can be used as an input to the FSM-based test generation methods [14]. The algorithms avoid unnecessary state explosion during the conflict resolution by creating the new sub-graphs only when needed.

By augmenting the original INDEEL, we significantly reduce the number of tests, while preserving all feasible transitions of G in G' after the INDEEL applied to eliminate inconsistencies (converting G' to G''). As a result, our method achieves the goal to *cover every state transition at least once* [10].

Eqs. 1 and 2 show the conditions (enclosed in angled brackets $\langle\rangle$) and actions (enclosed in curly braces $\{\}$) for transitions of Types 1 and 2. To reduce the test sequence length, additional two types of transitions merge non-timeout self-loops of v_p sharing the same time condition $\langle\phi_{p,l}\rangle$. An in-depth interpretation of these transition types is presented in Refs. [9, 10].

Type 1 *timeout transition* $e_i^j = (v_p, v_q)$*, defined for each timer* tm_j *(*e_i^j *may be a self-loop, i.e.,* $p = q$*)*

$$\begin{aligned} &\langle T_j \wedge (D_j - f_j < D_k - f_k)\rangle \\ &\{T_j = 0; f_k = f_k + \max(0, c_i + D_j - f_j); f_j = -\infty\} \end{aligned} \tag{1}$$

Type 2 *non-timeout transition* $e_i = (v_p, v_q)$*, which may be a self-loop that starts/stops a timer or a non-self-loop*

$$\langle f_k < D_k\rangle\{f_k = f_k + c_i\} \tag{2}$$

3 Fault Analysis

In our analysis, several well known assumptions [8, 14] on the specification and the IUT are valid: (1) the specification is strongly-connected, reduced, and deterministic; (2) the IUT has the same input alphabet as the specification; and (3) the faults do not increase the number of states in the IUT. The detection of transfer/output faults [16] depends on the state verification method [1, 15, 18], and is not part of the timing-fault analysis. If a timing fault results in a transfer/output fault, we assume that it is detected with high probability.

This paper utilizes the classification of timing faults from Refs. [8, 13]. We prove that under the above assumptions, all single 1-clock and n-clock timing faults [8] are detected when applicable. We also prove that certain faults due to incorrect settings for timer lengths are covered. Fault coverage for multiple simultaneous timing faults is an open problem regardless of the testing model.

At the testing of e_i: (a_i/o_i), output o_i is expected no later than θ after applying a_i. This behavior is controlled by a special-purpose timer in a test harness (outside the IUT), with the length θ: $0 < \theta < c_i + \epsilon$ for a non-timeout e_i, and $\max(0, D_j - f_j) < \theta < c_i + \max(0, D_j - f_j) + \epsilon$ for a timeout e_j^i.

3.1 1-Clock, Interval Fault

- *Timing requirement:* For transitions $e_i = (v_p, v_q; a_i/o_i)$ and h_k, transition e_i can trigger only after applying a_i within time boundaries $[b_1, b_2]$ measured from the execution of h_k.
- *Timing fault I:* a_i is applied at $b_{a_i} \notin [b_1, b_2]$; o_i is observed and v_q verified in less than $b_{a_i} + \theta$.
- *Timing fault II:* a_i is applied at $b_{a_i} \in [b_1, b_2]$; o_i is not observed or v_q not verified in less than $b_{a_i} + \theta$.

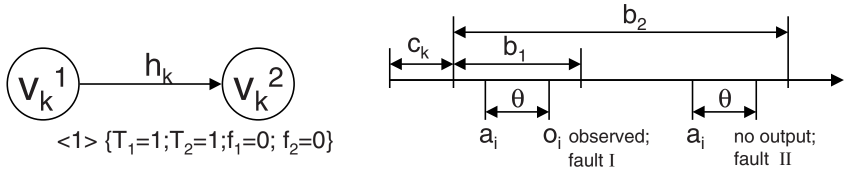

Fig. 1. Modeling 1-clock interval timing fault.

The 1-clock, interval timing requirement can be modeled as shown in Fig. 1. First, tm_1 (with $D_1 = b_1$) and tm_2 (with $D_2 = b_2$) are started in h_k: $\langle 1\rangle\{T_1 = 1; f_1 = 0; T_2 = 1; f_2 = 0\}$. As a result of the timing requirement, e_i triggers after tm_1 and before tm_2 expire, and in its actions stops tm_2 with output o_i, i.e., e_i: $\langle\neg T_1 \wedge T_2\rangle\{T_2 = 0\}$.

Let us consider the state and transition spaces (U_G, R_G), defined for V and variables $\mathcal{V} = \{T_1, f_1, \dots, T_{|K|}, f_{|K|}\}$.

$$U_G = \{(v_p, T_1, f_1, \dots, T_{|K|}, f_{|K|}) : v_p \in V, T_k, f_k \in \mathcal{V}\} \tag{3}$$

Let the sets of transitions between states in U_G and U'_G be denoted as R_G and R'_G, respectively. Let each $u_i \in U_G$ be represented by $(v_i,\ T_1, f_1,\ T_2, f_2)$. Transition e_i is represented in R_G by $g_i = (u_p, u_q)$, where $u_p = (v_p, 0, -\infty, 1, f_2 \in [b_1, b_2])$ and $u_q = (v_q, 0, -\infty, 0, -\infty)$. The time condition of e_i and the definition of u_p indicate that g_i is included (and can trigger) in a conflict-free graph only at a point in time f_2 within the boundaries of $[b_1, b_2]$. Transition e_i will not be included in a test sequence as originating from any u_p with $f_2 \notin [b_1, b_2]$.

Timing fault I is detected in three steps: (1) verifying state v_p, (2) observing o_i, and (3) verifying state v_q. These steps correspond to the execution of infeasible transition g_i, which is present in neither a conflict-free graph nor a test sequence. Timing fault II is detected in two steps: (1) verifying state v_p, and (2) observing $o_k \neq o_i$ or verifying state $v_k \neq v_q$. These steps are not expected as a result of executing transition g_i, which is included in a conflict-free graph and a test sequence. Therefore, all single 1-clock interval faults are detected.

The above analysis can be easily extended for two interval faults such as *time-constraint restriction* and *time-constraint widening* faults, which occur when the IUT changes either the upper or lower bound of a time constraint [8].

3.2 n-Clock Fault

- *Timing requirement:* For transitions $e_i = (v_p, v_q; a_i/o_i)$ and $h_1, \dots, h_n$, transition e_i can trigger after applying input a_i only when, for any $k < n$, h_k was executed before h_{k+1}.
- *Timing fault III:* a_i is applied, o_i is observed and v_q verified in less than $b_{a_i} + \theta$ time when, for at least one k: $2 \leq k \leq n$, h_k is executed before h_{k-1}.

For the n-clock timing requirement, timers $tm_1, \dots, tm_n$ with the infinite lengths are introduced (Fig. 2). Transition h_1 starts tm_1, i.e., h_1: $\langle 1\rangle\{T_1 = 1; f_1 = 0\}$. Each h_k $(2 \leq k \leq n)$ is split into h_k^1 and h_k^2: the former triggers

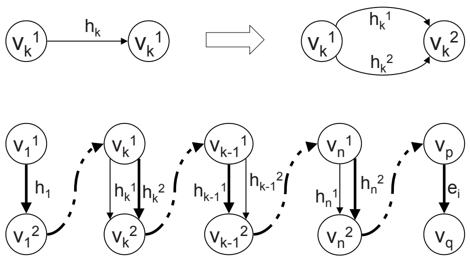

Fig. 2. Modeling n-clock timing fault. Transitions executed when the fault occurs appear in bold.

before tm_{k-1} expires, and starts tm_k, i.e., h_k: $\langle T_{k-1}\rangle\{T_k = 1; f_k = 0\}$; the latter triggers when tm_{k-1} is not running, and in its actions stops tm_k, i.e., h_k: $\langle\neg T_{k-1}\rangle\{T_k = 0\}$. Finally, e_i triggers only when tm_n is running.

The INDEEL allow only feasible test sequences to be generated. Consider two such sequences:

- $(h_1^1, \ldots, h_k^2, h_{k-1}^1, h_{k+1}^2, \ldots, h_n^2, h_{n+1} \neq e_i)$, with the outputs of $(o_{h_1}, \ldots, o_{h_k}, o_{h_{k-1}}, o_{h_{k+1}}, \ldots, o_{h_n}, o_{h_{n+1}})$, where $o_{h_{n+1}} \neq o_i$ or $v_q \neq \text{tail}(h_{n+1})$. Since h_k precedes h_{k-1}, h_k^1's condition of $\langle T_{k-1}\rangle$ cannot be satisfied because only h_{k-1}^1 can start tm_{k-1}. Transition e_i: $\langle T_n\rangle$ is infeasible, since h_n^2 does not start tm_n;
- $(h_1^1, \ldots, h_{k-1}^1, h_k^1, \ldots, h_n^1, e_i)$, with the outputs of $(o_{h_1}, \ldots, o_{h_{k-1}}, o_{h_k}, \ldots, o_{h_n}, o_i)$.

The outputs observed when the timing fault III occurs are as follows: $(o_{h_1}, \ldots, o_{h_k}, o_{h_{k-1}}, o_{h_{k+1}}, \ldots, o_{h_n}, o_i)$. The above multi-clock fault is detected by the first valid test sequence, either by observing $o_i \neq o_{h_{n+1}}$, or by verifying state $v_q \neq \text{tail}(h_{n+1})$ when $o_{h_{n+1}} = o_i$. Therefore, all single n-clock faults are detected.

3.3 Incorrect Timer-Setting Fault

- *Timing requirement:* For timer tm_j of length D_j, timeout transition $e_i^j = (v_p, v_q, -/o_i^j)$ will trigger exactly in D_j time units after tm_j is started in transition h_k.
- *Timing fault IV:* After h_k triggers, o_i^j is observed and v_q verified in less than D_j time.
- *Timing fault V:* After h_k triggers, o_i^j is observed and v_q verified in more than $D_j + c_i$ time.

There are several ways in which this type of timing faults can be detected. The first way is to take advantage of the special-purpose timer with the length of θ. Suppose that transition e_i^j triggers after $D'_j < D_j$ (timing fault IV). If $D_j <= f_j$, then $\theta > 0$ and the fault is not detected through the special-purpose timer. (It may, however, be detected by observing incorrect outputs.) If $D_j > f_j$, then $\theta > D_j - f_j$, and the fault is detected by observing o_i^j in less than θ time. Suppose that transition e_i^j triggers after $D''_j > D_j$ (timing fault V). If $D_j <= f_j$, then $\theta < c_i + \epsilon$ and the fault is detected for $D''_j - D_j > \theta$ by observing o_i^j in more than θ time.

When the fault cannot be detected by using the special-purpose timer, in many cases it may be detected by observing expected outputs from other transitions affected by the fault. Suppose that, for the timing fault IV, the specification allows the following test sequence: $(\ldots, h_k, \ldots, h_n, \ldots, e_i^j, \ldots)$. Consider two cases of timing fault IV: (1) tm_j expires in e_i^j before the implementation is able to execute h_n, and (2) the implementation executes the above test sequence in order. In the first case, e_i^j triggers instead of h_n. This error is detected by observing $o_i^j \neq o_{h_n}$ or verifying $v_q \neq \text{tail}(h_n)$. In the second case, whether the fault is detected depends on the state of the running timers at the time of error occurrence. If there are no running timers after tm_j expires prematurely, timing fault IV is not detected other than by the special-purpose timer.

Figure 3 shows a different case where there are running timers when the fault occurs. Timer tm_a with output o_a is started by h_k (which also starts tm_j), and timer tm_b with output $o_b \neq o_a$ is started by e_i^j. The outputs in a test sequence for this case are as follows: $(\ldots, o_k, \ldots, o_n, \ldots, o_i^j, \ldots, o_a, o_b, \ldots)$. When timing fault IV occurs and $D'_j > D_a - c_i - D_b$ (Case 2.1), the order of outputs in the test sequence is preserved by the implementation, and the discrepancy between D'_j and D_j is not large enough to be detected in this way. If $D'_j < D_a - c_i - D_b$ (Case 2.2), the difference $D_j - D'_j$ is large enough to cause o_b appear before o_a, which is detected by the above test sequence. The analysis for the timing fault V is analogous. Therefore, many single incorrect timer setting faults are detected.
Example 1 (Fault detection) Let us consider a system where 3 timeouts are required to occur in a specific order: timeouts for tm_1 followed by tm_2 and tm_3. A violation of this requirement results in a 3-clock timing fault. Our method can detect this 3-clock fault, which can occur due to several faulty timers as follows: (1) tm_3 expires then tm_1 then tm_2, (2) tm_2 expires then tm_1 then tm_3, or (3) tm_2 expires then tm_3 then tm_1, etc. In this example, the timer lengths are correct, but they are started incorrectly (too early or too late). Otherwise, the errors correspond to incorrect timer-setting faults. As proven in this section, any single n-clock faults are detected by our method.

We do not guarantee detection of all multiple n-clock or multiple incorrect timer-setting faults (or their combination). Consider the above error case tm_3 expires followed by tm_1 and then tm_2. Suppose also that there are incorrect timer settings: tm_1 and tm_2 are set to much shorter lengths than specified. In this case, the timers will all expire incorrectly (i.e., too early), but in the correct order.

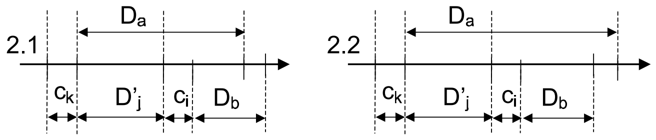

Fig. 3. Incorrect timer-setting fault IV: not detected (2.1); detected (2.2).

Such multiple faults cannot be detected in our model unless special external timers are present in the test harness to monitor timers' expiration times. ∎

4 Application to SDL

The timer mechanism in SDL is used to express timing behavior of a system with only one variable (*Timer*) and three operations (*Set*, *Stop* and *Expire*) [12]. The passage of time is implicit and various background procedures handle timer set, stop, and expiry. As a result, by inspecting an SDL specification, one cannot tell whether or not a timer will expire after the SUT arrives at a certain state. Therefore, infeasible test sequences can easily be generated unless time passage is being completely described by the timer variables. However, in our model, these three operations are represented by actions, with a major distinction that the passage of time is described by explicitly changing the variables' values. This important feature allows detection and elimination of infeasible tests.

Given an SDL specification, there are several steps involved in building a model from which a timed test sequence can be derived:

- extracting partial behavior of an SDL system as an FSM, which is represented by graph $G(V, E)$ (Section 4.1);
- extending thus obtained FSM with time-related variables T_1, f_1, $\ldots$, $T_{|K|}$, $f_{|K|}$, L_1, $\ldots$, $L_{|V|}$, $t^s_{1,1}$, $\ldots$, $t^s_{|V|,M_{|V|}}$. The resulting timed EFSM models the original system along with its time-related behavior, and is represented by graph $G'(V', E')$;
- augmenting G' algorithmically with the time conditions and actions.

4.1 FSM Derivation from SDL Specification

Extracting an FSM that represents partial behavior of an SDL system is beyond the scope of this paper. Instead, we adopt the techniques reported in the literature [3, 22]. One such technique [22] uses a special procedure to construct a single-entry multiple-exit graph for an EFSM modeling an SDL process. In the second stage, the extracted EFSM is converted to an FSM, which can be accomplished by using one of several known approaches [6, 14].

One can also use an FSM extractor called the FEX tool [3]. In general, transitions' predicates in an SDL specification depend on both internal (context) variables and input parameters. The approach behind FEX is to extract an FSM by the partial unfolding of variables of enumerated types, while using the predicates as part of the corresponding FSM inputs. The behavior of the SDL specification is thus approximated by an FSM (called an *approximating machine*), where the FSM's input is defined as a pair <*input signal, predicate*>, and most states correspond to the control states of the SDL specification. The FEX tool constructs such an approximating machine from a given SDL specification with the help of a normalization algorithm. The procedure is especially effective for the specifications with relatively simple predicates, which is the case for most protocols.

Assuming an efficient procedure for the FSM extraction (like the ones described above), we will now illustrate test derivation for SDL specifications, where guarding (Section 4.2) and delaying (Section 4.3) timers are taken into account. Test derivation for the functional timers appears in Ref. [11].

4.2 Guarding Timers

Let us consider the benchmark time-constraint specification in Fig. 4, given in TTCN-3 notation (taken from Ref. [12]), which uses two guarding timers. The SUT has two *Points of Control and Observation (PCOs)*, called A and B, through which a tester entity can send and receive messages to and from the SUT. A tester handling PCO A sends input a to the SUT, and sets two guarding timers, *T_Guard_Min* and *T_Guard_Max*, to 2 and 5 seconds, respectively. The expected behavior of the SUT is to wait for *T_Guard_Min* timeout (line 5), and then receive the output d before *T_Guard_Max* timeout (lines 13 and 14). If the SUT's reply d is too quick (i.e., before *T_Guard_Min* timeout in lines 6-10) or too slow (i.e., after *T_Guard_Max* in lines 15-19), the SUT fails the test.

The above functional requirement of the system can be formulated as a special case of the 1-clock, interval timing requirement [8]. Let us use the fault analysis from Section 3.1 as the basis for modeling the violation of the timing requirement. First, transition h is introduced as $h = (v_p, v_q, a/d)$ with the execution time of c_h (v_p is the initial state). Next, h is split into $h_1 = (v_p, v_{q'})$ with execution time of c_h and $h_2 = (v_{q'}, v_q)$ with execution time of 0, as shown in

```
1  timer T_Guard_Min; timer T_Guard_Max;
2  A.send (a);
3  T_Guard_Min.start (2); T_Guard_Max.start (5);
4  alt {
5  [ ] T_Guard_Min.timeout;
6  [ ] A.receive (d);
7          {
8          verdict.set (fail);
9          MyComponent.stop;
10         }
11     }
12 alt {
13 [ ] A.receive (d);
14         { T_Guard_Max.stop; }
15 [ ] T_Guard_max.timeout;
16         {
17         verdict.set (fail);
18         MyComponent.stop;
19         }
20     }
```

Fig. 4. Example SDL spec with two guarding timers (in TTCN-3 format).

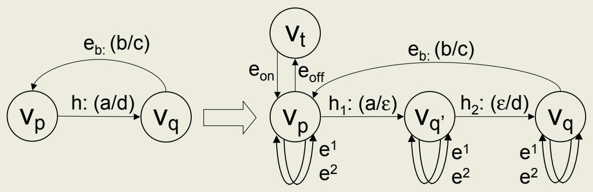

Fig. 5. Modeling 1-clock timing faults.

Figure 5. Then, the timing requirement can be reformulated as follows (ε denotes a null input or output):

- *Timing requirement:* For transitions $h_1 = (v_p, v_{q'}; a/\varepsilon)$ and $h_2 = (v_{q'}, v_q, \varepsilon/d)$, transition h_2 can trigger only within time boundaries $[b_1, b_2]$ measured from the execution of h_1.
- *Timing fault I:* d is observed in less than b_1 seconds after application of a.
- *Timing fault II:* d is observed in more than b_2 seconds after application of a.
- *Timing fault III:* d is not observed at all after application of a.

These timing faults can be modeled as transfer/output faults [16] for the graph in Figure 5. First, timers tm_1 and tm_2 are introduced with the lengths of $D_1{=}b_1$ and $D_2{=}b_2$, respectively. The timers are started in h_1: $\langle \neg T_1 \wedge \neg T_2 \rangle \{T_1 = 1; f_1 = 0; T_2 = 1; f_2 = 0\}$. Because of the timing requirement, h_2 can trigger only after tm_1 and before tm_2 expire, i.e., h_2: $\langle \neg T_1 \wedge T_2 \rangle \{\}$. The timeout transitions for tm_1 and tm_2 are e^1: $\langle T_1 \rangle \{\ldots\}$ and e^2: $\langle T_2 \rangle \{\ldots\}$, respectively. An expiry of tm_1 and tm_2 generates outputs o_1 (tm_1's timeout) and o_2 (tm_2's timeout).

State v_t is introduced as the initialization state, where a test sequence originates and terminates. A test sequence starts in state v_t with edge e_{on}: $\langle 1 \rangle \{T_1 = 0; T_2 = 0; f_1 = -\infty; f_2 = -\infty\}$, which initializes all timers. It terminates when traversing edge e_{off}: $\langle \neg T_1 \wedge \neg T_2 \rangle \{\}$, bringing the IUT from v_0 back to state v_t. The time condition of e_{off} ensures that at this point all timers are inactive.

The above transitions also have the following appended conditions and actions [9]. For simplicity, we only show the appended conditions and actions for transitions h_1 and h_2:

$$\begin{aligned} h_1 : \ & \langle f_1 < 2 \wedge f_2 < 5 \wedge \neg T_1 \wedge \neg T_2 \rangle \{f_1 = f_1 + c_h; \\ & f_2 = f_2 + c_h; T_1 = 1; f_1 = 0; T_2 = 1; f_2 = 0\} \qquad (4) \\ h_2 : \ & \langle f_1 < 2 \wedge f_2 < 5 \wedge \neg T_1 \wedge T_2 \rangle \{\} \qquad (5) \end{aligned}$$

Once G' is built, the infeasible transitions are removed from it by the IN-DEEL [6], which produces a conflict-free EFSM. Then, any FSM test-generation method [14] can be applied to obtain an efficient test sequence.

Consider a feasible test sequence and the associated I/O pairs:

$$(e_{\text{on}}, h_1, e_1, h_2, e_2, e_b, e_{\text{off}}) \qquad (6)$$

$$(o_{\text{on}}, a/\varepsilon, \varepsilon/o_1, \varepsilon/d, \varepsilon/o_2, b/c, o_{\text{off}}) \qquad (7)$$

Note that, while the presented framework does not distinguish input/output sequences $(a/\varepsilon, \varepsilon/o_1)$ and (a/o_1), no timing fault is undetected due to this property. On the other hand, when timing fault I occurs, the SUT traverses an infeasible path $(e_{\text{on}}, \ldots, h_1, h_2, \ldots)$, with the corresponding sequence of I/O pairs $(o_{\text{on}}, \ldots, a/\varepsilon, \varepsilon/d, \ldots)$. Similarly, the SUT traverses infeasible path $(e_{\text{on}}, \ldots, h_1, \ldots, h_2, \ldots)$, with the corresponding sequence of I/O pairs $(o_{\text{on}}, \ldots, a/\varepsilon, \ldots, \varepsilon/o_2, \varepsilon/d, \ldots)$, when timing fault II occurs. In the case of timing fault III, the SUT traverses another infeasible path $(e_{\text{on}}, \ldots, h_1, e_1, e_2, \ldots, e_{\text{off}})$, with the I/O pairs of $(\ldots, a/\varepsilon, \varepsilon/o_1, \varepsilon/o_2, \ldots, o_{\text{off}})$. All these infeasible paths are eliminated by the IN-DEEL from a conflict-free graph and valid test sequences. An observation of the infeasible I/O pairs is thus detected by test sequence (6) by comparing them with outputs in (7).

Specifically, observing d after applying a detects timing fault I; observing o_2 after applying a detects timing fault II; and observing a pair o_1, o_2 after applying a detects timing fault III. None of these observations correspond to those required by (7).

4.3 Delaying Timers

To prevent feasible transitions from becoming unreachable during testing, the transitions that start a guarding or functional timer may need to be delayed by certain amount of time [9]. The following rule is applied to graph G''s traversal:

- If e_i starts a timer and at least one other timer is running when e_i is to be traversed, delay e_i's traversal by the amount of time less than the time remaining until the earliest timeout.

The action of delaying such transitions allows us to explore various orderings of timers' expirations by causing certain timers to expire before others. The length of delaying timers is found algorithmically.

Example 2 (Delaying timers) The FSM in Figure 6 consists of three states v_0 (the initial state), v_1, and v_2, and five transitions e_1 through e_5. Suppose that all transitions take 1sec to traverse and each has the time condition $\langle 1 \rangle$ (i.e., *true*). There are two functional timers defined for the FSM: tm_1 (started by e_1) with the length of $D_1 = 4$ and the timeout transition e_3, and tm_2 (started by e_2) with the length of $D_2 = 2$ and the timeout transition e_4. Transitions e_3 and e_4 also explicitly stop timers tm_2 and tm_1, respectively.

Let us illustrate that e_3 may not be traversed if no delaying timers are used. When the IUT is in its initial state v_0 with all timers inactive, there is no need to delay e_1, since a delay cannot affect the time-related behavior of the system. Suppose that a tester does not delay e_2 either. In this case, when v_2 is visited, tm_1 and tm_2 have 3sec and 2sec left until expiration, respectively. Timer tm_2 expires first in the timeout transition e_4 that also stops tm_1. The system returns to v_0 with all timers stopped and e_3 never traversed.

On the other hand, when e_2 is delayed by more than 1sec, tm_1 and tm_2 have less than 2sec and exactly 2sec left until expiration, respectively. In this case, timer tm_1 expires first in the timeout transition e_3, which also stops tm_2. Thus by choosing the length of the delaying timer in e_2 as 0, a tester can traverse e_4. At another visit to v_1, the length greater than 1 will make e_3 feasible. ■

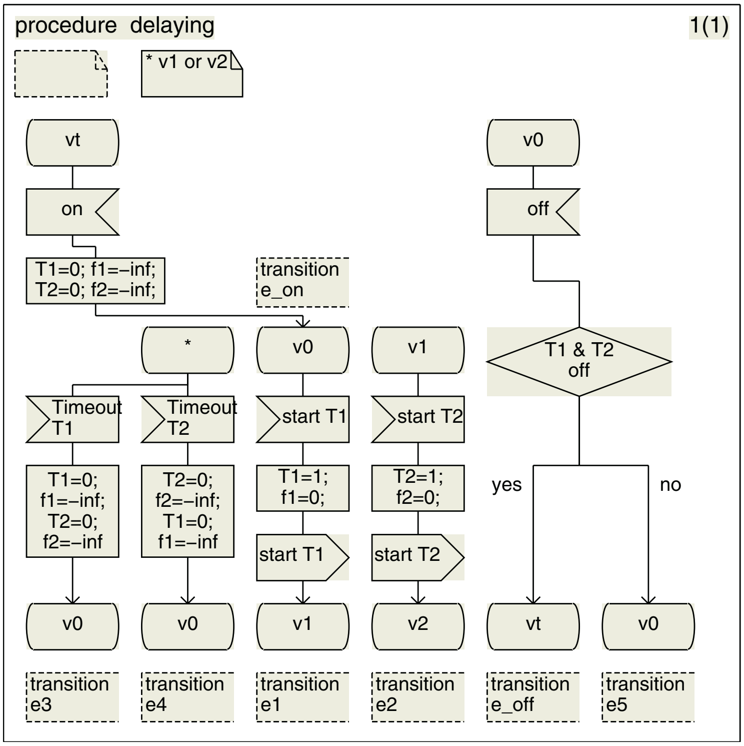

Fig. 6. Delaying affects reachability.

To generalize our previous results [9], let us consider the following state space defined for states in V and variables $\mathcal{V} = \{T_1, f_1, \ldots, T_{|K|}, f_{|K|}\}$:

$$W = \{(v_p, (\tau_{j_1}, y_{j_1}), \ldots, (\tau_{j_n}, y_{j_n})) : v_p \in V, T_{j_k} \in \mathcal{V}\}$$

where $\tau_{j_1}, \ldots, \tau_{j_n}$ are indices of running timers in the order of expiration. Each $y_{j_{k\neq 1}}$ is defined as the time between $tm_{j_{k-1}}$'s and tm_{j_k}'s timeouts; y_{j_1} is set to 0. Let W'_G be the subset of W reachable in G', with the set of transitions among states in W'_G denoted as X'_G. Each $x_i \in X'_G$ is derived [14] from the original G''s transition $e_i = (v_i^1, v_i^2)$ and labeled with the following parameters:

- e_i—original transition in E
- $w_i^1, w_i^2 \in W'_G$—x_i's start and end states in W'_G, respectively, where $w_i^1 = (v_i^1, \ldots)$ and $w_i^1 = (v_i^2, \ldots)$

Suppose that transition $x_i \in X'_G$ is to be traversed. It is clear that the time-related components (i.e., variables of T_i and y_i) of w_i^1 and w_i^2 are identical unless x_i is derived from a timeout transition or a transition that starts/stops a timer. Other transitions alter neither the order nor the time between timer expirations, which makes it unnecessary for a tester to delay their inputs.

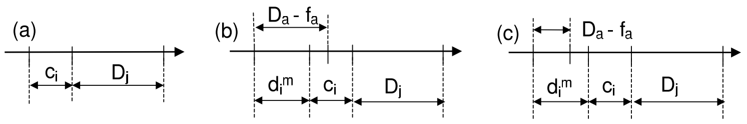

Fig. 7. Delaying transition x_i.

If x_i is a timeout transition for tm_j, the amount of time a tester can delay x_i is independent of the tester's action and equal to $D_j - f_j$. If x_i is not a timeout transition, one of the timers—say tm_a—is to expire first. Let d_i^m be the amount of time by which x_i is delayed in this case. It can be observed that if x_i is to be traversed instead of tm_a's timeout, d_i^m must be less than $D_a - f_a$ (Figure 7 (b)). In the case where none of the timers are running before traversing x_i (Figure 7 (a)), d_i^m will be set to 0 because time passage does not affect system behavior if all timers are inactive. (A delay greater than $D_a - f_a$ cannot be applied, as shown in Figure 7 (c)).

If x_i stops a timer, delaying x_i by any $0 < d_i^m < D_a - f_a$ does not result in the end state w_i^2 different from the end state for a zero delay. Therefore, d_i^m will be set to 0 as for a timeout transition. If x_i starts a timer, delaying x_i by any $0 \leq d_i^m < D_a - f_a$ is likely to result in multiple end states w_i^2 depending on the value of d_i^m. These multiple copies of w_i^2 need to be considered in G', since certain transitions in G' may be feasible only for a specific copy.

To satisfy the above requirement, each x_i will be replaced by a set of transitions X_i. Transition $x_i^{(0)} \in X_i$ handles the case with d_i^m set to 0 where all timers are inactive before traversing x_i. Transition $x_i^{(0)}$ has the following appended condition: $(\forall k)$ *timer* tm_k *not running.* Formally, this condition is $\langle \neg T_k \rangle$.

The case where d_i^m is upper bounded by a running timer tm_a with the shortest time to expire is handled by transitions $x_i^{(a)} \in X_i$, defined for each a: $1 \leq a < |K|$. In the above conversion, x_i is replaced with $|K| + 1$ transitions in X_i, out of which only one has a consistent condition, i.e., $x_i^{(0)}$ if no timer is running, or $x_i^{(a)}$ for a particular tm_a that is to expire first. Each $x_i^{(a)}$ has the following appended condition and actions:

- *condition:* for each timer $tm_{k \neq a}$: *timer* tm_a *running* <u>*AND*</u> *timer* tm_a *is to expire before* tm_k. Formally, this condition is $T_a \wedge (D_a - f_a < D_k - f_k)\rangle$;
- *action:* for each k, increment tm_k's current time by the introduced delay: $f_k = f_k + d_i^m$, where $0 \leq d_i^m < D_a - f_a$.

During an application of the INDEEL, the two inequalities of $d_i^m \geq 0$ and $d_i^m < D_a - f_a$ must be included in the consistency check of conditions involving d_i^m. The actual instantiation of parameter d_i^m, i.e., assigning a particular value from between d_i^m's bounds, takes place after generating a test sequence. In addition, if x_i stops or starts timer tm_j, the actions $\{T_j = 0; f_j = -\infty\}$ or $\{T_j = 1; f_j = 0\}$ must be appended to x_i's action list, respectively.

Let us now define a trace in state space W'_G, which will be instantiated as a test sequence in E [10].

Definition 1 *A* trace *of G' in state space W'_G is defined as a feasible sequence of tuples $t'_G = (d_1, x_1), \ldots, (d_m, x_m)$, where, for each x_k, tester delays applying a_k to an IUT by d_k. For a non-timeout x_k that starts a timer, $d_k \in \mathcal{R}^+$; for a timeout $x_i = x_i^j$, $d_k = \max(0, D_j - f_j)$, for the remaining transitions, d_k is 0.*

Example 2 (cont'd) For the FSM of Fig. 6, consider state space W'_G and its transition set X'_G. The IUT traverses x_1 to enter w_1. In w_1, if no delay is applied before traversing x_2, the IUT will move to state $w_2(0) = (v_2, (2,0), (1,1))$ regardless of a possible delay applied for x_1. If, however, delay d_2^m is applied before x_2 is triggered, the IUT may be in multiple states distinguished by the value of d_2^m. For the delay $1 < d_2^1 < 3$, x_3 is feasible and x_4 is not feasible in the states defined by $w_2(d_2^1) = (v_2, (1,0), (2,2))$. For $0 \le d_2^2 < 1$, in any state $w_2(d_2^2) = (v_2, (2,0), (1, 1 - d_2^2))$, x_4 is feasible and x_3 is not feasible.

A test sequence can be algorithmically derived in X'_G as the following parameterized trace:

$$t'_G(d_2^1, d_2^2) \quad = \quad (0, x_1), (d_2^2, x_2), (2, x_4), (0, x_1), (d_2^1, x_2), (3 - d_2^1, x_3) \qquad (8)$$

where $x_2 = x_2^{(1)} \in X_2$, $0 \le d_2^1, d_2^2 < 3$, and the accumulated conditions are

$$\begin{aligned} x_3 &: \langle (D_1 - f_1 < D_2 - f_2) \wedge \ldots \rangle \equiv \langle (d_2^1 > 1) \wedge \ldots \rangle \\ x_4 &: \langle (D_2 - f_2 < D_1 - f_1) \wedge \ldots \rangle \equiv \langle (d_2^2 < 1) \wedge \ldots \rangle \end{aligned}$$

The corresponding trace (test sequence) in E with the instantiated values of parameters $d_2^1 = 2$ and $d_2^2 = 0$ is as follows:

$$t'_G(2, 0) \quad = \quad (0, e_1), (0, e_2), (2, e_4), (0, e_1), (2, e_2), (1, e_3) \qquad (9)$$

Eq. (9) indicates that, when e_2 is traversed the first and second times, the length of its delaying timer is set to 2 and 0, respectively. ▪

4.4 Flexible Timeout Settings

Let us now illustrate the advantages of our approach with respect to flexible modeling of timeout settings. If the timer lengths are fixed in advance (as they are, e.g., in Timed Automata (TA) [2]) certain portions of the system may become unreachable. In particular, in a complex system or protocol it is difficult to predict and manually assign the correct lengths for functional timers. Our model offers the flexibility to define timer lengths as variables, and have the INDEEL find the appropriate timer ranges, as shown below.

Example 2 (cont'd) Suppose the timeout settings for the FSM of Fig. 6 are D_1=4 and D_2=5. For transitions x_3 and x_4, the accumulated conditions are

$$x_3 : \langle (d_2^1 < 3) \wedge \ldots \rangle, \quad x_4 : \langle (d_2^2 < -2) \wedge \ldots \rangle \qquad (10)$$

The only possible sequence of delay/edge pairs through the FSM of Fig. 6 is trace $t'_G(d_2^1, d_2^2) = (0, x_1), (d_2^1, x_2), (3 - d_2^1, x_3)$ containing x_3 with feasible condition in (10). During test generation, path $(0, x_1), (d_2^2, x_2), (5, x_4)$ that contains x_4 is also considered, but pruned because its condition in (10) is always false for the initial timeout settings. As a result, x_4 is always infeasible, and hence those portions of the graph reachable only through x_4 are untestable.

In our model, to find the timeout settings that make x_4 feasible, D_2 is a variable rather than a constant. In this case, the following parameterized trace will be obtained:

$$t'_G(d_2^1, d_2^2, D_2) = (0, x_1), (d_2^2, x_2), (D_2, x_4)(0, x_1), (d_2^1, x_2), (3 - d_2^1, x_3) \quad (11)$$

where $x_2 = x_2^{(1)} \in X_2$, with the following accumulated conditions:

$$x_3 : \langle (d_2^1 > 3 - D_2) \wedge \ldots \rangle, \qquad x_4 : \langle (d_2^2 < 3 - D_2) \wedge \ldots \rangle \quad (12)$$

A linear programming algorithm, which is used here to determine path feasibility, will find a feasible solution for (12) to instantiate (11). For example, trace (11) can be instantiated as the following test sequence in E:

$$t'_G(2.5, 0, 1) = (0, e_1), (0, e_2), (1, e_4), (0, e_1), (2.5, e_2), (0.5, e_3)$$

for d_2^1=2.5, d_2^2=0, and D_2=1. The methodology not only finds D_2=1, but also computes the appropriate lengths 2.5 and 0 for e_2's delaying timer. ∎

5 Conclusion

A recent model for testing systems with multiple timers is extended to compute proper input delays and timeout settings, and is applied to several types of timers required in a testing procedure. This model, combined with the INDEEL algorithms, allows the generation of feasible test sequences. It can be used to specify timers defined as timed extensions to SDL such as guarding and delaying timers. In cases where it is difficult to predict and manually assign correct timer lengths, the model also offers the flexibility to define timer lengths as variables, and have the INDEEL find the appropriate timer ranges.

References

1. A.V. Aho, A.T. Dahbura, D. Lee, and M.U. Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours. *IEEE Trans. Commun.*, 39(11):1604–1615, 1991.
2. R. Alur and D.L. Dill. A theory of timed automata. *Elsevier J. Theoret. Comput. Sci.*, 126:183–235, 1994.
3. G.v. Bochmann, A.F. Petrenko, O. Bellal, and S. Maguiraga. Automating the process of test derivation from SDL specifications. In *Proc. SDL-Forum Symp.*, pp. 261–276, Evry, France, 1997.
4. M. Bozga, S. Graf, L. Mounier, I. Ober, J.-L. Roux, and D. Vincent. Timed extensions for SDL. In SDL'01 [17].

5. S. Budkowski and P. Dembiński. An introduction to Estelle: A specification language for distributed systems. *Elsevier J. Comput. Networks ISDN Syst.*, 14(1):3–24, 1991.
6. A.Y. Duale and M.U. Uyar. Generation of feasible test sequences for EFSM models. In TestCom'00 [21], pp. 91–109.
7. A.Y. Duale and M.U. Uyar. INDEEL: A software system for inconsistency detection and elimination. In ATIRP'01 [20], pp. 3.29–34.
8. A. En-Nouaary, R. Dssouli, and F. Khendek. Timed Wp-method: Testing real-time systems. *IEEE Trans. Softw. Eng.*, 28(11):1023–1038, 2002.
9. M.A. Fecko, P.D. Amer, M.U. Uyar, and A.Y. Duale. Test generation in the presence of conflicting timers. In TestCom'00 [21], pp. 301–320.
10. 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. Network.* (to appear).
11. M.A. Fecko, M.U. Uyar, A.Y. Duale, and P.D. Amer. Efficient test generation for Army network protocols with conflicting timers. In ATIRP'01 [20], pp. 3.47–52.
12. D. Hogrefe, B. Koch, and H. Neukirchen. Some implications of MSC, SDL and TTCN time extensions for computer-aided test generation. In SDL'01 [17].
13. A. Khoumsi, A. En-Nouaary, R. Dssouli, and M. Akalay. A new method for testing real-time systems. In *Proc. IEEE RTCSA: Int'l Conf. Real-Time Comput. Syst. Appl.*, pp. 441–450, Cheju Island, S. Korea, 2000.
14. R. Lai. A survey of communication protocol testing. *Elsevier J. Syst. Softw.*, 62:21–46, 2002.
15. G. Luo, G.v. Bochmann, and A.F. Petrenko. Test selection based on communicating nondeterministic finite state machines using a generalized Wp-method. *IEEE Trans. Softw. Eng.*, 20(2):149–162, 1994.
16. A.F. Petrenko and G.v. Bochmann. On fault coverage of tests for finite state specifications. *Elsevier J. Comput. Networks ISDN Syst.*, 29(1):81–106, 1996.
17. R. Reed and J. Reed, eds. *Proc. SDL-Forum Symp.*, vol. 2078 of *Springer LNCS*, Copenhagen, Denmark, 2001.
18. A. Rezaki and H. Ural. Construction of checking sequences based on characterization sets. *Elsevier J. Comput. Commun.*, 18(12):911–920, 1995.
19. J. Springintveld, F. Vaandrager, and P.R. D'Argenio. Testing timed automata. *Elsevier J. Theoret. Comput. Sci.*, 254(1-2):225–257, 2001.
20. A.J. Tardif and J.W. Gowens, eds. *ARL Advanced Telecommun./Information Distribution Research Program (ATIRP).* UMD Printing, 2001.
21. H. Ural, R.L. Probert, and G.v. Bochmann, eds. *Proc. IFIP TestCom: Int'l Conf. Test. Communicat. Syst.*, Ottawa, ON, 2000.
22. H. Ural, K. Saleh, and A. Williams. Test generation based on control and data dependencies within system specifications in SDL. *Elsevier J. Comput. Commun.*, 23(7):609–627, 2000.