Available online at www.sciencedirect.com

SCIENCE DIRECT®

Microprocessors and Microsystems 27 (2003) 523–531

MICROPROCESSORS AND MICROSYSTEMS

www.elsevier.com/locate/micpro

Generating test sequences using symbolic execution for event-driven real-time systems

Nam Hee Lee*, Sung Deok Cha

Division of Computer Science and AITrc, Department of Electrical Engineering and Computer Science, KAIST, 373-1, Gusung-Dong, Yusung-Gu, Daejon, South Korea

Received 11 November 2002; accepted 18 April 2003

Abstract

Real-time software, often used to control event-driven process control systems, is usually structured as a set of concurrent and interacting tasks. Therefore, output values of real-time software depend not only on the input values but also on internal and nondeterministic execution patterns caused by task synchronization. In order to test real-time software effectively, one must generate test cases which include information on both the event sequences and the times at which various events occur. However, previous research on real-time software testing focused on generating the latter information. Our paper describes a method of generating test sequences from a Modechart specification using symbolic execution technique. Based on the notion of symbolic system configurations and the equivalence definitions between them, we demonstrate, using the railroad crossing system, how to construct a time-annotated symbolic execution tree and generate test sequences according to the selected coverage criteria.

Keywords: Real-time system testing; Symbolic execution; Modechart

1. Introduction

Dependable real-time software must produce functionally correct results within the specified time intervals. Validation of real-time software can be particularly difficult since it often consists of cyclic and interacting tasks, exhibiting a large number of nondeterministic execution patterns for a given input set. Formal verification and testing is well-known and complementary software quality assurance approaches. Although formal verification made impressive technical advances to the degree that they are used on large and complex industrial projects [1,2], it does not eliminate the need for testing. It is our strong belief that testing will always remain an essential and indispensable component of any software quality assurance program.

Many testing techniques have been proposed, but relatively few results are provided for real-time software. Input values alone are sufficient as test cases for sequential software, and input sequences which can exercise a sequence of synchronization are necessary for concurrent software. However, for typical event-driven real-time software, repeated runs using the same input values and sequences do not necessarily follow the identical execution paths and produce the same results. Therefore, both the event sequences and the time of each event occurrence must be included in test cases for real-time software.

There are only a few works on the testing of the temporal behavior of real-time systems [3–5]. In Ref. [3], a technique for generating test cases from TRIO specification is introduced. It extends the classical temporal logic to deal explicitly with time measures, and TRIO formulas can be automatically checked for satisfiability or validity. During the interpretation of a formula F specifying a property of a system, behaviors of the system compatible with F are generated: they are called *histories*. These histories are used as test cases. In Ref. [4], a system is modelled with a formally defined SA/SD-RT notation and translated into the time reachability tree for representing the behavior of the system. Each path from the root of the tree to its leaves represents a potential test case.

In Ref. [5], a technique for testing timing constraints of real-time systems is presented. A constraint graph is used for describing the various timing constraints the system

* Corresponding author.
E-mail address: nhlee@salmosa.kaist.ac.kr (N.H. Lee).

0141-933/03/$ - see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/S0141-9331(03)00102-9

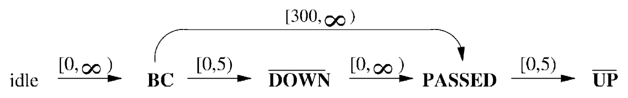

Fig. 1. A constraint graph for railroad crossing.

must satisfy. Timing constraints among various events are shown on the edges of the graph. For example, the constraint graph shown in Fig. 1 indicates that the system is initially in the idle state and that the input event BC, indicating the detection of a train approaching the crossing, may occur at an arbitrary and unknown time $[0, \infty)$. It also shows that the system is required to generate the output event $\overline{DOWN}$ within 5 time units following the receipt of the BC. Similarly, at an arbitrary time in the future, an input event PASSED is expected to occur. However, the occurrences of the events BC and PASSED must be separated by at least 300 time units. Finally, it is a system requirement to generate the output event $\overline{UP}$ within 5 time units following the occurrence of the input event PASSED.

Clarke [5] treated the time interval as another input domain and used traditional domain testing technique. While constraint graph-based approach is useful in generating test cases for real-time software, Clarke did not address how constraint graphs can be automatically generated from a formal specification. This paper bridges the gap by illustrating how such task can be accomplished on a Modechart specification [6] (Fig. 2).

Some authors have proposed coverage criteria to the problem of testing real-time software [3–5,7]. In Ref. [7], test adequacy criteria based on coverage measures of Petri nets topology for concurrent and real-time systems are presented. Specifically, these criteria are based on firing or transition coverage over Petri nets. In Ref. [3], since the actual test case is a subset of the allowed traces of the system, some heuristic techniques are defined to select a subset of all possible test cases for a given specification. These criteria are based on the constructs of the TRIO

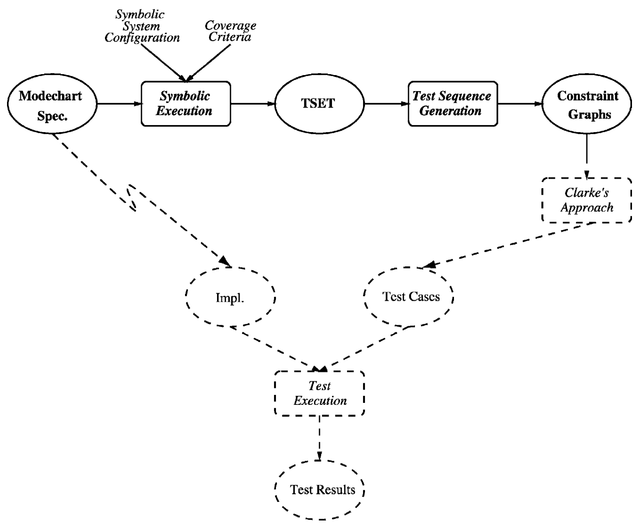

Fig. 2. Our approach for generating test sequences using symbolic execution.

specifications. Ref. [4] describes how to restrict test cases according to different coverage criteria because the time reachability graph would become large even for small-scale systems.

In this paper, we propose coverage criteria designed to test Modechart specification and discuss how symbolic execution technique can be used to generate test cases. We choose Modechart because it provides a rich set of visual constructs with which one can represent various modes the system components can be in and timing constraints among them. We use symbolic execution technique, which treats the time as symbolic values, for generating test sequences. The notion of symbolic system configuration is defined, and a time-annotated symbolic execution tree (TSET) is generated by symbolically executing Modechart specification. Finally, the test sequences are generated from TSET based on the selected coverage criteria and represented as the constraint graph.

The rest of our paper is organized as follows. Modechart specification language is briefly introduced in Section 2. Some coverage criteria of event-driven real-time systems are defined in Section 3, and procedures for constructing TSET are explained in Section 4. We demonstrate, using the railroad crossing system, how to generate constraint graphs. Section 5 concludes the paper.

2. Modechart

Modechart is a visual language devised to specify the behavior of real-time systems [6]. Modechart constructs include *modes*, which are analogous to states in Statecharts [8]; *actions*, which assign values to data variables and require at least one time unit to complete; and *events*, which are instantaneous. Events can be classified into *external*, *mode entry*, *mode exit*, *start*, and *stop* events. External events represent changes in the system environment, mode entry and exit events mark entry into or exit from a mode, and start and stop events mark the start and stop of an action. Modechart borrows compact Statecharts notations for representing concurrent states and provides constructs to denote various timing requirements such as delays, deadlines, and time intervals. In addition, Modechart uses a discrete time model: its delay and deadlines are represented as non-negative integers. Modechart formalism is supported by a software toolset including an editor, a simulator, a verifier, and a code generator [9].

In Modechart, modes may be composed in either serial or parallel manner. Modechart's notion of serial and parallel modes corresponds to OR and AND composition in Statecharts. If M is a *serial* mode with child modes M_1 and M_2, at any given time the system can be in exactly one of M_1 and M_2. If M is a *parallel* mode with child modes M_1 and M_2, then when the system is in M, it is simultaneously in both modes M_1 and M_2. Mode transition, indicating a change in system configuration, is considered an instantaneous event whose trigger condition is specified as

$c_1 \vee c_2 \vee \cdots \vee c_n$

where each disjunct c_i is either (1) a triggering condition for taking the transition, or (2) a lower/upper bound restriction on when the transition may be taken.

- Triggering condition: A disjunct c_i denoting a triggering condition is of the form

  $p_1 \wedge p_2 \wedge \cdots \wedge p_m$

  where the p_j specify a condition for taking the transition depending on the occurrence of an event and/or the truth values of certain predicates. In particular, each subcondition p_j is one of the three forms: a state variable is true (or false) at time t, one of the specified modes at time t, and the occurrence of an event E at time t.
- Lower/upper bound: A condition c_i denoting a lower/upper bound restriction is of the form (r,d), where r is a non-negative integer denoting a delay, and d is a positive integer or ∞ denoting a deadline.

Fig. 3 shows a Modechart specification of a railroad crossing system. The system model consists of two parallel modes CONTROLLER and ENVIRONMENT, and the ENVIRONMENT mode is further decomposed into two parallel modes TRAIN and GATE. The timing constraint (30,50), shown on a mode transition from G.GOING-DOWN to G.DOWN, indicates that it takes at least 30 time units to physically lower the gate and that it never takes more than 50 time units in doing so. A mode transition can also be triggered by other mode entry events such as → T.BC.

The Modechart verifier [10] generates the *computation graph*, whose size is known to be finite, which is analogous to time reachability tree. Fig. 4 shows a part of computation graph of the railroad crossing example. Vertices, also called the points, are labeled with the modes active at the time, and they represent different configurations the system can be in.[1] The edges represent the occurrence of transitions.

Computation graph itself is inadequate as test cases. For example, in node (C.UP, T.PASSED, **G.GOINGUP**), there exist two paths, (C.UP, **T.APPROACH**, G.GOINGUP) and (C.UP, T.PASSED, **G.UP**). Unfortunately, computation graph does not contain information as to which path is taken when.

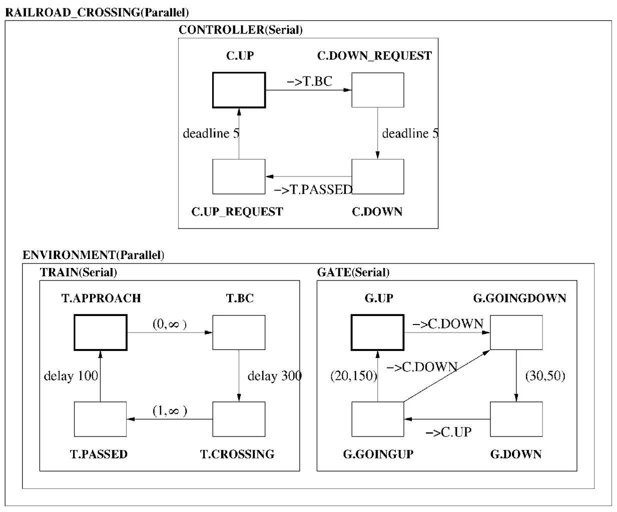

Fig. 3. A modechart specification of railroad crossing.

3. Coverage criteria for event-driven real-time system specifications

For a given Modechart specification, there exist an extremely large, usually infinite, number of potential event sequences the system can encounter in operation. This is especially true of real-time software whose structure often consists of cyclically and concurrently executing set of tasks. Research on software testing has developed the concept of coverage criteria to facilitate cost-effective testing of industrial software, and similar coverage criteria can be defined for event-driven real-time software.

All-event-sequences criterion is satisfied when all possible traces (e.g. combination of events) of the system are tested. Although ideal in principle, it is clearly impractical in practice. Since real-time software typically reacts continuously with the environment, some event sequences are repeated. More realistic criterion would be an *all-cyclic-event-sequences* coverage criterion, which includes all repeated event sequences at least once.

In the Modechart specification, we classify the events into input/output events from the point of view of system under test. Triggering conditions causing an entry (i.e. change) to modes belonging to the environment is considered an input event. Events such as → T.BC and → T.PASSED shown in Fig. 3 are such examples. Likewise, triggering conditions causing mode changes in the controller are considered output events (e.g. → C.DOWN and → C.UP) because such changes occur when the system generate outputs.

Utilizing the notion of input and output events, different criteria, in scope and in strength, can be enforced. For example, all-I/O-events criteria would require that test cases include all input and output events at least once. However, such criterion, while relatively simple to satisfy, may not be of much value in practice when testing real-time software because of inherent nondeterminism. Therefore, existing criteria could be 'strengthened' to include *all-cyclic-event-sequences-N_times* and *all-I/O-events-N_times* criteria. The multiplicity in the criteria would depend on the degree of

[1] The mode entered at the point is indicated in the figure by having its name appear in bold.

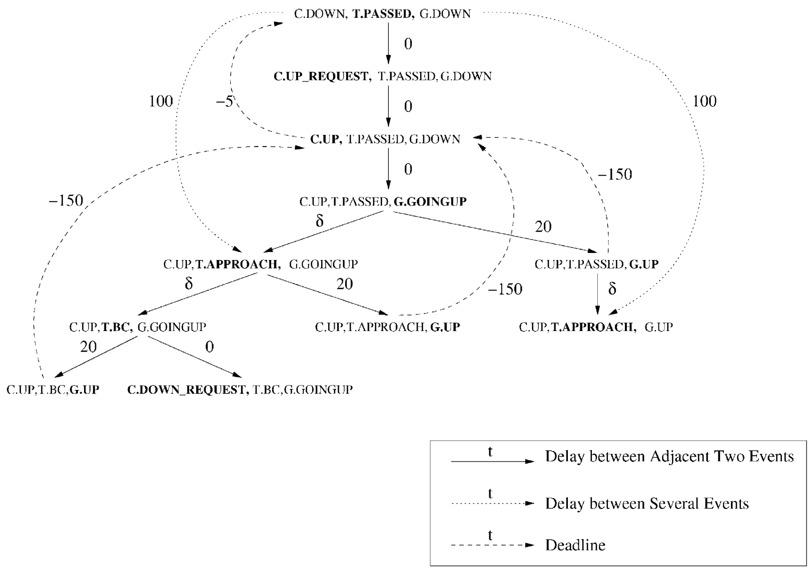

Fig. 4. A part of computation graph of the railroad crossing.

nondeterminism present in the implementation and the degree of assurance the system must provide.

In addition, structural coverage criteria such as *all-modes* and *all-transitions* can be defined in a manner analogous to similar to those of sequential software (Fig. 5).

Having presented various coverage criteria applicable for real-time software, interrelationships among them can be analyzed based on the notion of subsumption[2] as shown in Fig. 5. Given a Modechart specification M and test sequence set C, coverage criterion K_1 *subsumes* $K_2, K_1 \longrightarrow K_2$, iff every pair (M, C) that satisfies K_1 also satisfies K_2.

Lemma 1. *All-transitions subsumes all-cyclic-event-sequences and all-I/O-events.*[3]

Proof.

(1) all-transitions $\longrightarrow$ all-cyclic-event-sequences. (M, C), a pair satisfying all-transitions, satisfies all-cyclic-event-sequences because the cyclic transitions are the subset of transitions.

(2) all-transitions $\nleftarrow$ all-cyclic-event-sequences. Consider a serial mode in M which has a *sink mode* in which no outgoing transition exists. (M, C), a pair satisfying all-cyclic-event-sequences, does not cover this sink mode. As a result, (M, C) does not satisfy all-transitions.

(3) all-transitions $\longrightarrow$ all-I/O-events. (M, C), a pair satisfying all-transitions, satisfies all-I/O-events because the transitions triggered by the external events are the subset of transitions.

(4) all-transitions $\nleftarrow$ all-I/O-events. In Modechart specification, the transition is triggered not only by the external events but also by timing constraints. Because these timing constraints can drive several execution paths, (M, C), a pair satisfying all-I/O-events, does not satisfy all-transitions.

□

Lemma 2. *All-cyclic-event-sequences-N_times, all-I/O-events-N_times, and all-transitions are incompatible.*

Proof.

(1) all-cyclic-event-sequences-N_times $\nrightarrow$ all-I/O-events-N_times. (M, C), a pair satisfying all-cyclic-event-sequences-N_times, does not satisfy all-I/O-events-N_times because a sink mode triggered by an external event can exist.

(2) all-cyclic-event-sequences-N_times $\nleftarrow$ all-I/O-events-N_times. (M, C), a pair satisfying all-I/O-events-N_-times, does not satisfy all-cyclic-event-sequences-N_times due to the same reason of (4) of Lemma 1.

(3) all-cyclic-event-sequences-N_times $\nrightarrow$ all-transitions. Likewise (1), because (M,C), a pair satisfying all-cyclic-event-sequences-N_times, does not cover the sink mode, (M, C) does not satisfy all-transitions.

(4) all-cyclic-event-sequences-N_times $\nleftarrow$ all-transitions. (M, C), a pair satisfying all-transitions, does not satisfy all-cyclic-event-sequences-N_times by definition.

(5) all-I/O-events-N_times $\nrightarrow$ all-transitions. (M, C), a pair satisfying all-I/O-events-N_times, does not

[2] Intuitively obvious relations such as all-cyclic-event-sequences-N_times subsuming all-cyclic-event-sequences, is omitted.

[3] Proof of other subsumption relations is omitted because it can be done intuitively.

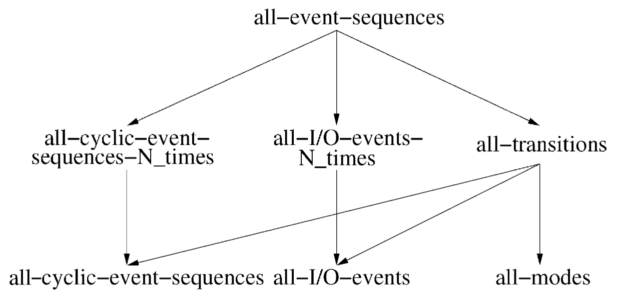

Fig. 5. Coverage criteria subsumption hierarchy.

satisfy all-transitions due to the same reason of (4) of Lemma 1.

(6) all-I/O-events-N_times $\nleftarrow$ all-transitions. (M, C), a pair satisfying all-transitions, does not satisfy all-I/O-events-N_times by definition.

□

4. Test sequence generation

Besides the coverage criteria, when testing real-time software, one must decide how to effectively handle timing issue. Treating time as a monotonically increasing variable is natural. Therefore, we define the notion of symbolic system configuration, similar to the one used in Ref. [11], to capture information on the specific state the system can be in as well as its symbolic timing constraints.

4.1. Symbolic system configuration

Symbolic definition of system configuration helps us avoid redundant and explicit generation of the same state occurring at different times.

Definition 1 (Symbolic state). Let A be a set of atomic modes, SV a set of symbolic values, μ a function from each serial mode to $\langle a, \iota \rangle$, where $a \in A \wedge \iota \in SV$, and C a constraint for ι. A *symbolic state* S is defined as $\langle \mu, C \rangle$.

Based on the symbolic representation of a state, system configuration can be defined to include a set of events and constraints among them that resulted in the system's entry to the current symbolic state.

Definition 2 (Symbolic system configuration). A symbolic system configuration (SC) is defined as $\langle S, E \rangle$, where

- S is a symbolic state and
- E is a set of events that triggered the system's entry to the current symbolic state.

For example, in the case of railroad crossing example shown in Fig. 3, the initial state can be symbolically shown as below. It should be noted that the path condition, C_0, is given concrete value, zero, only because the system is in known and fixed initial state. E_0 is assigned an empty set because there were no external or internal events that caused the system's entry to SC_0.

$$SC_0 = \left\{ \begin{array}{l} S_0 : \left(\begin{array}{l} \mu(CONTROLLER) = \langle C.UP, \tau_{C_0} \rangle \\ \\ \mu(TRAIN) = \langle T.APPROACH, \tau_{T_0} \rangle \\ \\ \mu(GATE) = \langle G.UP, \tau_{G_0} \rangle \\ \\ C_0 = (\tau_{C_0} = 0) \wedge (\tau_{T_0} = 0) \wedge (\tau_{G_0} = 0) \end{array} \right) \\ \\ E_0 : \phi \end{array} \right\}$$

Starting from the initial system configuration, the Modechart is symbolically executed and TSET is constructed by carrying out the following steps:

Step 1: [Initialization]. SC_0, the initial SC, becomes the root node of the TSET.

Step 2: ['Next' SC Computation]. While there are nodes remaining to be expanded in the TSET and the given coverage criterion has not been met, select a node, either in breadth-first or depth-first order, as the current node, and determine when the mode exit events from the current SC may occur. If there are no nodes left to be expanded in the TSET, go to step 4.

Step 3: [Equivalence Check and Coverage Decision]. In order to avoid repeated expansion of TSET nodes, newly generated SCs are compared against the existing one for symbolic equivalence. Steps 2 and 3 are repeated until there are no more nodes to be expanded or the selected criterion has been met.

Step 4: [Derivation of Constraint Graphs]. Each path in the TSET is represented with constraint graph, where symbolic timing constraints associated with adjacent SCs in the TSET are used to derive timing constraints of the constraint graph. description

4.2. Computing the next symbolic system configuration

Given the initial symbolic system configuration SC_0, the only enabled transition is $t_{\mathtt{T.APPROACH} \rightarrow \mathtt{T.BC}}$. The system enters a new symbolic system configuration SC_1. Exact information as to when the transition from SC_0 to SC_1 takes place is unknown, but the modes of the `CONTROLLER` and the `TRAIN` have been changed. Therefore, a new symbolic time value, τ_{T_1}, is assigned. It should be noted that the path condition, C_1, has been updated and that additional constraint, $(\tau_{C_1} = \tau_{T_1})$, is included to faithfully reflect

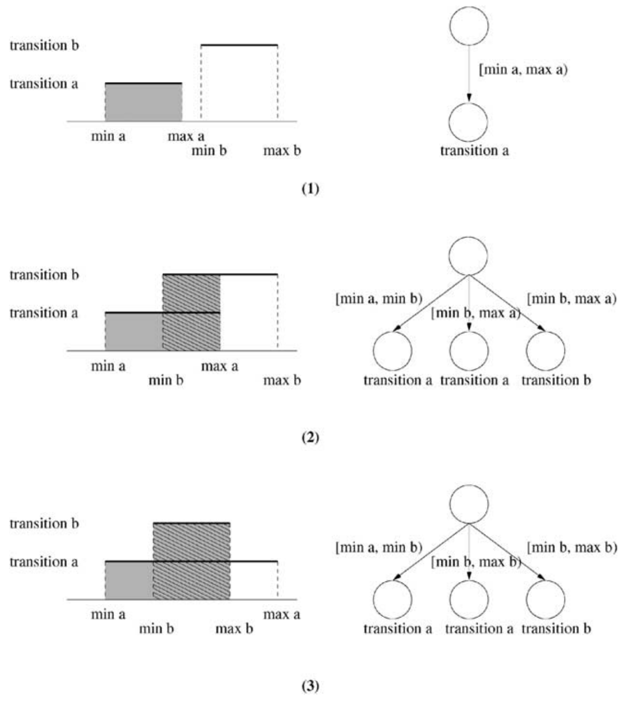

Fig. 6. Subinterval selection for two transitions.

the step semantics of Modechart.

$$SC_1 = \left\{ \begin{array}{l} S_1 : \left(\begin{array}{l} \mu(\mathit{CONTROLLER}) = \langle \mathit{C.DOWN\_REQUEST}, \tau_{C_1} \rangle \\ \mu(\mathit{TRAIN}) = \langle \mathit{T.BC}, \tau_{T_1} \rangle \\ \mu(\mathit{GATE}) = \langle \mathit{G.UP}, \tau_{G_0} \rangle \\ C_1 = C_0 \wedge (0 \le \tau_{T_1} < \infty) \wedge (\tau_{C_1} = \tau_{T_1}) \end{array} \right) \\ E_1 : \{ \rightarrow \mathit{T.BC}, \rightarrow \mathit{C.DOWN\_REQUEST} \} \end{array} \right\}$$

When multiple transitions are enabled, construction of the TSET must consider relationship among their timing constraints (Fig. 6). One possibility, shown in case 1, is that the transition `a` always takes place before the transition `b` does. In this case, only one branch is added to the TSET. Otherwise, as shown in cases 2 and 3, one cannot determine which transition will take place first during the time intervals. Therefore, the TSET must be constructed to handle all possible behaviors.

Continuation of symbolic execution at SC_1 indicates that there are two transitions enabled (deadline 5 and delay 300) and that the TSET contains only one branch, leading to SC_2 since the 'deadline 5' event is guaranteed to take place before the 'delay 300' event (Fig. 6, case 1). However, the occurrence of the former event does not prevent the latter event from taking place. That's why the TSET contains a dashed edge, from SC_1 to SC_4, to indicate the timing constraint of the latter event the system must satisfy over a sequence of several SCs.

$$SC_2 = \left\{ \begin{array}{l} S_2 : \left(\begin{array}{l} \mu(\mathit{CONTROLLER}) = \langle \mathit{C.DOWN}, \tau_{C_2} \rangle \\ \mu(\mathit{TRAIN}) = \langle \mathit{T.BC}, \tau_{T_1} \rangle \\ \mu(\mathit{GATE}) = \langle \mathit{G.GOINGDOWN}, \tau_{G_1} \rangle \\ C_2 = C_1 \wedge \tau_{T_1} \le \tau_{C_2} < 5 + \tau_{T_1} \wedge \tau_{G_1} = \tau_{C_2} \end{array} \right) \\ E_2 : \{ \rightarrow \mathit{C.DOWN}, \rightarrow \mathit{G.GOINGDOWN} \} \end{array} \right\}$$

At SC_6, whose detailed definition of SC is given below, there are two potentially overlapping events that are enabled simultaneously. One event is `→G.UP` whose timing constraints is (20,150), and the other is `→T.APPROACH` event with a given delay of 100 time units. Since the order of these events cannot be statically determined, three edges are created capturing all possible system behaviors (Fig. 6, case 2).

$$SC_6 = \left\{ \begin{array}{l} S_6 : \left(\begin{array}{l} \mu(\mathit{CONTROLLER}) = \langle \mathit{C.UP}, \tau_{C_4} \rangle \\ \mu(\mathit{TRAIN}) = \langle \mathit{T.PASSED}, \tau_{T_3} \rangle \\ \mu(\mathit{GATE}) = \langle \mathit{G.GOINGUP}, \tau_{G_3} \rangle \\ C_6 = C_5 \wedge \tau_{T_3} \le \tau_{C_4} < 5 + \tau_{T_3} \wedge \tau_{G_3} = \tau_{C_4} \end{array} \right) \\ E_6 : \{ \rightarrow \mathit{C.UP}, \rightarrow \mathit{G.GOINGUP} \} \end{array} \right\}$$

SC_7 represents the possibility of event `→G.UP` occurring between time interval [20,100), while SC_8 indicates the same event occurring at [100,150) but prior to the occurrence of the event `→T.APPROACH`. Similarly, SC_9 covers the possibility of detecting another train approaching the crossing while the gate has not been completely raised.

$$SC_7 = \left\{ \begin{array}{l} S_7 : \left(\begin{array}{l} \mu(\mathit{CONTROLLER}) = \langle \mathit{C.UP}, \tau_{C_4} \rangle \\ \mu(\mathit{TRAIN}) = \langle \mathit{T.PASSED}, \tau_{T_3} \rangle \\ \mu(\mathit{GATE}) = \langle \mathit{G.UP}, \tau_{G_4} \rangle \\ C_7 = C_6 \wedge 20 + \tau_{T_3} \le \tau_{G_4} < 100 + \tau_{T_3} \end{array} \right) \\ E_7 : \{ \rightarrow \mathit{G.UP} \} \end{array} \right\}$$

$$SC_8 = \left\{ \begin{array}{l} S_8 : \left(\begin{array}{l} \mu(\mathit{CONTROLLER}) = \langle \mathit{C.UP}, \tau_{C_4} \rangle \\ \mu(\mathit{TRAIN}) = \langle \mathit{T.PASSED}, \tau_{T_3} \rangle \\ \mu(\mathit{GATE}) = \langle \mathit{G.UP}, \tau_{G_5} \rangle \\ C_8 = C_6 \wedge 100 + \tau_{T_3} \le \tau_{G_5} < 155 + \tau_{T_3} \end{array} \right) \\ E_8 : \{ \rightarrow \mathit{G.UP} \} \end{array} \right\}$$

$$SC_9 = \left\{ \begin{array}{l} S_9 : \left(\begin{array}{l} \mu(\mathit{CONTROLLER}) = \langle \mathit{C.UP}, \tau_{C_4} \rangle \\ \mu(\mathit{TRAIN}) = \langle \mathit{T.APPROACH}, \tau_{T_4} \rangle \\ \mu(\mathit{GATE}) = \langle \mathit{G.GOINGUP}, \tau_{G_3} \rangle \\ C_9 = C_6 \wedge 100 + \tau_{T_3} \le \tau_{T_4} < 155 + \tau_{T_3} \end{array} \right) \\ E_9 : \{ \rightarrow \mathit{T.APPROACH} \} \end{array} \right\}$$

4.3. Equivalence of symbolic system configuration

Since it is common practice to structure industrial real-time software as cyclic processes, it is useful to define equivalence relation between the two SCs as follows.

Definition 3 (Equivalence of symbolic system configuration). We consider the two SCs are equivalent if and only if

- they are in the same set of atomic modes
- they have the same set of events, and
- they have the same timing constraints except the symbolic values.

For example, following SC_{14} is equivalent to SC_1, and they need not be further expanded.

$$SC_{14} = \left\{ \begin{array}{l} S_{14}: \left(\begin{array}{l} \mu(CONTROLLER) = \langle C.DOWN\_REQUEST, \tau_{C_5} \rangle \\ \mu(TRAIN) = \langle T.BC, \tau_{T_6} \rangle \\ \mu(GATE) = \langle G.UP, \tau_{G_4} \rangle \\ C_{14} = C_{10} \wedge (0 \leq \tau_{T_6} < \infty) \wedge (\tau_{C_5} = \tau_{T_6}) \end{array} \right) \\ E_{14}: \{ \rightarrow T.BC, \rightarrow C.DOWN\_REQUEST \end{array} \right\}$$

Theorem 1. *When a Modechart specification consists of iterating serial modes, an equivalent symbolic system configuration for a SC can be founded in finite step.*

Proof. The theorem can be proved by induction on the number of iterating serial modes. When a serial mode is only one iterating serial mode and the number of atomic modes in that is N_1, we can find an equivalent symbolic system configuration at most N_1 step because we ignored the symbolic values. Consequently, we can find an equivalent symbolic system configuration for a SC in maximum $N_1 \times \cdots \times N_k$ steps, where k is the number of iterating serial modes. □

Theorem 2. *The TSET is finite.*

Proof. If a Modechart specification has a finite set of atomic modes, the maximum number of possible symbolic system configurations is a product of the cardinality of each serial mode. A SC can be partitioned by several subintervals, but the number of subinterval is finite. As a result, if all serial modes are not iterating modes, then the TSET is finite. Therefore, if one of the serial modes is an iterating mode, then the TSET is also finite because we find an equivalent system configuration in finite step by Theorem 1. □

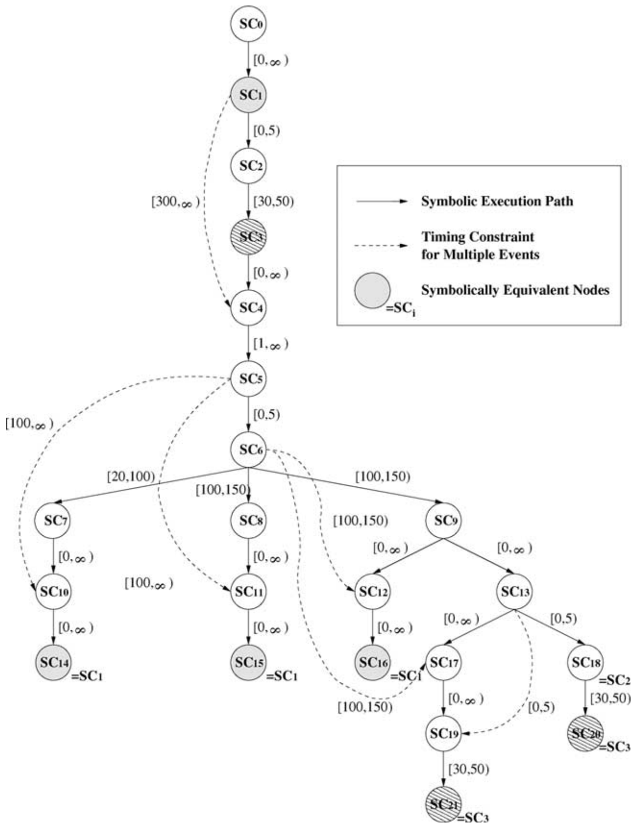

Fig. 7. TSET of the railroad crossing by all-cyclic-event-sequences.

Fig. 7 is the TSET obtained by symbolically executing the Modechart specification shown in Fig. 3 to meet the *all-cyclic-event-sequences* criterion. It should be noted that the time intervals shown in the TSET edges represent relative timing constraints indicating when the transition can take place with respect to the time at which the system had entered the previous SC. Timing constraints that apply to a sequence of transitions, shown in dashed lines in TSET, are also derived.

4.4. Generating test sequences

Each path in the TSET forms a test sequences and it can be represented with constraint graph. In the railroad crossing, there are five unique paths in the TSET shown in Fig. 7, and each path is converted into a constraint graph as shown in Fig. 8. The constraint graph `C0` captures the event sequences common to all paths and is shown separately to avoid duplication. `C1` through `C5` capture different event sequences the system may encounter at each cycle.

Once the TSET is constructed to meet the selected coverage criterion, constraint graphs can be derived by identifying the external event, including the passage of time, that initially triggered transition between the two adjacent SCs. For example, the event set E_1, included in the definition of SC_1, contain two events $\rightarrow$ `T.BC` and $\rightarrow$

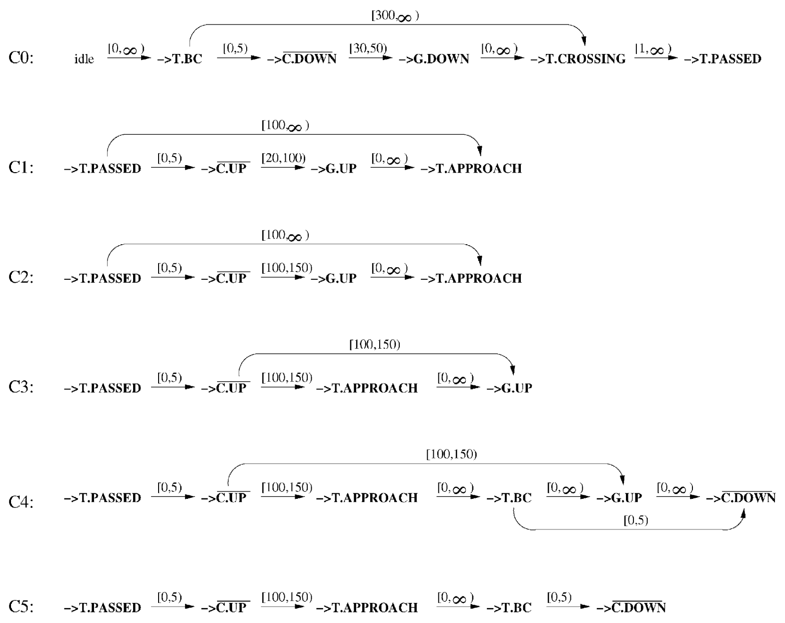
C0: idle [0,∞) ->T.BC [0,5) ->C.DOWN [30,50) ->G.DOWN [0,∞) ->T.CROSSING [1,∞) ->T.PASSED; [300,∞)
C1: ->T.PASSED [0,5) ->C.UP [20,100) ->G.UP [0,∞) ->T.APPROACH; [100,∞)
C2: ->T.PASSED [0,5) ->C.UP [100,150) ->G.UP [0,∞) ->T.APPROACH; [100,∞)
C3: ->T.PASSED [0,5) ->C.UP [100,150) ->T.APPROACH [0,∞) ->G.UP; [100,150)
C4: ->T.PASSED [0,5) ->C.UP [100,150) ->T.APPROACH [0,∞) ->T.BC [0,∞) ->G.UP [0,∞) ->C.DOWN; [100,150); [0,5)
C5: ->T.PASSED [0,5) ->C.UP [100,150) ->T.APPROACH [0,∞) ->T.BC [0,5) ->C.DOWN

Fig. 8. Constraint graphs for the railroad crossing example.

C.DOWN_REQUEST. Since the latter is triggered by the former, the constraint graph C0 contains only the former event and the timing constraint given in the TSET. The constraint graph C1 examines whether or not the system can successfully raise the gate within [20,100) time units after processing the signal that the train has passed the crossing but prior to the arrival of the next train at the crossing. C2 validates similar system behavior for the time interval [100,150), while the purpose of the test case C3 is to test the system behavior in case the next train approaches the crossing before the gate has been completely raised in the interval tested in C2.

There are five cyclic paths, $SC_1 ... SC_{14}$, $SC_{1...15}$, $SC_1 ... SC_{16}$, $SC_3 ... SC_{20}$, and $SC_3 ... SC_{21}$, in Fig. 7. If we were to refer them to as t_1 through t_5, respectively, and if we choose the value of N to be 2 for *all-cyclic-event-sequences-N_times*, there would be a total of 25 test sequences such as $t_1 \cdot t_1$, $t_1 \cdot t_2$, $\cdots$, and $t_5 \cdot t_5$.

5. Conclusions

In this paper, we proposed that various criteria can be defined for event-driven real-time software and that a set of constraint graph can be automatically derived. We used symbolic execution technique for generating the test sequences which include the exact timing intervals for each event. Clarke has previously shown how to generate test cases and test real-time software provided that various and complex timing dependencies have been captured in the constraint graphs. However, his approach failed to describe how accurate constraint graphs can be derived from specification. Our approach, utilizing well-established idea of symbolic execution, nicely complements the work done by Clarke. The combined approach convincingly demonstrates that it is possible to automatically generate test cases for real-time software to meet the specified coverage criteria.

We do not claim in this paper that we solved state explosion problem. It is an inherent and fundamental problem all concurrent analysis encounter. For example, TSET would also face state explosion problem as the number of parallel modes and the overlapping intervals in the Modechart increase. However, several large scale industrial systems we have performed case studies on tend not to have complex timing constrains to the degree that test case generation based on the proposed approach would result in state explosion. Examples include digital TV software (800,000 + lines of application code alone written in Java and C), shutdown system for a nuclear power plant (7000 + lines of Modula code), or satellite controller (3 subsystems and a total of 40,000 + lines of C code). Primary factor is that such systems, like most event-driven and time-critical software, require high assurance, and their design tend to be conservative, analyzable, and as simple as possible. Therefore, we feel that in practice, such state explosion problem is not as serious as one may fear. Primary contribution of this paper is not the state explosion problem, but it is the derivation of complex (and sometimes subtle) timing constraints that might not be adequately addressed if test cases are generated manually and informally. Once TSET is built and a set of constraint graphs is generated, one is expected to choose the criteria, one of those proposed in this paper, based on the resource available in testing and level of assurance desired.

Modechart specification language provides a rich set of constructs to represent various timing requirements. Time-out is the only feature with one can represent timing requirements in Statecharts. Therefore, our approach can be easily applied to specifications written in Statecharts as well as other variant languages based on the finite state machine concept.

Acknowledgements

This work was partially supported by the Advanced Information Technology Research Center (AITrc).

References

[1] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, J.D. Reese, Model checking large software specifications, IEEE Trans. Software Engng 24 (7) (1998).
[2] S. Owre, N. Shankar, J. Rushby, User guide for the pvs specification and verification system, Technical Report, SRI International, 1993
[3] D. Mandrioli, S. Morasca, A. Morzenti, Generating test cases for real-time systems from logic specifications, ACM Trans. Comput. Syst. November (1995) 365–398.
[4] V. Braberman, M. Felder, M. Marre, Testing timing behavior of real-time software, Proceedings of the 10th International Software Quality Week'97, 1997, pp. 143–155.
[5] D. Clarke, I. Lee, Testing real-time constraints in a process algebraic setting, 17th International Conference on Software Engineering, 1995.
[6] F. Jahanian, A.K. Mok, Modechart: a specification language for real-time systems, IEEE Trans. Software Engng 20 (12) (1994) 933–947.
[7] S. Morasca, M. Pezzè, Using high-level petri nets for testing concurrent and real-time systems, in: H. Zedan (Ed.), Real-Time Systems, Theory and Applications, Elsevier, North Holland, 1989, pp. 119–131.
[8] D. Harel, Statecharts: a visual formalism for complex systems, Sci. Comput. Program. (1987) 231–274.
[9] P.C. Clements, C.L. Heitmeyer, B.G. Labaw, A.T. Rose, MT: a toolset for specifying and analyzing real-time systems, Proceedings of Real-Time Systems Symposium, December, 1993, pp. 12–22.
[10] F. Jahanian, D.A. Stuart, A method for verifying properties of modechart specifications, Proceedings of Real-Time Systems Symposium, December, 1988.
[11] C. Ghezzi, S. Morasca, M. Pezzè, Validating timing requirements for time basic net specifications, J. Syst. Software 27 (1994) 97–117.

Nam Hee Lee received the BS and MS degrees in computer science from Korea Advanced Institute of Science and Technology (KAIST), Korea, in 1991 and 1998, respectively. He is currently a PhD candidate at KAIST. His research interests include formal methods, software testing, and quality assurance.

Sung Deok Cha received the BS, MS and PhD degrees in information and computer science from the University of California, Irvine, in 1983, 1986, and 1991, respectively. From 1990 to 1994, he was a member of the technical staff at Hughes Aircraft Company, Ground Systems Group, and the Aerospace Corporation, where he worked on various projects on software safety and computer security. In 1994, he became a faculty member of the Korea Advanced Institute of Science and Technology, Electrical Engineering and Computer Science Department. His research interest includes software safety, formal methods, and computer security.