Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

ELSEVIER

Computer Networks 42 (2003) 405–417

COMPUTER NETWORKS

www.elsevier.com/locate/comnet

Generating test sequences from a set of MSCs

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

Abstract

We propose an approach to generate test cases from a set of Message Sequence Charts (MSCs) by constructing a semantically equivalent finite state machine for testing reactive and embedded software. Test cases are expressed as a sequence of messages to be exchanged among various system entities. We use scenario activation conditions and state assignments to generate only the feasible states and transitions. This paper uses complex digital TV software to illustrate how test cases are automatically generated.
© 2003 Elsevier Science B.V. All rights reserved.

Keywords: Task interaction testing; Scenario-based testing; Message sequence charts

1. Introduction

Usage of large and complex software in controlling embedded systems has become popular. This trend is especially true in home appliances such as digital TV, cellular phones, and Web phones. When developing and testing reactive software for home appliances, engineers are usually under significant time pressure because these products have a relatively short life time. In addition, such software is often subject to changes to meet various customer needs. Furthermore, embedded software usually consists of a large number of concurrently executing and interacting tasks each of which is executed under different conditions. Cost-effective testing of embedded software is of critical concern in maintaining a competitive edge, and it is desirable to automate as much of the testing as possible.

Message Sequence Chart (MSC) notation [1] is an attractive formalism when specifying requirements related to interaction scenarios of embedded software, because MSC is an internationally standardized language, and its graphical notation is easy to understand. MSC includes various constructs to express complex requirements in a hierarchical manner. It shows entities in a system separately and expresses the interactions naturally and explicitly.

Much of the past research on MSC mainly focused on extending notations to concisely describe various requirements and defining formal semantics. Examples of the former research include various in-line expressions, high-level MSCs, and condition constructs. In addition, guarding (scenario activation) conditions and data variables are newly introduced in MSC-2000. Guarding

* Corresponding author.
E-mail addresses: nhlee@salmosa.kaist.ac.kr (N.H. Lee), cha@salmosa.kaist.ac.kr (S.D. Cha).

1389-1286/03/$ - see front matter © 2003 Elsevier Science B.V. All rights reserved.
doi:10.1016/S1389-1286(03)00250-0

conditions, indicating when each scenario is to be activated, are used to specify the state-dependent behavior of embedded software. The process algebra-based standard semantics is defined by ITU-T [2], but it is difficult to handle the guarding conditions and data variables other than by automata-based semantics. Several approaches to generating test cases from MSC also have been studied [3–5]. However, MSC has been primarily used to specify partial behavioral requirements of the system, and was not used in generating test cases. Instead, formal models such as SDL [6] have been used to generate test cases.

Test sequences of reactive software have to consist of sequences of the external input events, because task interactions are determined by the external input events and the current state of the system. As a consequence, to generate executable test cases we need a global system behavior derived from a set of MSCs, each of which captures the required but partial scenario. We propose an approach to construct a global system behavior (Global Finite State Machine—GFSM) from a set of MSCs, and generate test sequences from the GFSM using two test selection criteria. The system behavior is specified by a set of scenarios which are separately expressed for each external input event. Therefore, each scenario consists of a set of hierarchically composed basic MSCs (bMSCs) describing internal interactions, activation conditions, and state information of the system. The global system behavior is obtained and analyzed by combining these MSCs. We choose an automata-based interpretation to formalize semantics of MSCs and to transform a set of bMSCs into a behavior automata, and ultimately a GFSM is derived. Finally, test cases are generated from the GFSM, in terms of the required external events' sequences and task interactions corresponding to the events.

1.1. Related works

There are two different approaches to generate test cases from MSCs. A state space exploration is started which simulates both the given MSCs and the SDL system [4,5,7]. In this case, alternative receive events (which violate the MSCs but are valid according to the SDL specification) are added to the test case representation with a TTCN inconclusive verdict. If a state space exploration is not applicable, the MSCs have to be translated directly into test cases [3,8]. Tau from Telelogic AB is one of the major commercial SDL, MSC and TTCN tool sets. It provides a complete environment for the development of SDL specifications, MSC descriptions and TTCN test suites. In the case of a state exploration, a complete SDL specification is required. However, in the real world, only partial specifications exist for most systems. Often SDL specifications are not available at all.

In [8], a basic MSC is used to identify potentially concurrent events, and such information is used to generate test cases. The system entities are divided into Implementation Under Test (IUT) and Environment (ENV), and interactions among various tasks included in the IUT are ignored. That is, test cases consist of externally visible input and output events only, and flaws related to internal task interactions would be difficult to detect. Our work differs from the above work in that we model the whole system interaction behavior using a set of MSCs, because various MSC constructs provide sufficient features to express complex requirements of embedded software in a hierarchical manner.

Our paper is organized as follows. In Section 2, we briefly review the syntax of MSC. In Section 3, we explain algorithms used to transform a set of MSCs to a semantically equivalent GFSM. In Section 4, we illustrate how to apply the proposed approach to the software, with an example of embedded software running on a digital TV. Finally, Section 5 concludes the paper.

2. Message sequence charts

MSC is a graphical formalism which is used to visualize communication behavior (message sequences). MSC has been standardized by International Telecommunications Union (ITU) in Recommendation Z.120 [1]. Each system component participating in the communication is called an instance and is represented by a vertical time line. Time passes from the top of the chart to its bottom, and all events along each instance axis are totally ordered. Events are either a sending or re-

ceiving message, or a condition (possibly involving more than one instance). The only ordering constraint other than the one imposed by the instance axis is inferred from messages: a message has to be sent before it is received. Thus, a complete MSC defines a partial order on the events it contains. Graphically, messages are represented by arrows pointing in the direction of the message flow, and conditions are depicted by hexagonal lozenges.

For example, bMSC B_1, shown in Fig. 1(a), contains four instances, named 'E1', 'I1', 'I2' and 'E2', and two messages, i_1 and m_1. When the bMSC B_1 is activated, message exchange of i_1 must be completed before the message m_1 can be sent because they are temporally ordered in the same instance. In addition to the simple message exchanges, the MSC standard provides various structuring mechanisms. The 'alt' in-line expression in the bMSC B_1 indicates that after exchanging two messages, either B_3 or B_4 bMSC is selectively executed.

In this paper, we separately specify the interaction sequences for each external event of the target software with a hierarchy of bMSCs. In Fig. 1, the bMSCs B_1, B_3, and B_4 represent a sequence of interactions for an external input event i_1, and the bMSC B_2 represents the other event i_2. Furthermore, we add a top-level bMSC (Fig. 2) that indicates which bMSC is activated when the target software receives external events from the environment.

As the embedded software is typically structured as a set of cyclic tasks, the top-level bMSC is activated repeatedly. Thus, it is represented by a 'loop' in-line construct. Furthermore, we connect all external events to the top-level bMSC by an 'alt' in-line. This means that an external event occurs only after the system has completely processed the previous event. This interpretation corresponds to the synchrony hypothesis used in Statecharts [9]. Embedded software accepts data or command inputs from one or more sources, and the outputs are generated by processing the inputs under the current state of system. As the speed of processing, however, is usually much faster than the minimum inter-arrival time between the input events, internal processing is assumed to occur synchronously.

We make some assumptions when specifying and interpreting MSC specifications. First, we assume that no bMSC is referenced recursively. Otherwise, our algorithm to transform a set of MSCs into a

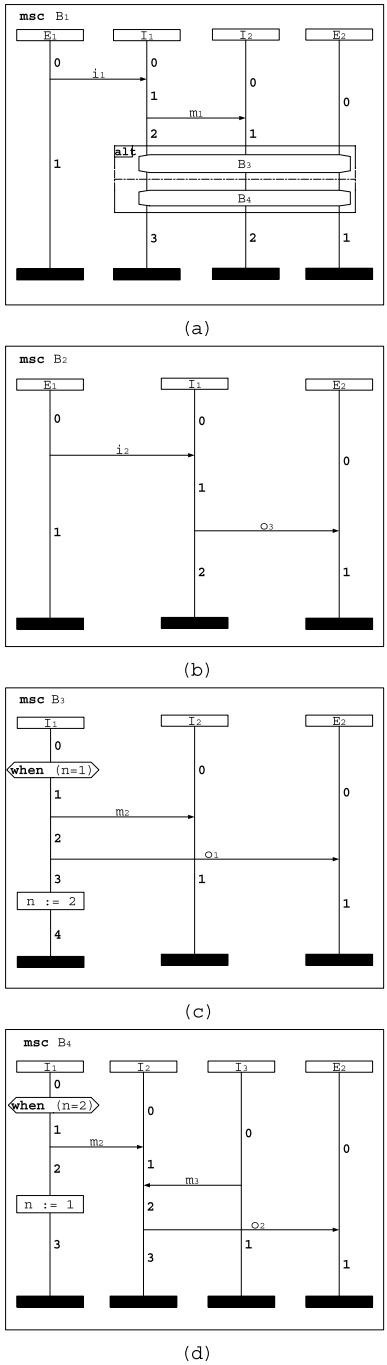

Fig. 1. Example: (a) B_1, (b) B_2, (c) B_3, and (d) B_4.

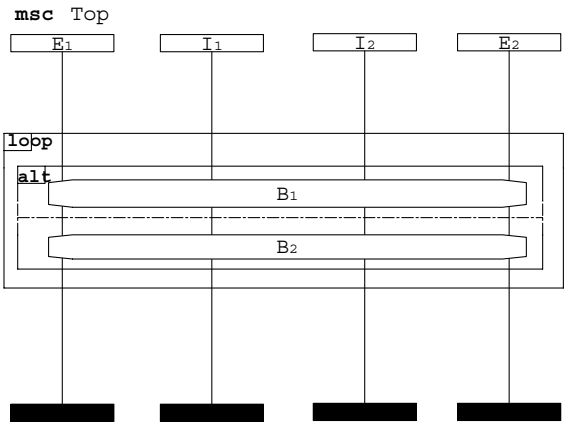

Fig. 2. Top level scenario for example.

GFSM would never terminate. We have not yet found situations under which it is essential to allow recursive referencing of bMSCs. Second, as the asynchronous concatenation model of the standard semantics causes the state explosion problem [10], we use the synchronous concatenation model between the referencing bMSCs and the referenced bMSCs. It means that the referenced bMSCs must be enabled after all messages in the referencing bMSCs are exchanged. Finally, each of the bMSCs representing alternative and optional scenarios has its own scenario activation conditions, and those are represented by 'condition' constructs. Furthermore, all messages in those MSCs are exchangeable only when the activation condition is satisfied.

We define the syntax of MSC as follows. A set of instances, events and labels are represented by P, E and Σ, respectively. The event consists of 'message', 'condition' and 'action' constructs of bMSC represented by E^M, E^C and E^A, respectively. Other constructs such as 'reference' and 'in-line' except 'par' are represented by E^I. Finally, a set of data variables is represented by V.

Definition 1 (*MSCs structure*). Overall structure of the specification consists of the five-tuple $MSCs = (S, s_0, \Sigma^R, n_r, S_r)$, where S is a finite set of bMSCs, $s_0 \in S$ is the top level scenario, Σ^R is a set of names, $n_r : S \to \Sigma^R$ is a *bMSC mapping function* of the names to each bMSC, and $S_r = \{(s_1, s_2) |\ s_1, s_2 \in S\}$ is a reference relation between bMSCs.

Definition 2 (*bMSC*). Each bMSC $S_p \in S$ consists of a six-tuple $S_p = (P_s, E_s, \Sigma_s, p_s, l_s, \langle_s)$, where $P_s \subseteq P$ is a finite set of instances, $E_s \subseteq E$ is a finite set of events, $\Sigma_s \subseteq \Sigma$ is a finite set of labels, $p_s : E_s \to 2^{P_s}$ is an instance mapping function for each event to one or more instances, $l_s : (P_s \cup E_s) \to \Sigma_s$ is a labelling function for each instance and event. $\langle_s : \bigcup \langle_{P_s} \cup \{(m_s, f(m_s)) | m_s \in E_s^M\}$ is an ordering relation between events, where $\langle_{P_s} = \{(e_1, e_2) | e_1, e_2 \in E_s\}$ represents the ordering relation in an instance, and f returns a receiving event for a sending event.

3. Test sequence generation

For generating the test sequences from a set of bMSCs, we first translate the set of bMSCs to a GFSM providing useful information for testers. Our approach reduces the number of states and transitions by considering the characteristics of reactive and embedded systems. For example, an execution of scenarios may cause the system configuration to be changed. The number of states is minimized by encoding such information in the GFSM. In addition, we adopt synchronous interpretation to reduce the number of transitions, but we interpret message exchanges asynchronously only when it is absolutely necessary. Finally, the test sequences are generated from the reduced GFSM using two test selection criteria.

3.1. GFSM construction

The first step in generating a GFSM is to define *behavior automata* (*BA*) for a bMSC by assigning non-negative and monotonically increasing location indicators for each event appearing in the bMSC. For example, in Fig. 1(a), the bMSC B_1 is in its initial location $[0, 0, 0, 0]$ for each instance. If the event i_1 occurs, then the system enters the location $[1, 0, 0, 0]$.

Definition 3 (*Behavior automata*). The behavior automata for each bMSC is defined as a five-tuple $BA = (Q, \Sigma_{BA}, \delta, q_0, F)$, where Q is a finite set of

states defined in terms of instance's location indicators, $\Sigma_{BA} = E_s$ is a finite set of events, $\delta \subseteq Q \times \Sigma_{BA} \times Q$ is a set of transitions, $q_0 \in Q$ is an initial state, and $F \in Q$ a final state.

Furthermore, the BA needs to be extended to properly handle various in-line expressions. This is accomplished by combining BAs generated separately from the corresponding bMSCs [11]. Fig. 3 represents the combined BA of the example in Fig. 1. As our assumptions on synchronous concatenation and activation conditions, the instance I_3 of bMSC B_4 cannot send the message m_3 before the activation condition, 'when $(n = 2)$', is evaluated. It should be noted that the BA is annotated with path information. For example, the system enters the state '$\langle B_1 \rangle B_3[1,0,0]$' when the value of '$n$' is 1.

Most previous approaches [2,10,12–14] used the BA for the global behavior of the system. While straightforward in concept, this approach quickly becomes impractical due to state explosion problem if several bMSCs are to be combined. Furthermore, this brute-force location encoding in the behavioral model provides little useful information to the testers. That's why we proposed to encode the state by considering the values of state

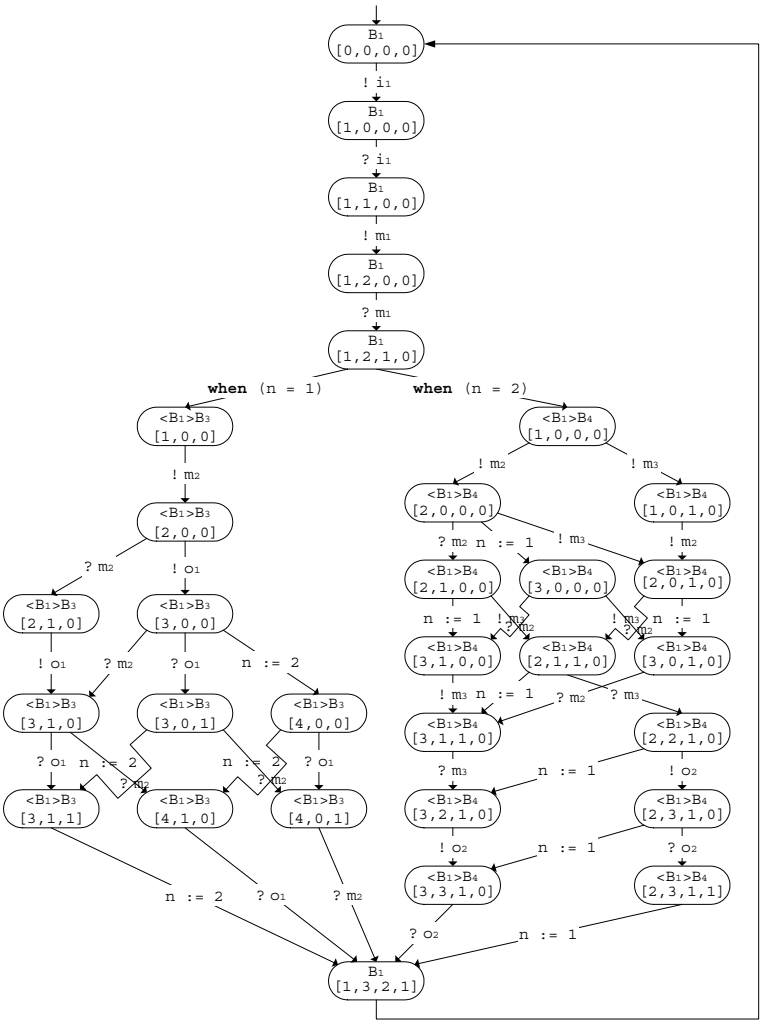

Fig. 3. Behavior automata for bMSCs B_1.

variables. Changes made to the values of the state variable then are shown as transitions.

The basic algorithm to generate a GFSM from a behavior automata corresponding to each bMSC is straightforward in that all the feasible paths and the resulting states are identified recursively. We use state variables and their values such that transitions are created only when the values of state variables are changed. Therefore, a state in the GFSM is defined as the set of values of the system state variables. Likewise, a transition represents an ordered sequence of messages that must be exchanged. It should be noted that multiple transitions are created if there are unordered and potentially concurrent message exchanges.

Definition 4 (*GFSM*). GFSM is a four-tuple $GFSM = (S_{GFSM}, s_{GFSM_0}, \Sigma_{GFSM}, T_{GFSM})$, where S_{GFSM} is a finite set of states, $s_{GFSM_0} \in S_{GFSM}$ is an initial state, $\Sigma_{GFSM} = \langle E_1; E_2; \ldots \rangle, E_i \in E^M$ is a set of labels, and $T_{GFSM} \subset S_{GFSM} \times \Sigma_{GFSM} \times S_{GFSM}$ is a finite set of transitions.

Let A be the domain of all data values, then a state of GFSM is a function from the set of variables to the set of data values, $\sigma : V \to A$. Denote the set of all possible states *States*, $States : 2^{V \to A}$, then the state set of GFSM is a subset of *States*, $S_{GFSM} \subseteq States$. In order to make sure that the number of states of GFSM remain finite, A must be finite in size. Therefore, only assignments of explicit data values are permitted inside 'action' constructs.

When the system is in a state $q \in Q$ of BA, we can find an enabled event set using a satisfy function $\mathrm{SAT} : States \times E \to \{0, 1\}$, where E is the activation conditions described in outgoing transitions of q. If an activation condition is not identical to the current valuation, the satisfy function returns '0', otherwise it returns '1'. Furthermore, if the outgoing transition includes a state change event represented by the 'action' construct, the system's configuration is changed by an assignment function $\mathrm{ASS} : States \times E \to States$. For other events the system configuration is not changed.

Construction of a GFSM is a process of traversing all possible paths in the BA using the valuation of the state variables. First, states and transitions are initialized as the empty sets (line 1 in GFSM Construction Algorithm). The initial state is defined and pushed onto the stack (lines 2–3). For reactive and embedded systems, the initial state of the system is often fixed and known in advance. Therefore, it makes sense to generate a global finite state machine by utilizing the initial state information. As long as all possible paths of the BA have not been previously analyzed (line 4), all feasible event sequences starting from the start state to the final state of the BA are identified (lines 8–24). When traversing the paths meets the final state of the BA, if there are assignments made to the system variable, new states are created and recorded to indicate that the BA must be analyzed later (lines 10–16). The valuation function σ is used to get the current values for each state variable.

For example, if we were to assume that the initial value of '*n*' is equal to 1, Fig. 3 has eight possible paths. Because we are only interested in

GFSM Construction Algorithm

```
1: initialize S_GFSM, T_GFSM and Stack_GFSM to empty set;
2: get an initial state s_GFSM0 from the user;
3: push s_GFSM0 onto Stack_GFSM;
4: while Stack_GFSM ≠ φ{
5:    pop σ from Stack_GFSM;
6:    initialize Stack_BA and Stack_path to empty set;
7:    push q0 and null onto Stack_BA and Stack_path;
8:    while Stack_BA ≠ φ{
9:    pop q and path from Stack_BA and Stack_path;
10:      if q ∈ F then {
11:         if σ ∉ S_GFSM then {
12:            S_GFSM ← S_GFSM ∪ {σ};
13:            push σ onto Stack_GFSM;
14:         }
15:         T_GFSM ← T_GFSM ∪ {path};
16:      }
17:      for all outgoing transitions p ⊆ δ of q do {
18:         extract e ∈ Σ_BA from p;
19:         if SAT(σ, e) then {
20:            push ASS(σ, e) onto Stack_BA;
21:            push path + e onto Stack_path;
22:         }
23:      }
24:   }
25: }
```

identifying the sequence of message exchanges taking place between the changes in the global system states, transitions corresponding to the identical sequences of message exchanges are eliminated when generating a GFSM. The finite state machine ends up preserving only three transitions as shown in Fig. 5.

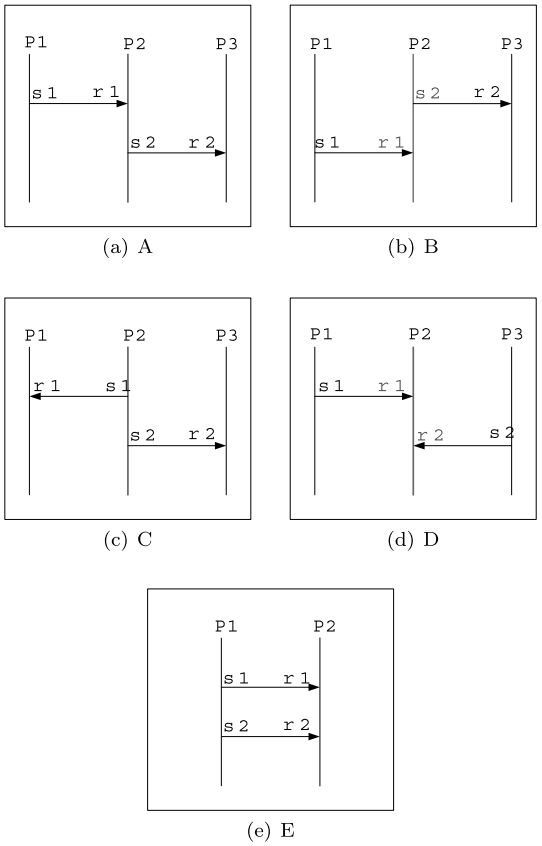

Fig. 4. Five types of message exchange.

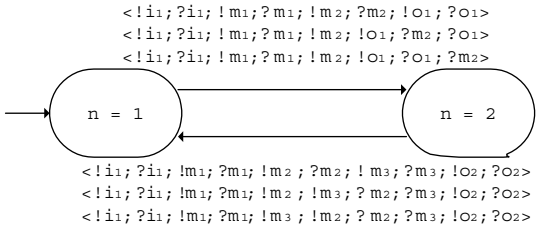

Fig. 5. GFSM for bMSC B_1.

3.2. Synchronous vs asynchronous interpretation

Embedded software typically consists of a set of tasks which communicate and synchronize with each other using shared variables or message exchanges. However, use of shared variables can introduce subtle flaws due to race condition. Besides shared variables, message passing techniques are often used in practice. Message passing can be interpreted in either synchronous or asynchronous manner. In the case of the former, sending and receiving of the message is said to take place at the same time, while the asynchronous interpretation treats sending and receiving of the message as distinct events taking place at different times. When implementing embedded and reactive software, tasks communicate with each other asynchronously using FIFO queue to store messages, and there are five distinct communication patterns (see Fig. 4) [15].

In this figure, the task P2 is to exchange messages with P1 or P3. In the case of Fig. 4(a), as far as the process P2 is concerned, asynchronous and synchronous interpretations result in the same behavior. P2 must wait for the arrival of the first message r1, before it can send a message. In Fig. 4(c), there are three different possibilities, $s1 < r1 < s2 < r2$, $s1 < s2 < r1 < r2$, and $s1 < s2 < r2 < r1$, interpreted asynchronously if we use '<' to denote the ordering relationship in the timeline. If we were to employ an asynchronous interpretation, they all need to be represented as distinct transitions, thus resulting in the transition explosion problem. However, from the viewpoint of the task P2, there is no difference. As far as the behavior of a GFSM is concerned, even if communication among the tasks take place asynchronously using FIFO queues, such communication can be justifiably interpreted synchronously. Likewise, synchronous interpretation of message exchanges do not result in inaccurate analysis.

The two exceptions, the inferred order, is shown in Fig. 4(b) and (d). In case (d), the task P2 waits for the arrival of messages from different tasks. As the exact times when the messages arrive cannot be statically predicted, there are four different possibilities when interpreted asynchronously. They are $s1 < r1 < s2 < r2$, $s1 < s2 < r1 < r2$,

$s2 < s1 < r2 < r1$, and $s2 < r2 < s1 < r1$. [1] On the other hand, the only possible interpretation for synchronous communication is $s1 < r1 < s2 < r2$. Therefore, the asynchronous interpretation must be enforced to accurately model the true behavior of the system. The case (b) is the same with (d). If the bMSCs B_1 is transformed into a finite state machine, there are three transitions, as opposed to six. Fig. 6 represents the resulting GFSM for bMSCs B_1 and B_2 in Fig. 1.

3.3. Coverage criteria for GFSM

The generation of test sequences from finite state machines is a well understood problem, and various techniques exist to generate sequences [16]. The algorithms of these methods (e.g., Wp-method [17] and Unique-Input–Output (UIO) method [18]) are generally too complex and produce long test cases because a checking sequence of each state is necessary for completing the test cases. However, if we have a status message, then the checking sequence can be easily obtained by simply constructing a covering path of the transition diagram of the specification machine.

In case of GFSM, the checking sequence is dispensable because each state is encoded by considering the value of state variables of the target system. Therefore, *state tour* and *transition tour* [2] [19] coverage criteria are sufficient methods in the GFSM. The former requires that each state in the GFSM is visited at least once, and it covers all external input events of the target system. The latter requires that each transition is visited at least once, and it covers all combination of the activation conditions in the specification.

In Fig. 6, we can generate a test sequence, $(n=1)\xrightarrow{i_1;m_1;m_2;o_1}(n=2)\xrightarrow{i_1;m_1;m_2;m_3;o_2}(n=1)$, for the state tour coverage criteria, and this means a sequence of external I/O events, $(i_1/o_1, i_1/o_2)$. For the transition tour coverage criteria, on the other hand, three test sequences, 1: $(n=1)\xrightarrow{i_2;o_3}(n=1)$, 2: $(n=1)\xrightarrow{i_1;m_1;m_2;o_1}(n=2)\xrightarrow{i_2;o_3}(n=2)\xrightarrow{i_1;m_1;m_2;m_3;o_2}(n=1)$, and 3: $(n=1)\xrightarrow{i_1;m_1;m_2;o_1}(n=2)\xrightarrow{i_1;m_1;m_3;m_2;o_2}(n=1)$, are generated.

4. Case study

A digital TV (DTV) developed by Samsung Electronics (SEC) implements the Digital television Application Software Environment (DASE) standards [20,21], defined by Advanced Television Systems Committee (ATSC). The DASE includes software modules that allow decoding and execution of application programs that deliver interactive and data broadcast services. The DTV software consists of Java modules and native applications written in C. The former provides DASE Application Programming Interface (API), and the latter provides the vendor specific functionality such as Web browser and e-mail, etc. Overall structure of the DTV software is shown in Fig. 7. It is quite complex in that it has more than 800 000 lines of code and consists of 17 tasks and threads excluding operating systems, windows libraries, and Java Virtual Machine (VM).

The application manager subsystem accepts data input such as HTML files, Java class files, or other contents and checks whether or not the applications should be launched. It is structured as a Java thread, a native task, and a communication thread between them. It is expected to maintain a list of active applications. The transport stream parsers decode the content of specific protocols, Program and System Information Protocol (PSIP) [21] and Data Broadcast (DB) [20]. The presentation engine is a content decoder for (possibly dynamic) HTML content. This engine is implemented as a native task, and provides application execution services in a uniform and platform-independent manner. It is integrated with the window manager. In the case of Java class files, an Xlet viewer handler for each Java application is

[1] In communication software, messages are exchanged with three steps; (1) a sender puts a message into a sending queue, (2) the network transmits the message and puts it into a receiving queue of a receiver, and (3) the receiver gets the message from the queue. In embedded software, however, it is completed within two steps; (1) a sender puts a message into a queue of a receiver and (2) the receiver gets the message from the queue. Consequently, $s2 < s1 < r1 < r2$ and $s1 < s2 < r2 < r1$ are impossible in embedded software.

[2] A tour means a sequence from/to an initial state, and if the tour reaches the initial state, it is completed.

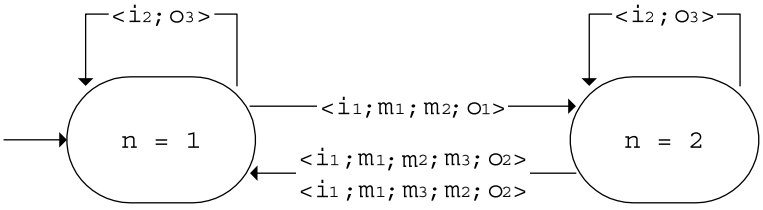

Fig. 6. GFSM for bMSCs B_1 and B_2 applying synchronous interpretation.

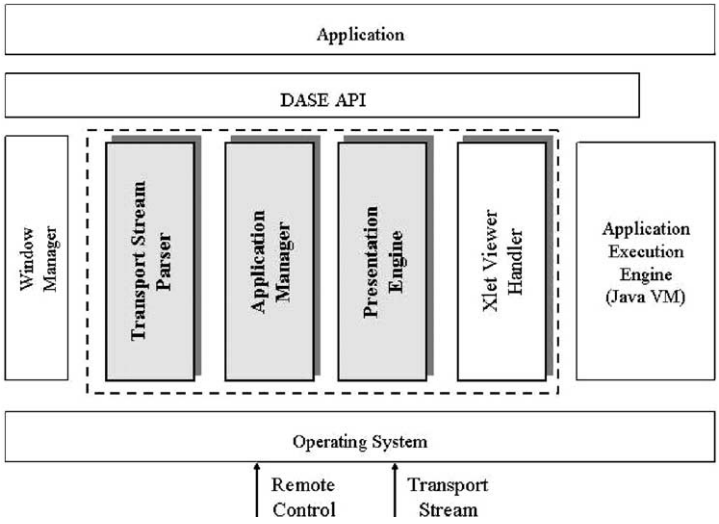

Fig. 7. The module structure of SEC DTV software.

allocated. It is integrated with the application execution engine, Java VM. In this paper, implementation under test (IUT) modules are highlighted as a dotted rectangle in Fig. 7. Other modules, window libraries and DASE API, etc., are excluded because integration testing technique focuses on detecting potential flaws on task interactions and message passing.

At first, a top-level bMSC is specified as shown in Fig. 8. It indicates which MSC is activated when the DTV software receives four events from the remote control unit or an event from the transport stream. In Fig. 8, all external events are connected by the 'alt' in-line.

The detailed interaction sequences for each external input event are described and connected to the top level bMSC. For example, Fig. 9 represents a part of interactions when the external event 'TV_Video' occurs. The DTV software performs such interactions when the system's state should be met with an activation condition. The activation condition means that the number of HTML applications in the DTV software is one ('length = 1').

The 'alt' in-line expression indicates that after exchanging four messages, either 'Start_HTML_Installed' (Fig. 10) or 'Start_HTML_Paused' (Fig. 11) bMSC is selectively activated. The exact conditions under which the bMSC is activated are specified in the referenced bMSC by a guard. In Fig. 10, upon completing the required message exchange, the state of the HTML application is set to 'ACTIVE' using the variable 'getState( )'.

The partial specification used in this example consists of 18 bMSCs. Fig. 12 shows the hierarchy of bMSCs, where two connected bMSCs

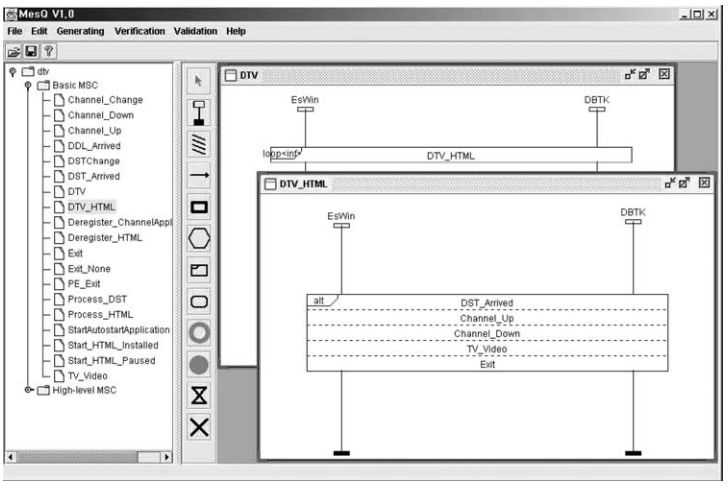

Fig. 8. Top level scenario for HTML applications.

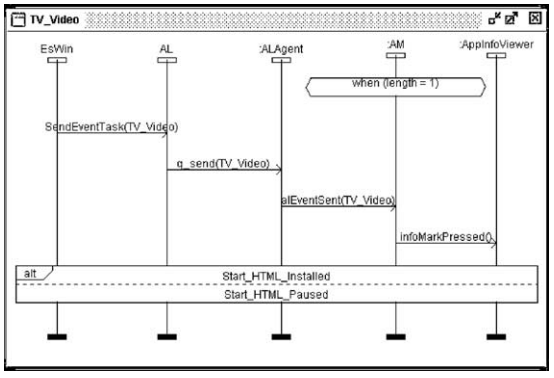

Fig. 9. Execution of HTML applications.

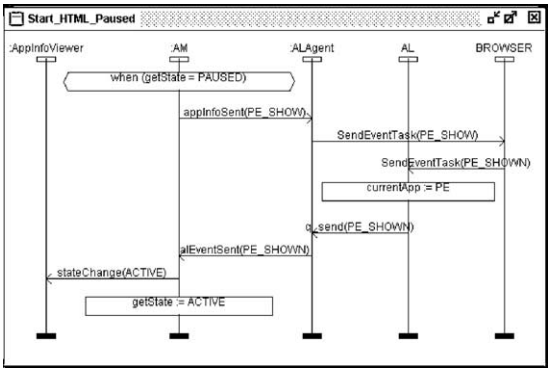

Fig. 11. HTML executions in PAUSED state.

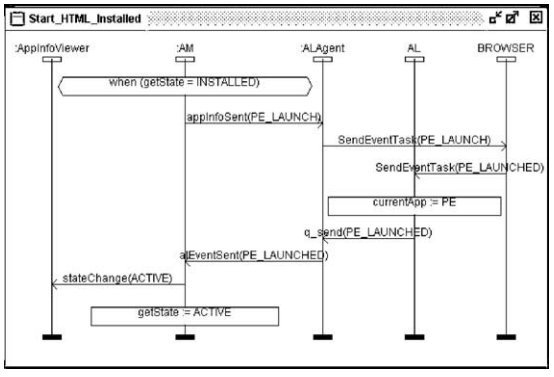

Fig. 10. HTML executions in INSTALLED state.

mean that the above bMSC refers to the other.

In our example, there are six state variables and five input events. Our algorithm to construct GFSM starts from the known and fixed initial state of the system, and generates only the feasible sequence of states and transitions. In this example, our algorithm constructed a GFSM (Fig. 13) which contains eight states and 22 transitions.

Finally, one test sequence, $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_5} s_6 \xrightarrow{t_6} s_7 \xrightarrow{t_8} s_2 \xrightarrow{t_0} s_3 \xrightarrow{t_5} s_4 \xrightarrow{t_6} s_5 \xrightarrow{t_4} s_0$, is generated to satisfy the state tour coverage criteria. For transition tour coverage criteria, seven test sequences are generated as follows:

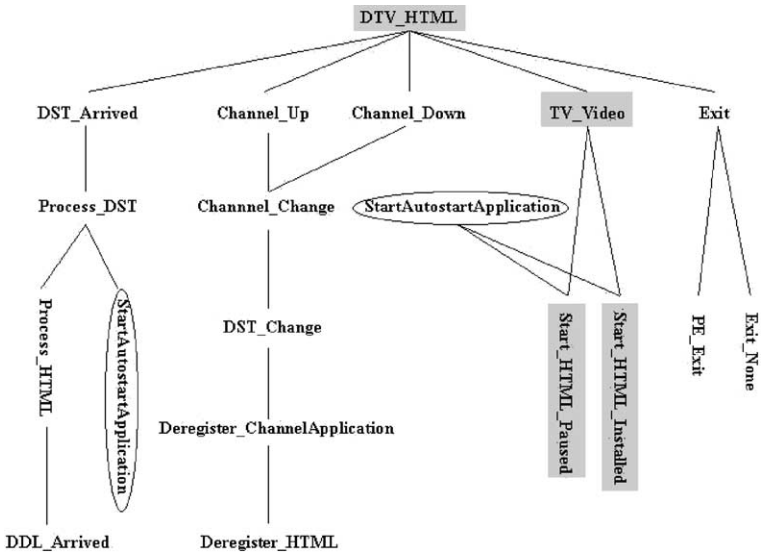

Fig. 12. MSCs structure for HTML applications.

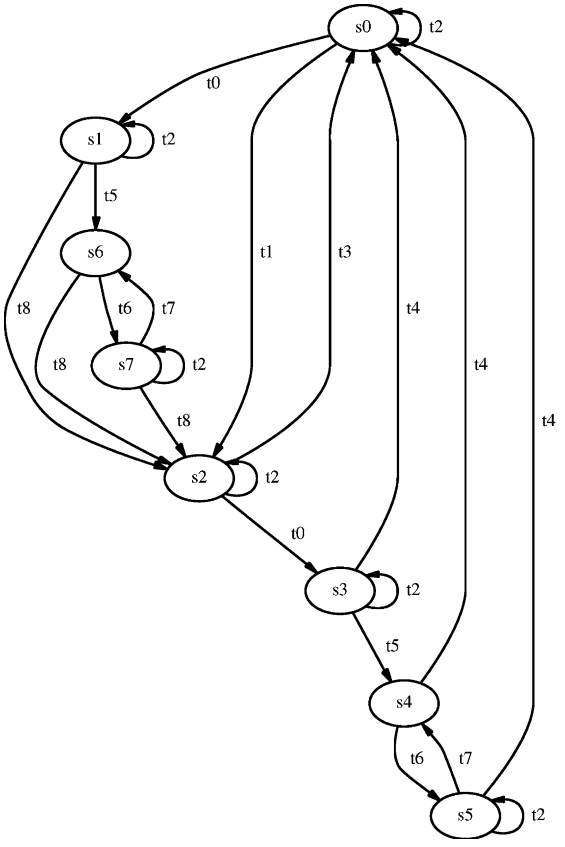

Fig. 13. Resulting GFSM for HTML applications.

(i) $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_2} s_1 \xrightarrow{t_5} s_6 \xrightarrow{t_6} s_7 \xrightarrow{t_2} s_7 \xrightarrow{t_7} s_6 \xrightarrow{t_8} s_2 \xrightarrow{t_0} s_3 \xrightarrow{t_2} s_3 \xrightarrow{t_5} s_4 \xrightarrow{t_6} s_5 \xrightarrow{t_2} s_5 \xrightarrow{t_4} s_0$

(ii) $s_0 \xrightarrow{t_1} s_2 \xrightarrow{t_3} s_0$

(iii) $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_8} s_2 \xrightarrow{t_3} s_0$

(iv) $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_5} s_6 \xrightarrow{t_6} s_7 \xrightarrow{t_8} s_2 \xrightarrow{t_3} s_0$

(v) $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_5} s_6 \xrightarrow{t_6} s_7 \xrightarrow{t_8} s_2 \xrightarrow{t_0} s_3 \xrightarrow{t_4} s_0$

(vi) $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_5} s_6 \xrightarrow{t_6} s_7 \xrightarrow{t_8} s_2 \xrightarrow{t_0} s_3 \xrightarrow{t_5} s_4 \xrightarrow{t_4} s_0$

(vii) $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_5} s_6 \xrightarrow{t_6} s_7 \xrightarrow{t_8} s_2 \xrightarrow{t_0} s_3 \xrightarrow{t_5} s_4 \xrightarrow{t_6} s_5 \xrightarrow{t_7} s_4 \xrightarrow{t_4} s_0$

The test sequence generated by state tour coverage criteria did not uncover flaws in the DTV software design. However, the test sequence based on the transition tour coverage criteria detected a defect that was previously unknown. It had to do with a deadlock situation that may occur when two consecutive requests to change TV channels are received while the first request is still being processed.

Although details are omitted, we have applied the proposed approach to a larger subsystem of the DTV software that consists of 56 bMSCs, 10 remote control unit input events, seven stream data inputs, and nine state variables. There were 17 and 875 test sequences needed to satisfy the state and transition tour coverage criteria, respectively.

5. Conclusion

This paper demonstrated that Message Sequence Charts formalism is useful when describing reactive behavior for large and complex embedded software. We also showed that MSCs can be algorithmically translated into a GFSM. From the GFSM, we can generate test cases automatically according to the state and transition tour coverage criteria. State variables, used to specify various activation conditions of basic MSCs, were used to limit the number of states and transitions included in the GFSM. We have applied the proposed approach to specify a substantial portion of complete requirements for an embedded software running on a digital TV. This case study convincingly demonstrated that the proposed approach is scalable to other industrial systems.

However, as we applied the proposed approach to a large and complex software such as DTV, we found that the proposed technique could be further extended. There are constructs included in the MSCs 2000 standard but not yet supported in the proposed approach. 'par' in-line (parallel) and timing constraints construct are examples. If the 'par' in-line is applied, the size of resulting GFSM would definitely grow, and further reduction techniques to control the degree of the transition explosion problem would be needed.

Acknowledgements

Partially supported by the Advanced Information Technology Research Center (AITrc).

References

[1] ITU-T Recommendation Z.120, Message Sequence Chart (MSC), 2000.
[2] ITU-T Recommendation Z.120, Annex B, Formal Semantics of Message Sequence Charts, 1998.
[3] J. Grabowski, D. Hogrefe, R. Nahm, Test case generation with test purpose specification by MSCs, in: 6th SDL Forum, 1993, pp. 253–266.
[4] M. Schmitt, A. Ek, B. Koch, J. Grabowski, D. Hogrefe, Autolink—putting SDL-based test generation into practice, in: IWTCS'98, Kluwer Academic Publishers, 1998, pp. 227–243.
[5] R.L. Probert, H. Ural, A.W. Williams, Rapid generation of functional tests using MSCs, SDL and TTCN, Computer Communications 24 (3–4) (2001) 374–393.
[6] ITU-T Recommendation Z.100, Specification and Description Language (SDL), 1999.
[7] J. Grabowski, SDL and MSC based test case generation—an overall view of the SAMSTAG method, Technical report, University of Berne, IAM-94-0005, 1994.
[8] I.S. Chung, H.S. Kim, H.S. Bae, B.S. Lee, Testing of concurrent programs based on message sequence charts, in: International Symposium on Parallel and Distributed Software Engineering (PDSE'99), 1999.
[9] D. Harel, Statecharts: a visual formalism for complex systems, Sciences of Computer Programming 8 (1987) 231–274.
[10] R. Alur, M. Yannakakis, Model checking of message sequence charts, in: Proceedings of the Tenth International Conference on Concurrency Theory, 1999, pp. 114–129.
[11] N.H. Lee, T.H. Kim, S.D. Cha, Construction of global finite state machine for testing task interactions written in message sequence charts, in: The Fourteenth International Conference on Software Engineering and Knowledge Engineering (SEKE'02), Ischia, Italy, 2002.
[12] P.B. Ladkin, S. Leue, Interpreting message flow graphs, Formal Aspects of Computing 7 (5) (1995) 473–509.
[13] J. Klose, H. Wittke, An automata based interpretation of live sequence charts, in: TACAS2001, Lecture Notes in Computer Science, Vol. 2031, Springer, Berlin, 2001, pp. 512–527.
[14] B.M. Kim, H.S. Kim, W.Y. Kim, Construction of global state transition graph for verifying specification written in message sequence charts for telecommunications software, IEICE Transactions on Information and Systems E84-D (2) (2001) 249–261.
[15] R. Alur, G.J. Holzmann, D. Peled, An analyzer for message sequence charts, Software Concept and Tools 17 (2) (1996) 70–77.
[16] D. Lee, M. Yannakakis, Principles and methods of testing finite state machines—a survey, Proceedings of the IEEE 84 (8) (1996) 1090–1123.
[17] G. Luo, G. Bochmann, A. Petrenko, Test selection based on communication nondeterministic finite-state machines using a generalized Wp-method, IEEE Transactions on Software Engineering 20 (1994) 149–162.
[18] K.K. Sabnani, A.T. Dahbura, A protocol testing procedure, Computer Networks and ISDN Systems 15 (4) (1988) 285–297.
[19] S. Naito, M. Tsunoyama, Fault detection for sequential machines by transition-tours, in: FTCS (Fault Tolerant Computing Systems), 1981, pp. 238–243.
[20] ATSC T3/S13 Doc. 010, Data Broadcast Specification, Technical report, ATSC, 1999.

[21] ATSC Doc. A/65, Program and System Information Protocol for Terrestrial Broadcast and Cable, Technical report, ATSC, 1997.

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.