Testing of Concurrent Programs After Specification Changes

In Sang Chung
School of Information and Computer Eng.,
Hansung University
insang@hansung.ac.kr

Hyeon Soo Kim
School of Computer and Software Eng.,
Kumoh National University of Technology
hskim@cespc1.kumoh.ac.kr

Hyun Seop Bae, Yong Rae Kwon
Department of Computer Sci.,
Korea Advanced Institute of Science and Technology
{hsbae, yrkwon}@salmosa.kaist.ac.kr

and Dong Gil Lee
Electronics and Telecommunications Research Institute
dglee@etri.re.kr

Abstract

This paper describes a specification-based regression testing technique that can be applied for revalidating concurrent programs after specification changes. This kind of regression testing technique requires sequencing constraints which specify the precedence relations among synchronization events. In our method, the sequencing constraints are extracted automatically from Message Sequence Charts(MSCs) that are considered to be partial and nondeterministic specifications. We show how to identify the sequencing constraints affected by the modifications of a specification rather than creating new sequencing constraints from scratch to reduce the cost of regression testing. We also describe how to determine whether the affected sequencing constraints are satisfied by the program being tested.

1. Introduction

The primary concern of regression testing is to illustrate that changes introduced in a program are correct and do not adversely affect the unchanged portions of the program. During the process of regression testing, all previously developed test cases may be deployed for revalidating a modified program. The revalidation methods of this form tend to consume large amounts of time and computing resources. There has been a significant amount of research on the design of effective regression testing techniques to reduce the cost of regression testing[10]. Many of such techniques have focused on regression testing of sequential programs.

Furthermore, most prior work on regression testing involves code-based techniques that select test cases using the code of the original and modified programs[10]. Consequently, little support has been provided for coping with specification changes. In practice, frequent modifications are made to specifications for various reasons: to correct the errors in specifications, to enhance or change functionalities of the corresponding programs. A program is modified to accommodate changes in its specification, leading to retesting the modified program. For effective fault detection, information on specification changes should be also considered in selecting test cases. It has been shown that code-based testing and specification-based testing complement each other[3]. This is because specification-based testing is based on valid behaviors that should be exhibited during executions of a program, while code-based testing depends on feasible behaviors exercised by program execution.

This paper presents a specification-based regression testing technique for concurrent programs using information on specification changes. Specification-based regression testing of concurrent programs investigates whether a (modified) program conforms to its (changed) specification. Specifications for concurrent programs include the sequencing constraints among synchronization events that have to be satisfied during program execution. There has been research on testing concurrent programs using sequencing constraints[3, 4]. For example, Tai et al. proposed a testing methodology based on the use of CSPE(Constraints on Succeeding and Preceding Events) that specifies restrictions on the event sequences of a concurrent program[3]. This testing method uses sequencing constraints, given by a user or automatically derived from formal specifications, and then generates event sequences as test cases according to the sequencing constraints. However, they did not address the problem of re-testing a (modified) program after changes are made to its specification.

Changes in specifications require us to create new sequencing constraints from scratch, which can be expensive in terms of human and machine effort. One approach to reducing the cost of regression testing is to identify sequencing constraints affected by specification changes and to perform testing on the modified program only for the affected sequencing constraints. This approach, called incremental regression testing, can avoid the costly construction of new sequencing constraints.

In order to illustrate our ideas, we will use Message Sequence Charts(MSCs) with partial and nondeterministic (specification) semantics. MSC is a popular and good means of describing the selected execution behavior of concurrent/distributed programs. Compared to other formal specification techniques, MSC is a relatively easy way to show interactions among tasks. The testing technique we will present in this paper regards MSC specifications as a suite of sequencing constraints to restrict the execution behaviors of a concurrent program. Based on this consideration, we present some techniques for impact analysis that allow us to identify which sequencing constraints have been affected by the modification.

2. Brief Overview of Message Sequence Charts

MSC provides a graphical means of visualizing selected system runs in distributed systems such as telecommunications systems. It has been standardized by the ITU(International Telecommunication Union) which maintains MSC recommendation Z.120[12]. Even before the approval of the MSC recommendation, it has used for a long time to specify system requirements, often in combination with SDL. One major reason for popularity of MSCs lies in their clear graphical layout that gives an intuitive understanding of partial system behavior.

MSCs mainly concentrate on showing the interchange of messages among system components and their environment. In this paper, we assume that system components refer to tasks, while they can be modeled by various constructs such as services, processes, and blocks in SDL. The main features of MSCs can be illustrated in terms of basic language elements and structural language elements. The basic language elements of MSCs include the constructs that are necessary to specify communications and local actions, whereas the structural elements include constructs that can be used to show the hierarchical refinement relations among MSCs.

Figure 1[1] shows an example of MSC usage. This example describes a part of a phone conversation scenario using most of the basic primitives of standard MSC. There are four task instances: *caller*, *line_a*, *trunk* and *line_b*, each of which is depicted as a vertical line. A horizontal arrow represents a communication via passing a message between two task instances.

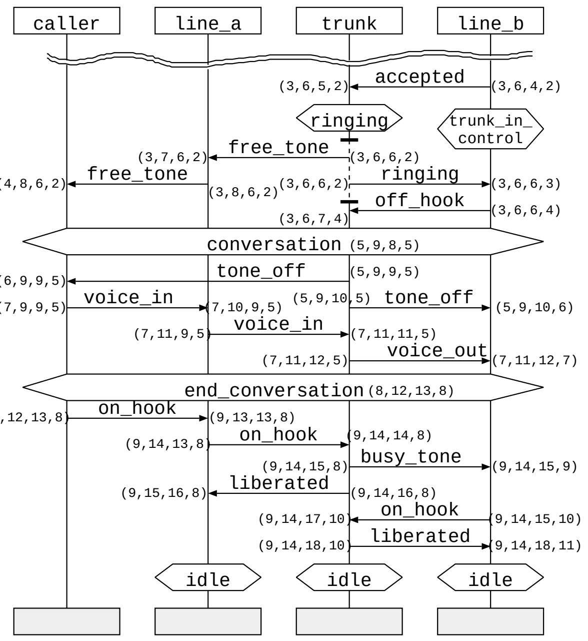

Figure 1. A Message Sequence Chart for Phone Conversation

For example, the task *line_a* passes a *free_tone* message to a *caller* task. A hexagon crossing over a task instance represents a condition to be satisfied by the instance during execution. For example, the task *trunk* must be in the *ringing* state after it received an *accepted* message from the task *line_b* in the phone conversation scenario. Shared conditions are represented by hexagons crossing over two or more tasks such as *conversation* and *end_conversation*. Finally, there is a coregion that is depicted by a dotted vertical line. Events in a coregion are not ordered while events in a task instance are totally ordered along the vertical line. For example, a task *caller* cannot send a *voice_in* message until receiving a *tone_off* message, while a task *trunk* can send a *free_tone* message before sending a *ringing* message and vice versa.

[1]Note that the MSC is annotated with integer vectors. These vectors contribute to determining the causal relations among events in MSCs. How to annotate MSCs with integer vectors and how to catch the causal relations using the vectors are explained in the following section.

3. Constraint-based Regression Testing

In this section, we present a regression testing methodology that uses sequencing constraints between events. The methodology consists of four steps: constraints elicitation, impact analysis, nondeterministic testing, and deterministic testing.

- **Constraints elicitation:** Events in MSC specifications are ordered by logical time stamps. By comparing the time stamps of two events, we can determine the sequencing constraints between them. Such sequencing constraints are used as validity constraints that an implementation has to satisfy.
- **Impact Analysis:** In specification-based regression testing, impact analysis is required to identify the affected constraints after a change is made to a specification. We can reduce the cost of regression testing by re-testing a concurrent program against only the affected constraints rather than testing all constraints.
- **Nondeterministic testing:** Nondeterministic testing executes a concurrent program P with a given input X many times so that as many (distinct) event sequences as possible can be exercised. During repeated nondeterministic execution of P, the event sequences exercised by P are collected and then scrutinized to determine whether they satisfy the sequencing constraints obtained during impact analysis.
- **Deterministic testing:** The deterministic testing technique forces a program to follow a given event sequence with a given input data[9, 11]. This technique is useful for testing the sequential constraints that nondeterministic testing did not cover.

3.1. Constraints Elicitation

We adapt Fidge's vector time stamping method[7] and attach a logical time stamp to each event in the MSC specification to detect the precedence relations among events. In this paper, we assume that the readers are familiar with Fidge's method. Constraints elicitation step can be explained using five rules according to the semantics of MSC constructs. Figure 1 shows the results of applying these rules to the phone conversation example.

- **Rule 1 : Initialization Rule**
  Initially, each task is given zero-time. That is, $T_i = (0, \cdots, 0)$, where T_i denotes the current time of task p_i. The dimension of T_i is the same as the number of tasks included in the given MSC specification. In Figure 1, there are four tasks in the specification and all events have four dimensional time stamps.
- **Rule 2 : Inheritance Rule**
  Basically, the time stamp of each event is inherited from that of the preceding event. This rule depends on whether events are located on a coregion or not. Let T_{ij} denote the time stamp attached to the j-th event of task p_i.
  - **Rule 2-1 : Normal Inheritance**
    Whenever the j-th and the $(j+1)$-th events are not in a coregion, $T_{ij+1} = T_{ij}$ + i-th *unit vector*. In other words, each event not included in a coregion inherits its time stamp from that of the event just preceding with one increment of the i-th column.
  - **Rule 2-2 : Coregion Inheritance**
    Let $j+1$-th, . . ., m-th events of the task p_i are in a coregion. If event j is the latest event which is not on the coregion, then for $\forall x \in \{j+1, \ldots, m\}$, T_{ix} = T_{ij} + i-th *unit vector*. That is, all events in a coregion have the same time stamp inherited from that of the event preceding coregion with one increment of the i-th column. Consider the coregion in the *trunk* task of Figure 1. Two sending events *trunk-send(free_tone)* and *trunk-send(ringing)* have the same time stamp (3,6,6,2) which is inherited from the *trunk-recv(accepted)* event.
- **Rule 3 : Communication Rule**
  Whenever two tasks communicate with each other, there is an implicit precedence relationship.
  - **Rule 3-1 : Sending Rule**
    If the j-th event of task p_i is a sending event, the time stamp T_{ij} is carried to the receiver task after applying Rule 2.
  - **Rule 3-2 : Receiving Rule**
    If the n-th event of task p_m is a receiving event, T_{mn} takes the pairwise maximum between its own time stamp evaluated by Rule 2 and the time stamp carried from the sending event.

  For example, let's consider the time stamp (3,6,6,3) of the *line_b-recv(ringing)* event. This is computed by PAIRWISE_MAX$\{$(3,6,4,3),(3,6,6,2)$\}$ in Figure 1, where the time stamp (3,6,4,3) is inherited from the *line_b-send(accepted)* while (3,6,6,2) is carried from the *trunk-send(ringing)*.
- **Rule 4 : Shared Condition Rule**
  All tasks $(p_j, \cdots, p_l)$ sharing a condition maximize their time stamps using every other involved task. That is, for $\forall x \in \{j, \ldots, l\}$, T_{xc} = PAIRWISE_MAX$\{T_j, \cdots, T_l\}$, where T_{xc} is the time stamp of the shared condition event. For example, the shared

condition *conversation* has a time stamp (5,9,8,5) which is calculated by PAIRWISE_MAX{(5,8,6,2), (3,9,6,2), (3,6,8,4), (3,6,6,5)}.

- **Rule 5 : Creation Rule**
  Whenever task p_i creates task p_j, p_j takes the current time from p_i. That is, $T_j = T_i$.

Figure 1 includes the results of attaching vector time stamps using these rules. After attaching the time stamp, we can determine the precedence between two events by comparing their time stamps according to the following rule.

- **Comparison Rule**
  For two given events a and b included in the tasks p_i and p_j, respectively, the precedence relation between them can be decided as follows:
  $a \rightarrow b \Leftrightarrow T_{ia}(i) \leq T_{jb}(i) \wedge T_{ia}(j) < T_{jb}(j)$
  where T(k) means the k-th column of the time stamp T.

3.2. Events Precedence Graph

Using the time stamping rules and the comparison rule, we can determine the precedence relations among events in MSCs. The precedence relations included in a given MSC M is denoted by a partially ordered set, shortly poset, (E_M, $\rightarrow_M$) where E_M is the set of events in M and $\rightarrow_M$ is an infix notation for the precedence relation. Each event in E_M is identified by the task which it is associated with, the type of action such as $send$ or $recv$, and the name of message. For example, the events "task t_1 sends a message m_1" and "task t_2 receives a message m_2" are denoted by "$t_1 : send(m_1)$" and "$t_2 : recv(m_2)$", respectively. Events involved with (shared) conditions are used only for establishing precedence relations between communication events, but are not included in E_M.

A poset (E_M, $\rightarrow_M$) can be represented as a directed acyclic graph $G_M = (E_M, A_M)$, where A_M is a set of edges such that $A_M = \{(u,v) | u, v \in E_M, u \rightarrow_M v\}$. G_M will be referred to as the event precedence graph or EPG in this paper. The edge (u,v) in A_M means a direct precedence relation between two events u and v, but transitive precedence relations are not explicitly shown in G_M. For each edge $(u,v) \in A_M$, we say that u is a direct predecessor of v and v is a direct successor of u. PRED(v) denotes the set of all direct predecessors of event v, and SUC(v) the set of all direct successors of event v. For example, Figure 2 shows the EPG corresponding to the MSC of Figure 1. Actually, each node must be labeled with an event name such as "$caller : recv(free\_tone)$". For the simplicity, however, we use the edge label corresponding the message name and give a number to each node. Thus, the start node with edge label m stands for the event "$t_i : send(m)$" and the end node represents the event "$t_j : recv(m)$". For example, a node labeled number 9 stands for the event "$line\_a : send(free\_tone)$" and a node labeled 10 represents the event "$caller : recv(free\_tone)$". In this EPG, PRED(8)={3,5,7} and SUC(3)={4,8}.

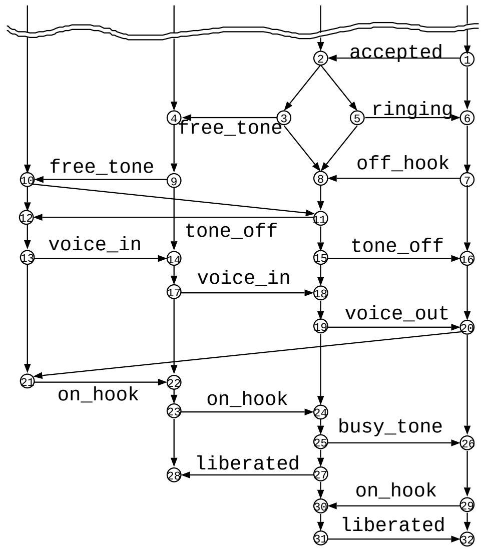

Figure 2. An Event Precedence Graph for the MSC of Figure 1

3.3. Impact Analysis for Primitive Modifications

This section describes an impact analysis technique to identify the affected sequencing constraints using G_M after a change is made to an MSC M. In order to facilitate the presentation, we need following notations for the given G_M.

- B(G_M/V) : set of all predecessors of the events in V.
  B(G_M/V) = $\{w \mid \text{for } v \in V, w \in E_M \wedge w \rightarrow^{\star}_M v\}$
- F(G_M/V) : set of all successors of the events in V.
  F(G_M/V) = $\{w \mid \text{for } v \in V, w \in E_M \wedge v \rightarrow^{\star}_M w\}$

In this paper, a sequencing constraint is denoted by an event pair (a,b) such that $a, b \in E_M$ and is used to specify restrictions over the execution orders of events a and b that have to be satisfied during program execution. Let the original MSC be M and the modified MSC M'. Then, we say that a sequencing constraint or an event pair (a,b) such that $a, b \in E_M \cap E_{M'}$ is *affected* if a modification causes:

(A1) $a \not\rightarrow_M b \wedge b \not\rightarrow_M a \wedge a \rightarrow_{M'} b$

(A2) $a \rightarrow_M b \wedge a \not\rightarrow_{M'} b \wedge b \not\rightarrow_{M'} a$

(A1) means that two concurrent events a and b are sequentialized by the modification. Similarly, **(A2)** means that two sequential events a and b become concurrent after the modification. As usual, two primitive actions are taken to modify an MSC M: adding a message to or deleting a message from M. First, we consider the case in which a message is added to M. Adding a message m can be interpreted as an operation of creating a new edge e such that $e = (t_i : send(m), t_j : recv(m))$ in the corresponding G_M. This provides the precedence relation between the events "$t_i : send(m)$" and "$t_j : recv(m)$": $t_i : send(m) \rightarrow_M t_j : recv(m)$. Using the fact that the relation is transitive, it is observed that the newly created edge e establishes the precedence relation between two sets of events: the predecessors of $t_i : send(m)$ and the successors of $t_j : recv(m)$. This indicates that all sequencing constraints affected by adding a message m are of the form **(A1)** above and can be obtained by the cartesian product of the sets "B$(G_M/PRED(t_i : send(m)))$" and "F$(G_M/SUC(t_j : recv(m)))$" which gives us the universe of sequencing constraints of interest.

Let us denote "B$(G_M/PRED(t_i : send(m)))$" by $\Delta_{B_m}[M]$ and "F$(G_M/SUC(t_j : recv(m)))$" by $\Delta_{F_m}[M]$. In addition, the cartesian product of those two sets is denoted by $\Delta_m[M]$. That is $\Delta_m[M] = \Delta_{B_m}[M] \times \Delta_{F_m}[M]$ Note that Δ_m does not contain the newly added events "$t_i : send(m)$" and "$t_j : recv(m)$". All event pairs of Δ_m are not necessarily considered to be affected because there might be cases where no new precedence relations are introduced, but still maintain the relations established before adding the message m to the MSC M. Such cases might be encountered when there exist other paths from events in $\Delta_{B_m}[M]$ to events in $\Delta_{F_m}[M]$ in G_M. This indicates that for $(a, b) \in \Delta_m[M]$, the relation, $a \rightarrow_M b$, that previously hold before adding the message m, remains unchanged even after the modification. In order to formulate the problem of finding affected constraints precisely, we shall use graph-theoretic techniques. The problem can be rephrased as follows.

> How many edges must be removed from the EPG G_M in order to disconnect those two subgraphs that are induced by $\Delta_{B_m}[M]$ and $\Delta_{F_m}[M]$, respectively?

This question can be answered by finding a *cutset* whose removal disconnects two subgraphs, but no proper subset of which disconnects them. It is well-known fact that more than one cutset may exist. However, for our purpose, we consider only a cutset that contains $e = (t_i : send(m), t_j : recv(m))$ as an its element. After finding a cutset C that disconnects those subgraphs that are induced by $\Delta_{B_m}[M]$ and $\Delta_{F_m}[M]$ with respect to G_M, we have to check whether C contains only one edge e, that is, e is a *bridge*. If e appears to be a bridge, no paths from events in $\Delta_{B_m}[M]$ to events in $\Delta_{F_m}[M]$ exist except the paths containing the edge e. In this case, all event pairs of $\Delta_m[M]$, which have been considered to be concurrent, become related by the precedence relation after the modification. Otherwise, it is clear that some events in $\Delta_{B_m}[M]$ can reach some events in $\Delta_{F_m}[M]$ via the edges in the set $C \Leftrightarrow \{e\}$, say C_e. This means that there exist unaffected sequencing constraints in Δ_m even after the message m is added.

Locating unaffected sequencing constraints can be easily done by considering the transitive closure of the events that are incident to the edges in C_e. Suppose that C_e contains the edges (e_{s_1}, e_{t_1}), $\cdots$, (e_{s_p}, e_{t_p}). Then, we can obtain the unaffected sequencing constraints by computing $\bigcup_{i=1}^{p}$ B$(G_M/\{e_{s_i}\}) \times$ F$(G_M/\{e_{t_i}\})$ which will be referred to as $\Delta_{U_m}[M]$. All event pairs in $\Delta_{U_m}[M]$ preserve their precedence relations even after the modification. That is, if $a \rightarrow_M b \in \Delta_m[M]$ before the modification and $a \rightarrow_M b$ still holds after the modification, then $(a, b) \in \Delta_{U_m}[M]$. As a result, the difference of Δ_m and Δ_{U_m}, denoted by $\Delta_{A_m}[M]$, is the desired set that contains only the event pairs or the sequencing constraints affected by adding the message m to the MSC M. That is, $\Delta_{A_m}[M] = \Delta_m[M] \Leftrightarrow \Delta_{U_m}[M]$

For example, consider the case that a message m from a task *line_b* to a task *line_a* is added to the position underneath the message *tone_off* and over the message *voice_in* in the MSC of Figure 1. After adding the message the EPG contains additional nodes representing $line\_b : send(m)$ and $line\_a : recv(m)$, and an edge $(line\_b : send(m), line\_a : recv(m))$. The node representing $line\_b : send(m)$ is located below node 16 and above node 20 and the node representing $line\_a : recv(m)$ is below node 14 and above node 17 in the EPG. $\Delta_m[M]$ for this case is computed as $\Delta_m[M] = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 15, 16\} \times \{17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32\}$. And the *cutset* C_e has the edges (16, 20), (15, 18), (11, 17). Then $\Delta_{U_m}[M] =$ B$(G_M/\{16\}) \times$ F$(G_M/\{20\}) \cup$ B$(G_M/\{15\}) \times$ F$(G_M/\{18\}) \cup$ B$(G_M/\{11\}) \times$ F$(G_M/\{17\})$. Therefore $\Delta_{A_m}[M] = \Delta_m[M] \Leftrightarrow \Delta_{U_m}[M] = \{(15, 17), (16, 17), (16, 18), (16, 19)\}$.

As discussed above, the set $\Delta_{A_m}[M]$ does not consider the precedence relations arisen due to the events that are newly added: $t_i : send(m)$ and $t_j : recv(m)$. Let $\Delta_{NA_m}[M]$ be the set of newly created precedence relations after adding message m to MSC M. Then, we can construct $\Delta_{NA_m}[M]$ as follows:

$$
\begin{aligned}
\Delta_{NA_m}[M] \quad = \quad & \{t_i : send(m), t_j : recv(m)\} \\
\cup \quad & \{(a, t_i : send(m)) | a \in \Delta_{B_m}[M]\} \\
\cup \quad & \{(a, t_j : recv(m)) | a \in \Delta_{B_m}[M]\} \\
\cup \quad & \{(t_j : recv(m), a) | a \in \Delta_{F_m}[M]\} \\
\cup \quad & \{(t_i : send(m), a) | a \in \Delta_{F_m}[M]\}
\end{aligned}
$$

For example, $\Delta_{NA_m}[M]$ for the case above is {*(line_b:send(m), line_a:recv(m)), (16, line_b:send(m)), (line_a:recv(m), 17), (line_b:send(m), 20), (14, line_a:recv(m))*}. Also, $\Delta_{ND_m}[M]$ described below is $\{(16, 20), (14, 17)\}$.

On the contrary, when a message m is deleted from the MSC M, no new precedence relations become established, but may remove existing relations that have arisen due to the message m. Consider the situation in which the precedence relation, represented by an edge "($t_i : send(m)$, $t_j : recv(m)$)" in the corresponding G_M, is removed. As a result, some events that have been related by the precedence relation may be concurrent as depicted in **(A2)** above. As in the case of adding a message, we can compute the sequencing constraints affected by deleting the message m as follows: $\Delta_{D_m}[M] = \Delta_m[M] \Leftrightarrow \Delta_{U_m}[M]$

Although computation of $\Delta_{A_m}[M]$ and $\Delta_{D_m}[M]$ is done in the same way, $\Delta_{D_m}[M]$ is composed of the event pairs each being concurrent: we know that $\Delta_{A_m}[M]$ is composed of the event pairs each being related by the precedence relation. Of course, such event pairs have been previously related by the precedence relation, but deleting the message m leads them to be converted to being concurrent.

For example, consider the situation in which an edge $(trunk : send(busy\_tone), line\_b : recv(busy\_tone)$" in the G_m is removed. For this case $\Delta_m[M]$ and $\Delta_{U_m}[M]$ are computed as the same way of adding a message m, too. Here $\Delta_m[M] = \{1, \ldots, 24\} \times \{29, \ldots, 32\}$. Cutset has (20, 29), (24, 30). Then $\Delta_{U_m}[M] = \mathrm{B}(G_M/\{20\}) \times \mathrm{F}(G_M/\{29\}) \cup \mathrm{B}(G_M/\{24\}) \times \mathrm{F}(G_M/\{30\})$. Therefore $\Delta_{D_m}[M] = \{(21, 29), (22, 29), (23, 29), (24, 29)\}$.

We can also define the set, denoted by $\Delta_{ND_m}[M]$, of event pairs that are no more any valid precedence relations due to the removal of message m. Since deleting message m removes its associated events, $t_i : send(m)$ and $t_j : recv(m)$ in the corresponding G_M, the precedence relations associated with those two events are exactly the same as $\Delta_{NA_m}[M]$.

Although both of sets $\Delta_{NA_m}[M]$ and $\Delta_{ND_m}[M]$ are identical, their meanings are different. Each event pair of $\Delta_{NA_m}[M]$ represents a newly created precedence relation that exists only in the modified MSC, whereas each event pair of $\Delta_{ND_m}[M]$ represents a precedence relation that exists in the original MSC, but does not exist any more in the modified MSC. Thus, we can characterize sets $\Delta_{NA_m}[M]$ and $\Delta_{ND_m}[M]$ formally as follows:

(A3) if $(a, b) \in \Delta_{NA}[M']$ then $a \notin E(M)$ or $b \notin E(M)$, but $a \rightarrow_{M'} b$.

(A4) if $(a, b) \in \Delta_{ND}[M']$ then either $a \rightarrow_M b$ or $b \rightarrow_M a$, but $a \notin E(M')$ or $b \notin E(M')$.

3.4. Impact Analysis for Complex Modifications

To be practical, we have to consider the effect of complex modifications on MSCs. A complex modification is a sequence of primitive modifications made by either inserting or deleting messages. In the previous section, we described how to identify the sequencing constraints that must be retested after a primitive modification is made to an MSC specification. It is worth noting that although a complex modification may be translated into a sequence of primitive modifications, testing is not performed after each primitive modification but rather it is performed once per complex modification. Moreover, later primitive modifications may invalidate the effects of previous primitive modifications during a complex modification. Therefore, we suggest how to compose the sequencing constraints affected by primitive modifications and determine the constraints which have to be re-tested after a complex modification.

There are two kinds of primitive modifications as described in the previous section: adding a message m denoted by α_m and deleting a message m denoted by δ_m. In the sequel, we will assume the following complex modification which translates the original MSC M into the final MSC $M^\star$ through n primitive modifications. In the modification sequence, M_i $(0 < i \leq n)$ denotes the modified MSC after applying a single primitive modification γ_{m_i} to M_{i-1} where $\gamma_{m_i} \in \{\alpha_{m_i}, \delta_{m_i}\}$.

$$M = M_0 \overset{\gamma_{m_1}}{\rightarrow} M_1 \overset{\gamma_{m_2}}{\rightarrow} \ldots \overset{\gamma_{m_n}}{\rightarrow} M_n = M^\star$$

In order to figure out the effects of a complex modification, consider a complex modification where $n = 2, \gamma_{m_1} = \alpha_{m_1}, \gamma_{m_2} = \delta_{m_2}$. We first consider the relationship between $\Delta_{A_{m_1}}[M_0]$ and $\Delta_{D_{m_2}}[M_1]$. The first primitive modification α_{m_1} may introduce precedence relations between some of the events in M_0 that have been concurrent before the modification. On the other hand, the later primitive modification δ_{m_2} plays a role in invalidating some of the precedence relations that have been introduced due to α_{m_1}. For the same reason, $\Delta_{A_{m_2}}[M_1]$ can be used to invalidate some of the previously identified sequencing constraints, $\Delta_{D_{m_1}}[M_0]$, in a way that each event pair of $\Delta_{D_{m_1}}[M_0]$ representing two concurrent events is forced to be related by the precedence relation. We can also find that the same situations exist between $\Delta_{NA_{m_1}}[M_0]$(or $\Delta_{ND_{m_2}}[M_1]$) and $\Delta_{ND_{m_2}}[M_1]$(or $\Delta_{NA_{m_1}}[M_0]$).

Based on this consideration, we can define the sets of affected sequencing constraints by a complex modification in the inductive manner. Here, $\Delta_A[M_i]$ and $\Delta_D[M_i]$ include the affected constraints which satisfy **(A1)** and **(A2)** after i-th primitive modification, respectively. In the following equations, $\lfloor S \rfloor_M = \{(a, b) | (a, b) \in S, a, b \in E(M)\}$.

$$(P1)\ \Delta_A[M_i] \;=\; \Delta_D[M_i] = \emptyset,$$

$$\text{if} \quad i = 0$$

$$\begin{aligned}(P2)\ \Delta_A[M_i] &= \lfloor(\Delta_A[M_{i-1}] \Leftrightarrow \Delta_{D_{m_i}}[M_{i-1}]) \\ &\cup \ (\Delta_{A_{m_i}}[M_{i-1}] \Leftrightarrow \Delta_D[M_{i-1}])\rfloor_{M_0} \\ \Delta_D[M_i] &= \lfloor\lfloor(\Delta_D[M_{i-1}] \Leftrightarrow \Delta_{A_{m_i}}[M_{i-1}]) \\ &\cup \ (\Delta_{D_{m_i}}[M_{i-1}] \Leftrightarrow \Delta_A[M_{i-1}])\rfloor_{M_i}\rfloor_{M_0} \\ &\text{if} \quad 1 \leq i \leq n\end{aligned}$$

We can easily observe that computation of $\Delta_A[M_i]$ and $\Delta_D[M_i]$ is order-independent. By order-independent we mean that the computation always produces the same results for any modification sequences if they could transform M_0 into M_n. Based on this fact, we can reduce the computation steps by transforming the original modification sequence into another but smaller sequence that also produces the same MSC.

$\Delta_A[M_i]$ is the extension of $\Delta_{A_m}[M]$ in the sense that they include the events pairs which are converted from concurrent relation to sequential relation by modification. Similarly, $\Delta_D[M_i]$ and $\Delta_{D_m}[M]$ include commonly the events pairs which become concurrent after modification. It means that $\Delta_A[M_i]$ and $\Delta_D[M_i]$ reference only the events included in the original MSC M but not new events introduced by modifications. The flooring operator $\lfloor\ \rfloor_{M_0}$ used in (P1) and (P2) ensures this property.

In the previous section, we defined $\Delta_{NA_m}[M]$ in order to compute the sequencing constraints which are associated with the new events introduced by a primitive modification. We also defined $\Delta_{ND_m}[M]$ to detect the sequencing constraints which are invalidated by eliminating some events from MSC by a primitive modification. Similarly, we need two sets $\Delta_{NA}[M_i]$ and $\Delta_{ND}[M_i]$ which include sequencing constraints associated with new events or eliminated events by complex modification. They can be computed as follows:

$$\begin{aligned}(P3)\ \Delta_{NA}[M_i] &= \Delta_{ND}[M_i] = \emptyset, \\ &\text{if} \quad i = 0 \\ (P4)\ \Delta_{NA}[M_i] &= (\Delta_{NA}[M_{i-1}] \Leftrightarrow \Delta_{ND_{m_i}}[M_{i-1}]) \\ &\cup \ (\Delta_{NA_{m_i}}[M_{i-1}] \Leftrightarrow \Delta_{ND}[M_{i-1}]) \\ \Delta_{ND}[M_i] &= (\Delta_{ND}[M_{i-1}] \Leftrightarrow \Delta_{NA_{m_i}}[M_{i-1}]) \\ &\cup \ (\Delta_{ND_{m_i}}[M_{i-1}] \Leftrightarrow \Delta_{NA}[M_{i-1}]) \\ &\text{if} \quad 1 \leq i \leq n\end{aligned}$$

Finally, we have to consider a situation where the same message is inserted and deleted in turn or vice versa at the same position. In this situation, we should not have any effects of two primitive modifications. Some spurious events pairs, however, may be introduced when other primitive modifications interpose between the insert and the delete operations. Thus it must be removed from $\Delta_A[M^\star]$. In the case of $\Delta_D[M^\star]$ such problems do not happen because of the flooring operations. Final results of $\Delta_A[M^\star]$ and $\Delta_D[M^\star]$ are defined respectively as follows:

$$\Delta_A[M^\star] = \Delta_A[M_i] \Leftrightarrow X$$

$$\Delta_D[M^\star] = \Delta_D[M_i]$$

where a set X is defined such that $\{(a,b)|(a,b) \in \Delta_A[M_i]a \ \rightarrow^*_{M_i} \ x, \ y \ \rightarrow^*_{M_i} \ b \ for(x,y) \ \in \ S \ = \bigcup_{k=1}^{n} \Delta_{NA}[M_k] \cap \bigcup_{k=1}^{n} \Delta_{ND}[M_k]\}$.

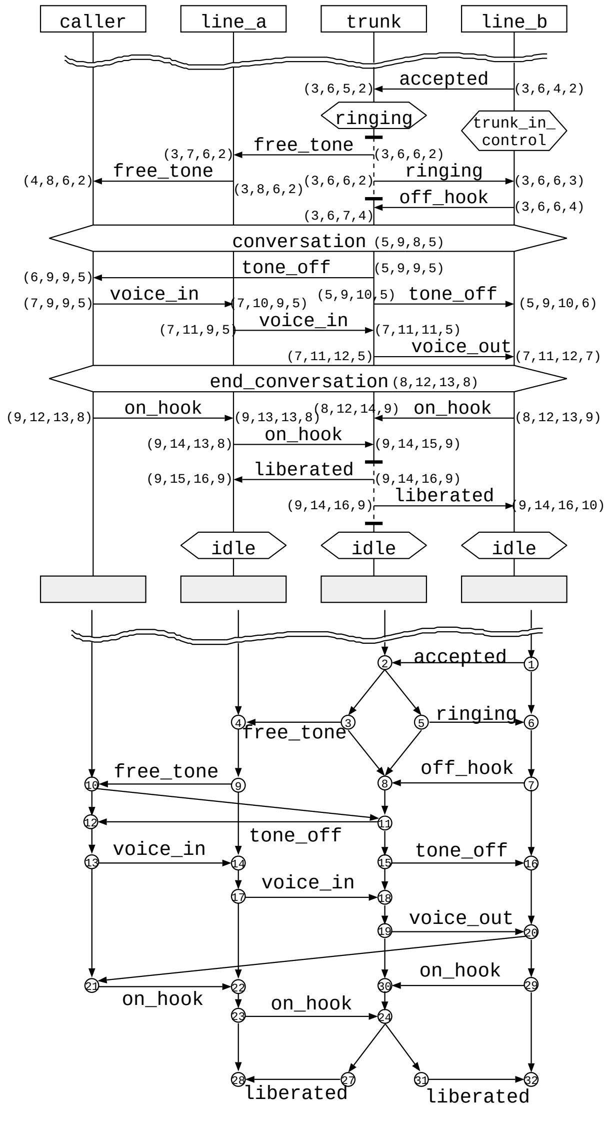

Figure 3. A modified MSC and the corresponding EPG

As mentioned above, changes to an MSC specification consist of more than one primitive changes. For example, consider two MSCs of Figure 1 and Figure 3. Both of them show telephone conversation scenarios. The MSC of Figure 1 shows a situation in which a caller hangs up(on_hook) phone in first, and then a callee($line\_b$) hangs it up after receiving busy tone($busy\_tone$). But, practically, a caller may hang up prior to the callee and vice versa. This scenario is reflected in the MSC of Figure 3 in which a caller and a callee can hang up the phone concurrently. Small changes like this accompany many primitive changes to the MSC specification. Transforming from Figure 1 to Figure 3 is composed of 5 primitive changes as follows:

1) Delete on_hook from the $line\_a$ to the $trunk$
2) Add on_hook from the $line\_a$ to the $trunk$
3) Delete $liberated$ from the $trunk$ to the $line\_a$
4) Add $liberated$ from the $trunk$ to the $line\_a$
5) Delete $busy\_tone$ from the $trunk$ to the $line\_b$

i	$\Delta_A[M_i]$	$\Delta_D[M_i]$
1	$\emptyset$	{(21,29),(22,29), (23,29),(24,29)}
2	$\emptyset$	{(16,27),(20,27),(21,27), (22,27),(21,29),(22,29), (21,30),(22,30),(21,31), (21,32),(22,31),(22,32)}
3	{(28,31),(28,32)}	{(16,27),(20,27),(21,27), (22,27),(21,29),(22,29), (21,30),(22,30)}
4	{(28,31),(28,32)}	{(21,29),(22,29), (21,30),(22,30)}
5	{(28,31),(28,32)}	{(21,29),(22,29), (21,30),(22,30)}
$M^\star$	$\emptyset$	{(21,29),(22,29), (21,30),(22,30)}

i	$\Delta_{NA}[M_i]$	$\Delta_{ND}[M_i]$
1	{(20,29),(24,27)}	{(24,25),(25,26), (26,29),(20,26),(25,27)}
2	{(22,28),(19,27)}	{(22,23),(23,24), (24,27),(23,28),(19,24)}
3	{(28,23),(23,24), (24,31),(30,24)}	{(30,31)}
4	{(19,30),(22,23)}	{(19,27),(27,28), (28,23),(22,28),(27,30)}
5	{(24,27), (27,28),(23,28)}	$\emptyset$
$M^\star$	{(24,27),(19,30), (20,29),(24,31), (30,24)}	{(19,24),(20,26),(24,25), (25,26),(26,29),(25,27), (27,30),(30,31)}

Of course, these changes are made randomly to the MSC of Figure 1. However, after these 5 changes are made to it, the modified MSC always has the same figure and same precedence relation even if any change order is performed. The results after 5 primitive changes have been processed are as follows:

3.5. Testing Based on Affected Constraints

After computing the affected sequencing constraints $\Delta_A[M]$, $\Delta_D[M]$, the added constraints $\Delta_{NA}[M]$, and the invalidated constraints $\Delta_{ND}[M]$, we need to test whether the modified concurrent program complies with the modified MSC specifications using those constraints. Below details about how to cover each constraint using either deterministic testing or a combination of nondeterministic testing and deterministic testing will be given.

Before describing testing techniques in detail, it is worthwhile to discuss on the implications of $\Delta_A[M]$, $\Delta_D[M]$, $\Delta_{NA}[M]$, and $\Delta_{ND}[M]$ to testing phase. The sets $\Delta_A[M]$ and $\Delta_{NA}[M]$ are composed of constraints of the form $a \rightarrow_M b$, which implies that the event a always precedes the event b whenever both of the events occur. On the other hand, each event pair of $\Delta_D[M]$ denotes two events that become concurrent. Let (c, d) refer to a constraint of $\Delta_D[M]$. This constraint indicates that the event c may precede d in some executions of P while d may precede c in other executions. Finally, $\Delta_{ND}[M]$ includes invalidated constraints. The constraints must not appear during executions of P.

There are two approaches in testing concurrent programs with sequencing constraints extracted from specifications: constraints-oriented approach and trace-oriented approach. In the constraints-oriented approach, test criteria is defined upon the coverage of individual constraint[3]. An individual constraint (a, b) is said to be *covered* by an execution of P if both of the events occur during the execution and a happens before b. Similarly, an execution of P *violates* a constraint (a, b) if b precedes a during the execution. In this framework, coverage of four sets $\Delta_A[M]$, $\Delta_D[M]$, $\Delta_{NA}[M]$, and $\Delta_{ND}[M]$ can be defined as follows:

- $\Delta_A[M]$ and $\Delta_{NA}[M]$: each sequencing constraint $(a, b) \in \Delta_A[M] \cup \Delta_{NA}[M]$ has to be covered at least one execution of P. Moreover, $v.b$ is infeasible whenever v is a valid and feasible prefix of P without including a.

- $\Delta_D[M]$: for each sequencing constraint $(a, b) \in \Delta_D[M]$, a happens before b in some executions of P and vice versa.

- $\Delta_{ND}[M]$: events pairs in $\Delta_{ND}[M]$ never occur during executions of P.

Nondeterministic testing of P can cover constraint $a \rightarrow_M b$ if at least one of the collected event sequences has s as its subsequence such that s preserves the order specified by $a \rightarrow_M b$. As an example, event sequence

$(e_1, e_2, e_3, e_4, e_5)$ covers $e_3 \rightarrow_M e_5$ because one of its subsequences, e.g., (e_3, e_4, e_5), preserves the specified precedence relation. Nondeterministic testing, however, does not exercise all feasible execution sequences. Furthermore, we cannot show the infeasibility of the given event sequences using nondeterministic testing[3]. It was pointed out that deterministic testing may complement the weak sides of nondeterministic one[3]. In fact, we can perform deterministic testing to show the coverage of $a \rightarrow_M b$ as well as the infeasibility of the given event sequences.

A trace-oriented approach tries to ensure the conformance between specifications and programs in the viewpoints of execution traces rather than individual constraints. This approach may accommodate various conformance relations between specifications and programs[4].

Although most of prior works on trace-oriented testing methods are concentrated on the equivalence conformance relations[1, 2, 5, 8], there are many situations in practice where it is difficult to apply testing techniques based on the equivalence between specifications and programs. First, when partial specifications are employed for concurrent program testing, we are unable to mark a feasible sequence as invalid because it is impossible to induce all valid sequences from a partial specification. In addition, specification languages for concurrent programs usually have partial order semantics for describing sequencing constraints among synchronization events. Previous testing frameworks, based on the equivalence relation, require that all totally ordered sequences induced from a partially ordered set be observed during program execution. One major problem with this approach is that only a subset of totally ordered sequences may be realized in the corresponding implementation according to the design decision. In this case, a target program still meets its specification although the feasibility of all valid sequences is not satisfied.

To cope with the partial and nondeterministic characteristics of MSCs, we consider two types of nondeterminacy in the specification according to their intentions[4]. One is intended to be implemented into the program exactly as described in the specification while the other is just introduced for specification convenience. For example, consider the two concurrent events in Figure 1: *line_a-send(on_hook)* and *line_b-send(on_hook)*. In this case, the program must implement the nondeterminacy since the caller can disconnect the phone call before the callee does, and vice versa. This kind of nondeterminacy is called *obligatory nondeterminacy*. In contrast, the two nondeterministic events *line_a-recv(free_tone)* and *line_b-recv(ringing)* in Figure 1 have different intentions. It does not matter whether the caller hears the *free_tone* signal before the callee hears the *ringing* signal or vice versa. This kind of *optional nondeterminacy* may be serialized by design decisions.

Based on these considerations, we can define two conformance relations, which are devised for testing concurrent programs using partial and nondeterministic specifications as follows:

- Concurrent program *P* has behavioral conformance to MSC *M* if and only if the feasible event sequences exercised by executions of *P* are valid with respect to *M*.

- Concurrent program *P* has nondeterminacy conformance to MSC *M* with respect to a obligatory event pair (e_1, e_2), where events e_1 and e_2 are concurrent, if and only if two feasible event sequences s_1 and s_2 of P such that event e_1 is followed by e_2 in s_1 and vice versa in s_2 are valid with respect to M. Similarly, P has nondeterminacy conformance to M if and only if P has nondeterminacy conformance to M with respect to all obligatory nondeterministic event pairs.

Testing based on the behavioral relation can be done by first performing nondeterministic testing of P, which will cover some of constraints in $\Delta_A[M] \cup \Delta_{NA}[M]$. For uncovered constraints, say (a, b), we generate a valid sequence $v.a.\beta.b$ and perform deterministic testing of P with this sequence.

Similarly, nondeterministic executions of P will cover some of constraints in $\Delta_D[M]$. For uncovered constraints, say (c, d), we generate two valid event sequences $v.c.\beta.d$ and $v.d.\beta.c$ and perform deterministic testing of P with those sequences. If P succeeds for one of the sequences, P satisfies the behavioral conformance relation with respect to the sequencing constraint (c, d). Note that behavioral conformance requires just one of the two sequences to be feasible. We repeat this process until all uncovered constraints are satisfied by P.

Since the events pairs in $\Delta_A[M]$ or $\Delta_{NA}[M]$ are sequentialized, nondeterminacy conformance testing is applied only against $\Delta_D[M]$. As in the behavioral conformance testing, uncovered constraints by nondeterministic testing are forced to be exercised by deterministic testing. For example, if nondeterminacy between e_1 and e_2 that constitute an event pair of $\Delta_D[M]$ appear to be obligatory and only one sequencing constraint, say $e_1 \rightarrow_M e_2$, has been covered by nondeterministic testing of P, then deterministic testing of P with $v.e_2.\beta.e_1$ would be performed to satisfy the nondeterministic conformance relation with respect to (e_1, e_2).

4. Discussion

One important concern in concurrent program testing is to determine whether the system behaves correctly. In [6], the use of a graphical interval logic(GIL) was presented in specifying and verifying temporal properties of concurrent programs. Specifications in GIL describe temporal properties that every state sequence exercised during an execution of a program has to satisfy. In order to determine whether the properties specified in a GIL formula are satisfied, a test oracle is built by constructing an FSM that accepts precisely those state sequences satisfying the formula. This approach supports run-time monitoring and debugging in such a way that the traces are checked as they are generated and a violation of a specification is reported as early in a trace as possible. In order to obtain the traces exercised during an execution of a concurrent program, nondeterministic testing is employed by manually instrumenting the Ada source program.

Other interesting languages for specifying sequencing constraints were defined for concurrent program testing. One such example includes a constraint notation called CSPE (Constraints on Succeeding and Preceding Events)[3]. Analogous to our work, CSPE constraint-based testing uses a combination of nondeterministic and deterministic testing. Furthermore, it can be applied to those situations where constraints are not complete, even though constraints can be made complete if they contain every possible pair of events.

As shown above, although diverse techniques are deployed for concurrent program testing, most of the techniques did not address how to test a modified program against a modified specification. One important distinction that makes our approach different from previous testing techniques is that changes to a specification are considered. In order to reduce the cost of regression testing, impact analysis is performed to obtain sequencing constraints affected by modifications made to a specification. Additionally, our testing technique does not need languages designated for specifying sequencing constraints such as CSPE and TSL because sequencing constraints to be covered can be automatically derived through impact analysis.

Furthermore, since the specifications under consideration are supposed to be partial and nondeterministic, we have proposed two conformance relations: the behavioral conformance relation and the nondeterminacy relation. The behavioral conformance relation requires a concurrent program to implement at least one synchronization sequence for each unique behavior described in an MSC specification and the nondeterminacy conformance relation requires that obligatory nondeterminacy must be preserved in the corresponding implementation.

At present, the technique presented in this paper is still undergoing extensive evaluation. More experiments are planned to demonstrate the applicability of our technique. We believe that the technique will provide a basis for developing new regression testing techniques for various types of specifications in the literature.

References

[1] H. AboElFotoh, O. Abou-Rabia, and H. Ural, “A Test Generation Algorithm for Systems Modelled as Nondeterministic FSMs,” *IEE Software Eng. Journal*, pp. 184-188, July 1993

[2] G. v. Bochmann and A. Petrenko, “Protocol Testing: Review of Methods and Relevance for Software Testing,” *Proc. of Int’l Symp. on Software Testing and Analysis*, pp. 109-124, 1994

[3] R. H. Carver and K. C. Tai, “Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs,” *IEEE Trans. on Software Eng.*, **24(6)**, pp. 471-490, June 1998

[4] I. S. Chung, H. S. Kim, H. S. Bae, Y. R. Kwon, and B. S. Lee “Testing of Concurrent Programs based on Message Sequence Charts,” *Proc. of Int’l Symp. on Soft. Eng. for Parallel and Distributed Systems*, pp. 72-82, May 1999

[5] S. K. Damodaran-Kamal and J. M. Francioni, “Testing Races in Parallel Programs with an OtOt Strategy,” *Proc. of Int’l Symp. on Software Testing and Analysis*, pp. 216-227, 1994

[6] L. K. Dillon and Q. Yu, “Oracles for Checking Temporal Properties of Concurrent Systems,” *Proc. of 2nd ACM SIGSOFT Symp. on Foundations of Software Eng.*, pp. 140-153, Louisiana, 1994

[7] C. Fidge, “Logical Time in Distributed Computing Systems,” *IEEE Computer*, **24(8)**, pp. 28-33, August 1991

[8] G. H. Hwang, K. C. Tai, and T. L. Huang, “Reachability Testing : An Approach to Testing Concurrent Software,” *Proc. of Asia Pacific Software Eng. Conf. 1994*, pp. 246-255, Tokyo, 1994

[9] T. J. LeBlanc and J. M. Mellor-Crummey, “Debugging Parallel Programs with Instant Replay,” *IEEE Trans. on Computers*, vol. C-36, no. 4, pp. 471-482, 1987

[10] G. Rothermel and M. J. Harrold, “A Safe, Efficient Regression Test Selection Technique,” *ACM Trans. on Software Eng. and Methodology*, vol. 6, no. 2, pp. 173-210, 1997

[11] K. C. Tai, R. H. Carver and E. E. Obaid, “Debugging Concurrent Ada Programs by Deterministic Execution,” *IEEE Trans. on Soft. Eng.*, **vol. 17**, no. 1, pp. 45-63, January 1991

[12] ITU-T Recommendation Z.120: Message Sequence Chart (MSC), April 1996