International Scholarly Research Network
ISRN Software Engineering
Volume 2012, Article ID 324054, 22 pages
doi:10.5402/2012/324054

Research Article

Synthesis of Test Scenarios Using UML Sequence Diagrams

Ashalatha Nayak[1] and Debasis Samanta[2]

[1] *Department of CSE, Manipal Institute of Technology, Manipal 576104, India*
[2] *School of Information Technology, Indian Institute of Technology, Kharagpur 721302, India*

Correspondence should be addressed to Ashalatha Nayak, asha.nayak@manipal.edu

Received 20 December 2011; Accepted 8 February 2012

Academic Editors: S. D. Kim and H. Okamura

UML 2.0 sequence diagrams are used to synthesize test scenarios. A UML 2.0 sequence diagram usually consists of a large number of different types of fragments and possibly with nesting. As a consequence, arriving at a comprehensive system behavior in the presence of multiple, nested fragment is a complex and challenging task. So far the test scenario synthesis from sequence diagrams is concerned, the major problem is to extract an arbitrary flow of control. In this regard, an approach is presented here to facilitate a simple representation of flow of controls and its subsequent use in the test scenario synthesis. Also, the flow of controls is simplified on the basis of UML 2.0 control primitives and brought to a testable form known as intermediate testable model (ITM). The proposed approach leads to the systematic interpretation of control flows and helps to generate test scenarios satisfying a set of coverage criteria. Moreover, the ability to support UML 2.0 models leads to increased levels of automation than the existing approaches.

1. Introduction

Due to the increasing size and complexity of software applications, the design and specification have become an important activity in the software life cycle. Interaction-based specifications such as UML sequence diagrams have been found effective in this regard, as they describe system requirements in the most intuitive way. A sequence diagram captures dynamic aspects of a system by means of messages and corresponding responses of collaborating objects. In other words, method calls, parameters, return values, and the collaborating objects can be explicitly modeled in a sequence diagram. As a result, these specifications can be used not only for capturing system behaviors but also for generating test cases that can precisely check for faults at the implementation level.

As recommended by Jacobson, the requirements of a system can be represented by a set of use cases [1, 2]. A use case can have several scenarios describing the main (primary) and alternate (secondary) scenarios. Since a scenario represents the single trace of behavior of the system, completely describing a system requires all the possible scenarios. These multiple scenarios of a use case are not completely independent of one another and are related to realize the behavior of a use case [3–5]. So the test scenario synthesis from sequence diagram must not only consider the sequence diagram corresponding to a single scenario but consider the behavior from multiple scenarios that are related to a use case.

Before UML 2.0, the sequence diagram notations considered each scenario separately in a single sequence diagram [6, 7]. Probably, due to that, early research on test scenario synthesis from sequence diagrams has concentrated on each scenario in isolation. However, there have been many problems on maintaining such a large set of sequence diagrams. The redundant information leading to different errors in design documents is reported by some researchers [8, 9]. Mainly, any change in the specification induced a lot of modification in the associated diagrams. In the absence of scenario relationships, many studies are conducted to infer scenario relationships from multiple sequence diagrams. To get over-all behavior of the system, such independently written partial behaviors are combined in some of the approaches [10, 11]. However, constructing a single consistent model requires significant additional effort from the designers [10].

UML 2.0 specifications have introduced the notion of combined fragments to model multiple scenarios in a concise manner [7, 12]. These new modeling capabilities

have provided significant improvement in the ability to model large-scale software systems [6]. Using the standard constructs to model hierarchical capabilities, it is possible to support models at any arbitrary levels of complexity. At one end, maintaining a large set of diagrams and the subsequent effort in combining has been subsumed by new constructs. However, at the other end, it turns out that there remains a gap to generate test scenarios from the new models.

To address the above gap, this paper investigates the application of sequence diagrams in software testing. There are many approaches in the literature for generating test cases from new sequence diagrams. However, these approaches focus on interpreting each scenario independently rather than identifying dependencies among them. Consequently, none of the models in the existing approaches allow nested and hierarchical constructs of a sequence diagram ever since the inception of UML 2.0. Even the approaches [13–17] proposed after the introduction of UML 2.0 have only sketched how to derive test cases while paying little attention to the model itself. In UML-based testing, models serve as the blueprint from which test cases are derived. To increase the automation of this procedure and thereby to derive greater benefit from model-based testing, models have to be accurate. In contrast, the existing tools and techniques are not capable of handling new constructs. This can lead to inaccuracies while interpreting the models. Thus, the primary motivation for the proposed research comes from the need to support higher level of automation and model precision while adopting UML 2.0 standard.

Although 2.0 version of sequence diagrams has increased expressive power, it is very hard to interpret sequence diagrams with new structured control constructs [18, 19]. The interpretation of a single scenario is evident but the way it relates to other scenarios cannot be interpreted easily [20]. Obviously, the interpretation becomes harder for large and complex specifications. Hence, despite the advantages, test scenario synthesis from the sequence diagrams becomes a difficult and challenging task, especially when interactions include repetitive, alternative, and concurrent control flows. A combined fragment may also enclose nested fragment(s). Further, there can be several operands, each containing its combined fragment [7], which can again be a plain combined fragment or another nested fragment. This implies that the nesting can be arbitrarily complex. Also, there can be different types of operators designating each fragment and some of the scenarios may denote early exit paths from these fragments (break fragment). Based on the type and nesting of combined fragments, scenario identification requires identifying various control flows that are embedded within combined fragments. Moreover, an early exit path of a combined fragment needs detailed interpretation. For example, an early exit path from a concurrent fragment cannot be mapped into a one-to-one basis with a scenario. The parallel merging leads to more number of scenarios compared to that of an alternative fragment. Pan-Wei [21] pointed out that exploring every possible use case scenario is a challenging task. This necessitates the conversion of a sequence diagram to a representation suitable for test scenario generation. In other words, the objective is to develop a testable model that can manage the test scenario generation process from a sequence diagram.

This paper aims at providing methodological support for automating the test scenario generation process from a sequence diagram using a two-phase approach. The control flow analysis of a sequence diagram is accomplished in the first phase. In order to accomplish control flow analysis, it is desirable to view a sequence diagram in terms of units of interaction such as in control flow analysis of programs [22]. These units of interaction are termed as blocks and identify the messages of a sequence diagram in terms of blocks of messages. A directed graph representation known as scenario graph is presented as an outcome of the first phase. In the second phase, test scenarios are generated from the scenario graph.

The rest of the paper is organized as follows. A brief discussion on basic definitions and concepts used in proposed methodology is given in Section 2. Section 3 presents the proposed approach to test scenario generation. Section 4 presents an illustration to explain the proposed approach. Experimental results are presented in Section 5. In Section 6, the related work is described and compared with the proposed approach. Finally, Section 7 concludes the paper.

2. Basic Concepts

In this section, a brief review of UML 2.0 sequence diagram is presented. The scenario graph is then defined for mapping the control flow semantics of various fragments. Based on the control primitives appearing in the scenario graph, a classification scheme is proposed. Finally, the concept of testable model known as Intermediate Testable Model (ITM) is discussed.

2.1. UML 2.0 Sequence Diagrams. A sequence diagram also called interaction diagram graphically displays a sequence of messages among collaborating objects for various scenarios of a use case. A set of such messages forms an interaction. In order to specify the notion of interaction, the abstract syntax of sequence diagram is defined as follows.

Definition 1. A sequence diagram is a tuple $D = \langle P, E, l, F\rangle$ where one has the following.

(i) P is a set of objects denoting participants involved in an interaction.

(ii) E is a set of events where each event corresponds to sending or receiving a message.

(iii) l is a labeling function that maps each event $e \in E$ to one specific participant $p \in P$ such that $l(e) = p$. The participant is known as the sender while an event $e \in E$ corresponds to sending a message; otherwise it is known as the receiver.

(iv) F is a set of ordered (from top to bottom) fragments.

Each fragment is a set of operands such that $F_i = \{opd_1, \ldots, opd_q\}$ where q is the number of operands. An operand opd_i, $i = 1, 2, \ldots, q$ is in the form $\langle opr, guard, M\rangle$ where

opr denotes the interaction operator associated with the fragment and $guard$ denotes a boolean expression that may be associated with the operand. M is a finite set of messages that are associated with the operand and may contain any fragment(s) nested in the operand opd_i. Each M is in the form (m, s, r) where m is the label of the message and $s, r \in P$ denote sender and receiver, respectively. There is an ordering relation over the messages in a operand. For any two distinct events e_i and e_j let $e_i < e_j$ denote that e_j occurs after e_i. Two messages $M_i = (m_i, s_i, r_i)$ and $M_j = (m_j, s_j, r_j)$ in M can be described by the ordering relation $<$. If $s_i = s_j$ and $es_i < es_j$ or if $r_i = r_j$ and $er_i < er_j$ or if $r_i = s_j$ and $er_i < es_j$, then $M_i < M_j$. Here, es_i and er_i denote the sending event and receiving event for a message M_i, respectively.

Example 2. A typical sequence diagram is shown in Figure 1(a) with three participant objects involved in the interaction where $P = \{Object_1, Object_2, Object_3\}$. There are two alt fragments, each fragment with two operands. In addition, a main fragment named *sd* will exist by default [7] which holds these two alt fragments. The main fragment is thus divided into several operands which are ordered from top to bottom. These operands are marked on the rightmost side of Figure 1(a) as B_1 to B_7. In this work, it is considered that messages are generated by synchronous interactions among objects. For every message, there can be *Occurrence Specifications*, which specifies the occurrence of message events such as invoking and receiving of method calls [7]. The occurrence specifications, in this way, denote message end points along a lifeline. es_i is used here as the convention for labeling message end points. For example, events associated with message $m1$ are (es_1, er_1) where es_1 is the sender event and er_1 is the receiver event.

Now consider messages within operand $B_1 = (opr, guard, M = \{m_1, m_2\})$ where $opr = sd$ and $guard = true$. For both messages m_1 and m_2, the sender object is $Object1$ such that $m_1 = (m1, Object1, Object3)$ and $m_2 = (m2, Object1, Object2)$. Since $es_1 < es_2$, $m_1 < m_2$. Next, there appears an alt fragment with two operands $B2$ and $B3$ where $B2 = (alt, [c1], \{m_3, m_4\})$ and $B_3 = (alt, [c_2], \{m_5\})$. Now, for messages m_3 and m_4, the receiver of $m3$ and sender of $m4$ are the same. Accordingly, the ordering is established as $m_3 < m_4$ since $er_3 < es_4$. Similarly, messages within all operands are ordered that reflect the visual order from top to bottom. To indicate this ordering, the messages are named sequentially as $m1$ to $m9$ in Figure 1(a). Thus, a sequence diagram precisely specifies the set of objects and the sequences of message exchanges that are involved in a use case.

2.2. Scenario Graph. In order to systematically investigate the comprehensive flow of control from a sequence diagram, the information contained in the sequence diagram is extracted and stored in a graph known as *scenario graph* [23]. The following nodes are considered while mapping a sequence diagram to a scenario graph.

(i) An *initial* node represents the beginning of a scenario graph.

(ii) A *block* node represents a sequence of messages such as messages within operands of a fragment.

(iii) A *decision* node represents a conditional expression such as Boolean expression that needs to be satisfied for selection among operands of a fragment.

(iv) A *merge* node represents an exit from the selection behavior such as an exit from an *alt* or an *opt* fragment.

(v) A *fork* node represents an entry into a *par* fragment.

(vi) A *join* node represents an exit from a *par* fragment.

(vii) A *final* node represents an exit of a scenario graph.

A scenario graph is defined as follows.

Definition 3. A scenario graph is a directed graph, $G = \langle A, E, in, F\rangle$. Here, *in* denotes the initial node such that there is a path from *in* to all other nodes and F denotes a set of all final nodes representing terminal nodes of the graph. Here, A is a set of nodes consisting of $BN \cup CN$ where BN is a set of block nodes, and $CN = DN \cup MN \cup FN \cup JN$ is a set of control nodes such that DN is a set of decision nodes, MN is a set of merge nodes, FN is a set of fork nodes, and JN is a set of join nodes. E denotes a set of control edges such that $E = \{(x, y) \mid x, y \in A\}$. It is assumed that each edge is labeled with the Boolean expression where the Boolean expression "true" is the default edge label attached to an edge.

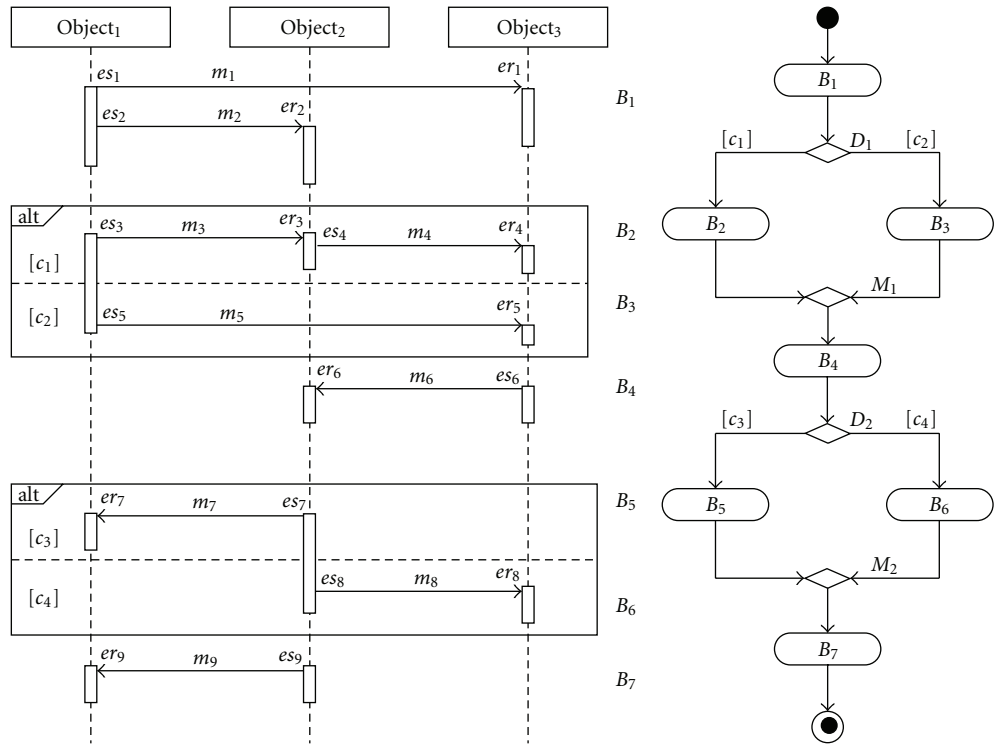
The structure of each node $A_i \in A$ is defined as follows. $\langle nodeId, nodeType, nodeDetails\rangle$ where one has the following.

(i) $nodeId$ is a unique label attached to each node in scenario graph.

(ii) $nodeType = \{decision, merge, fork, join\}$ for each $C_i \in CN$ and $nodeType = \{block, initial, final\}$ for all other nodes.

(iii) $nodeDetails = \{m_1, \ldots, m_q \mid q$ is a number of messages in $B_i \in BN\}$. Each $m_j \in nodeDetails$ is defined as a triple $\langle m, s, r\rangle$ with each message specifying its sender s, receiver r, and name of the message m for all block nodes $B_i \in BN$. $nodeDetails = \{alt, loop, break, opt, par\}$ associates an interaction operator to a control node, $C_i \in CN$.

Example 4. Figure 1(b) illustrates the scenario graph for the example sequence diagram of Figure 1(a). In a scenario graph, an operand of a fragment is denoted by a block node. A block node is shown in ovals and only node-id is mentioned for each of the nodes, for brevity. The guard associated with an operand is shown as an edge descriptor. For denoting fork and join nodes thick-line segments are considered whereas for denoting decision and merge nodes diamond symbols are used. A solid circle is used for denoting initial node and a solid circle enclosed within a hollow outer circle is used for denoting final nodes. Further, node label B_i is used for denoting block nodes. FN_i and JN_i labels refer to fork and join nodes, respectively. For denoting decision and merge nodes D_i and M_i are used as node labels. The node



(a) An example sequence diagram

(b) The scenario graph for the sequence diagram

Figure 1: An example illustrating a sequence diagram and its scenario graph.

structures for block nodes and control nodes are discussed hereinafter.

As shown in Figure 1(b), a block node is assigned one or more messages of an operand. Thus, the node structure for B_1 is assigned as ($nodeId = B_1$, $nodeType = block$, $nodeDetails = \{m_1, m_2\}$). It can be seen that a decision node D_1 is connected to the node B_1. Since D_1 is a control node, the structure ($nodeId = D_1$, $nodeType = decision$, $nodeDetails = alt$) is assigned to the decision node. In this way, a scenario graph provides an alternate representation for the sequence diagram by preserving all the details that are required for scenario generation.

2.3. Control Primitives in a Scenario Graph. A combined fragment encloses a group of messages that are associated with a specific type of operator and is expressed in terms of its operands. In the following, each combined fragment will be considered as a basic control primitive. This allows to express nested fragments in terms of basic primitives.

2.3.1. Loop Construct. A loop in a scenario graph is a set of nodes involved in an iterative computation.

2.3.2. Selection Constructs. The *alt* fragment and its variants such as *break* and *opt* result in a selection behavior, which are collectively referred here as a selection construct. A selection construct in a scenario graph can be identified as a set of nodes involved within a decision node and a merge node. However, there can be different variations of selection depending on the way a merge node converges. That is, there exist N, $N \neq 0$, such that N denotes outgoing flows from a decision node; a merge node may be used to converge K flows where K denotes incoming flows to a merge node and $0 \leq K \leq N$. A selection can be termed as *matched selection* if all outgoing flows from a decision node can be matched with each incoming flow of a merge node; that is, $K = N$. If $K < N$, then it is *K-out-of-N selection*. Further, there may not be a merge node; that is, flows from a decision node may not converge into any merge node. This situation is referred as *unmatched selection*.

2.3.3. Fork Construct. A pair of fork-join nodes is considered as a fork construct in a scenario graph. The occurrence of a fork node initiates multiple parallel flows whereas a join node synchronizes these parallel flows. The default action associated with join is AND and therefore the join node introduces a wait action. Only, when all other incoming flows are ready, the control is transferred on the outgoing edge of a join node.

2.4. Intermediate Testable Model (ITM). The scenario graph is built on the description of the message sequences that occur in a use case. In a scenario graph, a test scenario is corresponding to an execution thread. A sequence diagram can be arbitrarily complex, and in such a case, it is not

straightforward to interpret execution behavior directly from a scenario graph. In order to manage the complexity of a scenario graph, an intermediate representation of the scenario graph is proposed which is *intermediate testable model* (ITM). With ITM representation, each control construct can be analyzed independently. That is, a control construct in the scenario graph can be mapped to a special node in ITM termed as *composite node*. This region is termed as *Control Construct Graph* (CCG). To preserve the nesting structure, this mapping is done hierarchically such that a composite node may enclose zero or more composite nodes. An ITM can therefore be viewed as a concise representation of the scenario graph [23]. Since each fragment has been compressed into a composite node, an ITM is finally a chain of nodes $\langle a_1, \ldots, a_k \rangle$ such that for all i, $1 \leq i \leq k-1$, a_i is a predecessor of a_{i+1}.

Definition 5. An ITM $G_k = \langle A, E, in, f \rangle$ is a chain of nodes where one has the following.

(1) $A = A_B \cup A_C$: a finite set of nodes. Each node in A_B and A_C is a block node and composite node, respectively.

(2) $E \subseteq \{(a_i, a_j) \mid a_i, a_j \in A, i \neq j\}$: a set of directed edges between two nodes a_i and a_j.

(3) $in \in A$: the start node representing an initial node of the sequence diagram.

(4) $f \in A$: a final node representing a node without any successor node.

3. STUSD Methodology

In this section, the proposed approach for generating test scenarios from a sequence diagram is presented. The proposed methodology is termed as *STUSD* where *STUSD* stands for Synthesis of Test scenarios using UML Sequence Diagrams. Figure 2 gives an overview of the proposed methodology. A sequence diagram modeled using UML 2.0 design specifications is input to the test scenario synthesis methodology. The sequence diagram is composed by a designer using a CASE tool and stored in XMI format. The STUSD approach consists of two phases, *control flow analysis* and *test scenario synthesis*. The control flow analysis phase produces the scenario graph which is the directed graph representation of the given sequence diagram. In the first step of the analysis phase, the XMI parser reads the sequence diagram and retrieves details such as objects, messages, and fragments along with their corresponding guard information. In the subsequent step, the control flow interpreter deduces the sequence information from the temporal ordering of messages and forms a *scenario graph*. This graph, in turn, is used as input to the synthesis phase. In the first step of the synthesis phase, the ITM generator transforms the scenario graph into a testable model called the *Intermediate Testable Model* (ITM). Subsequently, the test scenario generator generates test scenarios from the ITM known as *abstract test cases*. Details of these steps are discussed in the following subsections.

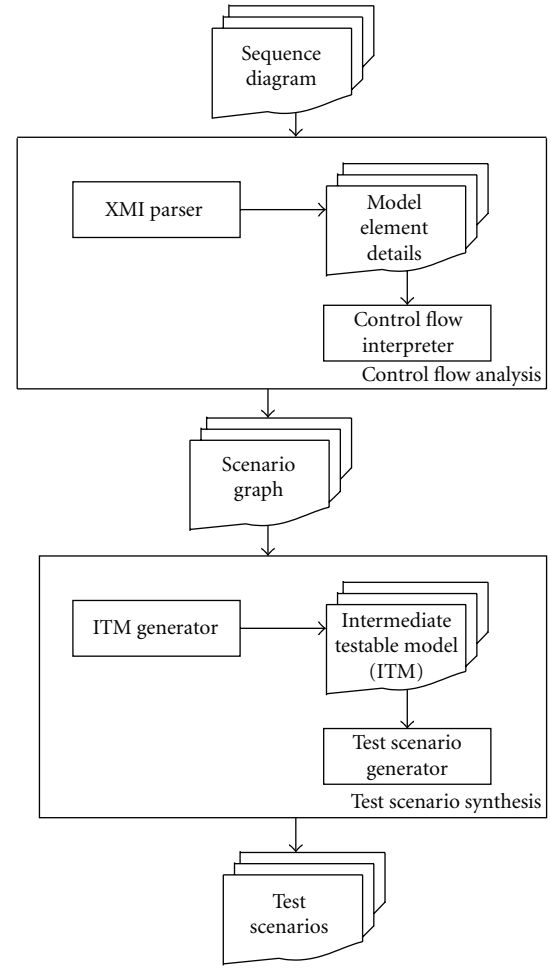


Figure 2: An overview of STUSD.

3.1. Capturing Model Element Details. In order to map a sequence diagram (exported in XMI format) to a scenario graph, a parser is designed to retrieve all information pertaining to a sequence diagram. The parser considers the following structures for messages and fragments to capture their information.

A message M_i in a sequence diagram is of the form $\langle m_i, sender, receiver \rangle$ (see Definition 1), where the message is $m_i = \langle methodName, paramList, rValue \rangle$. Here, $methodName$ denotes the name of a method from the object *sender* to the object *receiver*, $paramList = \{p_1, \ldots, p_n\}$ denotes a set of parameters associated with the method, and $rValue$ is the return value of the corresponding message on the sequence diagram. Both parameters and return values are denoted by $\langle name, type, value \rangle$. Here, *name* denotes the name of the attribute, $type$ denotes the data type associated with the attribute, and *value* is an instance of the value which is assigned to the attribute. Each participant is of the form $\langle objectName, className \rangle$.

Each fragment F_i is structured in terms of its operands, that is, $F_i = \{opd_1, \ldots, opd_q \mid q \text{ is the number of operands}\}$. An operand structure is assumed to contain four elements: $opd_i = \langle fragmentId, guard, M, F_i \rangle$ where

fragmentId denotes the interaction operator such as alt, par, and break that designate the type of fragment and *guard* denotes a Boolean expression that may be associated with each operand. M is a set of messages that are associated with the operand and F_i denotes an optional list of fragments indicating multiple instances of nested fragments within an operand.

The retrieved information in the format as stated previously is used to build the scenario graph, which is discussed in the following subsection.

3.2. Building Scenario Graph. The element structure stated in the previous section is used to build a scenario graph. It may be noted that the scenario graph representation preserves the sequencing among messages of a sequence diagram. Depending on the interaction operator, each fragment can be featured with its own flow of control. In order to extract this control flow, the transformation procedure is given for each fragment type. However, there are two particular issues to be considered in order to apply these transformations to a given fragment.

(i) Variability of an operand structure: an operand of the fragment may enclose subfragments varying the structure of each operand.

(ii) Variability of message structure: the number and type of messages contained within an operand of a fragment may vary.

To address the previous, messages within each fragment are confined as an unit of interaction and denoted by a *block* node. Irrespective of the message structure and operand structure, this allows us to construct the semantics of a fragment in terms of its constituent nodes such as block nodes and control nodes. This abstraction results in predefined transformation so that every fragment can be uniquely transformed to a set of nodes and edges. As a result, any nested fragment can be successively transformed in terms of its contained fragments.

The approach to build the scenario graph from a given sequence diagram is stated in algorithm *CreateScenarioGraph*. It is defined in terms of a recursive function *exitNode = ProcessFragment (fragmentId, entryNode)* to extract the control flow within nested fragments (Procedure 1 and Algorithm 1). Initially, the main frame that hosts the sequence diagram of a use case is supplied as a *fragmentId*. Depending on the contained elements within this main frame, the function is recursively called with the *fragmentId* of the element to be transformed. In each transformation, two nodes are distinguished—entry node and exit node. The entry node is the current node which is connected to the outside by incoming edges and therefore supplied as input to the function. The exit node is the node which is connected to the outside by outgoing edges and hence returned as output of the function. The fragments are processed until the termination condition is reached. When the termination condition for the main fragment is reached, the scenario graph is returned with initial node *in* as the entry node and the final node fn as the exit node (see Procedure 1).

Figure 3 depicts each of the fragment transformation in terms of a rule. In Figure 3 the left side of the rule is a fragment and the right side is a set of nodes and edges. After applying a particular rule, the fragment on the left side is transformed into a graph structure which is shown at the right side of the transformation rule. The nodes B_i and B_k in each of the transformation depict the nodes which are outside the fragment. For the sake of explicitly showing entry and exit nodes, the nodes B_i and B_k are marked as entry and exit nodes, respectively. An edge with arrow-head pointing to a node is shown to mark the entry node. Similarly, an outgoing edge from a node is shown to mark the exit node.

3.3. Building an ITM. While obtaining a scenario graph, the emphasis is on extracting the underlying control flow semantics of a sequence diagram. However, to generate test scenarios, it is required to analyze the dependencies that arise within nested fragments. It makes sense, therefore, to provide a structured representation for generating test scenarios. Based on the classification proposed in Section 2.3, a simplification on the structure of a scenario graph is carried out. A set of basic primitives are identified to express a scenario graph in terms of a hierarchical structure.

The simplifications lead to a testable model known as Intermediate Testable Model (ITM). The ITM is an intermediate form of the scenario graph. The objective is to identify all basic primitives in a scenario graph and reduce them to their corresponding *composite nodes* (see Section 2.4). Such a procedure of replacing the basic primitives by a composite node is termed as *composition*. To do the composition, the scenario graph G is traversed looking for basic primitives enclosed by its entry and exit node. For each such basic primitive in G, it is replaced by its composite node. Thus a new graph G_i is formed. If regions are nested with other regions, then the composition is done from the innermost region to the outermost region successively. The traversal of the scenario graph hence may be repeated several times until no more composition is possible. The composition procedure eventually reduces a scenario graph to a single chain of nodes known as ITM. The composite nodes corresponding to selection, loop, and concurrent constructs are referred as selection nodes, loop nodes, and concurrent nodes, respectively.

There are two main tasks in the composition: identifying different types of basic constructs and creating respective composite nodes for each of these types. In the following, the issues related to the compositions of various types of constructs are discussed and then the algorithm that makes use of composite nodes for building the ITM is presented.

3.3.1. Identifying Loops and Creating Composite Nodes. A loop in a scenario graph is modeled through a decision node. The branches that leave a decision node all have a conditional expression. Among these, there exists branch that exits the loop by connecting the branch to the node outside the region. Similarly, there exists branch that continues looping by connecting the branch to the node inside the region. These conditions are mutually exclusive so that the loop is either entered or exited. A loop fragment of a sequence diagram is

Input: D: Sequence diagram in XMI form // D is the main fragment
Output: **G**: scenario graph in the form $\langle A, E, in, F\rangle$
1: Create initial node in;
2: $x = ProcessFragment\ (D, in)$ // Process the main fragment with $fragmentId = D$
3: **if** $x \neq final\ node$ **then**
4: Create final node $fn \in F$;
5: Connect edge from x *to* fn;
6: **end if**
7: **return** G with entry node $in \in A$ and exit node $fn \in F$;
8: stop
Algorithm—The scenario graph generation algorithm

Procedure 1: Function Create Scenario Graph.

essentially a pretest loop as the decision node precedes the looping section.

For a pretest loop, the decision node is the entry node. A set of nodes and edges within the entry and exit nodes are recognized as a region and constitute a control construct graph. This region is reduced to a single node as shown in Figure 4(a). It may be noted that the edge entering to the entry node is connected to the composite node. Similarly, the edge leaving the exit node is originated from the composite node. In addition to this, information regarding the type of a composite node is associated with the composite node. For example, the type of the composite node would be loop here.

The example in Figure 4(a)(i) models a loop primitive. The identified loop is marked as composite node $L1$. The composition transforms the scenario graph structure to ITM as shown in Figure 4(a)(ii).

3.3.2. Identifying Fork Construct and Creating Composite Nodes. A fork node is used to model two or more parallel execution paths in a scenario graph. These parallel paths are combined into a single flow using a join node.

A fork in which all N execution paths are converged with a join node indicates that a fork node is an entry node and join node is an exit node. Once a construct is identified based on entry and exit nodes, then it is reduced to its corresponding composite node. Entry and exit edges are then attached to this newly created composite node. Figure 4(b)(i) shows such a parallel construct. The composite node of type fork known as $X1$ is associated with control construct graph as shown in Figure 4(b)(i). The ITM is then built using $X1$ as shown in Figure 4(b)(ii).

3.3.3. Identifying Types of Selection and Creating Composite Nodes. The fragments *alt*, *opt*, and *break* correspond to selection behavior. The selection in a scenario graph can be represented as a region enclosed by decision node and merge node. Based on the number of outgoing edges, a selection construct can lead to two-way or multiway selections. A two-way selection represents *if-then-else* logic by capturing nodes that are executed when the condition is true and the nodes that are executed when the condition is false. The decision node can also be connected to more than two outgoing edges so as to capture *switch-case* logic known as multiway selection. Since a selection can be of *matched*, *unmatched,* or *K-out-of-N* type, their composition is addressed separately as given hereinafter.

Case 1 (Matched selection). In this case, the entry and the exit node can be easily identified. Figure 4(c)(i) depicts a two-way selection where there are two branches leaving the decision node. The selection can thus be represented as a region enclosed by decision node and merge node. In the same way, a multiway merge forms a well-defined region enclosed by boundary node—entry node and exit node as shown in Figure 4(c)(ii). The region enclosed by selection construct is identified based on entry and exit node types. A composite node is then created to represent this region of the graph. For this purpose, the *type* information is attached to the composite node indicating that it represents a matched selection construct. Figure 4(c)(iv) shows an ITM where the region enclosed by multiway selection is reduced to composite node marked as $S1$.

Case 2 (Unmatched selection). In case of unmatched selection, some branch(s) may not converge at the end of the control flow. Such a branch is terminated at an exit node. Figure 4(c)(v) depicts an unmatched selection. A composite node of type unmatched selection is then created and made to associate with this region as shown in Figure 4(c)(v). The final node f is used here to indicate end of flow of control. It may be noted that there is no path from f. A composition of unmatched selection is shown in Figure 4(c)(vi).

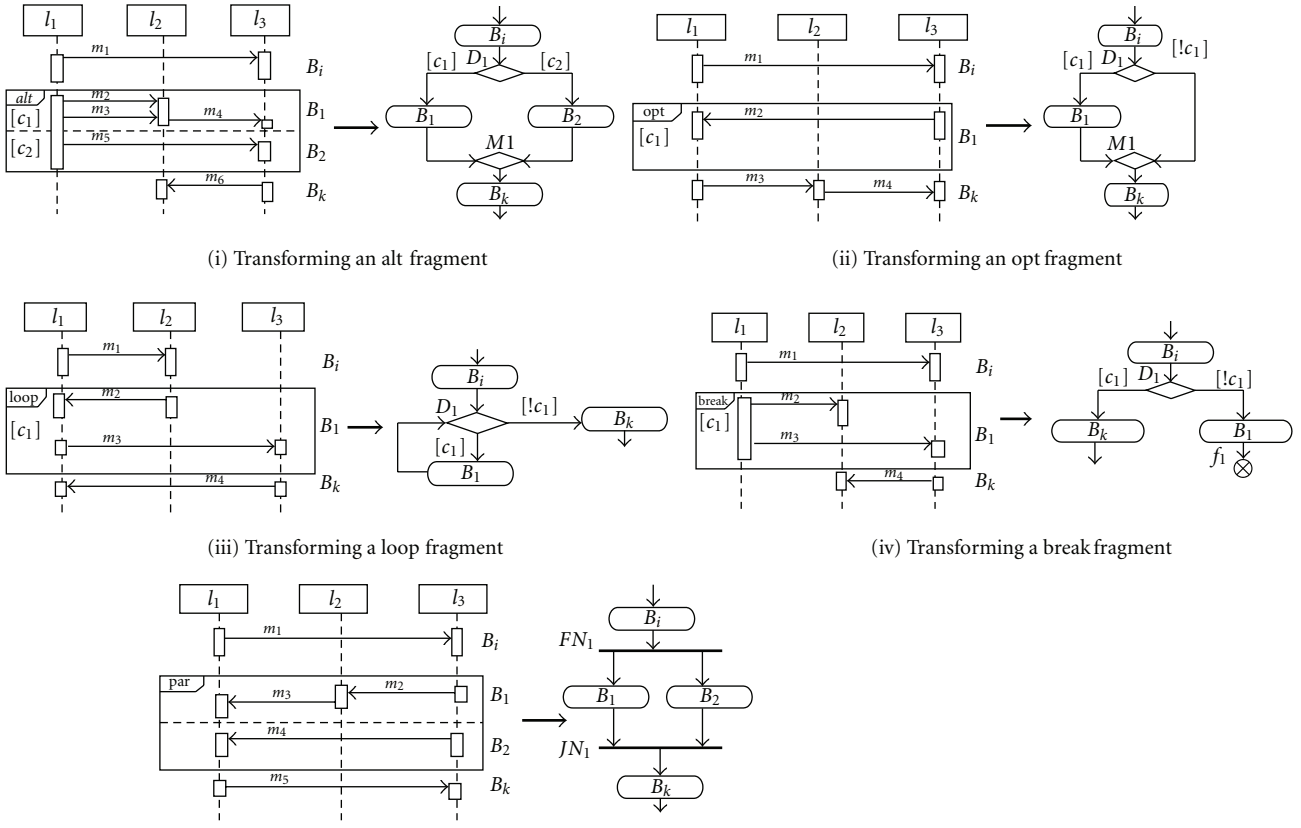
Case 3 (K-out-of-N selection). In this case, K branches are merged and remaining $N - K$ are not merged such that $K < N$. It can be composed in the same way as in the previous two cases with a little modification. For K branches, the exit node is a merge node. The remaining $N - K$ branches are with implicit final node as exit node. Two separate regions can be composed corresponding to matched and unmatched parts following Cases 1 and 2 of the selection construct, respectively. Figure 4(c)(iii) shows a situation where 3 out of 4 branches are merged. Figure 4(c)(iv) shows

Input: $fragmentId$: Fragment, a tag indicating the type of fragment
$curNode \in A$
Output: $exitNode \in A$
1: **While** ! End Offragment **do** // end of current fragment
2: $x = GetNextElement()$; // Read the next element in the fragment
3: **if** $x =' EOF'$ **then** // Termination condition
4: $exitNode = curNode$;
5: **return** $exitNode$
6: **end if**
7: **case:** $x =' message'$ // x is a message element
8: InsertAtPos(i, x, L);
9: x = GetNextElement();
10: **while** $x =' message'$ **do**
11: InsertAtPos$(i + 1, x, L)$;
12: x = GetNextElement();
13: **end while**
14: **if** $x! =' message'$ **then**
15: unread(x); break;
16: **end if**
17: $BN = CreateBlockNode(L)$ // Create a block node, BN with block of messages in partial order O;
18: ConnectEdge$(curNode, BN)$; // Edge from $curNode$ to the next node BN
19: $curNode = BN$;
20: **end case;**
21: **case:** $x =' loop'$ // x is a loop fragment
22: $DN = CreateDecisionNode(x.guard)$ // Create decision node with predicate $guard$ in x;
23: ConnectEdge$(curNode, DN)$;
24: $y = ProcessFragment(x, DN.TRUE)$;
25: ConnectEdge(y, DN); // Create back edge
26: $curNode = DN.FALSE$; // Out edge from decision node
27: **end case;**
28: **case:** $x =' opt'$ // x is an opt fragment
29: $DN = CreateDecisionNode(x.guard)$ // Create decision node with predicate $guard$ in x;
30: ConnectEdge$(curNode, DN)$;
31: $y = ProcessFragment(x, DN.TRUE)$;
32: $MN = CreateMergeNode()$;
33: ConnectEdge$(DN.FALSE, MN)$;
34: ConnectEdge(y, MN);
35: $curNode = MN$;
36: **end case;**
37: **case:** $x =' break'$ // x is a break fragment
38: $DN = CreateDecisionNode(x.guard)$ // Create decision node with predicate $guard$ in x;
39: ConnectEdge$(curNode, DN)$;
40: $y = ProcessFragment(x, DN.TRUE)$;
41: $fn = CreateFinalNode()$;
42: ConnectEdge(y, fn);
43: $curNode = DN.FALSE$;
44: **end case;**
45: **case:** $x =' alt'$ // x is an alt fragment
46: $DN = CreateDecisionNode(nil)$ // Create decision node with multioperands;
47: ConnectEdge$(curNode, DN)$;
48: $MN = CreateMergeNode()$;
49: **for** *each operand* $opd_i \in x$

Algorithm 1: Continued.

```
50:                         e_i = guard(opd_i);
51:                         y = ProcessFragment(opd_i, DN.e_i);
52:                         ConnectEdge(y, MN);
53:             end for
54:             curNode = MN;
55:         end case;
56:         case: x =′ par′                    // x is a par fragment
57:             FN = CreateForkNode()                    //Create fork
            node;
58:             ConnectEdge(curNode, FN);
59:             JN = CreateJoinNode();
60:             for each operand opd_i ∈ x
61:                         y = ProcessFragment(opd_i, FN);
62:                         ConnectEdge(y, JN);
63:             end for
64:             curNode = JN;
65:         end case;
66:  end while
67:  exitNode = curNode;
68:  return   exitNode
Function ProcessFragment(Fragment, A)
```

Algorithm 1: Function ProcessFragment (Fragment: fragmentId, A: curNode).



(i) Transforming an alt fragment

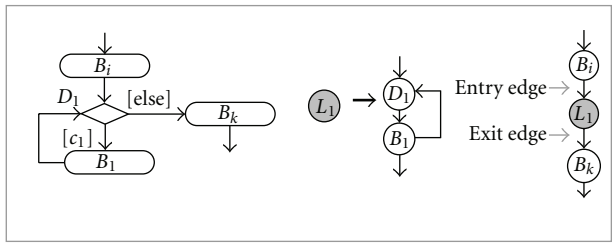
(ii) Transforming an opt fragment

(iii) Transforming a loop fragment

(iv) Transforming a break fragment

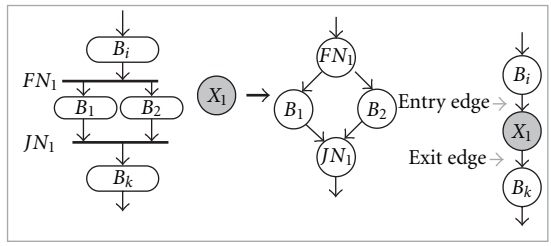
(v) Transforming a par fragment

Figure 3: Transforming fragments to their scenario graphs.



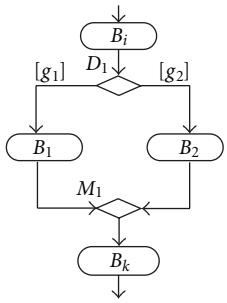
(i) A loop and its composition (ii) An ITM

(a) Creating composite node for a loop fragment

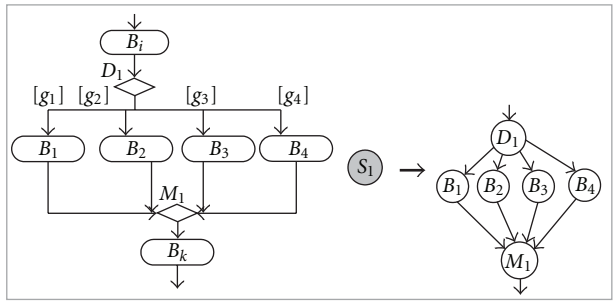


(i) Synchronized fork and its composition (ii) An ITM

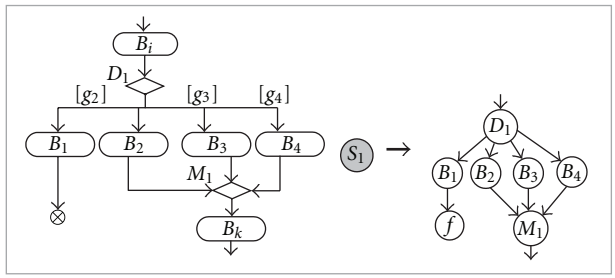
(b) Creating composite node for a par fragment



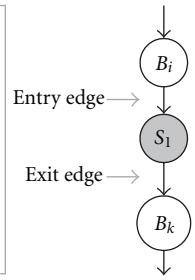
(i) A two-way selection



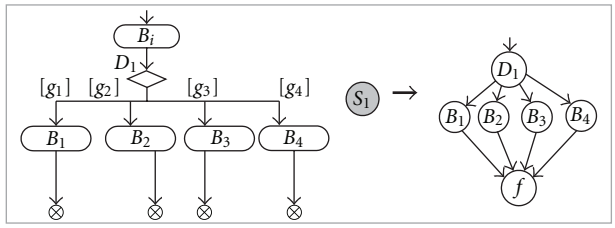
(ii) A multiway selection and its composition



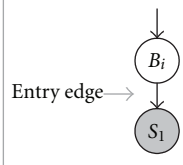
(iii) A K-out-of-N selection and its composition



(iv) An ITM for K-out-of-N and matched selection



(v) An unmatched selection and its composition



(vi) An ITM for unmatched selection

(c) Creating composite nodes for different types of selection fragments

Figure 4: Creating composite nodes for fragments.

the composite node creation of type K-out-of-N selection and their subsequent transformation to ITM.

3.3.4. Algorithm for Building an ITM. Building an ITM from a given scenario graph is stated in the form of pseudocode in algorithm *BuildITM* (Procedure 2).

Initially, the scenario graph G is presented as input to the *BuildITM* algorithm. While traversing through the scenario graph, the graph G_i is transformed to G_{i+1}, if at least one region is found. Otherwise, G_{i+1} is the same as G_i and this condition terminates the algorithm. A flag *RegionFound* is set for this purpose to indicate if

Input: G: The scenario graph in the form $\langle A, E, in, F\rangle$
Output: G_k: ITM
Data: *RegionFound*: a flag indicates whether a region is formed or not
1: RegionFound = TRUE // Initially assumes composition is possible
2: **while** *Region Found* **do**
3: $G_k = FormIntermediateGraph(G, RegionFound)$
4: $G = G_k$
5: **end while**
6: **return** G_k // No more composition possible
Algorithm—Building ITM

Procedure 2: Function *BuildITM*.

there exists at least one composition (which means that $G_{i+1} \neq G_i$). This is handled by the function $G_{i+1} = FormIntermediateGraph(G_i, RegionFound)$ which transforms G_i to G_{i+1}.

On each call of *FormIntermediateGraph()*, the function *ComposeRegion()* receives the control node which is the current entry node and returns the exit node of the region located (Procedures 3 and 4). Since a nested construct has successive entry nodes without their corresponding exit nodes, the function *ComposeRegion()* locates regions recursively. When all the control nodes are processed, the resulting graph G_k is returned to the caller, *BuildITM*.

3.4. Generating Test Scenarios from ITM. An ITM is a testable model having behavioral information of a system under test (SUT) corresponding to a use case. Alternatively, a path in the scenario graph can be mapped onto a test scenario. Thus, scenario graph can be used to formally define the test scenario of a sequence diagram as given hereinafter.

Definition 6. A sequence $S = \langle n_0, n_1, \ldots, n_q\rangle$ is a test scenario in a scenario graph, if $n_0 = in$, $n_q \in F$ and $(n_i, n_j) \in E$, for all i, j, $0 \leq i$, $j < q$, where E denotes a set of edges and F denotes a set of final nodes in a scenario graph.

An ITM which is a reduced form of a scenario graph encapsulates each of the control primitives in the form of special node known as a composite node. A composite node may also be nested with zero or more composite node(s). It may be noted that an ITM is a chain of nodes from the initial node to the final node. This chain is termed as *base path* and denoted by t_b. The base path denotes a sequence of nodes containing block nodes as well as composite nodes. Depending on the construct enclosed, there may be a number of paths enclosed in a region. A composite node is thus said to enclose a set of internal paths. A path beginning with an entry node in a region is termed as an *internal path*. The number of internal paths composed by each composite node can be derived from the coverage criteria. In the following, coverage criteria with respect to different cases are introduced. Next, an algorithm for test scenario generation that makes use of internal paths is presented.

3.4.1. Coverage Criteria. Let T be a set of test scenarios for a sequence diagram. T satisfies the following coverage criterion if the following condition holds good.

(i) *Selection Coverage Criterion.* For each selection node in an ITM, T must include one test scenario corresponding to each output branch of the decision node.

(ii) *Loop Adequacy Criterion.* For each loop node in an ITM, one has the following.

(1) T must include at least one test scenario in which control reaches the loop and then the body of the loop is not executed (zero iteration path).

(2) T must include at least one test scenario in which control reaches the loop and then the body of the loop is executed at least once before control leaves the loop (more than zero iteration path).

(iii) *Concurrent Coverage Criterion.* For each concurrent node in an ITM, T must include one test scenario corresponding to every valid interleaving of message sequences.

The messages in a par fragment can be interleaved as long as the ordering imposed by each operand is maintained [7]. In this respect, a valid interleaving sequence is the one which maintains ordering of message sequences within an operand.

3.4.2. Test Scenario Generation Algorithm. Algorithm *GenerateTestScenario* has been proposed to produce a set of paths satisfying each of the above-mentioned criteria (Procedure 5). The algorithm performs a depth first search in the control construct graph starting from its entry node. An *internal path set* is associated through each composite node, to expand a path into a number of paths. In that case, any path from the initial node to a composite node say, C_j, is said to be attached with a suffix which is the set of all internal paths corresponding to C_j.

The ITM representation lends itself to a simple method of test scenario generation. Initially, t_b is the trivial scenario, since t_b may contain zero or more composite node(s) with

```
Input: G: The scenario graph in the form ⟨A, E, in, F⟩
       RegionFound: a flag for deciding the termination criteria
Output: G_k: ITM
Data: curNode: Denotes the next node to be inspected
      exitNode: Denotes the trailer node when the region is formed
1 :  G_k = G                    // Initially, G is the ITM
2 :  curNode = in               // Start from the initial node of G_k
3 :  while curNode is not a finalNode do
4 :      if FindControlNode(curNode) then // curNode is not a
         control node
5 :          exitNode = ComposeRegion(G_k, curNode)
                        // Composing to be carried out at curNode
6 :      else
7 :          if exitNode = NULL then // Ensures that no composition was carried out so far
8 :              RegionFound = FALSE;
9 :          end if;
10 :         break;
11 :     end if
12 :     curNode = exitNode;
13 : end while
14 : return G_k
Function FormIntermediateGraph(G, RegionFound)
```

Procedure 3: Function FormIntermediateGraph.

```
Input: G_k: The input graph
       curNode: The current node to be inspected
Output: exitNode: The trailer node
1:  x = FindMinimalRegion(curNode)          // Takes
    care of nested regions
2:  if x = NULL then
3:      exitNode = ReduceRegion(curNode)        // Do
        the composition following composition rules
4:      Update(G_k)          // Update G_k with CCG at curNode
5:  else // Indicates a nested region is found
6:      exitNode = ReduceRegion(x)          // x is the entry node
7:      Update(G_k)
8:      exitNode = ComposeRegion(G_k, curNode)
                   // recursively repeat the composition
9:  end if
10: return exitNode
Function ComposeRegion(G, RegionFound)
```

Procedure 4: Function ComposeRegion.

or without nested composition. It may lead to many more test scenarios on exploring each test scenario successively. In each expansion, a composite node is replaced with one of its internal paths. The number of tests generated is to cover all the paths of an internal path set. Building a set of test scenarios can therefore be considered as replacing each of the composite nodes by each of its internal path. This procedure is carried out until we find that every composite node is expanded. The algorithm *GenerateTestScenario* outlines this test scenario generation approach.

4. Illustration of STUSD

In this section, the test scenario synthesis approach is illustrated using an example called Cell Phone System (CPS). It is considered that the cell phone being designed has keys for power on and off, digits 0–9, alphabets a–z, and buttons, such as talk, cancel, scroll up, scroll down, and end. There are two actors. They are (a) users (customers) who subscribe and request the services and (b) administrators who manage the payment and subscription information. From the user's

Input: ITM, an intermediate testable model of a given scenario graph with initial node, in.
Output: T, a collection of test scenarios forming the test suite.
Data: B, a set of base paths which is initially empty
P_j: a set of internal paths corresponding to a composite node, C_j.

1: Generate base path, t_b from ITM
2: $B = B \cup t_b$ // Add t $_b$ to the set of base paths
3: **while** $B \neq empty$ **do**
4: **for** path $t_i \in B$ **do**
5: Let $t_i = \langle n_0, n_{i1}, n_{i2}, \ldots, n_{ip}, \ldots n_{iq} \rangle$ be a sequence of adjacent nodes in t_i
6: **for all** $n_{ij} \in t_i$ **do**
7: **if** n_{ij} is not a composite node **then**
8: $T = T \cup t_i$ // t_i is fully expanded and it is added to the test suite T
9: Remove t_i from B
10: **else**
11: **for** composite node $n_{ij} \in t_i$ **do**
12: Get internal paths corresponding to the node n_{ij}
13: Let $P_j = \{p_1, \ldots, p_m\}$ denote m internal paths corresponding to the composite node n_{ij}
14: **for** $p_i \in P_j$ **do**
15: Replace the node n_{ij} in t_i with the internal path p_i and let the expanded form of t_i be t_i'
16: $B = B \cup t_i'$ // Add t_i' to B
17: **end for**
18: **end for**
19: **end if**
20: **end for**
21: **end for**
22: **end while**

Algorithm—Test scenario generation from ITM

Procedure 5: Function GenerateTestScenario.

perspective, different ways of interacting with the system are identified. Each of the use cases is represented with a corresponding sequence diagram.

The sequence diagram for *Make Call* use case is presented in Figure 5. The *Make Call* sequence diagram models the interactions taking place when the user hits a button. Then onwards, different variations and operational conditions are identified depending on the key pressed. In the following, the procedure involved in the construction of scenario graph and ITM is illustrated with respect to the *Make Call* use case. Following this, the test scenario generation from the ITM is illustrated.

4.1. Scenario Graph Construction. The scenario graph G obtained from the *Make Call* sequence diagram is shown in Figure 6. Here, the Boolean expressions on the outgoing edges of the decision nodes denote the guard conditions appearing on interaction fragments. Every block node records one or more messages. For example, the first block node $B1$ denotes sequence of messages $m1$ and $m2$. Then the *alt* fragment is shown which is mapped onto a decision node with two outgoing edges—D_1 to B_2 and to B_3, respectively. For "$key = cancel$" combination, there is an unconditional *break* fragment which brings the user screen to a *menuScreen* mode closing the interaction. This is shown by a block node, B_2 as the target of edge labeled "$key = cancel$" and a final node. Similarly, the other combination "$key = digit$" is maintained as a block bode, B_3. In the same way, all the messages and fragments are parsed to produce the scenario graph.

4.2. ITM Construction. The working of *BuildITM* algorithm is traced for the sequence diagram shown in Figure 5. The scenario graph of Figure 6 is used as input G. Figure 7 gives the intermediate graphs produced and composite nodes. In these figures, we have sticked to circle notation for denoting nodes. To emphasize on composition procedure, each node is shown with its node identifier and any edge descriptor is ignored. A shaded node denotes a composite node. The composite node is labeled according to the type of composite node. That is, selection nodes are labeled using S_i as node labels. For concurrent nodes X_i is used as node label and for loop nodes L_i is used as node label. The composition procedure starts looking for basic primitives in G classifying them into various types of composite nodes. A loop node named $L1$ and selection nodes named $S1$ and $S2$ are created

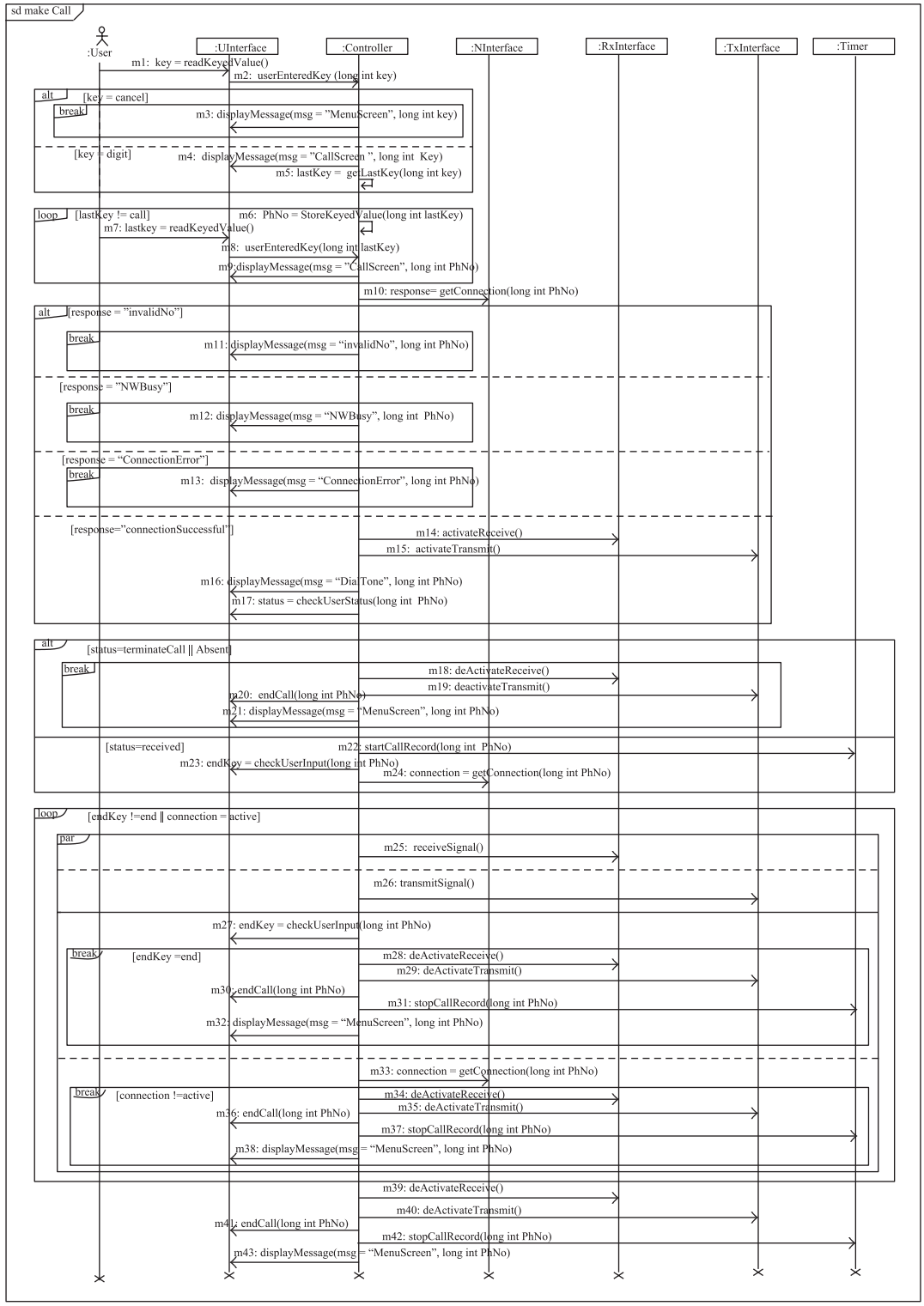


Figure 5: Sequence diagram of *MakeCall* use case in CPS.

as shown in Figure 7(b). This forms the first iteration and the resulting graph at iteration level 1 is called as G_1. In the second iteration, G_1 forms the input producing G_2 as output. In G_2, a concurrent node named $X1$ is identified as shown in Figure 7(d). In this way, the composition procedure continues until it finds that all constructs are reduced. At this level, the final ITM is formed which is shown in Figure 7(l).

4.3. Test Scenario Generation. As per test scenario generation algorithm, the first step is to generate base path, which

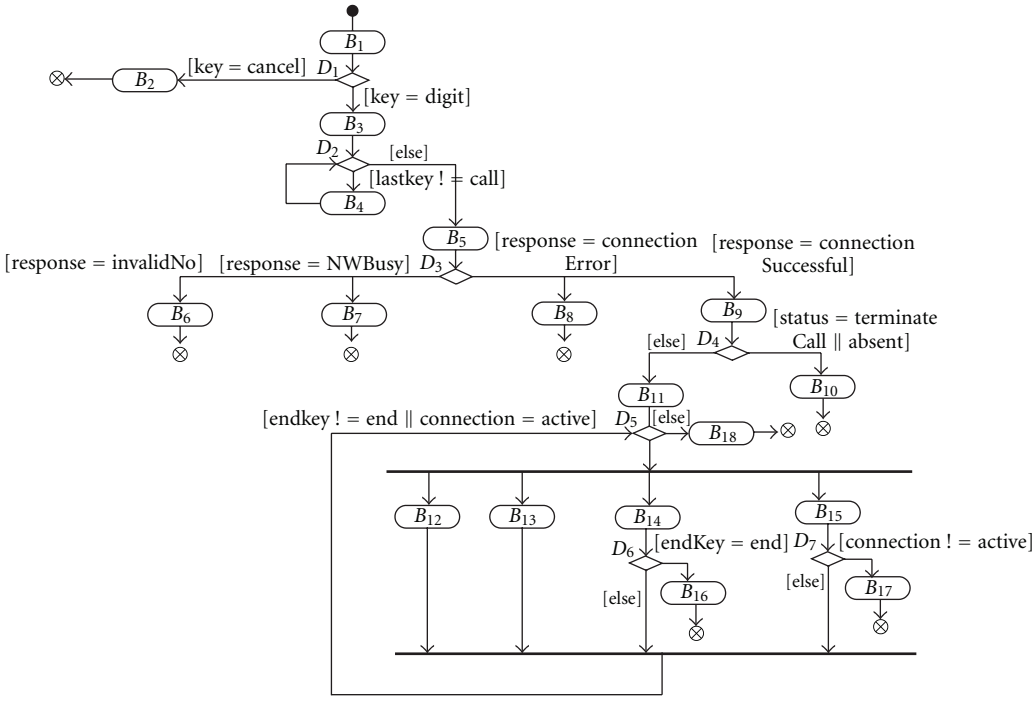


FIGURE 6: Scenario graph of *MakeCall* use case.

is $B = \{\langle in, B1, S5\rangle\}$ (see Figure 7(l)). In the next step, the composite node $S5$ is expanded. As the node type of $S5$ indicates an unmatched selection, all possible test scenarios corresponding to the selection coverage criterion are covered. As a result, $S5$ will be replaced with each of its internal paths. In this step, it gives rise to two paths into B. The first of these contains sequences of basic nodes only. Therefore, it is moved into set, T. The second path contains a composite node named $L1$ and $S4$. $L1$ is then replaced with its internal paths conforming to loop adequacy criteria giving rise to two internal paths, $\langle D2\rangle$ and $\langle D2, B4\rangle$. Next, as per selection coverage criterion, four internal paths are produced for selection node, $S4$. The test scenarios are thus evolved iteratively from the base path until B becomes empty. Table 1 shows few iterations of the test scenario generation algorithm. In this table, the underlined nodes denote composite nodes which are expanded in the succeeding steps.

5. Experimental Results

In this section, an experiment to investigate the effectiveness of generated test scenarios is presented. Four use cases are considered in the proposed experiment and the test suites for each of these are obtained using the proposed approach. In the following, the use cases considered in this experiment are listed.

5.1. Subject Programs. The four use cases we considered in our experiment are *Credit Card Payment, Paper Registration, PIN Validation,* and *Make Call*. Each of these use cases has several control primitives and accordingly their sequence diagrams exhibit moderate complexity. Further, use cases were fairly chosen so as to include different variations and combinations of control primitives. The use cases are the following:

(1) Credit Card Payment use case in Online Shopping System (OSS),

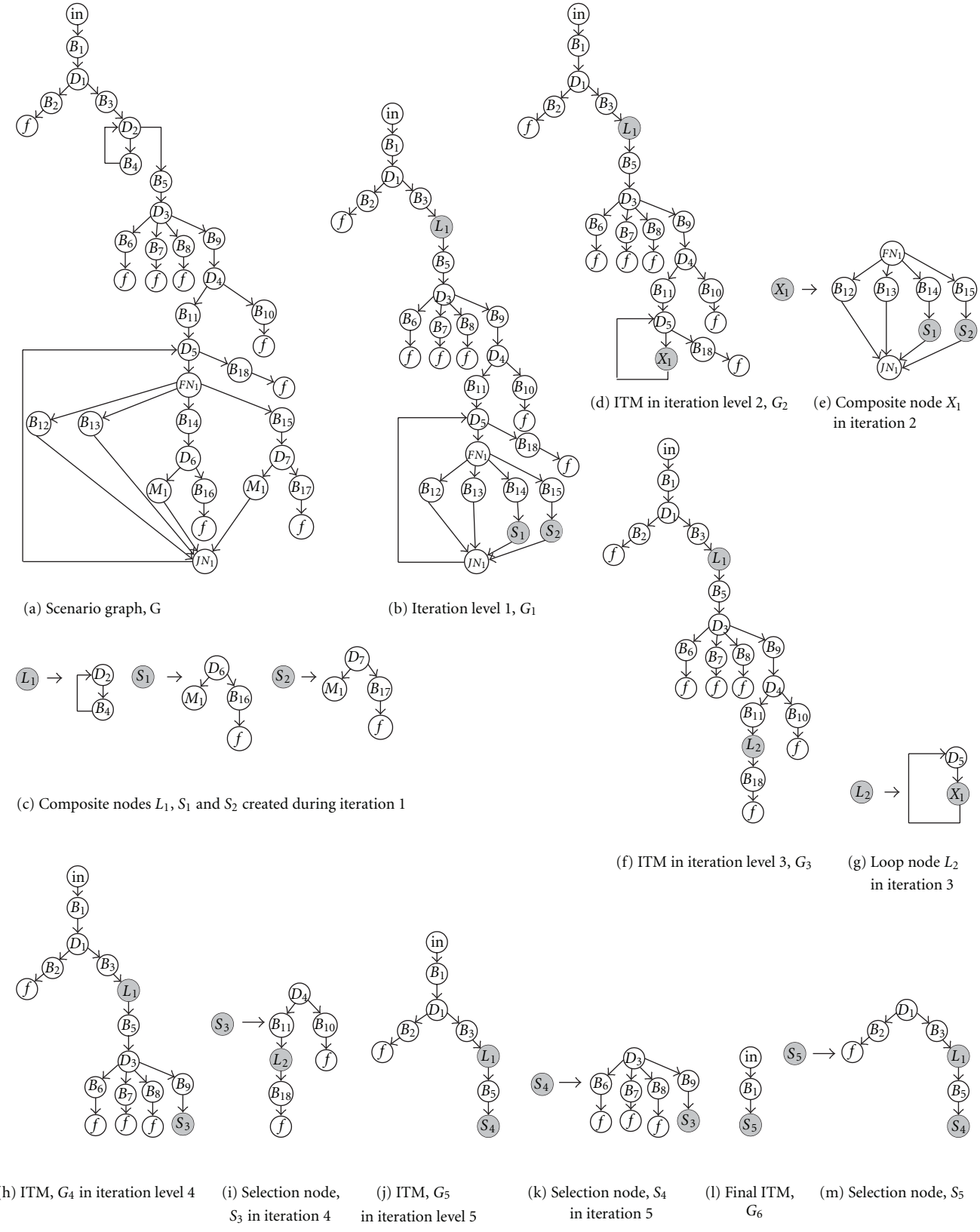
(2) Paper Registration use case in Conference Management System (CMS),

(3) PIN Validation use case in Automatic Teller Machine (ATM),

(4) Make Call use case in Cell Phone System (CPS).

The objective of the proposed experiment is to observe fault detection ability of the generated test suite. For this, a set of mutation operators (stated in Section 5.2) are considered. A mutation operator induces small syntactic change [24] to the programs. A *mutant program* or *mutant* is produced by applying a single mutation operator only once to the original program. Applying the mutation operators, a set of mutants are generated. The implementation corresponding to the above-mentioned use cases is in Java and considered as the subject programs. The subject programs we considered are moderately sized Java programs ranging from 200 to 500 lines of code. These lines of code indicate the functional part of the program where the faults are specifically seeded. It excludes code corresponding to user interface portions, input-output handling, API inteface, and so forth. The additional details of these subject programs are given in Table 2.

5.2. Fault Model. The fault model of proposed automatic test synthesis technique is described here for mutation testing.



(a) Scenario graph, G

(b) Iteration level 1, G_1

(c) Composite nodes L_1, S_1 and S_2 created during iteration 1

(d) ITM in iteration level 2, G_2

(e) Composite node X_1 in iteration 2

(f) ITM in iteration level 3, G_3

(g) Loop node L_2 in iteration 3

(h) ITM, G_4 in iteration level 4

(i) Selection node, S_3 in iteration 4

(j) ITM, G_5 in iteration level 5

(k) Selection node, S_4 in iteration 5

(l) Final ITM, G_6

(m) Selection node, S_5

Figure 7: Composition procedure for building ITM.

Table 1: Generating test scenarios for *Make Call*.

Step	Set of base paths, B	Set of test scenarios, T
1	$\{\langle in, B1, \underline{S5}\rangle\}$	—
2	$\{\langle in, B1, D1, B2, f\rangle, \langle in, B1, D1, B3, \underline{L1}, B5, \underline{S4}\rangle\}$	—
3	$\{\langle in, B1, D1, B3, \underline{L1}, B5, \underline{S4}\rangle\}$	$\{\langle in, B1, D1, B2, f\rangle\}$
4	$\{\langle in, B1, D1, B3, D2, B5, \underline{S4}\rangle, \langle in, B1, D1, B3, D2, B4, B5, \underline{S4}\rangle\}$	$\{\langle in, B1, D1, B2, f\rangle\}$
5	$\{\langle in, B1, D1, B3, D2, B5, D3, B6, f\rangle, \langle in, B1, D1, B3, D2, B5, D3, B7, f\rangle, \langle in, B1, D1, B3, D2, B5, D3, B8, f\rangle, \langle in, B1, D1, B3, D2, B5, D3, B9, \underline{S3}\rangle, \langle in, B1, D1, B3, D2, B4, B5, D3, B6, f\rangle, \langle in, B1, D1, B3, D2, B4, B5, D3, B7, f\rangle, \langle$in, B1, D1, B3, D2, B4, B5, D3, B8, f$\rangle, \langle$in, B1, D1, B3, D2, B4, B5, D3, B9, $\underline{S3}\rangle\}$	$\{\langle in, B1, D1, B2, f\rangle\}$
6	$\{\langle$in, B1, D1, B3, D2, B5, D3, B9, $\underline{S3}\rangle, \langle$in, B1, D1, B3, D2, B4, B5, D3, B9, $\underline{S3}\rangle\}$	$\{\langle in, B1, D1, B2, f\rangle, \langle in, B1, D1, B3, D2, B5, D3, B6, f\rangle, \langle in, B1, D1, B3, D2, B5, D3, B7, f\rangle, \langle in, B1, D1, B3, D2, B5, D3, B8, f\rangle, \langle in, B1, D1, B3, D2, B4, B5, D3, B6, f\rangle, \langle in, B1, D1, B3, D2, B4, B5, D3, B7, f\rangle, \langle in, B1, D1, B3, D2, B4, B5, D3, B8, f\rangle\}$
7	...	...

Table 2: Details of subject programs.

Subject program name	Number of classes	Number of methods	Lines of code
Credit Card Payment	4	16	209
Paper Registration	5	36	384
PIN Validation	4	21	295
Make Call	6	43	478

A fault can be detected when the test scenarios are applied to the system under test. Since the test cases can be used to reveal faults during execution of the code, mainly two types of faults are identified known as *interaction faults* and *message sequence faults*. The interaction faults are the results of simple changes to the source program. Previous studies [25–27] have investigated the fault detection effectiveness of these general purpose mutation operators. The message sequence faults are associated with control constructs of the language. In order to inject faults into the control constructs, message sequence faults are investigated in this paper. Various mutation operators are used to cover these faults for finding the fault detection effectiveness of proposed test cases.

5.2.1. Mutation Operators. The following eleven mutation operators are considered [28, 29].

(1) *Language Operator Replacement (LOR)*. It replaces an arithmetic, logical, or relational operator by another operator from the same group. It also includes insertion or deletion of unary and logical operators to *if, while* statements, and so forth.

(2) *Literal Change Operator (LCO)*. It increases or decreases numeric values. For Boolean literals, it changes true to false or false to true.

(3) *Method-Name Replacement Operator (MRO)*. This operator replaces a method name with other method names that have the same parameters and return type.

(4) *Parameter Change Operator (PCO)*. This operator changes the order, size, and type of parameters in method declarations or invocations.

(5) *Type Replacement Operator (TRO)*. This operator replaces a type with compatible types.

(6) *Variable Replacement Operator (VRO)*. This operator replaces a variable name with other similar or compatible types.

(7) *Control-Flow Disruption Operator (CDO)*. This operator is used to disrupt normal control flow by changing break, continue, or return statements.

(8) *Statement Swap Operator (SSO)*. It swaps statements in a block such as, statements inside one switch block with another switch, and so forth.

(9) *Statement Delete Operator (SDO)*. It deletes one statement in a block such as, in the body of a synchronised method.

(10) *Loop Change operator (LCO)*. This operator increases or decreases the intended number of loop iterations by applying the changes to the loop initialization, loop expression, or an update statement.

(11) *Concurrency-Related Operator (CRO)*. This operator deletes a call to the wait or notify method. It can also be used to remove the *synchronized* attribute from a method in Java.

The mutation operators are put into two groups based on the type of faults they cover. The first six mutation operators are in $Group_I$ and this group of mutation operators covers interaction faults that are of general nature which can be applied for the variables, types, classes, expressions, or methods in the source program. In order to directly mutate the control constructs of the source program in Java that handle loops, threads, if-then statement, and so forth, the $Group_{II}$ operators numbered from 7 to 11 are used. The type and number of mutants produced depend on the source code. Table 3 lists the type and number of mutants generated for different use cases.

5.3. Metric for Evaluation. The objective of the proposed experiment using mutation testing is the generation of a test set that can distinguish the resulting output of the mutants from the original program. In order to evaluate the effectiveness of generated test sets, mutation score is used. The mutation score M for a test suite T is the ratio of *distinguished* (also known as killed) mutants over the total number of mutants.

5.4. Subject Test Suites. For each subject program mentioned in Section 5.1, sequence diagrams of these use cases are drawn with MagicDraw 16.8 [30] and the diagrams are exported in XML format. Following the STUSD methodology discussed in Section 3, subject test suites are constructed. This required instantiating the scenarios with specific values. Using dynamic domain reduction procedure [31], a feasible domain is mapped to each variable in the sequence diagram. The domain analysis for variables was carried out for this purpose to describe operational domains. A valid test input from the domain is chosen for each of the test scenario.

During test execution, any changes in the output are recorded. In order to support this, the subject program is instrumented in such a way that each method returns its method identifier only when the method produces the correct response. By making use of mutation operators, faults are injected in the subject programs. For example, in the implementation of *Make Call* use case, 61 mutant programs are created with each mutant containing one of the 61 faults. The test suites are executed on each of the mutants. The method identifier is then used to track any deviation in the method execution from the expected response. This has solved the test oracle problem which was crucial for evaluating the generated test suite. The programwise results for all subject programs with respect to each test suite are shown in Table 4.

5.5. Analysis of Results and Its Limitation. For each subject program, Table 4 lists a number of mutants generated along with number of mutants killed. Different mutants injected under each type of fault are given in Table 3. In order to clearly specify the generated mutants and their killing rate, mutants are shown under two groups. First group referred to as $Group_I$ includes mutants of generic nature as discussed in Section 5.2. Although mutants in $Group_I$ try to inject faults and exercise different parts of the source program, these mutants cannot directly cause faults into the control constructs. In addition, mutants numbered 7–11 labelled as $Group_{II}$ exercise different parts of the source program and cause faults into the statement block such as loop and if-then-else.

Intuitively, assuming that the model covers the required behavior, the faults that are injected in those part of the code can only be detected. This means that faults in remaining part of the implementation, if not covered by the model, cannot be killed. For example, in *Make Call* and *PIN Validation*, additional faults based on Binder's Modal class test [32] are injected. An object of modal class can accept messages only in a particular state which is referred here as valid state and rejects the messages in any other state referred as invalid state. For each modal class under test, the state of the object before test is then checked against the state of the object after the test [33]. In order to track this, the state of each object is specified while the message was received. Following this, the Modal Class Faults (MCFs) are created as follows.

(i) Valid state of the receiving object: this fault is caused to check that the messages are received in the valid state of an object.

(ii) Invalid state of the receiving object: this fault is caused to check that the messages received in the invalid state of an object are rejected.

At present, faults under MCF category cannot be detected by the proposed approach. Four faults of type *MCF* are seeded in *Make Call* subject program and it is observed that the proposed test cases are not able to distinguish these faults.

In Table 4, the variations in the fault detecting ability of the subject test suites are due to faults in MCF category. Nevertheless, the path coverage investigated in this experiment is effective at ensuring adequate coverage of the model. This is indeed correlated to the fault detection capability as path coverage is the strongest criteria. These variations among the test suites for a given model and its implementation suggest that achieving lesser coverage may not be appropriate. However, the limitations are due to the modal classes. In order to overcome this, it is required to specify invariants on classes and their attributes in OCL. This information can then be used to validate when the constraints are violated during test execution.

Intuitively, a test suite that covers all the test paths of a model requires more testing effort. Based on mutation analysis, however, it achieves adequacy of the test suite. Using fault detection effectiveness, the experimental results have confirmed this. In order to generate test cases, a sequence diagram of arbitrary control flow presents challenges due to various types of control primitives and their nesting. The test cases together cover all the coverage criteria and are adequate as confirmed by mutation testing results.

6. Related Work

In UML, scenario-based models and state-based models are regarded as two prominent ways to specify behavioral requirements [34]. Considering the relevant research on

Table 3: Mutants for subject programs.

Sl. No.	Mutation operator	Number of mutants for each subject program			
		Make Call	Credit Card Payment	Paper Registration	PIN Validation
1	LOR	5	3	5	6
2	LCO	5	3	5	6
3	MRO	2	0	0	0
4	PCO	4	0	6	2
5	TRO	5	1	8	2
6	VRO	5	1	8	2
7	CDO	14	2	9	7
8	SSO	4	1	5	4
9	SDO	10	1	4	4
10	LCO	2	1	4	2
11	CRO	5	0	0	5
	Total	61	13	54	40

Table 4: Mutant killing results for subject programs.

Subject programs	Number of killed mutants under $Group_I$ (M_I)	Number of killed mutants under $Group_{II}$ (M_{II})	Mutation score (M)	Test suite size
Make Call	26	35	0.93	191
Credit Card Payment	8	5	1.00	16
Paper Registration	32	22	1.00	67
PIN Validation	18	22	0.80	121

state machines, there exist some encouraging results. The algorithms and tools for synthesis of state machines from multiple scenarios are described [35–37]. However, exploiting the conjunction of scenarios to synthesize test scenarios has not been addressed till now. The state machines represent the behavior of individual objects (intraobject) whereas scenario-based models represent interaction among a set of objects (interobject). Hence, the state- and scenario-based specifications are complementary perspectives to model behavioral aspects of a system [36]. In this regard, scenario-based specifications captured by UML interaction diagrams are to be reviewed.

Among the interaction diagram-based approaches proposed for test case generation, the proposed survey reveals that interaction overview diagrams and timing diagrams have not yet contributed significantly to test case generation. The communication diagrams although similar to sequence diagrams use none of the structuring mechanisms like combined fragments. So, communication diagrams can only correspond to simple sequence diagrams. In this regard, a brief survey of test generation using sequence diagram is discussed hereinafter.

Among sequence diagram-based test generation techniques, it is worth mentioning that previous research uses sequence diagrams of UML 1.x [32, 38–41]. Binder describes general test requirements to develop test cases from a sequence diagram [32]. In Basanieri and Bertolino approach [38], test cases are derived from UML use case and sequence diagrams. The test cases are obtained corresponding to message sequences from the sequence diagram by following the temporal order of messages. In this way, starting from sequence diagrams corresponding to subuse cases, the test cases are incrementally derived for the entire system. By consulting class diagram for method and its possible input values, test specifications are then derived. A use case test suite collects all test cases of a given use case. By incrementally constructing test suites, a test frame is defined to hold all use case test suites of the system. Fraikin and Leonhardt approach [39] discuss an approach known as SeDiTeC that uses sequence diagrams to test Java programs. To facilitate testing in an early stage in development, a test stub concept is proposed. This allows to generate stubs for the classes of the system under development when all methods are not implemented. Another approach is proposed by Cheng et al. known as generic tester [40] for testing distributed systems using sequence diagrams. A generic testing scheme is proposed for handling more than one standard of communication specification in an implementation. A class diagram that supports interface definitions of all modules along with sequence diagrams is input to the generic tester. According to the method call in a sequence diagram, input data values are generated and validated against the data types in a class diagram. In the approach proposed by Briand and Labiche [41], various UML artifacts that are produced at the development stage are included for deriving test requirements. The sequential constraints between use cases are expressed using activity diagrams. The activity diagram is then transformed into a directed graph from which use case sequences are extracted. These sequences representing execution paths are then augmented with test-scale information by assigning

actual parameters with symbolic values. In the next step, each use case is described using a sequence diagram in which OCL notation is used for the description of guard conditions.

However, the aforementioned approaches [32, 38–41] are not capable of modeling most of the control sequencing flows in their input models. For example, the early termination of scenarios known as breaking scenarios is not possible to handle in UML 1.x approaches. In addition, parallel and choice of selection behavior could not be specified. Also, there is no facility for expressing nested flows in UML 1.x models. Consequently, there is no further scope to study how model complexity affects test case generation in these approaches. Fortunately, the problem of complex control flows can be expressed adequately in UML 2.0 approaches. In the following, UML 2.0 approaches are compared with the proposed work.

UML 2.0 sequence diagrams have been used in literature for testing UML design artifacts [13, 42]. In design-testing approach, Pilskalns et al. [13] discuss an approach to find inconsistencies between behavioral view specified by a sequence diagram and structural view specified by a class diagram. In their approach, an aggregate model is defined that uses Object Constraint Language (OCL) expressions and combines the information from structural and behavioral views. From a sequence diagram modeled with synchronous calls and interaction operators, alt, loop, and opt, the testing approach is described to generate and execute the tests. For detecting dynamic faults, the tests are conducted to the implementation. However, dynamic faults are limited in finding invalid values to arguments of a method call. In this regard, it is unclear how behavioral view specifically helps in detecting design faults that can be propagated into implementation. In the validation approach proposed by Dinh-Trong, test inputs are derived from UML class and sequence diagrams [42]. The symbolic execution is applied to generate path constraints where paths are selected to satisfy test adequacy criterion in the sequence diagram. However, there is no emphasis on interaction fragments in their approach. There are other approaches to generate test cases for testing implementation instead of validating the software model. These are discussed hereinafter.

Cartaxo et al. present test case generation for feature testing of mobile phone applications [14]. The basis for their approach is UML 2.0 sequence diagrams. A feature is specified as a smallest part of the service requirement. The input model is transformed to a labeled transition system from which test cases are generated for feature testing. The transition system works on the basis of annotations and they restrict the message exchanges between only two entities—user and system. The input models are ad hoc in their approach and represent only repetitive and conditional sequences without using formalism such as combined fragments. Due to this semantic gap, the models lack expressiveness of the language.

Nebut et al. present test case generation from use cases along with other formalism such as contracts [15]. For each use case parameterized with the actors, contracts are specified using preconditions and postconditions. To express these contracts, they make use of first, order logical expressions as their language and combine it with predicates and logical operators. By defining contracts for each use case, they build a transition system known as Use Case Transition System (UCTS) that can represent all valid sequences of the use case. These are known as test objectives and constitute the first phase of their approach. In the second phase, they transform test objectives to test scenarios. For this phase, they attach sequence diagram as the additional artifact to obtain sequences of message calls on the system under test. As requirements are captured by means of contracts, the sequence diagrams focus on interaction between system and actors describing the expected exchange of messages using system level sequence diagrams resulting in system testing. For this, they depict each scenario using a separate sequence diagram. The complexity of building editor tool to express declarative contracts demands high precision and rigor in their approach.

Javed et al. [16] propose a model-driven approach using model transformation technology to test software applications. As a first step, they transform a sequence of method calls in a sequence diagram into an xUnit model using model-to-model transformation tool, Tefkat. Tefkat is an Eclipse Modeling Framework (EMF) model transformation engine that executes horizontal (model-to-model) transformations on sequence diagram. In the second step, model-to-text transformations are applied using MOFScript tool. At this step, the tester needs to provide a file defining packages, and so forth, for compiling and executing the generated test cases. This file known as *code header* file along with test data file which specifies parameter values and expected return values of method calls is provided to generate JUnit test cases. In a similar way, Dai [17] discusses the transformation of UML models using UML 2.0 profile for the testing called the UML 2.0 Testing Profile (U2TP) to support model-driven testing. The testing approach proposed by Javed and Dai [16, 17] has not considered interaction fragments into their test synthesis approach. There have also been approaches which use sequence diagrams for control flow analysis [43]. However, their analysis does not focus on multiple scenarios and their dependency.

Compared to the previous approaches, the proposed approach strictly follows the standard syntax and semantics of combined fragments. Adhering to the semantics, various combinations of nesting can be recognized within fragments. The approach proposed in this paper recognizes breaking scenarios as special type of regions (unmatched constructs). It is noteworthy to observe that none of the approaches studied the problem of unmatched and nested constructs in their input models.

7. Conclusions

In this paper, a procedure to generate test scenarios using UML 2.0 sequence diagram has been proposed. Existing test scenario synthesis techniques from sequence diagrams limit their applicability on specifying and transforming a single scenario for generating information about test cases. Consequently, fragments and/or their nesting are not included

into test synthesis. These techniques can easily be automated as their input models do not exhibit much complexity. However, exploiting the expressive power embedded in sequence diagrams of arbitrary control flow presents some challenging problems while deriving scenarios conforming to a set of coverage criteria. A methodology has been proposed in this paper towards the automation of this procedure. For this purpose, the sequence diagrams representing valid behaviors of the system are analyzed. This includes scenarios that explore boundary tests as they represent valid tests of the system.

Representing the structure of a sequence diagram with scenario graph and its simplified model, ITM has many advantages. The scenario graph provides a graphical representation of the messages among all participating lifelines during the life time of the system. The hierarchical structure of ITM makes it possible to generate test cases based on simple path concatenation technique instead of exhaustive graph search techniques. This has been facilitated by the introduction of separate control flow graph for each fragment. The process also leads to maintainable and reusable models. Moreover, the proposed approach can be used as a reference model for managing a sequence diagram with nested fragments.

The two-phase approach proposed in this paper can also be easily tailored to synthesize test cases from other UML models. Due to the incremental nature of our approach, the analysis phase which is based on specific UML models needs to maintain changes whereas the generic synthesis phase can easily be applied to generate test cases from the resulting model.

Future work includes translating abstract test cases into executable test scripts for a test execution environment. Future work will also consist in extending the approach to maintain model-based traceability techniques that can assist in mapping requirements with generated test cases. It may also be helpful to support changes to design artifacts for creating relationships between requirements, design, generated test cases, and code. This problem may be especially important due to the iterative nature of software development process.

References

[1] I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.

[2] I. Jacobson and P. Ng, *Aspect-Oriented Software Development with Use Cases*, Addison Wesley, 2004.

[3] A. Cockburn, *Writing Effective Use Cases*, Addison-Wesley, 2001.

[4] P. Metz, J. O'Brien, and W. Weber, "Specifying use case interaction: types of alternative courses," *Journal of Object Technology*, vol. 2, no. 2, pp. 111–131, 2003.

[5] X. Bai, W.-T. Tsai, R. Paul, K. Feng, and L. Yu, "Scenario based modeling and its applications," in *Proceedings of the 7th International Workshop on Object-Oriented Real-Time Dependable Systems*, 2002.

[6] B. Selic, "What's new in UML 2.0?" Tech. Rep., IBM Rational Software, April 2005.

[7] "UML. UML 2.3 Superstructure-Final Adopted Specification," Object Management Group, May 2010, http://www.omg.org/spec/UML/2.3.

[8] F. J. Lucas, F. Molina, and A. Toval, "A systematic review of UML model consistency management," *Information and Software Technology*, vol. 51, no. 12, pp. 1631–1645, 2009.

[9] M. Elaasar and L. Briand, "An overview of UML consistency management," Tech. Rep. SCE-04-18, Department of Systems and Computer Engineering, Ottawa, Canada, 2004.

[10] S. Uchitel and M. Chechik, "Merging partial behavioural models," in *12th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pp. 43–52, November 2004.

[11] R. Mizouni, A. Salah, S. Kolahi, and R. Dssouli, "Merging partial system behaviours: composition of use-case automata," *IET Software*, vol. 1, no. 4, pp. 143–160, 2007.

[12] D. Bell, "UML's sequence diagram," Technical Library, IBM Rational Software, February 2004.

[13] O. Pilskalns, A. Andrews, A. Knight, S. Ghosh, and R. France, "Testing UML designs," *Information and Software Technology*, vol. 49, no. 8, pp. 892–912, 2007.

[14] E. G. Cartaxo, F. G. O. Neto, and P. D. L. Machado, "Test case generation by means of UML sequence diagrams and labeled transition systems," in *IEEE International Conference on Systems, Man, and Cybernetics*, pp. 1292–1297, October 2007.

[15] C. Nebut, F. Fleurey, Y. Le Traon, and J. M. Jézéquel, "Automatic test generation: a use case driven approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 140–155, 2006.

[16] A. Z. Javed, P. A. Strooper, and G. N. Watson, "Automated generation of test cases using model-driven architecture," in *29th International Conference on Software Engineering and the 2nd International Workshop on Automation of Software Test*, May 2007.

[17] Z. Dai, "Model-driven testing with UML 2.0," in *Proceedings of the 2nd European Workshop on Model Driven Architecture*, 2004.

[18] Z. Micskei and H. Waeselynck, "The many meanings of UML 2 Sequence Diagrams: a survey," *Software and Systems Modeling*, vol. 10, no. 4, pp. 489–514, 2011.

[19] B. Cornelissen, A. Van Deursen, L. Moonen, and A. Zaidman, "Visualizing testsuites to aid in software understanding," in *11th European Conference on Software Maintenance and Reengineering*, pp. 213–222, March 2007.

[20] D. Harel and I. Segall, "Visualizing inter-dependencies between scenarios," in *4th ACM Symposium on Software Visualization*, pp. 145–154, September 2008.

[21] N. G. Pan-Wei, *Hunting for Use-Case Scenarios*, The Rational Edge, Lexington, Mass, USA, 2003.

[22] F. E. Allen, "Control flow analysis," *ACM Sigplan Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[23] A. Nayak and D. Samanta, "Model-based test cases synthesis using UML interaction diagrams," *ACM SIGSOFT Software Engineering Notes*, vol. 34, no. 2, pp. 1–10, 2009.

[24] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering Methodology*, vol. 1, no. 1, pp. 5–20, 1992.

[25] M. R. Lyu, Z. Huang, S. K. S. Sze, and X. Cai, "An empirical study on testing and fault tolerance for software reliability engineering," in *Proceedings of the 14th International Symposium on Software Reliability Engineering*, November 2003.

[26] S. Ali, L. C. Briand, M. J. Rehman, H. Asghar, M. Z. Z. Iqbal, and A. Nadeem, "A state-based approach to integration testing based on UML models," *Information and Software Technology*, vol. 49, no. 11, pp. 1087–1106, 2007.

[27] A. Paradkar, "Case studies on fault detection effectiveness of model based test generation techniques," in *1st International Workshop on Advances in Model-Based Testing*, May 2005.

[28] S. Kim, J. Clark, and J. McDermid, "The rigorous generation of Java mutation operators using HAZOP," in *Proceedings of the 12th International Conference on Software and Systems Engineering and their Applications*, pp. 9–19, Paris, France, December 1999.

[29] M. Delamaro, M. Pezze, and A. M. R. Vincenzi, "Mutant operators for testing concurrent Java programs," in *Proceedings of the Brazilian Symposium on Software Engineering*, pp. 272–285, Rio de Janeiro, Brazil, 2001.

[30] MagicDraw, Magicdraw home page, 2010, http://www.magicdraw.com.

[31] A. J. Offutt, Z. Jin, and J. Pan, "The dynamic domain reduction procedure for test data generation," *Software-Practice and Experience*, vol. 29, no. 2, pp. 167–193, 1999.

[32] R. V. Binder, *Testing Object Oriented Systems: Models, Patterns and Tools*, The Addison-Wesley Object Technology Series, Addison-Wesley, 1999.

[33] R. M. Hierons, K. Bogdanov, J. P. Bowen et al., "Using formal specifications to support testing," *ACM Computing Surveys*, vol. 41, no. 2, article no. 9, 2009.

[34] Y. Bontemps and A. Egyed, "Scenarios and state machines: models, algorithms, and tools: a summary of the 4th workshop," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 1–4, 2005.

[35] D. Harel and H. Kugler, "Synthesizing state-based object systems from LSC specifications," *International Journal of Foundations of Computer Science*, vol. 13, no. 1, pp. 5–51, 2002.

[36] D. Harel, H. Kugler, and A. Pnueli, "Synthesis revisited: Generating statechart models from scenario-based requirements," in *Formal Methods in Software and Systems Modeling*, pp. 309–324, Springer, 2005.

[37] J. Whittle and P. K. Jayaraman, "Synthesizing hierarchical state machines from expressive scenario descriptions," *ACM Transactions on Software Engineering and Methodology*, vol. 19, no. 3, article no. 8, 2010.

[38] F. Basanieri and A. Bertolino, "A practical approach to UML-based derivation of integration tests," in *Proceedings of the Software Quality Week*, November 2000.

[39] F. Fraikin and T. Leonhardt, "SeDiTeC-testing based on sequence diagrams," in *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, pp. 261–266, 2002.

[40] F. T. Cheng, C. H. Wang, and Y. C. Su, "Development of a generic tester for distributed object-oriented systems," in *IEEE International Conference on Robotics and Automation*, pp. 1723–1730, September 2003.

[41] L. Briand and Y. Labiche, "A UML-based approach to system testing," *Journal of Software and Systems Modeling*, pp. 10–42, 2002.

[42] T. T. Dinh-Trong, S. Ghosh, and R. B. France, "A systematic approach to generate inputs to test UML design models," in *17th International Symposium on Software Reliability Engineering*, pp. 95–104, November 2006.

[43] V. Garousi, L. C. Briand, and Y. Labiche, "Control flow analysis of UML 2.0 sequence diagrams," in *1st European Conference on Model Driven Architecture—Foundations and Applications*, pp. 160–174, 2005.