Softw Syst Model (2011) 10:63–89
DOI 10.1007/s10270-009-0133-4

REGULAR PAPER

Synthesis of test scenarios using UML activity diagrams

Ashalatha Nayak · Debasis Samanta

Received: 12 April 2008 / Revised: 15 August 2009 / Accepted: 21 September 2009 / Published online: 16 October 2009
© Springer-Verlag 2009

Abstract Often system developers follow Unified Modeling Language (UML) activity diagrams to depict all possible flows of controls commonly known as scenarios of use cases. Hence, an activity diagram is treated as a useful design artifact to identify all possible scenarios and then check faults in scenarios of a use case. However, identification of all possible scenarios and then testing with activity diagrams is a challenging task because several control flow constructs and their nested combinations make path identification difficult. In this paper, we address this problem and propose an approach to identify all scenarios from activity diagrams and use them to test use cases. The proposed approach is based on the classification of control constructs followed by a transformation approach which takes into account any combination of nested structures and transforms an activity diagram into a model called *Intermediate Testable Model* (ITM). We use ITM to generate test scenarios. With our approach it is possible to generate more scenarios than the existing work. Further, the proposed approach can be directly carried out using design models without any addition of testability information unlike the existing approaches.

Keywords UML · Software testing · Model-based testing · Activity diagram · Test case generation

Communicated by Prof. Lionel Briand.

A. Nayak (✉) · D. Samanta
School of Information Technology, Kharagpur, India
e-mail: anayak@sit.iitkgp.ernet.in

D. Samanta
e-mail: dsamanta@iitkgp.ac.in

1 Introduction

Software testing is a major activity in any software development life cycle and plays a crucial role in software quality assurance. Due to the increasing complexity of software, there is a growing need to improve the testing process. Hence, various test strategies have been evolved over the years. With the popularization of object-oriented design, model-based test case generation has recently emerged as the promising approach to improve the effectiveness of the testing process. In object-oriented design methodology, model artifacts provide a systematic approach to analyze, design and implement software. In model-based testing, design artifacts are applied for testing purpose [5]. In other words, test data can be generated at an early stage of the software development process. A direct consequence of this is that testing and coding activities can be carried out in parallel and hence reducing the software development time.

To support the object-oriented development practices, Object Management Group (OMG) introduced Unified Modeling Language (UML) in the year 1996. In recent years, UML has emerged as the *de facto* standard for modeling software systems and has received significant attention from researchers as well as practitioners. Using UML, software designers can capture different views of a system. The different views that can be modeled using UML are: user, structural, behavioral, implementation and environmental views [28]. The UML structural models have traditionally been used for testing object-oriented programs at the unit and integration level [5,12,18,23,24]. The behavioral view, on the other hand, has been recognized to be particularly useful to test a system at cluster and system level [6,11,13,26,30].

In this work, we have considered the synthesis of test scenarios from UML activity diagrams for system level testing. An activity diagram is used to depict all possible flows of

executions in a use case. Possibly, UML activity diagram is the only design artifact which is good at describing the flow of control in an object-oriented system. Due to this reason, among several UML diagrams activity diagrams are treated as one of the most important design artifact. In an activity diagram, the system behavior is depicted in terms of activities combined with control flow constructs. An activity represents a task to be accomplished whereas a control flow construct such as decision, looping, concurrency and synchronization represents the order of executions of activities [28]. A construct can be nested with any other constructs and thereby representing any complex flow of activities.

A sequence of activities in an activity diagram of a use case can be used to generate test cases. Such a sequence of activities is called a test scenario or simply a scenario. Each scenario represents a trace of an execution of the system and hence can be mapped to a test case. In other words, it is required to identify all possible begin-to-end paths in an activity diagram in order to cover all activities and flow of constructs to test a use case satisfactorily. However, the path identification task is a non-trivial task. An activity diagram may have arbitrarily complex control flow due to different nested combinations of control constructs. In particular, there may be early termination of flows from a nested sequence of activities. In order to identify all possible scenarios it is required to unambiguously interpret such exit paths. The path generation becomes more complicated if there are multiple exit paths involving nested structures. Multiple entry or exit paths within loops have been discussed in the literature to cause unstructured cycles [2].

In this paper, we present a new technique to automatically generate test scenarios from an activity diagram. The method is based on analyzing the control flow graph underlying the given activity diagram. In order to facilitate the control flow analysis, various control constructs in an activity diagram are distinguished. A classification of control constructs is proposed for this purpose. The motivation is to determine the control constructs for providing a structured representation. A structured construct can be distinguished as having a single entry point and a single exit point [31]. The classification is general enough to identify control constructs which can be analyzed independently of each other. Based on this analysis, control constructs of an activity diagram can be partitioned into separate graphs known as regions. The regions have been used extensively for structuring programs on the basis of their control flow graphs (for example, see [31]). A transformation approach is proposed for this purpose which transforms the flat structure of an activity model into a well formed hierarchical structure which we call as Intermediate Testable Model (ITM). A major benefit of the structuring is that the resulting graph, that is ITM, maintains a simple control flow and the test scenario generation is systematic from ITM. In other words, the ITM is meant to unambiguously interpret the semantics of flow of activities. This helps to generate test scenarios from ITM precisely and completely. The scenarios generated are at an abstract level. The actual test cases are synthesized by augmenting each scenario with test data. This gives test values to test the target system. In this work, we have not addressed the problem of automatic test data generation.

The contributions of this paper to activity diagram based testing are as follows: First, classification of control flow constructs in an activity diagram is presented. In our classification of control flow constructs, we refine basic (non-nested) constructs based on their exit paths. Second, a transformation approach is proposed that takes into account any combinations of nested structures. Third, based on the classification and transformation, we define a testable model that is suitable for generating test scenarios. This is indeed with regard to arbitrary control flows.

The rest of the paper is organized as follows. A brief discussion on basic definitions and concepts used in our methodology is given in Sect. 2. Section 3 presents our proposed approach to test case generation. Section 4 presents a case study to demonstrate the use of our methodology. In Sect. 5, we summarize our experimental results. In Sect. 6, the related work is described and compared them with the proposed approach. Discussions regarding effectiveness, time complexity, applicability and limitations of the proposed approach are presented in Sect. 7. Finally, Sect. 8 concludes the paper.

2 Basic concepts and definitions

In this section, we introduce a few definitions and notations that are referred to in our discussions. We begin with activity diagram defining various control constructs and then introduce our testable model called ITM as a systematic way of representing these control constructs.

2.1 UML activity diagram

An activity diagram is the design artifact used for describing the dynamic behavior of a system. We can interpret the abstract syntax of an activity diagram as defined below.

Definition 1 UML activity diagram can be described as 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. A is a set of nodes consisting of $\langle AN, ON, CN \rangle$ where AN is a set of action nodes, ON is a set of object 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 $E = \{(x, y) | x, y \in A\}$.

From the definition of a UML activity diagram, it may be noted that the various types of activity nodes occurring in an activity diagram include action nodes, object nodes and control nodes [28]. Among these, action nodes specify the behavior that needs to be executed. An action begins execution by taking data from its incoming edges. An action node can have multiple incoming and multiple outgoing edges. When the execution of an action is complete, data is made available to all its successor nodes. Object nodes specify the values passing through activity diagram. Object nodes are denoted by rectangle symbol with the name of the node written inside. A shorthand notation for object nodes called input pins and output pins [28] are used for providing inputs and outputs of an action, respectively.

Control edges are used to specify how controls flow from one node to the next. All nodes in an activity diagram are connected by control edges. A control edge links two nodes in the activity diagram and is graphically represented by a directed edge. However, nodes required to be combined in many other ways to alter the sequential flow of execution. Control nodes can achieve this to form control flow constructs such as loop, conditional sequence and concurrent execution of activities. The UML superstructure [28] describes several kinds of control nodes such as initial node, final node, decision node, merge node, fork node and join node. An initial node denotes the beginning of a flow of execution. A flow of execution follows a path depending on the various guard conditions that are satisfied during the traversal and then ends in one of the final nodes. There are two types of final nodes: activity final nodes and flow final nodes. Activity final nodes are used to denote the termination of activity whereas flow final nodes denote the termination of a flow. Decision nodes choose one of the alternate flows whereas merge nodes act as a source node combining all incoming flows but without synchronizing these flows. A fork node is inserted to split the current flow into multiple concurrent flows. A join node is used to synchronize several flows into a single flow.

2.2 Control constructs in an activity diagram

A UML 2.0 activity diagram may consist of several control constructs. In the following sub section, we state and define these control constructs for references in our subsequent discussions. It may be noted that an activity diagram is a directed and connected graph. We make use of the concept of dominance and post-dominance [2] for defining regions in activity diagrams. Let x and y be two distinct nodes in a directed graph. Node x is said to *dominate* the node y if x is on every path from the initial node to y. Node x is said to *post-dominate* the node y if x is on every path from y to a final node. A region denotes a set of nodes and edges with a distinguished node called an entry node to the region and another distinguished node called an exit node from the region. In other words, a region is a sub graph of a graph between two distinct nodes. We formally define a region in an activity diagram as follows.

Definition 2 A region $R = \langle A_1, E_1, x_1, y_1 \rangle$ of an activity diagram, $G = \langle A, E, in, F \rangle$ is a subgraph with two distinct nodes $x_1 \in A1$ and $y_1 \in A1$ where $A_1 \subseteq A$, $E_1 \subseteq E$ and

- x_1 is an entry node in A_1 that dominates all other nodes in A_1.
- y_1 is an exit node in A_1 that post-dominates all other nodes in A_1.

The two requirements in this definition ensures that every path from the initial node into the region passes through the entry node and every path from inside the region to a final node passes through the exit node.

For every node n in the region, entry node dominates n and exit node post-dominates n and hence the region is connected. It may be noted that one or more regions may be contained in a region. Suppose $R_1 = \langle A_1, E_1, x_1, y_1 \rangle$ and $R_2 = \langle A_2, E_2, x_2, y_2 \rangle$ are two regions of G. We say that R_2 contains R_1 denoted as $R_1 < R_2$ if for every node in the region R_1, the entry node x_2 dominates and exit node y_2 post dominates where $A_1 \subset A_2$ and $E_1 \subset E_2$. Otherwise, we say that the regions are distinct and denote it as $R_1 \not< R_2$.

An activity diagram can be arbitrarily complex where control constructs may be nested containing many sub constructs. To be able to uniquely distinguish these constructs, we identify every basic (non-nested) construct and define it as the smallest region known as a *minimal* region. This allows us to express nested constructs in terms of basic types without introducing any more new types unlike the existing approaches [7,9]. We define a minimal region as given below.

Definition 3 A region R with two distinct nodes, namely entry and exit nodes in it is called a minimal region if and only if it does not contain any other region with two distinct nodes as entry and exit nodes. Or in other words, a minimal region in a graph G is a subgraph of G with a single entry node and a single exit node.

It may be noted that a minimal region is corresponding to a basic control construct (that is, without nesting) and entry and exit nodes are two control nodes in the construct.

In the following, we introduce basic control constructs in a typical activity diagram. We denote each control construct as a minimal region with a distinguished entry node and exit node.

2.2.1 Loop constructs

A loop in an activity diagram can be distinguished as a set of all nodes involved in iterative computation. It is a strongly

connected region with two distinguished nodes—entry node x and exit node y and a distinguished edge (y, x) known as back edge. Depending on whether the decision section precedes or follows the loop section, we classify a loop into two types: *pre-test* and *post-test* loops. For a pre-test loop, a decision node forms the entry node into the region and for a post-test loop, a decision node forms an exit node.

Definition 4 A loop is a strongly connected minimal region $\langle L = A_1, E_1, x_1, y_1\rangle$ of an activity diagram $\langle G = A, E, in, F\rangle$ where $A_1 \subseteq A$ represents a set of all nodes involved in the iteration. E_1 represents a set of directed edges between nodes in A_1. The node $x_1 \in A_1$ forms the entry node and the node $y_1 \in A_1$ is the exit node such that

- if $x_1 \in DN$ then the region denotes a pre-test loop.
- if $y_1 \in DN$ then the region denotes a post-test loop.

2.2.2 Selection constructs

A selection construct in an activity diagram can be identified as a set of all nodes involved within a decision node and a merge node. It can thus be identified as a region with decision node as the entry node and merge node as the exit 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 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 $0 \leq K \leq N$. We term a selection 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 we call it as *K-out-of-N selection*. Further, there may not be a merge node paired with a decision node. In other words, flows from a decision node may not converge into any merge node. We refer to this situation as *unmatched selection*. In the following, we define a selection construct.

Definition 5 A selection construct is a minimal region $S = \langle A_1, E_1, x_1, y_1\rangle$ of an activity diagram $\langle G = A, E, in, F\rangle$ where $A_1 \subseteq A$ represents a set of all nodes in the selection construct and E_1 denotes a set of edges between nodes in A_1. The entry node $x_1 \in A_1$ is the decision node with N outgoing edges from it and exit node, $y_1 \in A_1$ is the merge node with K incoming edges into it such that

- if $K = 0$ then the region denotes an unmatched selection.
- if $K = N$ then the region denotes a matched selection.
- if $0 < K < N$ then the region denotes a K-out-of-N selection.

2.2.3 Fork constructs

A pair of fork-join in an activity diagram is treated as a concurrent construct. 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. The concurrent flows coming out of a fork may also be merged rather than synchronized. In that case, a merge node is placed instead of a join node which allows multiple flows, one for each incoming flow arriving at a merge node specifying an "OR" action. Depending on the number of flows converging, a fork can also be classified in the same way as a selection construct. That is, for some $N, N \neq 0$ there are N outgoing flows from a fork node, a merge or join node may be used to converge $K, 0 \leq K \leq N$ flows where K denotes incoming flows to a merge or join node. We say a fork is *matched fork* if all outgoing flows from a fork node can be matched with each incoming flow of a join or merge node, that is, $K = N$. Further, a matched construct is said to be a synchronized fork if a join node is used to converge outgoing flows of a fork node, otherwise it is termed as a merged fork construct. If $K < N$, then we call it as *K-out-of-N fork*. If there is no converging node at the end of these flows, that is, $K = 0$, then we call it as *unmatched fork*. The fork construct is precisely defined as below.

Definition 6 A synchronized (merged) fork construct is a minimal region $X = \langle A_1, E_1, x_1, y_1\rangle$ of an activity diagram $\langle G = A, E, in, F\rangle$ where every node in $A_1 \subseteq A$ represents a set of all nodes involved in the concurrency and E_1 denotes a set of edges between nodes in A_1. The entry node $x_1 \in A_1$ is the fork node with N outgoing edges from it and exit node $y_1 \in A_1$ is the join (merge) node with K incoming edges into it such that

- if $K = 0$ then the region denotes an unmatched fork.
- if $K = N$ then the region denotes a matched fork.
- if $0 < K < N$ then the region denotes a K-out-of-N fork.

Example Figure 1a–i shows examples for various basic control constructs defined in this section. Each example depicts a minimal region constituting a single construct in which node labels with $B_1, B_2, \ldots$, etc. denote nodes that are enclosed in the region and node labels with A_i and A_k refers to nodes that are not in the region. Here, action nodes are shown with node labels A_i, A_k and $B_1, B_2, \ldots$ For denoting decision and merge nodes, D_1 and M_1 are made use as node labels. Similarly, FN_1 and JN_1 denote fork and join nodes, respectively.

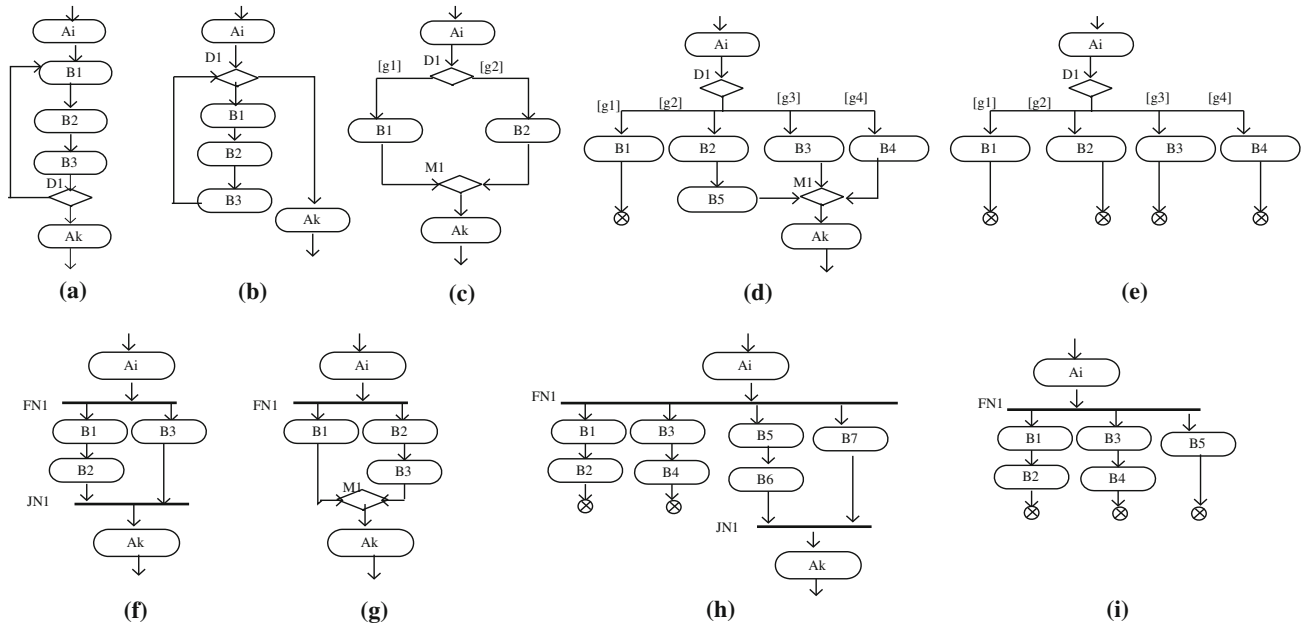


Fig. 1 Control constructs in an activity diagram. **a** Post-test loop, **b** pre-test loop, **c** matched selection, **d** K-out-of-N-selection, **e** unmatched selection, **f** synchronized fork, **g** merged fork, **h** K-out-of-N fork, **i** unmatched fork

2.3 Intermediate testable model (ITM)

As far as scenario generation from an activity diagram is concerned, we need to associate a scenario corresponding to a flow of execution. However, it is not straightforward to interpret execution order directly from an activity diagram. In this section, we begin with our structured representation and then detail the problems that are faced while interpreting execution order directly from an activity diagram.

2.3.1 Structure of ITM

In order to unambiguously interpret the semantics of flow of actions, we propose a testable representation from the activity diagram, which we call ITM.

Let us denote distinct control constructs of an activity diagram as $C = \{C_1, \ldots, C_n\}$ where each C_i is a minimal region that corresponds to a single control construct or a nested control construct enclosing several regions $C_i = \{R_1, \ldots, R_n\}$. As each minimal region with a single entry and single exit node property can be analyzed independently of each other, we uniquely represent each such region by a special node termed as composite node. This means that every composite node is mapped to a minimal region that corresponds to a basic control construct of an activity diagram. In that case, every minimal region can be represented as a separate graph we term it as *Control Construct Graph* (CCG). The use of composite nodes thus provides a representation for each minimal region. However, in order to retain the nesting relation for a nested control construct of an activity diagram, this mapping is carried out in successive steps. In each step, all R_i's that denote distinct minimal regions are mapped to their respective composite nodes. Depending on the nesting relationship among the regions, we say that a composite node R_i encloses zero or more composite nodes. An outer construct which contains inner construct is mapped to a composite node by enclosing composite node corresponding to the inner construct.

The nodes of ITM can be distinguished into two classes of nodes called *basic nodes* and *composite nodes*. A basic node is a node whose internal structure corresponds to that of an activity node. A composite node is a mapping to a control flow construct. Each ITM can therefore be viewed as a concise representation of the activity model which is obtained by replacing all minimal regions with their corresponding composite nodes. Since every control construct is represented by a composite node, an ITM is a chain of nodes $\langle n_1, \ldots, n_k \rangle$ such that for all i, $1 \leq i \leq k-1$, n_i is a predecessor of n_{i+1}. The node n_k may as well be a composite node or a final node of an activity diagram. We define the ITM as follows.

Definition 7 An intermediate testable model, $G_t = \langle N, R, in, f \rangle$ is a chain of nodes where

1. $N = N_B \cup N_C$: a finite set of nodes. Each node in N_B and N_C is a basic node and composite node, respectively.
2. $R \subseteq \{(n_i, n_j) | n_i, n_j \in N, i \neq j\}$: a set of directed edges between two nodes n_i and n_j.
3. $in \in N_B$: the start node representing an initial node of the activity diagram.

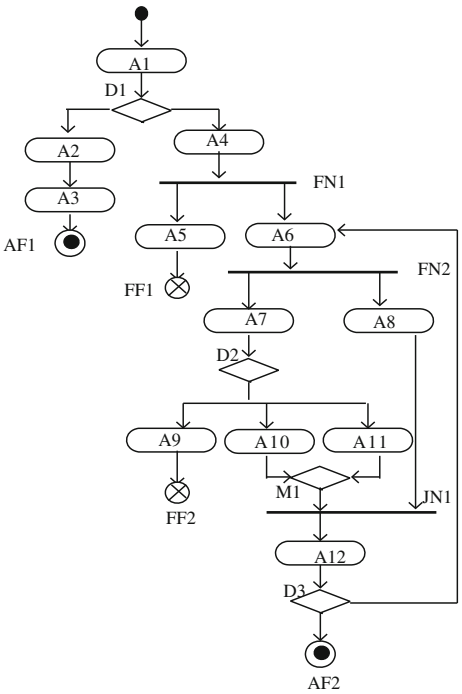


Fig. 2 Example activity diagram

4. $f \in N$: a final node representing a node without any successor node.

We define the structure of each node $N_i \in N$ as $\langle nodeId, nodeType, name, outEdge \rangle$ where

- $nodeId$ is a unique label attached to each node in ITM.
- $nodeType = \{action, initial, flowFinal, activityFinal, decision, merge, fork, join\} for N_i \in N_B$ and $nodeType = \{MatchedSelection, KNSelection, UnmatchedSelection, PreTest, PostTest, SynchronizedFork, MergedFork, UnmatchedFork KNFork\}$ for $N_i \in N_C$
- $name$ corresponds to any textual description attached to a node
- $outEdge = \langle outNode, D_I, D_O, predicate \rangle$ where $outNode$ specifies the $nodeId$ corresponding to successor of each node, D_I and D_O specifies the set of input and output object information attached to each node, and $predicate$ specifies the Boolean expression attached to outgoing edge.

2.3.2 Significance of ITM

In an activity diagram, there can be various combinations of nested constructs in which an innermost construct can be surrounded by different types and combinations of outer constructs. For example, there can be a post-test loop followed by a fork construct within a pre-test loop or a pre-test loop within a fork construct etc. Let us consider an example activity diagram as shown in Fig. 2 to understand scenario generation involving nested constructs. Figure 2 shows two fork nodes $FN1$ and $FN2$. However, there is only one join node which is marked as $JN1$. Although in this example it appears that the join node $JN1$ may correspond to the most recent fork node $FN2$, it is not clear when exactly concurrency initiated by $FN1$ ends in the absence of its join node. Therefore, the concurrent construct involving fork $FN1$ we call it as unmatched construct. In addition, there is an exit path from each of these concurrent constructs shown by two flow final nodes $FF1$ and $FF2$. Again, it is not clear whether paths from the initial node through $FF1$ and $FF2$ should be translated into two scenarios or more? In other words, do we need to interpret early exit paths through concurrent construct the same way as selection since there is another exit path marked as $AF1$? If not, how do we distinguish different kinds of exit paths? Now, consider the loop with node $A6$. Again, it is not clear whether nodes $A5$ and $A9$ form part of the loop or not. More precisely, what are the nodes contained in this loop? In fact, the loop may be interpreted differently by different people.

From the above discussion it is clear that it is difficult to interpret activity diagrams directly. In order to avoid ambiguous interpretations, it is needed to provide automated methods for analyzing complex dependencies that arise within nested structures. It makes sense, therefore, to identify each of the constructs and map it to a well structured model before generating test cases. Thus, there is a need for their transformation into a structured representation. The proposed approach is to identify such regions and transform it into a structured representation known as ITM. It may be noted that an ITM represents the essential structure of an activity diagram indicating a high level flows of control. Each composite node within an ITM encloses a control construct. Consequently, depending on the type of control construct enclosed, a composite node calls for further analysis. Such a detailed analysis can be based on the CCG. In summary, ITM is an intermediate representation of an activity diagram and is meant for interpreting the semantics of flow of controls unambiguously and hence for the generation of test scenarios.

Example Let us consider the example activity diagram of Fig. 2 to illustrate nesting of composite nodes in a nested construct. We denote control constructs of an activity diagram as $C_1 = \{S1, X1, L1, X2, S2\}$. In the following, we refer composite nodes by the minimal region it is mapped to. We say that composite node $S2$ is an outer construct as it encloses all of the inner composite nodes. Figure 3 depicts the composition procedure steps during ITM construction. The final ITM is shown in Fig. 3e. For each identified region, Fig. 4 shows the mapping of composite nodes pictorially in the order of their creation. Figure 4a refers to the composite

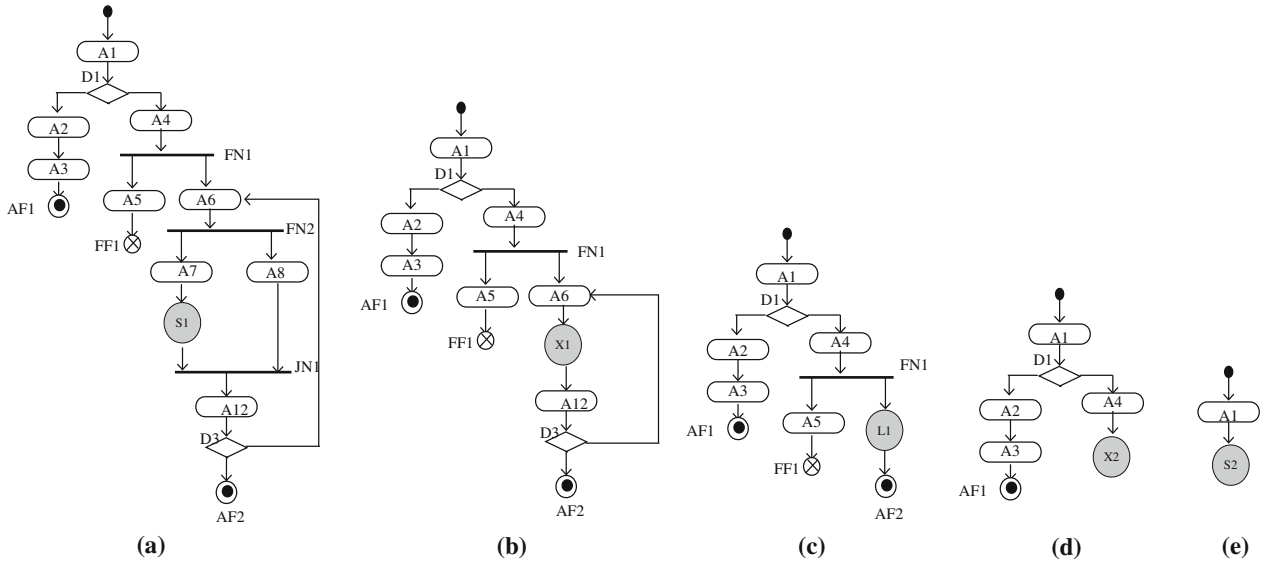


Fig. 3 Minimal region identification steps. **a** Step 1 – minimal region S1, **b** Step 2 – minimal region X1, **c** Step 3 – minimal region L1, **d** Step 4 – minimal region X2, **e** Step 5 – minimal region S 2

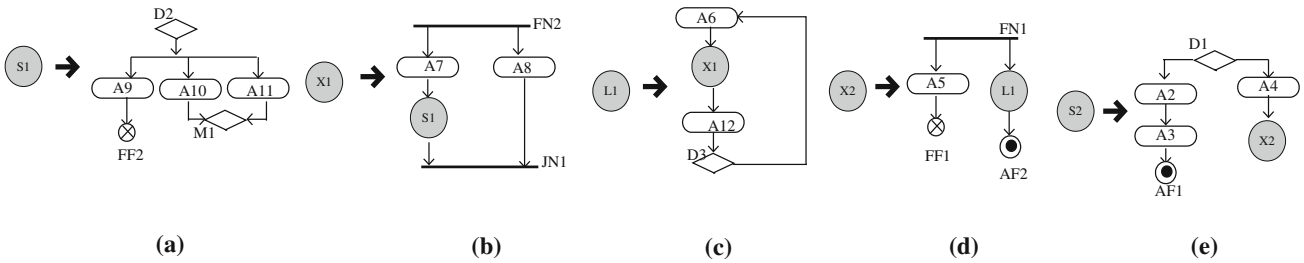


Fig. 4 Composite nodes of example activity diagram **a** S1 mapped to a selection region, **b** X1 containing S1, **c** L1 containing X1, **d** X2 containing L1, **e** S2 containing X2

node *S*1 mapped to a CCG which is a selection region. The subsequent mapping encloses composite nodes corresponding to inner constructs as shown in Fig. 4b–e.

3 STAD methodology

In this section, we present our approach for generating test scenarios from an activity diagram (AD). We name our proposed methodology as *STAD* where *STAD* stands for Synthesis of Test scenarios from Activity Diagrams. Figure 5 gives an overview of our proposed methodology. In the first step of our synthesis approach, all model elements, such as action nodes, control nodes, object nodes along with their corresponding edge information are retrieved. In the next step, the ITM generator uses the retrieved data and transforms them into a testable model called ITM. Subsequently, the test scenario generator generates test scenarios from the ITM. Details of these steps are discussed in the following sub sections.

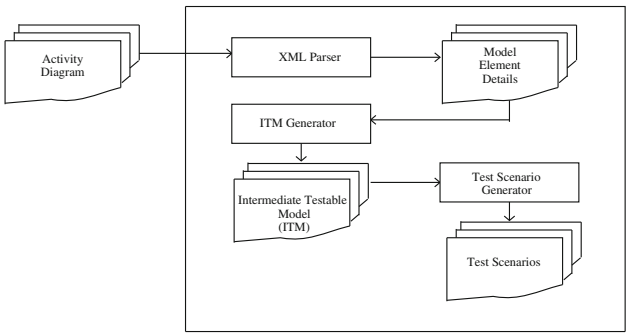


Fig. 5 An overview of STAD

3.1 Capturing model element details

We consider an activity diagram as input to our test scenario synthesis methodology. The activity diagram is modeled using UML 2.0 design specifications. We develop an activity diagram using a CASE tool and export it into a XML file. It can be composed by a designer using a case tool and be stored in XML format as in [8]. Depending on the type

of tags, different model elements e.g. action, control, object nodes and edges between nodes are extracted. For this purpose, a XML parser is considered. This parser stores the above mentioned information in a graphical representation. This graph representation, in fact, depicts the connectivity between different nodes. We use this intermediate form of the parsed activity diagram in our subsequent task. This representation is in accordance with Definition 1 and we refer it as scenario graph G.

3.2 Building an ITM

In this sub section, we discuss our approach of creating an intermediate testable model from the scenario graph G. We begin with identification of different control constructs and their composition. Then, we present transformation procedure of scenario graph to ITM. We also discuss about a few properties of transformation procedure.

3.2.1 Identifying control constructs and creating their compositions

Our objective is to identify all minimal regions and reduce them to their corresponding *composite nodes*. Such a procedure of replacing the minimal regions by a composite node, we term as *composition*. We refer the composite nodes corresponding to selection, loop and concurrent constructs as selection nodes, loop nodes and concurrent nodes, respectively. There are two main tasks in our composition: identifying different types of basic constructs and creating respective composite nodes for each of these types. In the following, we discuss our approach to the compositions of various types of constructs.

Identifying loop constructs and creating their compositions A loop in an activity diagram consists of a set of activities that can be executed repeatedly. It is modeled through a decision node. There exist two branches that leave a decision node. Among these, there is a branch that exit the loop by connecting the branch to the node outside the region. Similarly, there is a branch that continues looping by connecting the branch to the node inside the region. A loop can be modeled using the decision node as the first node in the region which is known as an entry node (see Definition 4). Such a loop we term as *pre-test* loop. The decision node can also appear as the last node in the region forming an exit node. Such a loop is known as a post-test loop. For each of the loop types, we create composite nodes as discussed below.

Suppose x is a decision node and there is another node y which is the predecessor node of x. If there exist a branch b emanating from x which follows a sequence of nodes and finally merge into the node x, then we say that node x is the decision node corresponding to a loop. If the predecessor node y is on the branch b, then it is a post-test loop else pre-test loop. If it is a pre-test loop then x is the entry as well as exit node of the loop region. On the other hand, if it is a post-test loop, then the node immediately after the node x on the branch b is the entry node and x is the exit node. Once the entry and exit nodes of a loop region is identified, the set of nodes and edges within the entry and exit nodes (both inclusive) are recognized as a region called control construct graph of a loop. we replace this region in the intermediate ITM with a composite node (for example, see Fig. 6). We attach type information to the composite node as pre-test or post-test loop accordingly.

The example in Fig. 6a uses a single control construct modeling a post-test loop. During loop identification phase, entry and exit nodes are detected which are shown to be nodes Bi and $D1$, respectively. Accordingly, a composite node is created which is marked as the node $L1$. The nodes and edges comprising the loop construct form a control construct graph and are associated to node $L1$. By connecting node $L1$ with control flow through entry and exit edges, we build ITM as shown in Fig. 6c. The edges connecting node Ai to $L1$ and $L1$ to Ak are marked as entry and exit edges in Fig. 6c. Figure 6b depicts a pre-test loop and its composition.

Identifying selection constructs and creating their compositions A selection is used to model two or more mutually exclusive alternative paths. It is depicted by using a decision node with two or more outgoing edges (see Definition 5). Here, the execution order is based on the value of conditional expression. For each value that can be evaluated, an edge is originated from the decision node. Based on the number of outgoing edges, a selection construct can lead to two-way or multi-way selections. A two-way selection represents *if-then-else* logic by capturing nodes that are executed when

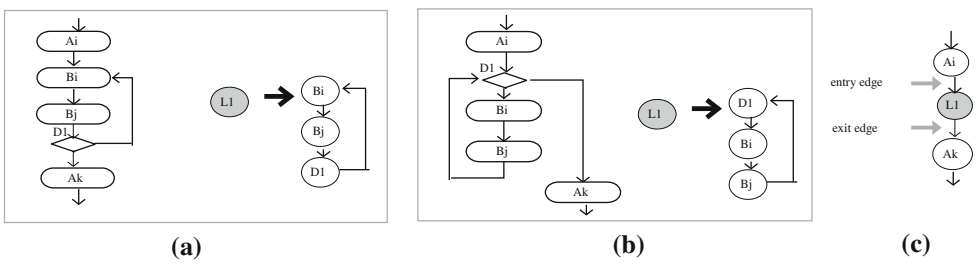


Fig. 6 Creating composite node for post-test and pre-test loops. **a** A post-test loop and its composition, **b** a pre-test loop and its composition, **c** an ITM

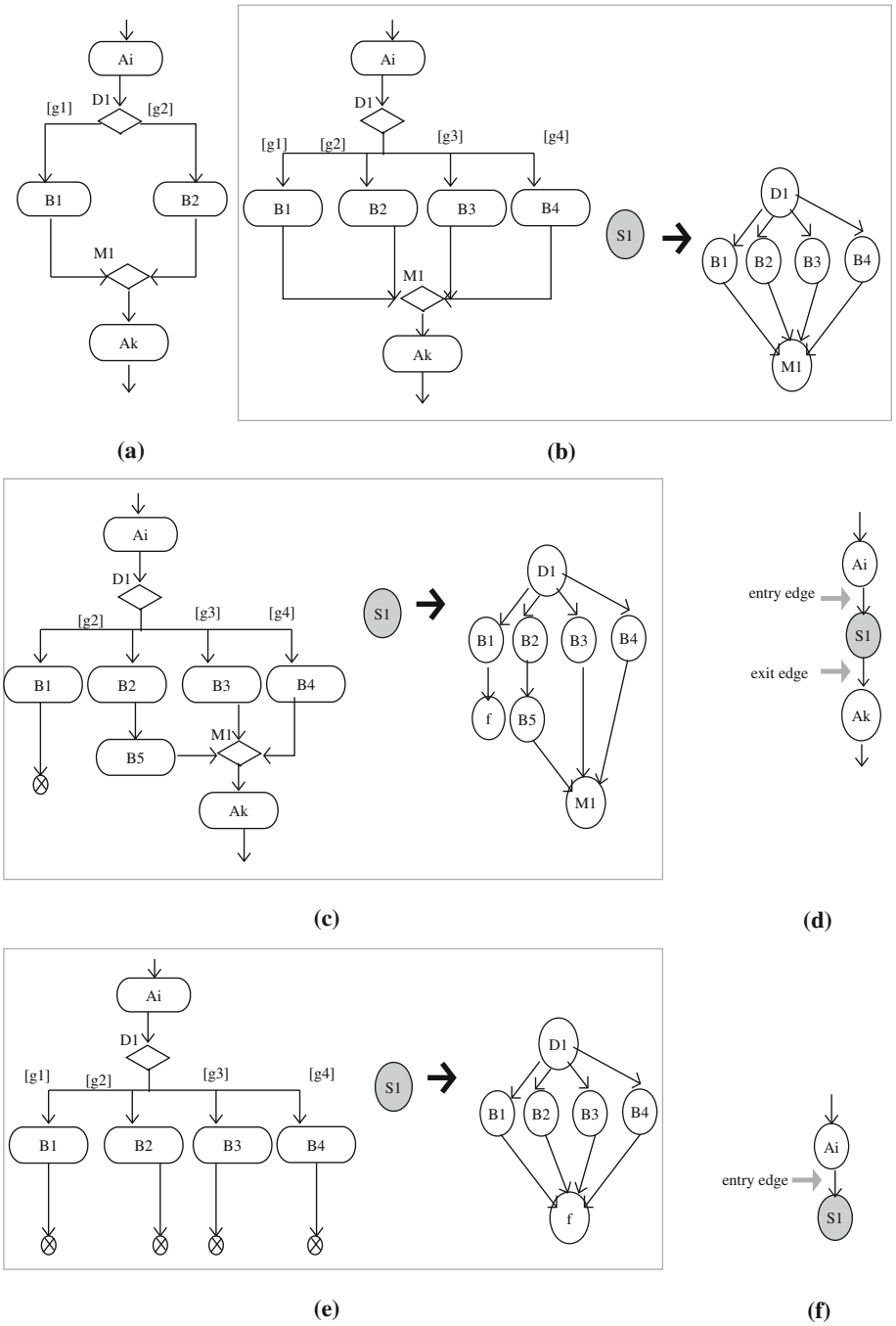


Fig. 7 Creating composite nodes for different types of selections. **a** A two-way selection, **b** a multi-way selection and its composition, **c** a K-out-of-N selection and its composition, **d** an ITM for K-out-of-N and matched selection, **e** an unmatched selection and its composition, **f** an ITM for unmatched selection

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 multi-way selection. Since a selection can be of *matched*, *unmatched* or *K-out-of-N* type, their composition is addressed separately as given below.

Case 1: Matched selection

Suppose x is a decision node. If there exists a merge node such that all branches emanating from the node x merge into x then we say that x is the decision node corresponding to a matched selection construct. In this case, x is the entry node and the merge node is the exit node. A set of nodes and edges between the entry and exit nodes both inclusive constitutes a control construct graph for a matched selection construct. This control construct graph is then replaced in the intermediate ITM by a composite node. The *type* information is attached to the composite node indicating that it represents a matched selection construct. Figure 7b shows the region enclosed by multi-way selection and its corresponding ITM

is shown in Fig. 7d. In ITM, the multi-way selection region is reduced to a composite node marked as $S1$.

Case 2: Unmatched selection

In case of unmatched selection, there is no converging node at the end of the control flow. However, the control flow represented by this region is reduced to its composite node by knowing that each of these branches is terminated at some point. We use this information to create composite node for an unmatched selection construct. Let x be a decision node. If all branches emanating from x through a sequence of nodes are terminated at flow terminal nodes, then we recognize the control construct starting at node x as an unmatched selection construct. In this case, x is the entry node and each flow terminal node is exit node of the control construct graph. We replace the control construct graph by a composite node and the type of node is tagged as unmatched selection. Figure 7e depicts an unmatched selection, where each of the branches from the decision node initiates an activity and then terminate. A composite node of type unmatched selection is then created and made to associate with this region as shown in Fig. 7e. The final node, f is used here to indicate end of flow of control. As each of the branches is terminated, subsequently there are no more activities taking place. This is reflected by the composite node, $S1$ without any exit edge as shown in Fig. 7f.

Case 3: K-out-of-N selection

In this case, K branches are merged and remaining $N-K$ are unmatched 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 denote unmatched selection with implicit final node as exit node. We compose a region corresponding to matched and unmatched parts following the Case 1 and Case 2 of the selection construct, respectively. Figure 7c and d shows a situation where 3 out of 4 branches are merged. Figure 7d shows the composite node creation of type K-out-of-N selection and their subsequent transformation to ITM.

Identifying concurrent constructs and creating their compositions A fork node is used to model two or more parallel execution paths in an activity diagram. These parallel paths may require to be combined into a single flow. For this purpose, a connecting node such as join or merge is used. The choice of which node to connect is often made based on the design requirement. A join node is used when it is required to wait for the completion of these parallel paths before proceeding. We refer to this construct as a *synchronized fork* construct (see Definition 6). Sometimes, it may not be required to wait for all paths to complete. A merge node is used for this purpose such that when a path arrives at the merge node, it is continued with the path following merge without any waiting. We call this construct as *merged fork* construct. There can also be special situations when some or all of these paths end at a flow final node. Accordingly, a fork may result into *K-out-of-N matched fork* or *unmatched fork*. We discuss our approach to create composite nodes for each of the above mentioned concurrent constructs in the following.

Case 1: Matched fork

To deal this case, we start with a fork node. Let the fork node be x. If all the branches starting from x and through a sequence of nodes merge into a join node (or a merge node) say y, then we identify the control construct as a matched fork construct. The fork node x is the entry node and the node y is the exit node. The region between x and y both inclusive is the control construct graph of the matched fork construct. We replace the control construct graph in the intermediate ITM by a composite node and *type* of node is marked as matched fork. Figure 8a and b shows a fork matched with a join node and a fork matched with a merge node, respectively. The composite node of type merged fork is associated to control construct graph as shown in Fig. 8b. In this way, the ITM is built as shown in Fig. 8g. The composition procedure can be extended to a *synchronized fork* in the similar way.

Case 2: Unmatched fork

The different execution paths emerging from a fork may not converge to a merge node (or join node). Instead, each one of them may terminate. We call this situation as *unmatched fork*. We identify an unmatched fork construct and perform its composition as follows. Suppose, x is a fork node. If all branches emanating from x and through a sequence of nodes ended at flow final nodes, then we identify that x is an entry node and all flow final nodes are exit nodes of a region corresponding to unmatched fork construct. In place of multiple exit points, we reduce to a single exit point by eliminating all except one and redirecting all edges to flow final nodes to that exit point (for example, see Fig. 8c). A set of nodes and edges from the entry node to the exit node both inclusive constitute a control construct graph corresponding to the unmatched fork construct. We replace the control construct graph in intermediate ITM to a composite node and mark the *type* of the composite node as unmatched fork. Figure 8c shows an unmatched fork construct and its composition, where $FN1$ is the entry node and f is the exit node. The ITM with the composition is shown in Fig. 8d.

Case 3: K-out-of-N matched fork

There may be a situation where a fork construct may not be completely unmatched. It happens when only some of the paths end with final nodes and the rest may be connected with merge (or join node). We consider this case as combination of above two cases and we refer it as K-out-of-N matched fork. In this case, K paths are converged using merge (or join).

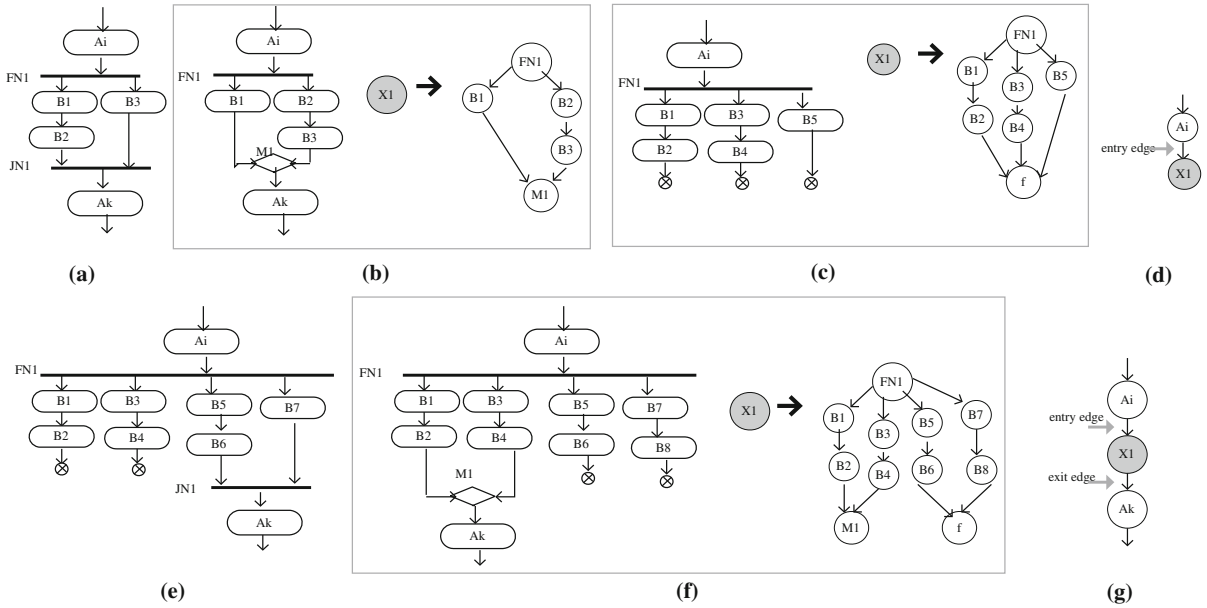


Fig. 8 Creating composite nodes for different types of forks. **a** Synchronized fork, **b** merged fork and its composition, **c** unmatched fork and its composition, **d** an ITM for unmatched fork, **e** K-out-of-N join fork, **f** K-out-of-N merge fork and its composition, **g** an ITM for K-out-of-N and matched fork

The remaining paths are unmatched. We compose a control construct graph corresponding to matched and unmatched parts following the Case 1 and Case 2, respectively of the fork construct. Figure 8f shows an example of K-out-of-N matched construct. Its composition into ITM is shown in Fig. 8g.

3.2.2 Transforming scenario graph to ITM

In this sub section, we discuss how we transform a scenario graph G to an ITM by means of compositions.

To do the transformation, the graph G is traversed looking for minimal regions enclosed by its entry and exit node. For each such minimal region in G, it is replaced by its composite node through the process of compositions as discussed in Sect. 3.2.1. After compositions, the immediate predecessors of composite node are the immediate predecessors of entry nodes and the immediate successors of composite nodes are the immediate successors of exit nodes. Thus a new graph G_1 is formed. Now again, graph G_1 is traversed looking for minimal regions and compositions are made like in the previous step. The procedure is continued in this way, until no more composition is possible. Thus, a sequence of graphs $\{G_1, G_2, \ldots, G_k\}$ are derived in k iterations, where $k \geq 0$. We say that graph G_k is the ITM.

3.2.3 Properties of the transformation procedure

In this section, we state some properties of the transformation procedure.

Let us denote the collection of transformation rules mentioned in Sect. 3.2.1 as S. Each rule s consists of the left-hand side $L(s)$ and right-hand side $R(s)$. Each $L(s)$ is uniquely defined with its entry node, exit node and edge characteristics (see Definitions 4–6). The right-hand side of the rule $R(s)$ is a composite node. After a transformation rule is applied to a graph G, each minimal region in G is reduced to a single node. In this context, we define applicability as given below.

Applicability A transformation rule is applicable if and only if the given graph contains a minimal region. Otherwise, the transformation rule is not applicable.

The transformation procedure outlined in Sect. 3.2.1 is applied successively to the input graph G until there exists a minimal region. The resulting graph say G_1 is unique, no matter in which of the order, the rules are applied. This property is known as *confluence* which means that the order of application of transformation rules does not influence the resulting graph ITM.

Confluence If transformation rules are applied to a given graph until no longer possible, then a unique graph results which is independent of the sequence of applications chosen.

We formulate the confluence as Finite Church Rosser (FCR) property due to Hecht and Ullman [14] and Sethi [27].

FCR property Let H be a set and $\Rightarrow$ be a binary relation on H. A pair $(H, \Rightarrow)$ is Finite Church Rosser if

(i) for each p in H, there is a constant $k \geq 0$ such that if $p \overset{i}{\Rightarrow} q$, then $i \leq k$. That is, there is a bound on

the number of applications of $\Rightarrow$, beginning with any element p (Finiteness property) and

(ii) for all p in H, if $p \Rightarrow p_1$ and $p \Rightarrow p_2$, then there is some q such that $p_1 \overset{*}{\Rightarrow} q$ and $p_2 \overset{*}{\Rightarrow} q$ (Commutativity property). Here $\overset{*}{\Rightarrow}$ denotes the reflexive transitive closure of $\Rightarrow$.

Let $G \in H$ be a scenario graph corresponding to an activity diagram. Let us define the relation $\overset{s}{\Rightarrow}$ by $G_1 \overset{s}{\Rightarrow} G_2$, if and only if, G_1 can be transformed into G_2 by one application of the transformation labelled s. Here, $s \in S$ denotes name of the transformation rule that corresponds to a control construct defined in Sect. 2. We refer to specific transformation by the labels in S where $S = \{L_{pre}, L_{post}, S_M, S_{UM}, S_{KN}, F_M, F_{UM}, F_{KN}\}$ denotes name of eight transformation rules. Let $\Rightarrow$ denote the union of these eight transformations.

Theorem 1 $(H, \Rightarrow)$ *is confluent.*

Proof Consider a scenario graph $G \in H$ with n nodes that corresponds to an activity diagram. We begin with finiteness property.

Each of the rule eliminates atleast two nodes. Hence, each of these rules may be applied to a maximum of $n-1$ times since initial node is unchanged. That is, k may be taken as $n-1$ and therefore, $\Rightarrow$ is finite.

To test for commutativity property, there are eight distinct transformation rules. Let us begin with all possible combinations of any two rules. Let us arbitrarily choose post-test loop L_{post} and matched selection S_M. In a graph, G, we may find application of either one of these choices: L_{post} or S_M or both L_{post} and S_M. Note that, if none of these choices are applicable, then the property is trivially established. There are k iterations in our transformation procedure which may be denoted as $G \overset{*}{\Rightarrow} G_1 \overset{*}{\Rightarrow} G_2 \overset{*}{\Rightarrow} \cdots \overset{*}{\Rightarrow} G_k$. In the following, we prove confluence of $G \overset{*}{\Rightarrow} G_1$, by considering these three cases.

Case 1: Only L_{post} is applicable.

Suppose L_{post} is applied to graph G to yield G_1. If L_{post} is applicable to G only once, then $G \overset{L_{post}}{\Rightarrow} G_1$ yields a graph G_1. In practice, we need to deal with more than one instances of L_{post} control construct and hence L_{post} transformation rule. Suppose there are two instances of L_{post} rules to a graph G known as L_{post_1} and L_{post_2}. Suppose L_{post_1} is applied to G to yield g_1 and L_{post_2} is applied to G to yield g_2, respectively. That is, $G \overset{L_{post_1}}{\Rightarrow} g_1$ and $G \overset{L_{post_2}}{\Rightarrow} g_2$ is applied to G in different orders. Clearly, if we apply transformations L_{post_2} to g_1 and L_{post_1} to g_2, we get the same graph G_1. That is, $g_1 \overset{L_{post_2}}{\Rightarrow} G_1$ and $g_2 \overset{L_{post_1}}{\Rightarrow} G_1$. This is because the new graphs g_1 and g_2 have their right-hand side of the rule in place of the left-hand side. Simultaneous composition of both inner and outer construct starting from graph G is ruled out according to our transformation procedure. Hence, a nested instance of L_{post} in which an outer construct enclosing an inner construct cannot hold good because, in that case we apply the rules in two different iterations. This means that the rules are applied to distinct regions, R_1 and R_2 where both $R_1 \not\prec R_2$ and $R_1 \not\prec R_2$. Hence, $A_1 \cap A_2 = \emptyset$ and $E_1 \cap E_2 = \emptyset$ where A_i and E_i are the set of nodes and edges in the left-hand side of ith application of L_{post_i}. This means that transformations do not interfere with each other. Hence, any applications of L_{post} results in the same graph.

Case 2: Only S_M is applicable.

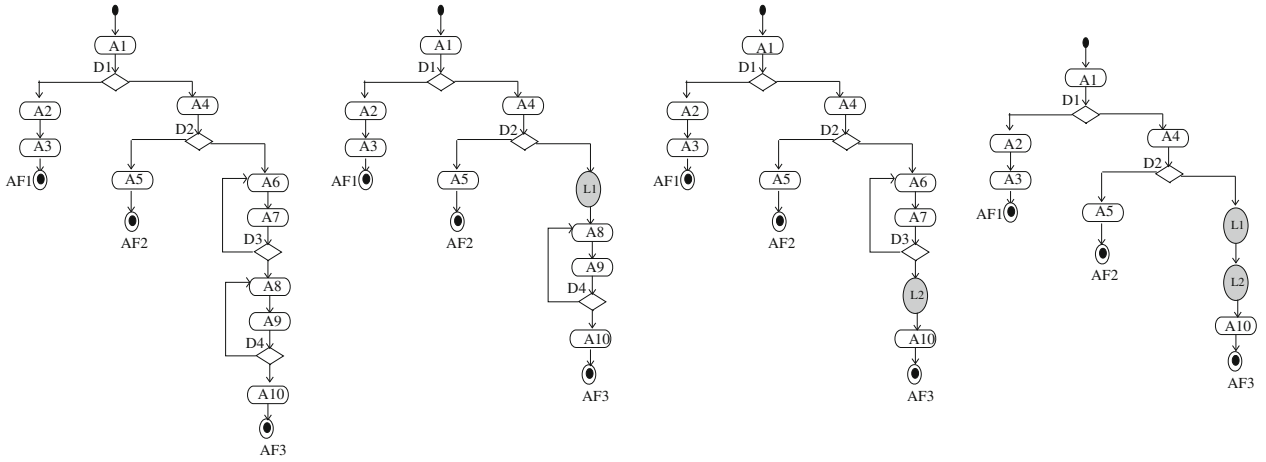
Applicability of S_M can be proved in the same way as outlined above. For a single instance of rule S_M, applying S_M yields the graph G_1. We assume two instances of matched selection construct, namely S_{M_1} and S_{M_2} in the graph G to establish their confluence. Let us apply the rule S_{M_1} and S_{M_2} to G. Let the resulting graphs be g_1 and g_2. That is, we have derived g_1 and g_2 by $G \overset{S_{M_1}}{\Rightarrow} g_1$ and $G \overset{S_{M_2}}{\Rightarrow} g_2$. Since, we assumed two instances of S_M, now let us continue with applying the other rule. There can be no other transformation rule applicable because we assumed only rule S_M to be applicable. In addition, applying a rule once means that the derived graph is same as the original graph except that the minimal region is transformed to a composite node. This means that result of applying S_{M_2} to g_1 and S_{M_1} to g_2 must derive the same graph G_1. Thus, $g_1 \overset{S_{M_2}}{\Rightarrow} G_1$ and $g_2 \overset{S_{M_1}}{\Rightarrow} G_1$.

Case 3: Both S_M and L_{post} are applicable.

In this case, let us assume that there is one instance each of a post test loop and a matched selection in G. Let the graph g_1 be the result of applying L_{post} to G and g_2 be the result of applying S_M to G. Now, applying S_M to g_1 and applying L_{post} to g_2 cannot interfere since nested instance is ruled out. In addition, no other changes to g_1 is possible because, only the rule S_M is applied to g_1. For the same reason, g_2 is also same as G except that the minimal region corresponding to L_{post} is replaced by a composite node. Since, we assumed both instances of S_M and L_{post}, it means that rule S_M to $g_1 L_{post}$ to g_2 are still applicable as there are no other changes possible. Hence, $g_1 \overset{S_M}{\Rightarrow} G_1$ and $g_2 \overset{L_{post}}{\Rightarrow} G_1$ holds.

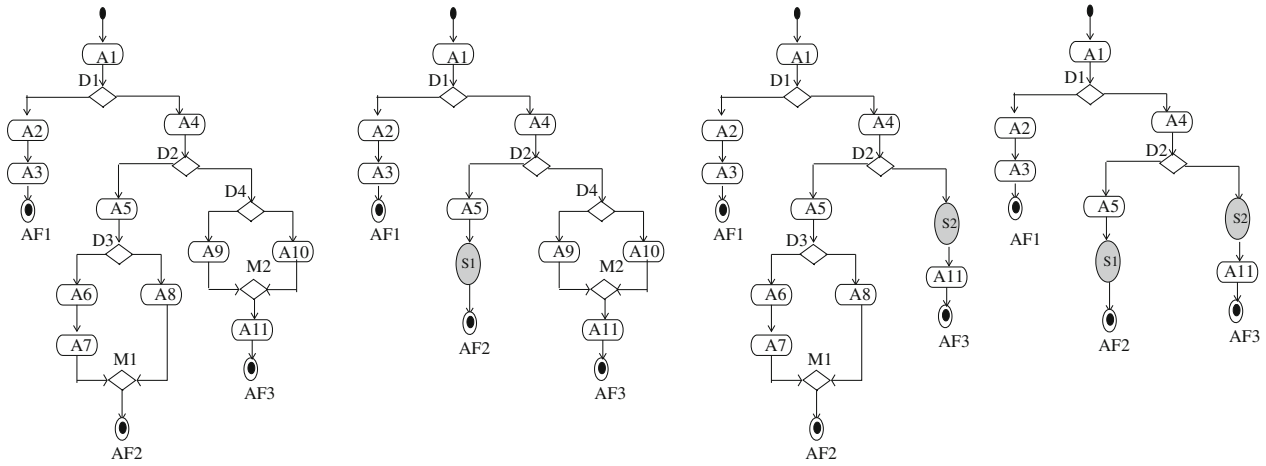
This includes possible three cases. We have shown that any possible order of applications results in the same graph G_1 beginning with $G \in H$. Since identical steps are carried out in $G_i \overset{*}{\Rightarrow} G_{i+1}$, the result is same no matter in which of the order, the rule is applied. Hence, confluence property for $G(G_0) \overset{*}{\Rightarrow} G_1 \overset{*}{\Rightarrow} G_2 \overset{*}{\Rightarrow} \cdots \overset{*}{\Rightarrow} G_k$ is established, where $G_i \in H$.

Example We illustrate the property of confluence with an example. Figure 9a illustrates Case 1 in which rule L_{post} is



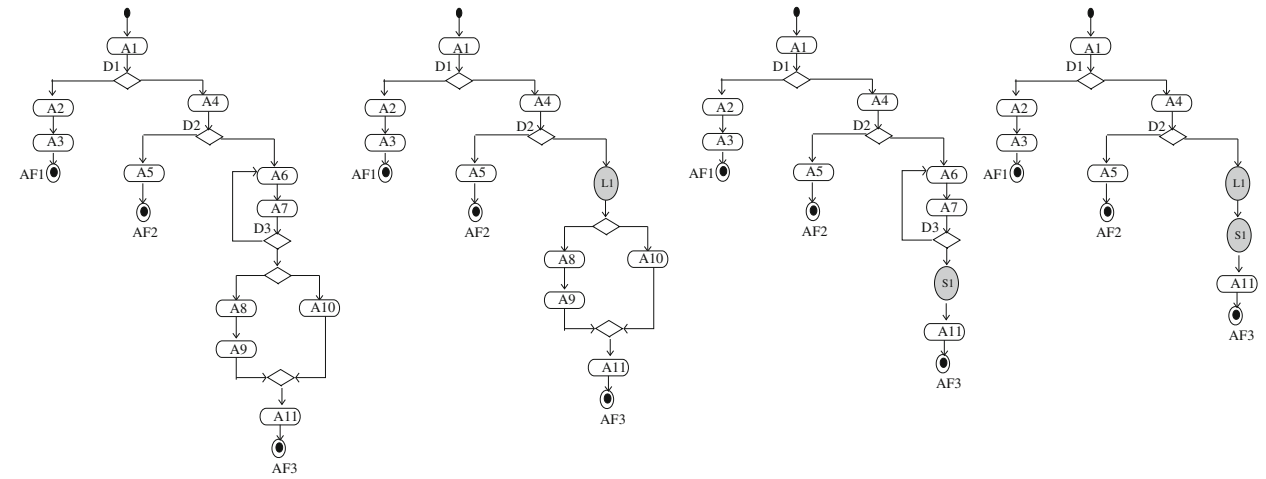
(i) Initial graph, G (ii) Graph g1 by applying post test loop L1 (iii) Graph g2 by applying post test loop L2 (iv) Graph G1

(a)



(i) Initial graph, G (ii) Graph g1 by applyiyng matched selection S1 (iii) Graph g2 by applyiyng matched selection S2 (iv) Graph G1

(b)



(i) Initial graph, G (ii) Graph g1 by applyiyng post test loop L1 (iii) Graph g2 by applyiyng matched selection S1 (iv) Graph G1

(c)

Fig. 9 Confluence of transformation rules. **a** Proof for case 1; **b** proof for case 2; **c** proof for case 3

applied. Figure 9a (i) shows graph G to which transformation rule L_{post_1} is applied. Graph g_1 in Fig. 9a (ii) shows the result of rule L_{post_1}. In the similar way, Fig. 9a (iii) shows $G \overset{L_{post_2}}{\Rightarrow} g_2$. Finally, graph G_1 in Fig. 9a (iv) shows that $g_1 \overset{L_{post_2}}{\Rightarrow} G_1$ and $g_2 \overset{L_{post_1}}{\Rightarrow} G_1$. Figure 9b illustrates Case 2 in which the applicable rule is S_M. In the same way, Fig. 9c (i) shows graph G to which two different rules are applied as per Case 3. Graph g_1 in Fig. 9c (ii) is the result of applying L_{post} to G. Next, the rule S_M is applied to graph G and the result g_2 is shown in Fig. 9c (iii). By applying S_M to g_1 and L_{post} to g_2 results in graph G_1 in Fig. 9c (iv).

Completeness The proposed transformation procedure is complete in the sense that for every control construct in an activity diagram, there exist a transformation rule.

We establish the completeness property as follows. The proposed approach considers activity diagrams that are reducible to structured control graphs. A structured control graph is a graph in which control structures are block oriented with one entry point and one exit point and the control structures are properly nested [2,31]. In a properly nested control structure, the sub graphs corresponding to control structures are either mutually disjoint or completely contained within one another [3]. On the other hand, an unstructured graph can be produced if the control structures have two or more entry and exit points and if the control structures are not properly nested. The control flow analysis of structured high level languages have demonstrated that edges interacting with structured constructs lead to unstructured graphs [14].

To formalize a structured activity diagram, we apply the concept of flow graph reducibility due to Hecht et al. [14]. We say that the control structure graph of an activity diagram is a structured graph, if and only if, the repeated application of composition reduces an activity diagram to a single chain of nodes. If the result is not a single chain of nodes, then the activity diagram is termed as an unstructured graph. Equivalently, we can say that the repeated application of the composition procedure ensures detection of an unstructured activity diagram.

In order to accomplish completeness, for every control construct in activity diagrams, there exists a transformation rule. A nested construct is reduced by means of repeated composition. Hence, there exists composition for every structured graph which implies completeness of control constructs in activity diagrams. It may be noted that given an activity diagram G, if it cannot be eventually reducible to a single chain of nodes, then a design revision is necessary.

Example We illustrate that the iterative procedure of deriving ITM can reveal insight about the control structure of the input graph G as well as output graph G_k. Since control constructs are distinguished in our approach as matched and unmatched constructs (K-out-of-N is a combination of both), it is needed to handle edges interacting with each case separately. The definition of a region disallows any edges into the middle of a construct from outside the region or an edge from inside the region to a node outside the region. Hence, an edge into a matched region cannot be composed and remain incomposable. However, this is not the same for an unmatched construct. Since all the edges from an unmatched region end with final nodes, the composition approach can always handle an edge into the middle of an unmatched construct. Continuing with our example 0activity diagram of Fig. 2, $FN1$ was identified as an unmatched construct since it lacked its join node. There is also an edge into the middle of the unmatched construct. However, the repeated composition has resolved the unstructuredness. As we see from Fig. 4d, the construct using $FN1$ is resolved as an unmatched region $X2$. Depending on the structure of a loop, a loop can also result into an unstructured loop. If two loops share the same entry node, then the nesting relationship no longer holds with them. The control flow analysis has identified such loops with more than one entries as irreducible loops [15]. Figure 10 presents an illustration depicting different ways in which two loops , $L1$ and $L2$ can be related. From Fig. 10a and b, it can be seen that the nesting or containment relationship (discussed in Sect. 2.2) holds with $L1$ and $L2$ here. It can also be stated that when two loops share the same entry node, the nesting relationship no longer holds with them. Figure 10c shows such a relationship where two loops share the same entry node $B1$. Our approach can identify such unstructured loops.

Assuming that G can eventually be reduced to a single chain of nodes, we deal with the nesting depth as given below.

Nesting Depth Suppose in the graph G, there exists a nested construct in which R_2 is an outer construct containing an inner region R_1 such that $R_1 < R_2$. In this case, region R_1 is

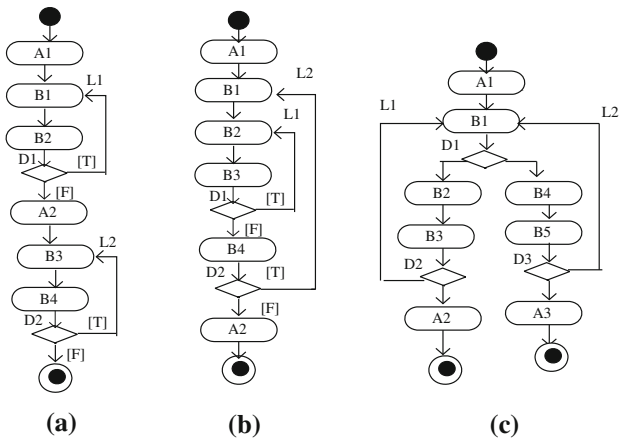


Fig. 10 Types of loop relationships. **a** Disjoint loops, **b** nested loops, **c** irreducible loops

only recognized as the minimal region (see Definition 3). The identified region R_1 is composed into a single node deriving a new graph G_1. Now, in G_1, the region R_2 is the minimal region as the inner region R_1 is already reduced to a single node. In G_1, the outer region is mapped covering the inner region. The composite nodes in final ITM are therefore outer regions and said to have a nesting depth. A nesting depth of a composite node gives the number of iterations required to build ITM. For a nesting depth of k such that $R_1 < R_2 < \cdots < R_k$ requires k iterations to build ITM. In other words, the number of iterations required to build ITM is equivalent to the nesting depth of a composite node. In general, there can be k composite nodes in ITM $\{c_1, \ldots, c_k\}$, each with a nesting depth $\{d_1, \ldots, d_k\}$, $d_i \geq 0$. The nesting depth of zero indicates that the minimal region is the outermost region. Hence, the number of iterations required to build ITM is the maximum nesting depth, that is, $\max(d_1, \ldots, d_k)$ where d_i is the nesting depth of composite node c_i.

3.3 Generating test scenarios from ITM

An ITM is a testable model having behavioral information of the system under test corresponding to a use case. Alternatively, a path in an ITM can be mapped onto a scenario. For brevity in our discussion, we denote a path by enclosing the sequence of nodes it contains in angular brackets. Thus, ITM can be used to formally define the scenario of an activity diagram as given below.

Definition 8 A sequence $S = \langle n_0, n_1, \ldots, n_q \rangle$ is a scenario in an activity diagram, 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 an activity diagram.

An ITM encapsulates all the control constructs in the form of special nodes known as composite nodes. Each composite node, in fact, is mapped to a single control construct. Moreover, a composite node may also be nested with zero or more composite node(s). It may be noted that an ITM, which is a reduced form of an activity diagram is a chain of nodes from the initial node to the final node. This path we term as *base path* and denote it as t_b. The base path denotes a sequence of nodes containing various types of basic nodes as well as composite nodes. We may recall that a composite node refers to a CCG enclosed within its entry and exit node. Depending on the construct enclosed, there may be a number of paths enclosed in a region. A path beginning with an entry node in a region we term as an *internal path*. A composite node is thus said to enclose a set of internal paths. The number of internal paths composed by each composite node can be derived from the coverage criteria. In the following, we mention the coverage criteria with respect to different cases and then describe our algorithm for test scenario generation.

3.3.1 Coverage criteria

Let T be a set test cases for an activity diagram. T satisfies the following coverage criterion if the following condition holds good.

C_1 : *Selection coverage criterion.*

- For each selection node in an ITM, T must include one scenario corresponding to each output branch of the decision node.

C_2: *Loop adequacy criterion.* For each pre-test loop node in an ITM,

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

For each post-test loop node in an ITM,

- T must include at least one scenario in which control reaches the loop and then the body of the loop is executed at least once before control leaves the loop ("one iteration" paths).
- T must include at least one scenario in which the body of the loop is repeated twice through the loop ("more than one iteration" paths).

C_3 : *Concurrent coverage criterion.*

- For each concurrent node in an ITM, T must include one scenario corresponding to every valid interleaving of action sequences.

So far the scenarios in a concurrent block is concerned, the number of such scenarios can be computed as below. Let D be an activity diagram consisting of a concurrent block of n independent threads $T_1, T_2, \ldots, T_n$ where thread T_i is a sequence of activities $T_{i1}, T_{i2}, \ldots, T_{imi}$. The total number of activities in thread T_i is therefore represented by m_i. Let $N = \sum_{i=1}^{n} m_i$ represent all the activities in the concurrent block. There are $N!$ possible interleaving of execution sequences for a collection of N activities. However, the nodes can be interleaved as long as the ordering imposed by each thread is maintained. In other words, there may be certain node combinations that are invalid because they do not recognize the ordering between nodes of a thread. We need to ensure coverage of all valid interleaving sequences. Let V be the set of all

scenarios that corresponds to valid interleaving sequences. The number of such elements is, $|V| = N!/(\prod_{i=1}^{n} m_i!)$. This calculation assumes that the nodes within a concurrent block have single incoming and single outgoing edge only.

3.3.2 Test scenario generation algorithm

Algorithm has been proposed to produce a set of internal paths satisfying each of the above mentioned criteria. The algorithm performs a depth first search in the control construct graph starting from its entry node. For an example, a composite node for a loop construct generates two internal paths ensuring the adequacy as per criteria C_2. We say a composite node is therefore said to enclose a set of internal 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. We associate, *internal path set* through each composite node, to expand a path into a number of paths.

The ITM representation, in this way, lends itself to a simple method of test scenario generation. Initially, t_b is the trivial test scenario, since t_b may contain zero or more composite node(s) with 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 path. 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. Algorithm 1 outlines our test scenario generation algorithm. It will now be shown that the algorithm *GenerateTestScenario* terminates.

Algorithm 1 GenerateTestScenario

Require: ITM, an intermediate testable model of a given activity diagram 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 ∪ t_b                    // Add t_b to the set of base paths
3: while B ≠ empty do {
4:   for each path t_i ∈ B{
5:    Let t_i = ⟨n_0, n_i1, n_i2, ..., n_ip, ..., n_iq⟩ be a sequence
      of nodes in t_i
6:    for all n_ij ∈ t_i {
7:     if n_ij is not a composite node then
8:       T = T ∪ 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:       Get internal paths corresponding to the node n_ij
12:       Let P_j = {p_1, ..., p_m} denote m internal paths corresponding to the composite node n_ij
13:       for each p_i ∈ P_j {
14:         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'
15:         B = B ∪ t_i'          // Add t_i' to B
16:       }                       // end of for loop
17:     }                         // end of else part
18:   }                           // end of for loop
19:  }                           // end of for loop
20: }                             // end of while loop
```

Algorithm—Test scenario generation from ITM

Theorem 2 *The algorithm GenerateTestScenario terminates after a finite number of iterations.*

Proof The main loop in the algorithm checks if base path set B is empty. At the first iteration, base path t_b is the only path in B. The basic operation in the *while* loop is to add new paths into B by expanding composite nodes. On each expansion, a composite node adds $m, m \geq 0$ paths into B. The coverage criteria defined in Sect. 3.3.1 determine internal paths and hence m is bounded. For example, a loop node returns two paths whereas a selection node returns m which is equal to the number of outgoing edges from the entry node. Since m is known, we can precisely compute the total number of paths added into B. For a base path of n nodes, there can exist atmost $n-1$ composite nodes. In terms of the basic operation of the algorithm, it means that $w = \prod_{i=1}^{k} m_i$ paths are generated at the end of first iteration where k is total number of composite nodes, $1 \leq k \leq n-1$ and m_i is the total number of internal paths of a composite node c_i. Note that $k = 0$ means that no more new paths can be evolved from the given path t_i. In this case, the path t_i under consideration is moved into output set (test scenario set) T and removed from B. Essentially, for an arbitrary path t_i, either w paths are added into B, if $k \geq 1$ or t_i is removed from B if $k = 0$ (that is, $w = 1$). There are two cases to demonstrate that eventually B becomes empty.

Case 1: $k = 0$ for a base path t_b implies that t_b is the the only scenario generated. Hence, B is empty in one iteration.

Case 2: $k \geq 1$ for a base path t_b adds w test paths requiring w iterations. In each iteration, a path t_i with $k = 0$ is removed. Thus, paths with $k \geq 1$ are only eligible in this iteration. The number of iterations for these paths is bounded by the nesting of composite nodes. The nesting depth of a composite node is equivalent to the nesting depth of control constructs. This implies an iteration of atmost $w * d$ iterations where $d \geq 1$ is the nesting depth which is explicitly shown in the activity diagram. This establishes the termination of *GenerateTestScenario* algorithm.

4 Case study

In this section, we illustrate the identification of scenarios according to our approach using an example called Cell Phone System (CPS). We consider 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 each user's perspective, we identify different ways of interacting with the system which is depicted as a use case diagram shown in Fig. 11. Each of the use case is represented with a corresponding activity diagram.

We present the activity diagram for *Make Call* use case which is shown in Fig. 12. The *Make Call* activity diagram models the flow of activities initiated when the user hits a button. Then onwards, different variations and operational conditions are identified depending on the key pressed. In our activity modeling, we consider a call to the new number requiring the user to enter the number from the key pad. A call can also be initiated from the list of dialed numbers, or from the phone book facility. However, the description of the system behavior is the same for these two options as they do not result in any new requirements and therefore, any new scenarios. In the following, we detail the procedure involved in ITM construction and test scenario generation with respect to the *Make Call* use case followed by sample test cases and their format.

4.1 ITM construction

The scenario graph G obtained by parsing the *Make Call* activity diagram as shown in Fig. 13a. Here, a label on a node denotes the identifier corresponding to the node. The

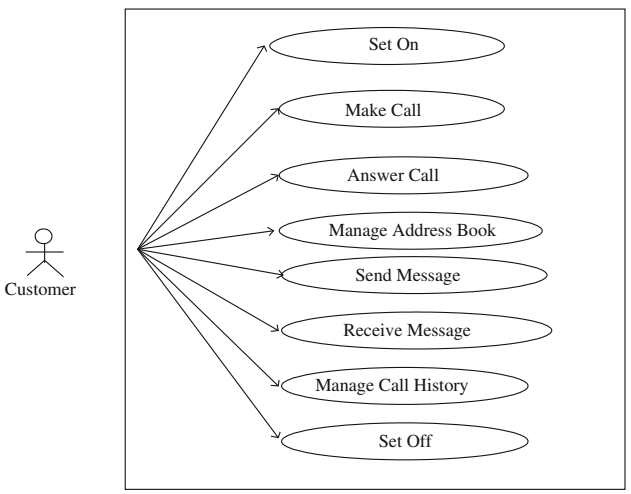


Fig. 11 Use case diagram of the CPS

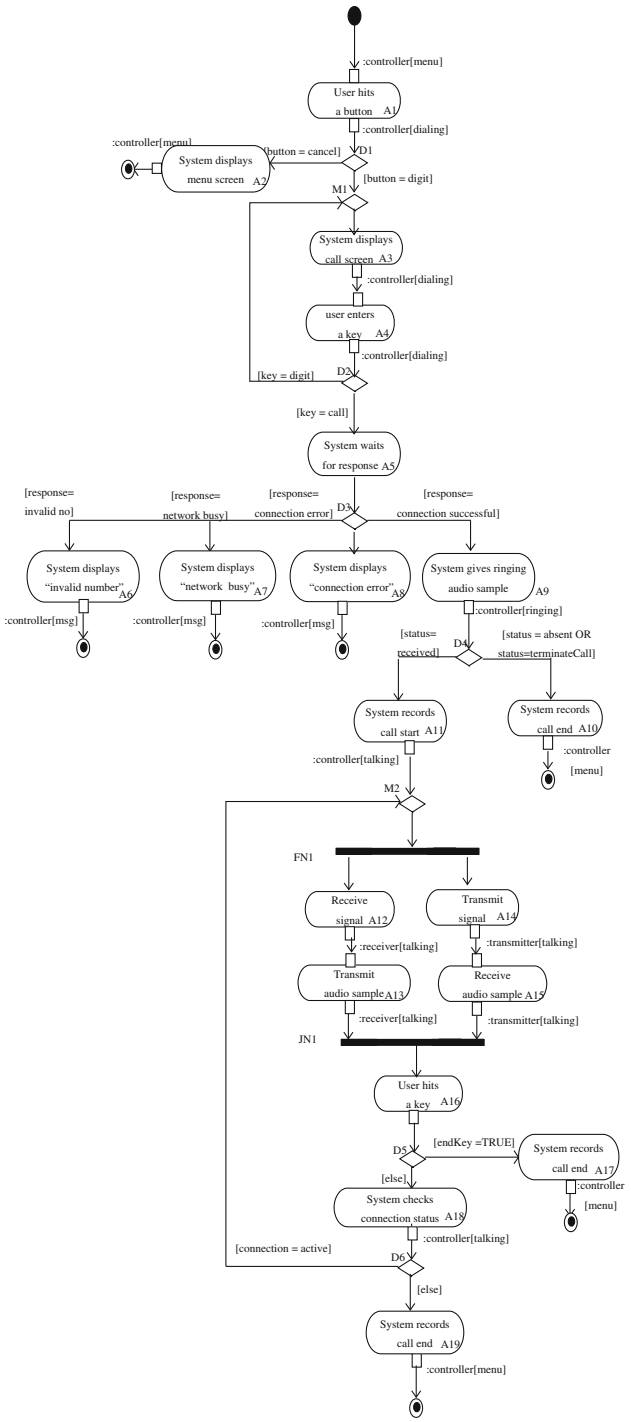


Fig. 12 Activity diagram of *Make Call* use case in CPS

composition procedure begins with taking the graph G as input and starts looking for minimal regions in G classifying them into various types of composite nodes. A loop, concurrent and selection construct are identified which are marked as $L1$, $X1$ and $S1$, respectively as shown G_1 in Fig. 13b and are mapped to respective regions as shown in Fig. 13g. This graph G_1 is the input for the second iteration. During the

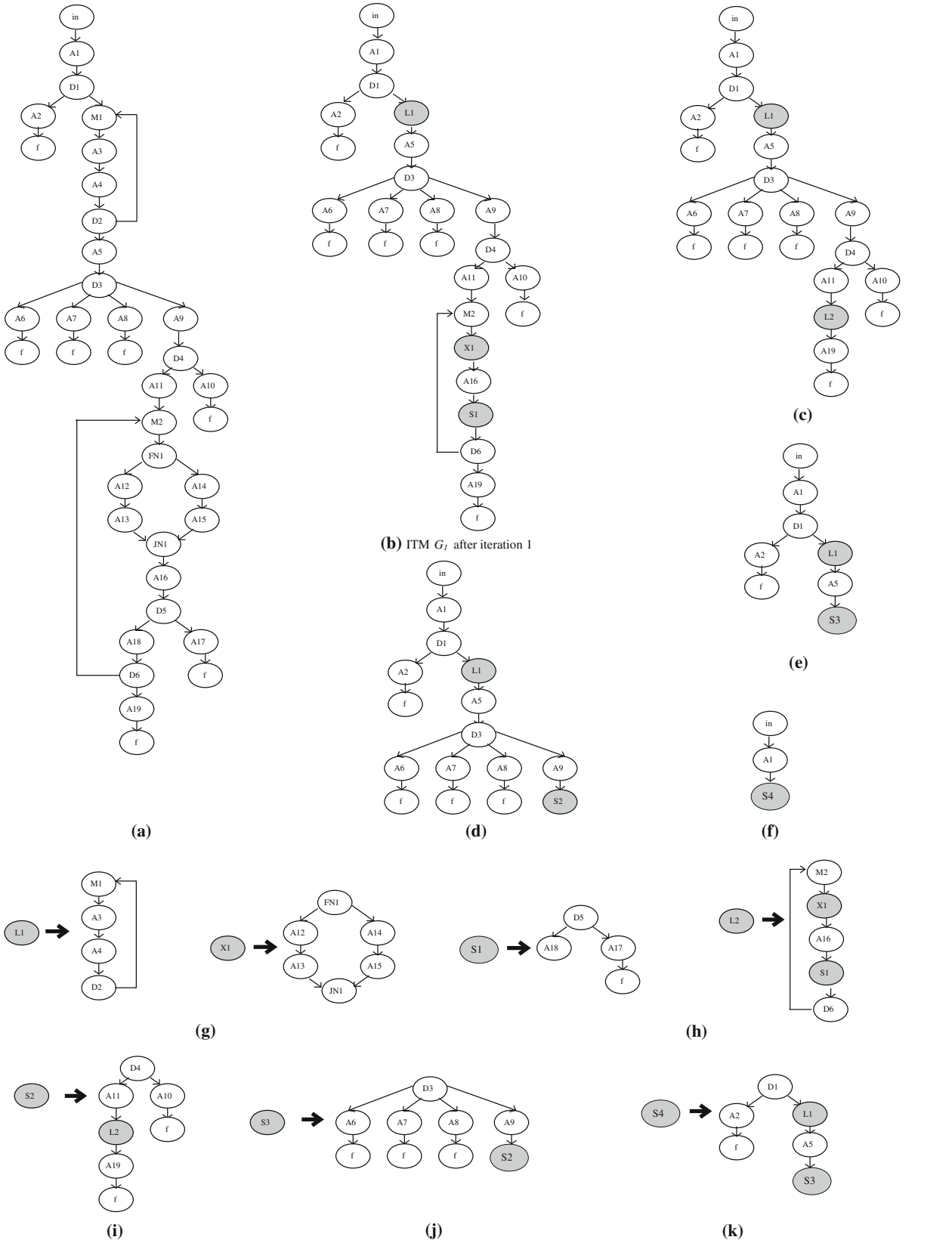


Fig. 13 Transformation procedure for *Make Call* use case. **a** Initial scenario graph, G, **b** ITM G_1 after iteration 1, **c** ITM G_2 after iteration 2, **d** ITM G_3 after iteration 3, **e** ITM G_4 after iteration 4, **f** final ITM, G_5, **g** composite nodes L_1, X_1 and S_1 created in iteration 1, **h** loop node L_2 in iteration 2, **i** selection node S_2 in iteration 3, **j** selection node S_3, **k** selection node S_4 in iteration 5

Table 1 Generating test scenarios for *Make Call*

Step	Set of base paths, B	Set of test scenarios, T
1	$\{\langle in, A1, \underline{S4}\rangle\}$	–
2	$\{\langle in, A1, D1, A2, f\rangle, \langle in, A1, D1, \underline{L1}, A5, \underline{S3}\rangle\}$	
3	$\{\langle in, A1, D1, \underline{L1}, A5, \underline{S3}\rangle\}$	$\{\langle in, A1, D1, A2, f\rangle\}$
4	$\{\langle in, A1, D1, M1, A3, A4, D2, A5, \underline{S3}\rangle,$ $\langle in, A1, D1, M1, A3, A4, D2, M1, A3, A4, D2, A5, \underline{S3}\rangle\}$	$\{\langle in, A1, D1, A2, f\rangle\}$
5	$\{\langle in, A1, D1, M1, A3, A4, D2, A5, D3, A6, f\rangle,$ $\langle in, A1, D1, M1, A3, A4, D2, A5, D3, A7, f\rangle,$ $\langle in, A1, D1, M1, A3, A4, D2, A5, D3, A8, f\rangle,$ $\langle in, A1, D1, M1, A3, A4, D2, A5, D3, A9, \underline{S2}\rangle,$ $\langle in, A1, D1, M1, A3, A4, D2, M1, A3, A4, D2, A5, \underline{S3}\rangle\}$	$\{\langle in, A1, D1, A2, f\rangle\}$
6	$\{\langle in, A1, D1, M1, A3, A4, D2, A5, D3, A9, \underline{S2}\rangle,$ $\langle in, A1, D1, M1, A3, A4, D2, M1, A3, A4, D2, A5, \underline{S3}\rangle\}$	$\{\langle in, A1, D1, A2, f\rangle,$ $\langle in, A1, D1, M1, A3, A4, D2, A5, D3, A6, f\rangle,$ $\langle in, A1, D1, M1, A3, A4, D2, A5, D3, A7, f\rangle,$ $\langle in, A1, D1, M1, A3, A4, D2, A5, D3, A8, f\rangle\}$
7	...	...

second iteration, a minimal region for a loop construct is identified bringing in a loop node, $L2$. The loop node $L2$ encloses composite nodes that are formed earlier namely, $X1$ and $S1$ as shown in Fig. 13c. The graph G_2 in Fig. 13c is looked for minimal regions now. All the subsequent iterations recognize different selection constructs which are marked as $S2$, $S3$ and $S4$. The procedure of composition is continued until all the constructs are reduced. Figure 13b–k show the composition involved in the transformation of G to the ITM at five different steps. It shows the intermediate graphs $G_1, \ldots, G_5$. The final ITM namely, G_5 is shown in Fig. 13f.

4.2 Test scenario generation

Let us now discuss how the above regions helps in generating test scenarios from ITM. To begin with, there is only one test scenario known as base path. However, there are composite nodes in the base path. These composite nodes denoting control constructs are decomposed by applying the respective coverage criterion to the region. This means that the regions containing loop, fork etc. are transformed into the node sequences satisfying the given coverage criterion. These node sequences known as internal paths are made use for every decomposition of a composite node. In the following, we walk through ITM and show how the test scenarios have been evolved from the base path.

As illustrated in the test scenario generation algorithm, the first step is to generate base path, which is $B = \{\langle in, A1, S4\rangle\}$ (see Fig. 13f). In the next step, the composite node $S4$ is expanded. As the node type of $S4$ indicates an unmatched selection, we cover all possible scenarios corresponding to the selection coverage criterion, $C1$. As a result, $S4$ will be replaced with each of its internal paths. In this step, it gives rise to two paths into B. 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. Step 2 shows the expansion of base path into two test scenarios. Among this, one of the path is moved into T as it cannot be expanded any further. The base path set, B is now left with only one test scenario as shown in *Step 3*. However, there are two composite nodes $L1$ and $S3$. Step 4 shows the subsequent expansion of composite node $L1$ yielding two paths. The expansion now proceeds with the other composite node $S3$. The node $S3$ is a selection node (see Fig. 13j) yielding four internal paths. The test scenarios are thus evolved iteratively from the base path until B becomes empty.

4.3 Sample test cases

In this example, we identify 7 paths and show the result of path expansion resulting from each of these. We see that 177 test scenarios are required to test all of these paths exhaustively as can be seen from Table 2. Each test scenario is used to test one of these paths that are specified in Table 2. The test scenarios which are derived from ITM we call them as abstract test cases. For each abstract test case, test inputs need

Table 2 Number of test cases for 7 scenarios in *Make Call*

Path Index	Path description	Total number of test scenarios
1	Cancel call	1
2	Invalid number	2
3	Network busy	2
4	Connection error	2
5	Terminate call	2
6	Connection successful	84
7	Call ends on connection error	84

Table 3 Sample test cases

TestId	Test inputs			Expected result
	ActivityId	Input state	Variable	
t_1	A_1	menu	button = cancel	dialing
	A_2	dialing		menu
t_2	A_1	menu	button = 1	dialing
	A_3	dialing		dialing
	A_4	dialing	key = call	dialing
	A_5	dialing	response = invalid no	dialing
	A_6	dialing		msg
t_3	…	…	…	…

Table 4 Summary of test scenarios generated for different use cases of CPS

Use case	Number of test scenarios
Set on	3
Make call	177
Answer call	12
Manage address book	54
Send message	38
Receive message	14
Manage call history	21
Set off	2

to be derived by substituting sample values so as to form a complete test case.

Table 3 lists few sample test cases. The sample values for a test case has been derived manually using category partitioning approach [25]. During test execution, any changes in the output is recorded in terms of changes in the activities and the conditions coordinating these activities. This can be supported in two ways. First, by instrumenting statements of the source program, all activities that comprise a test case can be tracked. Second, the input and output object pin information maintained for each activity in the activity description can further be used to verify the results. We make use of this information to keep track of changes in the output object state for each executed activity from the corresponding expected object state (derived from the activity diagram). The changes during activity execution may include—creating objects, storing as well as updating state of the object. We assume that state changes that happen inside an activity is specified in the output object state. Thus, for each activity with a given input state, its output object state corresponds to the expected object state. Although user participation is anticipated to provide expected results for activities that may not always be specified with object pin information, to some extent this can assist in verification process.

Intuitively, every sequence of activities corresponds to a particular execution of the use case. As each test case documents the sequence of activities corresponding to the generated path, the resulting test cases can be used to track any deviation in the activity execution as well as deviation in the activity sequence. This can be seen from the generated test cases in Table 3. Every row of this table corresponds to a test case. For example, the *Cancel Call* scenario shown as t_1 lists the expected sequence of activities $A1$ and $A2$ along with their expected results. The first column denotes *TestId*. The second column identifies different information that can together assist in deriving test inputs. Node identifier of each activity along with its input pin and predicate information is made use for this purpose. We may note that as per category partitioning, each category can be assigned with sample values. For example, in the test case t_1, we have assigned a possible choice for the variable $button$. Similarly, in the second test case the choice $digit$ for the variable $button$ is a place holder which is replaced with actual value, such as 1. This can be seen from the second test case t_2 in Table 3. In this way, different sample values are substituted to cause different system behavior.

The composition procedure is extended to all other use cases in CPS followed by test scenario generation. Table 4 summarizes the test scenarios generated for other use cases of CPS.

5 Experimental results

In this section, we present our experiment to investigate the effectiveness of generated test suites. Specifically, our experiments are designed to address the following issue.

Are test suites from the proposed approach effective in exposing faults?

In order to answer this issue, we considered four use cases from four different application domains and obtained test suites for each of these using our approach. In the following, we list the use cases considered in our experiments.

5.1 Subject programs

We considered four use cases, namely *Credit Card Payment, Paper Registration, PIN Validation and Make Call* in our experiment. These applications are designed and implemented by the PG students in their assignments for a course on *Information System Design* coordinated by the second author of this paper. Each of these use case has several control constructs and accordingly their corresponding activity diagrams exhibit moderate complexity. Further, use cases

were fairly chosen so as to include different variations and combinations of control constructs. The use cases are:

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).

Activity diagrams of these use cases are drawn with MagicDraw 10.0 [1] and exported the diagrams in XML format. Following the STAD methodology discussed in Sect. 3, we obtain an ITM and test scenarios are then generated. The objective of our experiment is to observe fault detection ability of the generated test suite. For this, a set of mutation operators are considered (stated in Sect. 5.2) in our experiment. A mutation operator induces small syntactic change to the original program [22]. The implementation corresponding to the above mentioned use cases are in Java and considered as the subject programs. A *mutant program* or *mutant* is produced by applying a single mutation operator only once to the original program. Applying the mutation operators we generate a set of mutants.

5.2 Mutation operators

The following eleven mutation operators are used in our experiment [10,17].

1. *Language Operator Replacement* (*LOR*): 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 etc.
2. *Literal Chage Operator* (*LCO*): Increases or decreases numeric values. For Boolean literals, 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*): To disrupt normal control flow by changing break, continue or return statements.
8. *Statement Swap Operator* (*SSO*): Swaps statements in a block such as, statements inside one switch block with another switch etc.
9. *Statement Delete Operator* (*SDO*): 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.

We group these mutation operators into two groups based on the type of faults they cover. The first six mutation operators are in $Group_1$ and this group of mutation operators covers some faults that are of general nature which can be applied for the variables, types, classes or expressions 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 etc. the $Group_{II}$ operators numbered from 7-11 are used. The type and number of mutants produced depends on the source code. Table 5 lists the type and number of mutants generated for different use cases.

5.3 Metric for evaluation

The objective of our experiment using mutation testing is the generation of a test set that can distinguish the behavior of the mutants from the original program. In order to evaluate the effectiveness of generated test suites, we make use of mutation score. 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 that are mentioned in Sect. 5.1, we generated test suites using the proposed approach. To compare the effectiveness, we also generated an alternate set of test suites for each of the subject programs. These alternate test suites are generated using traversal technique on graph representation of the activity diagram. Thus, there are two sets of test suites for each subject program—one using ITM method and the other using direct traversal of the activity diagram. We call test suites that are generated directly as random test suites since there is no analysis of the activity diagram involved prior to the test suite generation.

In order to compare fault detection ability of these two testing strategies, we considered several factors which can have direct impact on our analysis. Among them, we identified

Table 5 Mutants for subject programs

S.No.	Mutant 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 6 Mutant killing results for subject programs

Subject programs	Random approach				Proposed approach			
	M_I	M_{II}	M	T	M_I	M_{II}	M	T
Make call	26	26	0.81	94	26	35	1.00	177
Credit card payment	8	5	1.00	16	8	5	1.00	16
Paper registration	32	14	0.85	26	32	22	1.00	67
PIN validation	18	11	0.725	73	18	22	1.00	121

M_I Number of killed mutants for $Group_I$
M Mutation score
M_{II} Number of killed mutants for $Group_{II}$
T Test suite size

three factors as significant in varying the interpretation of results. The first factor, the test coverage criteria mentioned in Sect. 3.3.1 are made use uniformly for both testing strategies. To produce test cases from test scenarios, the choice of test data is important. We manually generated test data using category partition method [25] for both strategies. However, we considered only a single set of test data for both the strategies. Hence, a single test suite is made available from each set of scenarios. The second factor, same set of test oracles is considered for the same scenarios. Finally, the mutant programs are generated by seeding the faults mentioned in Table 5 in the subject programs. The above three factors ensured that the test cases can be considered different only if the underlying test scenarios are different. In other words, for the same scenario, the test cases using both approaches are identical. Random test suite is therefore used as a benchmark in our discussions.

Using 11 mutation operators we injected faults in the subject programs. For example, in the implementation of *Make Call* use case, we created 61 mutant programs with each mutant containing one of 61 faults. The test input for each of the test case is manually obtained and test suites are then executed on each of the mutants. For the same use case and its implementation, we thus obtain different mutation scores. The program wise results for all subject programs with respect to each test suite are shown in Table 6.

5.5 Analysis of results

For each subject program, Table 6 lists number of mutants generated along with number of mutants killed. In order to clearly specify the generated mutants and their killing rate, we have shown mutants under two groups. First group referred to as $Group_I$ includes mutants of generic nature as discussed in Sect. 5.2. Although mutants in $Group_I$ try to inject faults and exercise different parts of the source program, they 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 directly in the statement block such as loop,

if-then-else etc. In the following, we discuss program wise analysis and subsequently discuss the over-all findings according to our experiments.

5.5.1 Make call

It can be seen from Table 6 that random approach detected only 26 faults out of 35 in $Group_{II}$. The nested constructs cannot be separately distinguished when test scenarios are obtained directly from activity diagrams. This leads to less number of scenarios using random approach. Since all activities are covered using the random approach, faults related to $Group_I$ operators are distinguished. However, the results were different when it comes to $Group_{II}$ mutants. Different mutants injected under each type of fault is given in Table 5. We seeded only SDO faults into the thread. The mutation testing involving a thread required more testing effort, that is, more test cases. We have applied a single test data and observed the different paths in the concurrent executions. It is really impossible to decide different test data to trace different scenarios.

5.5.2 Credit card payment

The *Credit Card Payment* use case has simple conditions involving matched selections and loops. Since there are no concurrent constructs, the testing effort, that is, number of test cases is less. Hence the random approach as well as the proposed approach correctly produced the same number of test cases. We seeded totally 13 faults out of which five were targeted for revealing faults in the control constructs. Hence, the random test suite is equally effective if there are no nesting of constructs. The results are shown in Table 6.

5.5.3 Paper registration

The *Paper Registration* use case has nested constructs consisting of loops and selection. There are exit paths for these selection construct within the loops forming each selection as K-out-of-N selection construct. In order to exercise all possible paths, testing effort will be more for nested constructs. We recognize all of these control constructs and generate test cases accordingly, which has resulted in more test cases. However, random approach cannot identify test cases to distinguish different instances of nesting. As a result, the number of test cases generated with random approach is very less. It is clear from the results of mutation testing that these test cases are not sufficient in detecting all types of faults.

5.5.4 PIN validation

The activity diagram for this use case includes a fork with two outgoing edges. One of this checks for "card stolen" condition and another one checks for "account closed" condition. Therefore, it involves two selection constructs within the fork. Moreover, if any of these conditions are satisfied then the user interaction will be stopped. This leads to two exit paths within the fork. There is also a loop construct which checks for valid pin being entered for a maximum of four times. A loop within a nested structure or a loop that encloses different types of control structures are not recognized while employing random approach as evident from the mutant killing results.

5.5.5 Overall results

In this section, we discuss the implication of our results. We observe that there are differences in the size of test suites obtained from the random approach and the proposed approach. The differences are justified as below. Depending on the nesting depth and the type of constructs nested in, more test cases are required to ensure complete coverage. This is evident from the subject programs considered in our experiment. In *Credit Card Payment*, as the nesting depth is zero, the random approach is equally effective in detecting faults. However, the difference is significant for subject programs with non-zero nesting depth. Specifically, mutant killing results with $Group_{II}$ operators in Table 6 confirm our observation.

It is also to be noted that for any given subject program considered in our experiment, the test suites using a random approach are the sub sets of our test suites since the test cases contained in random test suites are all included in our approach as well. This means that, although we conducted our experiment in order to determine test effectiveness, we also analyzed whether differences in effectiveness could be due to differences in numbers of test scenarios. In other words, we have shown that a sub set of our test suites cannot ensure complete coverage. Therefore, our experiments indicate that a composition approach for increasing coverage is an effective method to detect faults. However, it is known that an increase in number of test scenarios increases the cost of testing. In addition, complete coverage of a test suite does not necessarily guarantee increased fault detection [5]. Nevertheless, a test suite satisfying a given coverage can be judged by its fault detecting ability. For this purpose, we compared these test suites in terms of their ability to detect faults. These variations among the test suites for a given model and its implementation indicate that achieving lesser coverage may not be appropriate.

The experimental results show that on average 15.37% faults can be detected with an increase in coverage. However,

there are many factors that affect the generalization of our results. Mainly, the factors identified as unvarying in Sect. 5.4 may play a crucial role in distinguishing faults thereby introducing variation in results. To address this we must vary these factors and conduct experiments on more than one set of test suites for the same subject program. By doing this, it is possible to consider several subsets for the same subject program. In a similar way, the subject programs and mutant operators considered in our experiments may not be representative. More subject programs from diverse application domains and a larget set of mutation operators must be considered.

6 Related work

In recent years, several researchers have concentrated on developing testing methodologies using UML models. As our work is related to test case generation using activity diagram, we survey only to those work related to UML activity diagrams. We present our survey in the following.

Hartmann et al. [13] describe an approach for generating system level tests from activity diagrams. In their approach, activity diagrams are manually annotated prior to the test case generation. The annotations help to determine different variables and possible data choices for these variables. Test cases are then generated considering all paths in the annotated diagram. They make use of category partition method for generating test data corresponding to a test case. They also discuss test case execution for which test cases are converted into executable test scripts or test procedures.

In a similar approach, Vieira et al. [29] focuses on GUI testing. An activity diagram representing each tab of the GUI is annotated with test requirements as in [13]. The categories and choices are represented by a class along with its attributes and are associated with the activity. Based on different combinations of choices, test cases are generated so as to cover the activity diagram. The test scripts are then generated in the form of test procedures as a set of XML files.

Linzhang et al. [20] propose an approach for generating test cases from activity diagrams and present UMLTGF, a prototype tool for supporting automation in test case generation. While traversing activity diagrams, they restrict that the loops be executed at most once and consider basic path coverage criterion for generating test cases. The approach relies on the assumption that any fork node can only have two outgoing edges and they generate two scenarios from such a fork structure.

Mingsong et al. [21] presents an approach for generating test cases from activity diagrams. The approach makes use of basic paths [20] for handling loops. Based on a partial order relation that sequences the activities of an activity diagram, they define a simple path which selects a representative path for handling concurrency. These simple paths are then matched with the execution trace of the corresponding program by mapping functions to activities. In this way, they find out coverage of simple paths in an activity diagram.

Kim et al. [16] propose an approach to generate test cases from activity diagrams by exposing all possible external inputs and outputs. They achieve this by translating activities into send signal and accept event actions. The model resulting from this transformation is known as I/O explicit Activity Diagram (IOAD) in their approach. From this IOAD model test cases are generated. They also show how to minimize the number of scenarios that can be derived from a fork-join structure by applying the knowledge of external event actions and their order. However, it involves considerable amount of interpretation while translating actions into external event actions.

Bai et al. [4] describe an approach for scenario-based software testing using activity diagrams. The scenarios are known as thin threads in their approach. To begin with, they describe a pre-processing step in which a flattened activity diagram is formed by replacing all sub-activity nodes with the corresponding activity diagram. From this flattened activity diagram, all branch and concurrency are replaced by their equivalent execution paths. During this process, any conditions associated with branch nodes are arranged as a separate condition tree. Similarly, all data objects are arranged as a separate data object tree. The resulting tree is known as a thin thread tree. They demonstrate the usage of the derived trees for prioritizing test cases based on risk analysis. In an extended paper, Li et al. [19] make use of anti-ant like behavior while exploring the activity diagram. They propose an algorithm for traversing activity diagrams that simulates the movement of a group of ant like agents. The objective of this traversal is to build condition tree, thin thread tree and data object tree from the activity diagram without any preprocessing.

Xu et al. [9] employ adaptive agents to build the test scenarios from an activity diagram in a tool known as TSGAD. In their approach, a test scenario is a sequence of action states and transitions from an initial state to a final state. Activity diagrams containing fork-join pairs including nested fork join pairs and branches are considered in their approach. To explore various test scenarios of an activity diagram, they simulate the behavior model of the bacteria. Moreover, their algorithm makes use of a simple fork-join (SFJ) type as a basis of decomposition and tries to express all other types in terms of the SFJ type.

Chandler et al. [8] describe an approach known as AD2US for generating usage scenarios from activity diagrams. Even though they do not formally define usage scenarios, each path through an activity diagram is viewed as usage scenario in their approach. From an input XML file, they describe how to extract and process diagram elements so as to generate usage

scenarios. In [7], an approach extending the work in [8] is presented, focusing on concurrent regions. It allows inclusion of loops, branches and nested concurrent regions. Accordingly, they propose categories to describe concurrent regions. They also analyze coverage of paths and conditions based on criteria, such as risk and usage. The scenario computation within a concurrent region, in their approach, includes only branch and nested branches into account. This is equivalent to our selection construct only. Moreover, how to calculate scenarios for various other combinations of loops and branches are not considered in their approach.

The research described in this paper addressed two limitations observed in the existing approaches. First, the existing approaches make it difficult to generate tests for complex models because the work assume simplified models [4,8,13,16,19–21,29] or make use of a subset of the constructs in the models [7,9]. Second, the reported researches are not able to capture the essential structure of the model prior to test case generation. Consequently, they describe scenario generation by ignoring unmatched constructs and various combinations of nested structures. Hence, existing algorithms are unable to handle arbitrary control flows. In our approach, ITM takes care of all these limitations by forming a hierarchical and concise representation of the flat model which can easily be tamed to generate test cases.

7 Discussion

In this section, we discuss the efficacy of the proposed approach. We also point out few limitations in our approach.

7.1 Effectiveness

Our experimental results reveals the fact that more testing effort is desirable to achieve test adequacy. Experimental results with our approach and random approach confirm that test suite according to our approach is more effective to detect faults than the random approach. This is indeed a significant observation. It might also be noticeable that test adequacy in our approach is at the cost of a large number of test cases. We may clarify this as follows. Intuitively, a test suite that covers a large number of coverage criteria requires more testing effort [5]. This is eventually reflected in our proposed approach. Our test cases, in fact, cover a large number of paths in the model artifacts satisfying several coverage criteria stated in Sect. 3.3.1. In contrast, random test suites cover a sub set of paths in the activity diagram and hence the fewer number of test cases. Nevertheless, this is at the cost of test effectiveness. In order to generate test cases from an activity diagram of arbitrary control flow presents challenges due to various types of constructs such as unmatched constructs and their nesting. The variations in the result suggest that in order to detect all types of faults, the test suite need to include all possible test paths. Our test cases together cover all the coverage criteria and are adequate as confirmed by mutation testing results.

7.2 Time complexity

Our composition starts with G by identifying all minimal regions. The composite nodes are then defined in G_1 to map the identified region. The procedure of composition then continues in G_1. In each iteration, we need to visit all nodes to check whether they meet the properties of a minimal region. The time for identifying minimal region is therefore $O(e)$ where e is the number of edges in the given graph. The nodes that satisfy properties of a minimal region must involve atleast two nodes—entry and exit nodes. Let us term regions with two nodes as trivial regions because, in practice there may exist one or more nodes between entry and exit nodes. Since we ignore atleast two nodes for each iteration, number of nodes in each iteration are reduced atleast by 2. Moreover, we continue iteration only if we are able to compose atleast one region. This means that as long as $G_{i+1} \neq G_i$, we continue the iteration. For $\{G, G_1, G_2, \ldots, G_k\}$ where G is the initial graph and G_k is the ITM, then number of iterations required is $k + 1$. In the worst case, the ITM reduces to a graph with atleast two nodes since initial node cannot form part of any minimal region. The maximum number of iterations in that case is $n/2$ where n is the number of nodes in the initial graph. Building ITM can therefore be performed in $O(n^2)$. This forms the worst case complexity where activity diagrams are non-trivial and there may be more than one occurrences of minimal regions. Supposing there are k such regions, the number of nodes in $(i + 1)$th iteration is reduced by $2{*}k$. Number of nodes in N_{i+1} iteration is $N_i - 2{*}k$ where N_i is the number of nodes in ith iteration. This means that number of nodes in the graph reduce substantially in each iteration. This is to say that average complexity falls below $O(n^2)$.

7.3 Applicability

In case of properly nested control constructs, it is obvious that the nesting relationship exists among control constructs. Composition transforms every control construct into a node with single incoming and single outgoing edge which we termed as a composite node. While composing, distinct minimal regions are identified. As a result, entry and exit nodes are uniquely defined for each region and therefore, there is no partial overlapping of regions. Supposing all the constructs are well defined, sequentially placed and do not have any nested structure, the composition finishes in one iteration. However, to be able to associate entry and exit nodes for unmatched and nested constructs leaves us with more than

one iteration. These constructs can only be resolved at the subsequent iterations. But this does not affect the way regions are composed in each iteration. So our approach is confluent in the sense that irrespective of the order in which constructs are identified, we get same set of composite nodes and therefore same number of tests.

The structured representation resulting from above mapping ensures that ITM captures the nesting or containment relationships within nested structures. It further simplifies the subsequent identification and the mapping. It enables unmatched constructs to be associated to a distinguished entry and exit nodes which are otherwise impossible due to the lacking control flow information. Our approach is based on region identification. The identified regions can be analyzed independently of each other without any ambiguity. As each of the region requiring further analysis has already been identified, the test case generation from the ITM is systematic. This further ensures that nested and unmatched structures are dealt unambiguously.

The base path with which we begin our test generation process gives us single test scenario. Depending on the type of composite node, the control construct graph is then traversed. The number of test scenarios generated depends on the underlying coverage criterion that governs each composite node. The separate control construct graph for every control construct allows localized control influencing the number of tests generated. Compared to the flat model of the activity diagram, the proposed approach can ensure adequate coverage. This can be seen from the experimental results in comparison to other approaches.

We believe that the approach presented in this paper is general enough to be applicable to synthesize test cases from other UML based interaction diagrams such as sequence diagrams, collaboration diagrams and interaction overview diagrams.

7.4 Limitations

Using our approach, it is has been possible to precisely identify and capture various types of control flow constructs in an activity diagram. However, edges interacting with control constructs may turn an activity diagram into an unstructured graph. Presently, our approach cannot transform such unstructured activity diagrams. We can say that G must contain atleast one such control construct if G cannot be reduced to a single chain of nodes. In this case, we will be left with nodes having more than one in degree and/or out degree.

Another limitation concerns our data selection for a given path. Based on various coverage criteria, we obtain test scenarios from the ITM. For every test scenario in the test suite, the test data generator needs to derive input data resulting in the execution of the selected test scenario. This transforms a test scenario into an executable test case. Presently, we have not focused on test data generation and our automation is thus limited to test scenario generation only.

8 Conclusions

In this paper, we have addressed the problem of generating test scenarios from UML activity diagrams. Identification of control constructs and their nested combinations is one of the key problems for deriving test scenarios from activity diagrams. We introduced an approach to capture all control structures of an activity diagram and to transform them into a testable model called ITM.

Representing the structure of an activity diagram with ITM has a number of advantages. The proposed approach provides a foundation and reference for managing any complex and arbitrary activity diagrams. The introduction of separate control flow graph for each control construct lends to a simple process of analysis and derivation of scenarios using coverage criteria. The impact of path explosion has been tackled by choosing representative path for a set of paths in the context of looping.

Our approach does not demand any modification in the input design and hence suitable for any unified process development environment. Moreover, our approach is capable of managing activity diagrams of any complexity for generating test cases efficiently.

We plan to extend our work to generate test vectors for each test case towards a fully automatic system testing. In this paper, we only considered the activity diagram generated from use cases to represent system behavior. The use cases are at a higher level of abstraction, which cannot be used to model message flows associated with actor-system interactions. It might be more useful to include sequence diagrams illustrating each activity in a use case so that detailed oracles can be derived to refine scenario generation phase further. Our results so far suggest that the approach is efficient enough to scale up to the requirements of a large system. However, further research is needed to apply this approach to a complete model illustrating set of use cases and their dependencies.

Acknowledgments The authors would like to acknowledge the anonymous reviewers, for their constructive feedback which helped in improving the presentation of the paper.

References

1. Magicdraw home page. http://www.magicdraw.com/
2. Aho, A., Sethi, R., Ullman, J.: Compilers Principles, Techniques and Tools. Addison-Wesley, Reading (1986)
3. Allen, F.E.: Control flow analysis. ACM Sigplan Notices **5**(7), 1–19 (1970)

4. Bai, X., Lam, C.P., Li, H.: An approach to generate the thin-threads from the UML diagrams. In: Proceedings of Computer Software and Applications Conference, pp. 546–552 (2004)
5. Binder, R.V.: Testing Object Oriented Systems: Models, Patterns and Tools. The Addison-Wesley Object Technology Series (1999)
6. Briand, L., Labiche, Y.: A UML- based approach to system testing. J. Softw. Syst. Model. **1**(1), 10–42 (2002)
7. Chandler, R., Lam, C., Li, H.: Dealing with concurrent regions during scenario generation from Activity diagrams. In: Proceedings of the International Conference on Systems, Computing Sciences and Software Engineering, pp. 415–420. Springer, Netherlands (2007)
8. Chandler, R., Lam, C.P., Li, H.: AD2US:an automated approach to generating usage scenarios from UML Activity diagrams. In: Proceedings of 12th Asia Pacific Software Engineering Conference, pp. 9–16. Taipei, Taiwan (2005)
9. Xu, D.H.L., Lam, C.P.: Using adaptive agents to automatically generate test scenarios from the UML Activity diagrams. In: Proceedings of 12th Asia Pacific Software Engineering Conference, pp. 15–17. Taipei, Taiwan (2005)
10. Delamaro, M., Pezze, M., Vincenzi, A.M.R.: Mutant operators for testing concurrent Java programs. In: Proceedings of Brazilian Symposium on Software Engineering, pp. 272–285. Rio de Janeiro, RJ, Brazil (2001)
11. Frohlich, P., Link, J.: Automated test case generation from dynamic models. In: Proceedings of 14th European Conference on Object-Orinted Programming (2000)
12. Harrold, M.J., Rothermel, G.: Performing data flow testing on classes. In: Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering, pp. 154–163 (1994)
13. Hartmann, J., Viera, M., Foster, H., Ruder, A.: A UML based approach to system testing. Journal, Innovations in Systems and Software Engineering pp. 12–24 (2005). Springer, London
14. Hecht, M.S., Ullman, J.D.: Flow graph reducibility. In: Proceedings of the fourth annual ACM symposium on Theory of computing, pp. 238–250. Denver, Colorado, United States (1972)
15. Hecht, M.S., Ullman, J.D.: Characterizations of reducible flow graphs. J. ACM **21**(3), 367–375 (1974)
16. Kim, H., Kang, S., Baik, J., Ko, I.: Test cases generation from UML Activity diagrams. In: Proceedings of 8th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, pp. 556–561 (2007)
17. Kim, S., Clark, J., McDermid, J.: 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 (1999)
18. Kim, Y., Carlson, C.R.: Scenario based integration testing for object-oriented software development. In: Proceedings of Asian Test Symposium, pp. 283–288 (1999)
19. Li, H., Lam, C.P.: Using anti-ant-like agents to generate test threads from the UML diagrams. In: LNCS,Volume 3502, pp. 69–80 (2005)
20. Linzhang, W., Jiesong, Y., Xiaofeng, Y., Jun, H., Xuandong, L., Guoliang, Z.: Generating test cases from UML Activity diagram based on gray-box method. In: Proceedings of the 11th Asia-Pacific Software Engineering Conference, pp. 284–291. IEEE (2004)
21. Mingsong, C., Xiaokang, Q., Xuandong, L.: Automatic test case generation for UML Activity diagrams. In: Proceedings of the 2006 International workshop on Automation of software test, pp. 2–8. Shanghai, China (2006)
22. Offutt, A.J.: Investigations of the software testing coupling effect. ACM Trans. Softw. Eng. Methodol. **1**(1), 5–20 (1992)
23. Offutt, J., Abdurazik, A.: Generating tests from UML specifications. In: Proceedings of the Second International Conference on Unified Modeling Language, Beyond the Standard, pp. 416–429 (1999)
24. Offutt, J., Abdurazik, A.: Using UML Collaboration diagrams for static checking and test generation. In: Proceedings of Third International Conference on UML, pp. 383–395. York, UK (2000)
25. Ostrand, T.J., Balcer, M.J.: The category-partition method for specifying and generating fuctional tests. Commun. ACM **31**(6), 676–686 (1988)
26. Ryser, J., Glinz., M.: A scenario-based approach to validating and testing software systems using statecharts. In: Proceedings of 12th International Conference on Software and Systems Engineering (1999)
27. Sethi, R.: Testing for the Church-Rosser property. J. ACM **21**(4), 671–679 (1974)
28. UML: UML 2.0 Superstructure-Final Adopted Specification. Object Management Group (2003). http://www.omg.org/docs/ad/03-08-02.pdf
29. Vieira, M., Leduc, J., Hasling, B., Subramanyan, R., Kazmeier, J.J.: Automation of GUI testing using a model-driven approach. In: Proceedings of the 2006 International workshop on Automation of Software Test, pp. 9–14. Shanghai, China (2006)
30. Weidenhaupt, K., Pohl, K., Jarke, M., Haumer, P.: Scenario usage in system development—a report on current practice. IEEE Softw. **15**(2), 34–45 (1998)
31. Zhang, F., D'Hollander, E.H.: Using hammock graphs to structure programs. IEEE Trans. Softw. Eng. **30**(4), 231–245 (2004)

Author Biographies

Ashalatha Nayak is an Assistant Professor in Department of Computer Science and Engineering at Manipal Institute of Technology, Manipal, India. She obtained her B.Tech. and M.Tech. in Computer Science and Engineering from Mangalore University, Karnataka state, India. She is pursuing her Ph.D. in the School of Information Technology, IIT Kharagpur in the area of Model Based Testing. Her research interests include program analysis and software testing.

Debasis Samanta is an Associate Professor in School of Information Technology, Indian Institute of Technology Kharagpur, India. He received his B.Tech., M.Tech. and Ph.D. all in Computer Science and Engineering from Calcutta University, Jadavpur University and Indian Institute of Technology Kharagpur, respectively. His research interests include information system design, human computer interaction, cloud computing, biometric processing etc.