Model Based Test Case Generation from UML Activity Diagrams

V. Mary Sumalatha
Research Scholar
Gitam University, Visakhapatnam
Andhra Pradesh, India
vmsumalatha2002@gmail.com

Dr G.S.V.P. Raju
Professor
Andhra University, Visakhapatnam
Andhra Pradesh, India
gsvpraju2011@gmail.com

Abstract

UML Activity diagrams became a standard to capture the system behaviour it is considered as well suited diagram in software testing. Software testing is one of the main activities to be carried out in the software development life cycle. It consumes more money and time, which leads to automation that reduces the human effort in finding bugs and errors. Automation in the last phase of system development is similar to manual testing. In both cases bugs are detected only after code has been complete. So rectifications and modification of the code takes lot of time. So testing process should be started from the beginning phase of software development life cycle and should continue till the last phase. Three approaches in testing are code based testing, specification testing and model based approach. So we focus on Model Based Approach for automatic test case generation. For this we generate test cases for object oriented software using UML diagrams like Activity diagram only after removing the flow ambiguities present in the model to be developed.

Keywords- UML, Activity diagrams, Activity Graph, Ambiguities, Concurrency, Software Testing, Test case Generation.

I. INTRODUCTION TO SOFTWARE TESTING

Testing is a means for assessing the quality of software, and is one of the most important phases during the software development process with regard to quality assurance. It can never show the absence of failures, but it aims at increasing the confidence that a system meets its specified behavior. Testing is an activity performed for improving the product quality by identifying defects and problems. Planning the test includes the planning of resources and the laying down of a test strategy, defining the test methods, the coverage criteria to be achieved, and the test completion criteria, structuring and prioritizing the tests environment. In addition to testing, validation, and verification are especially important quality assurance activities in the domain of software systems. The purpose of validation is to confirm that the developed product meets the user needs and requirements. Verification ensures that it is consistent, complete, and correct at the different steps of the life cycle. Testing means exercising an implementation to detect faults and can be used both for verification and for validation. The goal of testing is during the software development systems can be categorized into static testing, also called review, and dynamic testing, whereas the latter is distinguished between structural, functional, and
non-functional testing. After the review phase, the test goal is usually to check the functional behavior of the system and non-functional tests appear in later development stages.

1.1 Black box testing

Black box testing is mainly used for functional testing. Functional testing evaluates the correctness of the program by comparing the needs of users without any knowledge of how the software is implemented. In black box testing, testers test software through user interfaces or the application programming interfaces. The software is successful if the program executes the test cases and returns the expected output. The expected results are stated in the user requirements. In the analysis phase of the software development life cycle, software architects must elicit requirements and transform them into program features that meet customers’ needs. In the design phase developers receive these requirements and produce design documents for their programmers. A design document is a set of guidelines for testers to create their test cases and the expected results in black box testing.

1.2 White box testing

White box testing is testing for all possible coverage paths; i.e. all paths of the program must be tested through some criteria. White box testing aims to qualify a quality of a source code. It qualifies the test suite by coverage criteria. Code coverage criteria is defined using segment coverage, branch coverage, node testing, condition coverage, basis path testing, data flow testing, path testing and loop testing. The test procedure attempts to execute every part of the source code using the test data. The more test cases are generated, the more coverage is gained.

1.3 Grey box testing

Grey-box testing is a combination of black box-testing and white box-testing. As with black-box testing, grey-box testing uses a specification for creating test cases. The specification used in grey-box testing does not specify only the requirements of a system, but it also describes the behavior of the system. Grey-box testing is similar to white-box testing in this sense. The behavioral information embedded in a specification is also used for generating test cases.

1.4 Software testing approaches.

There are mainly three types of testing approaches namely Model based testing, Specification based testing and code based testing. In specification based techniques test cases are derived directly from the specification or from some other kind of model of what the system should do. The principle of code based testing is to have each and every statement in the program executed at least once during the test. Code based testing attempts to test all reachable elements in the software under the cost and time constraints. The testing process begins by first identifying areas in the program not being exercised by the current set of test cases, follow by creating additional test cases to increase the coverage. Methods proposed for structural code based testing include statement coverage, decision coverage and condition coverage. Model-based testing is testing in which the entire test specification is derived in whole or in part from both the system requirements and a model that describe selected functional aspects of the system under test. In this context, the term entire test specification covers the abstract test scenarios substantiated with the concrete sets of test data and the expected system under test outputs. It is organized in a set of test cases. Model-based testing is an approach that bases common testing tasks such as test case generation and test result evaluation of a model. A model of software is a depiction of its behavior where behavior can be described in terms of the input sequences accepted by the system, the set of actions, conditions, the flow of data through the application’s modules and routines. There are numerous such models, and each describes different aspects of software behavior. For example, control flow, data flow, and program dependency graphs express how the implementation behaves by representing its source code structure. Decision tables and state machines, on the other hand, are used to describe external so-called black box behavior. Examples of some software testing models are finite state machines, state charts, the unified modeling language (UML), Markov chains and grammars. In our paper we concentrate on model based approach. Activities in MBT are building a model, generating expected inputs and outputs, running the tests, comparing the actual outputs with expected outputs and stop testing after a criterion is reached.

II. RELATED WORK

Typically three types of techniques have been proposed in the literature to generate test cases. These are Model based, Specification based and Code based techniques. Model based techniques [1][4][5] takes UML and using the requirement specifications as input, and design/architectural information is used to generate test cases. These
techniques generate functional tests [1][5] and also integration tests [3]. Ravi et al [1] proposed a technique that presents a relationship between the structure of the UCAD and generated functional test cases. They used the ‘Behavioral slicing using unit of behavior’ concept in structuring a UCAD similar to the concept introduced in [6]. Specification Based Testing refers to the process of testing a program based on what its specification says its behavior should be. In particular, we can develop test cases based on the specification of the program’s behavior, without seeing an implementation of the program. Furthermore, we can develop test cases before the program even exists. Several techniques have been proposed to automatically generate test cases from software specifications of the envisaged systems such as Z [7] and ADL [8] etc. Hall proposed a test case generation technique based on the Z specification language. The proposed technique used partition analysis strategy to divide the input space into partitions, for test generation purposes. Liu, Miao and Zhan further extended the work of Stocks and Carrington for object-oriented specifications. It is based on Object-Z notation, and can be partially automated. The proposed framework in generates a valid input space (VIS) for class methods, and applies a strategy on VIS to generate test data. Legeard and Peureux present a case study on generating test sequences for Smart Card GSM 11-11 standard to evaluate the effectiveness of B Testing Tools testing environment on a large real-life application. Miao and Liu extended the work of Stocks and Carrington and Liu et al. They proposed a test class framework for object-oriented class testing using Object-Z specification. Their proposed framework is partially automatable. It generates a valid input space for class methods, and applies a testing strategy on VIS to generate test data. Code-based techniques use code analysis techniques [9][10] to build the control and data paths and thus generate test cases by traversing these paths. These techniques also use the symbolic execution to generate test cases for glass-box testing. The research being done on code based test case generation techniques is mainly for white-box testing. Basically, test cases for black box testing are generated using the specifications or the model based approaches. Test case generation using activity has been an area of much attention these days. Chen at 2010 and 2009[11,12], Samanta at al, 2009 and 2011[13,14], Kundu and Samantha in 2009[17], Kim, Kang, Baik & Kk, 2007[15], Linzhong et al, 2004[16], proposed their approaches on activity diagrams. Pretschner and Jan 2005[21] presented that the omission and encapsulation of data and details need to be considered in abstraction. However, unplanned omission leads to ambiguities and inconsistencies. Bock and Gruninger 2005[18] defined abstraction as a deliberate and clearly identified omission of unnecessary and redundant information, whereas ambiguity is an unintended and unidentified omission of useful information. In the case of unavailability of required information or when the intended system is not fully defined or understood, the modeler may need to make assumptions. Premniner and Pretschner 2005[20] pointed out that the ambiguities may arise from erroneous or partial behavioral assumptions. Bock and Gruninger 2005[18] also suggest that imprecise and implicit functional assumptions are the main cause of miscommunications and ambiguities. According to Heckel 2003[23], the obscure semantics of a modeling language is another source of ambiguities in a model. In a recent study, Lange and Chaudron 2005[22] noted that UML models in practice often contain several syntactical and semantical defects when used. They further stated that some of these issues stem from the inability or failure of the modeler to correctly and precisely specify the system in the model. Frisch et al. 2002[19] suggested that while developing models at certain levels of abstraction may lead to pitfalls in models and they suggested refinements of the model are required for addressing these problems. Ranjit kumar and Vikas panthi[25] suggests algorithms to generate test cases from UML Activity diagrams by traversing the activity diagram from beginning to end, showing choices, conditions, concurrent executions, loop statements.

III. UML DIAGRAMS

The Unified Modeling Language (UML) is a specification defined by the Object Management Group in the year 2010. UML 2 defines 13 diagrams, divided into two general sets:

3.1 Structural Diagrams

Structure diagrams define the static architecture of a model. They are used to depict its specification that make up a model including classes, objects, interfaces, physical components and to model the relationships and dependencies between elements.

1. Package diagrams are used to organize elements into packages and describe the interactions between them.
2. Class diagrams is a collection of basic building blocks of a model: the types, classes, contents and their relationships used to construct the model
3. Object diagrams show how instances of structural elements are related and used at run-time.
4. Composite Structure diagrams depicts the internal structure of a component, class or a use case by layering an element's structure and focusing on inner detail, construction and relationships.
5. Component diagrams are used to model components that compose the system including their interrelationships, interactions and interfaces.
6. Deployment diagrams show the execution architecture of the system including hardware and software execution environment and the physical disposition of significant artefacts within a real-world setting.

3.2 Behavioral Diagrams

Behavior diagrams capture the behavioural features of a software system or a business process.

1. Use Case diagrams are used to model user interactions. They define behavior, requirements and constraints in the form of scripts or scenarios. They show use cases, actors and their relationships.
2. Activity diagrams are used in representing the basic program flow, to capturing the decision points and actions within any generalized process. They are mainly used to represent high level business process, complex software systems and scientific systems.
3. State Machine diagrams are essential to understanding the instant to instant condition or "run state" of a model when it executes and describes the states of an object or an interaction and their transitions.

3.3 Interaction Diagrams.

Interaction diagrams are subset of behavioural diagrams that emphasize mainly on object interactions.

1. Communication diagrams show the communications between objects at run-time, instances of classes, their interrelationships and message flows between them.
2. Sequence diagrams are closely related to Communication diagrams and show the sequential flow in terms of time ordering and messages passed between objects.
3. Timing diagrams are a combination of Sequence and State diagrams to represent the change in state of an element over time in response to externals events.
4. Interaction Overview diagrams are a combination of Activity and Sequence diagrams to provide interaction diagrams to be easily combined with decision points and flows.

IV. UML DIAGRAMS

The Unified Modeling Language (UML) is a specification defined by the Object Management Group in the year 2010. UML is widely used for object-oriented analysis and design. UML consists of diagrams which describe structures and behavior of a system. The structural diagrams declare components of a system and how each component is connected to one another. Examples of a structural diagram are the class diagram, the component diagram, the composite structure diagram, and the deployment diagram. The behavioral diagrams depict the behavior of a system in different ways. Examples of behavioral diagrams are the activity diagram, interaction diagram, state machine diagram, and use case diagram. Software designers use a combination of structural diagrams and behavioural diagrams to specify the system.

4.1 UML activity diagram

Activity diagram is another important diagram in UML to describe dynamic aspects of the system. Activity is a particular operation of the system. Activity diagrams are not only used for visualizing dynamic nature of a system but they are also used to construct the executable system by using forward and reverse engineering techniques. It does not show any message flow from one activity to another. Activity diagram is basically a flow chart to represent the flow form one activity to another activity. The activity can be described as an operation of the system. So the control flow is drawn from one operation to another. This flow can be sequential, branched or concurrent. Activity diagrams deals with all type of flow control by using different elements like fork, join etc.

**Definition:** An activity diagram \( AD = (N, E, I, F) \) consists of a finite set \( N \) of nodes and edges where \( N \) consists of set of action nodes and control nodes. Control Nodes are further partitioned into set \( ND \) of decision nodes, merge nodes, fork nodes and join nodes. \( E \) is the set of control flows that exists between nodes. \( I \) is the initial node and \( F \) is the set of flow final and activity final nodes.

4.2 Basic notation used in activity diagram

There are two of elements in activity diagrams: nodes and edges. Each node is an ActionNode, ControlNode, or ObjectNode. Nodes in activity diagram are connected by edges which include ControlFlow edges or ObjectFlow edge.

a. Initial node. The filled in circle is the starting point of the diagram.
b. Activity final node. The filled circle with a border is the ending point. An activity diagram can have zero or more activity final nodes.

c. Activity. The rounded rectangles represent activities that occur.

d. Flow/edge. The arrows on the diagram.

e. Fork. A black bar with one flow going into it and several leaving it. This denotes the beginning of parallel activity.

f. Join. A black bar with several flows entering it and one leaving it. All flows going into the join must reach it before processing may continue. This denotes the end of parallel processing.

g. Condition. Text such as correct or incorrect, yes or no on a flow, defining a guard which must evaluate to true in order to traverse the node.

h. Decision. A diamond with one flow entering and several leaving.

i. Merge. A diamond with several flows entering and one leaving. The implication is that one or more incoming flows must reach this point until processing continues, based on any guards on the outgoing flow.

j. Swimlanes. Swimlanes, indicate who/what is performing the activities.

k. Flow final. The circle with the X through it. This indicates that the process stops at this point.

Figure 1. Activity Diagram Notations

4.3 Dealing with ambiguities in UML Activity Diagrams.

The development process of software systems usually occurs on two different levels. First a model of the system is built. It represents the required system behavior and usually represents an abstraction of the system. When the model is revealed to be correct, code is generated from the model. This is the software level. The reason for building those intermediate levels is the fact, that it is much cheaper and faster to modify a model than to change the final product. The entire process is called model-based development (MBD). Abstraction is a key approach to deal with the complexity in modeling [Booch 2010]. A complex model is developed with a simplified view of the system through the use of abstraction. However, unplanned abstraction can leave ambiguity and inefficiency while generating test cases, where abstraction is defined as a deliberate and clearly identified omission of unnecessary and redundant information, whereas ambiguity is an unintended and unidentified omission of useful information. In model-based development, effectiveness and efficiency of a model depends on the accuracy
information available in it. Models at a higher level of abstraction provide a simplified view of a complex system but it may omit vital and useful details resulting in ambiguities in the specification. On the other hand, a low level model with concrete information not only limits the usage of the specified model, but also increases the cost and effort needed to develop and maintain it. UML 2 has become an industry standard for modeling, analysis, design and development of software systems, business and scientific models. Although the syntax of UML languages empowers flexibility in expressing very complex systems at various levels of abstraction, it often causes error and ambiguities in the specification depicted in it. In the case of ambiguous information available, the system to be developed needs to take care of various types of assumption such as the data related assumptions (e.g. type, variable or value), functional assumptions (e.g. parameters, functional logic, procedure calls and error handling) and flow related assumptions (i.e. sequencing, loop and concurrency) [Dye 2002]. Basing on the assumption types, the ambiguities are categorized as functional ambiguities, flow/ambiguities and data ambiguities. In this paper we deal with the flow ambiguities in the pretransluation step. So the model that is used to generate test cases should cope with the standards of UML. The activity diagram that is used for testing is pretranslated to a well formed model. In the preprocessing stage we transform the activity diagram into a well formed diagram which ensures the connectivity between nodes and removing the dangling objects. In well formed diagram every initial node has no incoming and one outgoing edge. Action node has one incoming and one outgoing edge. Fork has a single incoming and multiple outgoing edges. Join has multiple inflows and a single outflow. Decision node has a single inflow and 2 or more outflows. Merge has multiple inflows and a single outflow. Flow final and activity final nodes have single inflow and no outflow. Any other deviations from these inflows are corrected in this pretranslation step by inserting the necessary implicit, explicit fork, join, decision and merge nodes where ever they are needed. To make the corrections every node’s indegree and outdegree is taken into consideration and the necessary nodes are added and the edges are adjusted accordingly. Below are some activity diagram nodes are their indegree and outdegree suggested by OMG group in their specification [Formal-05-07-05].

<table>
<thead>
<tr>
<th>Type of the node</th>
<th>Indegree</th>
<th>Outdegree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity Action</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Initial Node</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Activity Final Node</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Fork</td>
<td>1</td>
<td>Multiple</td>
</tr>
<tr>
<td>Merge</td>
<td>Multiple</td>
<td>1</td>
</tr>
<tr>
<td>Decision</td>
<td>1</td>
<td>2 or More</td>
</tr>
<tr>
<td>Flow Final</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Following are some ambiguities and their corrections.

Some of the ambiguous notations in activity diagrams and their modifications are as follows. Left side is the ambiguity and its correction on right side. A flow ambiguity between activity action nodes where the outdegree of an activity is greater than one is represented on the left and its correction is insertion of fork or a decision node.
A flow ambiguity between activity action nodes where the indegree of an activity is greater than one is represented on the left and its correction is insertion of join or a merge node.

**Ambiguous Initial and Activity Final nodes.**

![Diagram of ambiguous initial and activity final nodes]

Ambiguous fork node with indegree greater than one is corrected to insert an merge node above the fork node to control flow ambiguity.

![Diagram of ambiguous fork node correction]

### 4.4 Dealing with Concurrency in UML Activity Diagrams.

Activity diagrams can be classified into two types based on concurrency, non concurrent activity diagrams and concurrent activity diagrams. To generate simple paths for concurrent activity diagrams we have to consider the order in which activities occur in diagram[25]. Basing on the relations we generate all basis paths that obey ordering of relations. Let us consider an example activity diagram and its graph. The ordering of the relations is Initial<Fork, Fork<A, Fork<C, A<B, C<D, D<E, B<Join, E<Join and Join<Final. The set of paths that follow ordering are:

- Initial Fork A B C D E Join Final
- Initial Fork C D E A B Join Final
- Initial Fork A C B D E Join Final
- Initial Fork A C D B E Join Final
- Initial Fork A C D E B Join Final
- Initial Fork C A D E B Join Final
- Initial Fork C A D B E Join Final
- Initial Fork C D A B E Join Final.

© 2013, [http://www.journalofcomputerscience.com](http://www.journalofcomputerscience.com) - TJCSA All Rights Reserved
The ordering of the relations is Initial<Fork, Fork<A, Fork<C, A<B, C<D, D<E, B<Join, E<Join and Join<Final. The possible sets of allowed paths are

- Initial Fork A B C D E Join Final
- Initial Fork A C B D E Join Final
- Initial Fork A C D E B Join Final
- Initial Fork A C D B E Join Final
- Initial Fork C A D E B Join Final
- Initial Fork C A B D E Join Final

To generate all possible paths we either apply BFS or DFS in between fork or join. The set of paths when BFS is applied is

- Initial Fork A C B D E Join Final
- Initial Fork C A D B E Join Final.

The path that is generated when DFS is applied is

- Initial Fork A B C D E Join Final
- Initial Fork C D E A B Join Final.

Consider another example that has decision merge between fork and join and the paths are generated.

Paths By applying BFS

- Initial Fork A C B Decision D E Merge Join Final
Initial Fork C A Decision D E Merge Join Final.
By applying DFS

Initial Fork A B C Decision D Merge Join Final
Initial Fork A B C Decision E Merge Join Final
Initial Fork C Decision D Merge A B Join Final
Initial Fork C Decision E Merge A B Join Final.

But when we consider the paths the set of paths generated using BFS implements concurrency but produces erroneous paths. So we consider the paths that are generated using DFS which are efficient in nature.

V. PROPOSED METHODOLOGY

In this section we discuss our proposed approach to generate test cases from UML activity diagram after removing ambiguities and efficiently executing the concurrent paths.

1. The necessary information is gathered from the UML activity diagram.
2. Check whether any ambiguities are present in the diagram. Remove the ambiguities by adding the necessary nodes.
3. Convert the activity diagram to an activity precedence graph.
4. Generate test cases from the graph by applying DFS between Fork and Join Pair.

In the first step we try to gather the data from the activity diagram like node type, node name and the set of inflows and outflows, indegree and outdegree of every node is stored in a table. We check the indegree and outdegree of every node to find if it has any ambiguities. If ambiguity is present we insert Decision Fork, Join or Merge nodes according to the requirement. We check if there a missing fork for every join and missing join for every Fork. Nodes are inserted and the necessary changes are done to the inflows and outflows of evry node. Once again we check for whether the ambiguities are removed are not. By using the set of nodes and edges we construct the activity graph by applying the set of rules for mapping of an activity diagram to graph by applying DFS between initial node and activity final node and we execute very loop atleast for zero or one number of times so that true side and false side of every decision is covered.

<table>
<thead>
<tr>
<th>S.NO.</th>
<th>Constructs of Activity Diagram</th>
<th>Node of Activity Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initial Node</td>
<td>Node of type I with no incoming edge</td>
</tr>
<tr>
<td>2</td>
<td>Activity Final Node</td>
<td>Node of type E with no outgoing edge</td>
</tr>
<tr>
<td>3</td>
<td>Flow Final Node</td>
<td>Node of type E with no outgoing edge</td>
</tr>
<tr>
<td>4</td>
<td>Decision Node</td>
<td>Node of type D</td>
</tr>
<tr>
<td>5</td>
<td>Guard Condition associated decision node</td>
<td>Node of type C and associated with condition string. Its parent node is of type D</td>
</tr>
<tr>
<td>6</td>
<td>Merge Node</td>
<td>Node of type M and having single outgoing edge</td>
</tr>
<tr>
<td>7</td>
<td>Fork Node</td>
<td>Node of type F with single incoming edge</td>
</tr>
<tr>
<td>8</td>
<td>Guard condition associated with fork node</td>
<td>Node of type C and its parent node is of type F</td>
</tr>
<tr>
<td>9</td>
<td>Join Node</td>
<td>Node of type J and will have one outgoing edge</td>
</tr>
<tr>
<td>10</td>
<td>An object ‘OB’ at input/output pin of an activity ‘AC’</td>
<td>Node of type O and associated object name is ‘OB’. Its parent node will be of type ‘A’ and associated activity name ‘AC’. If same object ‘OB’ is in both input and output pin of the activity ‘AC’, then only one node is to be used.</td>
</tr>
<tr>
<td>11</td>
<td>Object state ‘S’ of an object ‘OB’</td>
<td>Node of type OS. If ‘OB’ is at input of an activity, then this node is left child of node of type O and associated object name is ‘OB’ otherwise this node is right child of parent node associated object name with ‘OB’.</td>
</tr>
<tr>
<td>12</td>
<td>Activity Node</td>
<td>Node of type A. Its associated string is activity name.</td>
</tr>
</tbody>
</table>

VI. CASE STUDY: LOGIN SCREEN

Login screen initially displays a message “Ask for Username and Password”. The user enters the username and password and the system checks if the entered username and password are valid or not. If they are valid and the first time to login it asks for change of password. If not the first time to login it displays the main page if wrongly it asks to login again.
A. The activity diagram for the above case study is

```
InitialNode: InitialNode
ActivityAction: Ask-for-username-and-password
ActivityAction: Verify-username-and-password
DecisionNode: Valid-login
ActivityAction: Notify-user
ActivityAction: Record-error
DecisionNode: Is-new
ActivityAction: Include-change-password
ActivityFinalNode: Welcome-user-to-the-system
JoinNode: JoinNode
```

Figure 2. The Activity Diagram for the Login Screen

**Table 3. Ambiguity Table.**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>In</th>
<th>Out</th>
<th>Ambiguity</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitialNode</td>
<td>InitialNode</td>
<td>0</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td>ActivityAction</td>
<td>Ask-for-username-and-password</td>
<td>2</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>ActivityAction</td>
<td>Verify-username-and-password</td>
<td>1</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td>DecisionNode</td>
<td>Valid-login</td>
<td>1</td>
<td>2</td>
<td>no</td>
</tr>
<tr>
<td>ActivityAction</td>
<td>Notify-user</td>
<td>1</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td>ActivityAction</td>
<td>Record-error</td>
<td>1</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td>DecisionNode</td>
<td>Is-new</td>
<td>1</td>
<td>2</td>
<td>no</td>
</tr>
<tr>
<td>ActivityAction</td>
<td>Include-change-password</td>
<td>1</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td>ActivityAction</td>
<td>Welcome-user-to-the-system</td>
<td>1</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td>ActivityFinalNode</td>
<td>ActivityFinalNode</td>
<td>1</td>
<td>0</td>
<td>no</td>
</tr>
<tr>
<td>JoinNode</td>
<td>JoinNode</td>
<td>2</td>
<td>1</td>
<td>no</td>
</tr>
</tbody>
</table>

B. The activity diagram is converted to well form activity graph and the graph is as follows.
Table 4. Ambiguity Table

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>In</th>
<th>Out</th>
<th>Ambiguity</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitialNode</td>
<td>InitialNode</td>
<td>0</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td>ActivityAction</td>
<td>Ask-for-username-and-password</td>
<td>1</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td>ActivityAction</td>
<td>Verify-username-and-password</td>
<td>1</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td>DecisionNode</td>
<td>Valid-login</td>
<td>1</td>
<td>2</td>
<td>no</td>
</tr>
<tr>
<td>Notify-user</td>
<td></td>
<td>1</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td>DecisionNode</td>
<td>Record-error</td>
<td>1</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td>Is-new</td>
<td></td>
<td>1</td>
<td>2</td>
<td>no</td>
</tr>
<tr>
<td>Include-change-password</td>
<td></td>
<td>1</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td>Welcome-user-to-the-system</td>
<td></td>
<td>1</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td>ActivityFinalNode</td>
<td>ActivityFinalNode</td>
<td>1</td>
<td>0</td>
<td>no</td>
</tr>
<tr>
<td>MergeNode</td>
<td>MergeNode</td>
<td>2</td>
<td>1</td>
<td>no</td>
</tr>
</tbody>
</table>

Activity graph

To generate test cases from an activity diagram we convert the activity diagram to an intermediate form known as activity graph. Activity graph is a directed graph that contains start node, end node, decision nodes guard condition node, join node, fork node. The edges represent the flow in the activity diagram.

The activity graph generated has been proposed by Kundu and Samantha. They proposed a set of rules for mapping an activity diagram to activity graph.

Algorithm to generate test cases

To generate test cases the set of basis paths are obtained from the activity graph by applying DFS. A Basis path is a path in which every loop is executed zero or one times. This ensures that all nodes in the activity graph are traversed at least one or twice to ensure that both true and false values of the loop are executed at least once. For the above diagram there are four basis paths.

The Possible paths generated from the above graph are:

- src node : Initial Node => Merge Node => Ask for Username and Login => Valid login => yes => Is new => no => Join Node => Welcome user to system => Activity Final Node : dst node.
- src node : Initial Node => Merge Node => Ask for Username and Login => Valid login => yes => Is new => yes => Include change password => Join Node => Welcome user to system => Activity Final Node : dst node.
- src node : Initial Node => Merge Node => Ask for Username and Login => Valid login => no => Notify user => Record error => Merge Node => Ask for Username and Login => Valid login => yes => Is new => no => Join Node => Welcome user to system => Activity Final Node : dst node.
- src node : Initial Node => Merge Node => Ask for Username and Login => Valid login => no => Notify user => Record error => Merge Node => Ask for Username and Login => Valid login => yes => Is new => yes => Include change password => Join Node => Welcome user to system => Activity Final Node : dst node.
Number of all possible and unique routes = 4

“Process Order” Use Case

Let us consider the process order use case description[24]: “Once the order is received the activities split into two parallel sets of activities. One set fills and sends the order while the other handles the billing. On the Fill Order side, the method of delivery is decided conditionally. Depending on the condition either the Overnight Delivery activity or the Regular Delivery activity is performed. Finally the parallel activities combine to close the order.”

Paths generated are
Receive Order => Fill Order => Rush Order => Yes => Overnight Delivery => Send Invoice => Receive Payment => Close Order
Receive Order => Fill Order => Rush Order => No => Regular Delivery => Send Invoice => Receive Payment => Close Order
Receive Order => Send Invoice => Receive Payment => Fill Order => Rush Order => No => Regular Delivery => Close Order
Receive Order => Send Invoice => Receive Payment => Fill Order => Rush Order => Yes => Overnight Delivery => Close Order
V. Mary Sumalatha, Dr. G.S.V. Raju, The International Journal of Computer Science & Applications (TJCSA)  ISSN – 2278-1080, Vol. 2 No. 10 December 2013

VII. CONCLUSION

In this paper we presented the test case generation of UML activity diagram. Moreover our method for test case generation by removing ambiguities produces a set of test cases that are more efficient and effective. In future, it is possible to build an automatic tool using this approach. This automatic tool will reduce cost of software development and improve quality of the software.

REFERENCES