An Efficient Approach for Model Based Test Path Generation

Aye Aye Kyaw and Myat Myat Min

Abstract—**Software testing plays a vital role in developing software that is freedom from bugs and defects. Manual test is a cost and time consuming process although it may find many defects in a software application. If the testing process could be automated, the cost of developing software could be reduced obviously. The most critical part of the testing process is generation of test cases. Model based test path generation approaches identify faults in the implementation at early design phase, reduce the software development time, and inspire developer to improve design quality. The proposed approach focuses on Model Based testing for automatic best test path generation. In this approach, the proposed system generates all possible test paths and optimizes a test path from these paths. The proposed system will reduce the processing time by using XMI based activity diagram, validate the generated test paths with Cyclomatic Complexity and choose the best test path without dependence on others.**

Index Terms—**Model-based testing, test path, test path generation, UML activity diagram.**

I. Introduction

Software testing is an essential part of the software development life cycle. Software testing is also an activity that should be done throughout the whole development process [1]. The testing process consists of three parts: test case generation, test execution, and test evaluation. Comparing with the other two parts, test case generation is more challenging and difficult [2]. Generating test data form high level design notations has several advantages over code-based test case design. Testing based on design models has the advantage that the test cases remain valid even when the code changes a little bit. Design models can be used as a basis for test case generation, significantly reducing the costs of testing [3].

There are many testing strategy and among them, some are Model-Based Testing (MBT) that depends on extracting test cases from different models (requirements models, usage models, and models constructed from source code), Specification-based testing (or black-box testing) that depends on requirements models, and Program-based testing (or white-box testing) that uses source code as the underlying model [4].

Unified Modeling Language (UML) is a de-facto standard for modeling analysis and design artifacts. UML models are an important source of information for test case design, which is satisfactorily exploited, can go a long way in reducing testing cost and effort and at the same time improve software quality [5]. 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, activity diagrams are treated as one of the most important design artifact among several UML diagrams. As UML activity diagram captures the key system behavior, so it is well suited for the system level testing of systems [6]. Activity diagram is an important diagram among 13 diagrams supported by UML 2.0. It is used for business modeling, control and object flow modeling, complex operation modeling etc. Main advantage of this model is its simplicity and ease of understanding the flow of logic of the system [7].

In recent trend, Model-Based test case attracts many researchers by using some data mining concept to produce an automated optimal test case. Most of existing automation tool using MBT is performed step by step. (e. g. constructing dependency table, creating dependency graph, generating possible paths). The efficient and effective approaches are needed still although there are many existing automatic test case generation approaches [8]. With this motivation, we aim our work at generating all possible test paths and optimizing the best test path from activity diagram. Our approach minimizes the size of test, time and cost, and reduces steps at the evolution of generated test paths.

The rest of the paper is organized as follows: The next section reviews the use of activity diagrams for software testing and optimization techniques. Section III scribes the architecture of our proposed system with an illustration. Section IV presents a case study to demonstrate the use of our methodology with the Home Insurance Proposal management system. The paper concludes with Section V.

II. UML Activity Diagram Modeling

This section describes the fundamentals about activity diagram. Activity diagrams are static-behavioral. Compared to use case diagrams, activity diagrams are capable of documenting the flow within a use case or within a system. An important characteristic of activity diagrams is their ability to show dependency between activities. An activity diagram has two kinds of modeling elements: Activity nodes and Activity edges.

A. *Activity Nodes*

There are two main kinds of nodes in activity diagrams:

- Action nodes (AN): Action nodes consume all input data/control tokens when they are ready to generate new tokens and send them to output activity edges. Action, an individual step within an activity, can posses input and output information. The output of one action can be the input of a subsequent action within an activity.
- Control nodes (CN): Control nodes route tokens through

Manuscript received May 5, 2014; revised July 23, 2014.

The authors are with the University of Computer Studies, Mandalay, Myanmar (e-mail: ayeayekyaw2009@gmail.com, myatiimin@gmail.com).

the graph. The control nodes include constructs to start the diagram, to terminate the diagram, to choose between alternative flows (decision/ merge), to split or merge the flow for concurrent processing (fork / join). Control node is an activity node used to coordinate the flows between other nodes. Control nodes are Initial node, Flow final node, Activity final node, Decision node, Merge node, Fork node, Join node.

B. *Activity Edges*

Edges represent flow of control through the activity. It connects the individual components of activity diagrams [9].

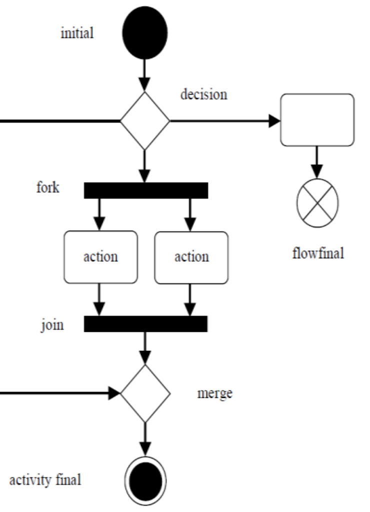


Fig. 1. Elements of activity diagram.

III. Model Based Test Path Generation

Various authors have used the following architecture for generating the test path.

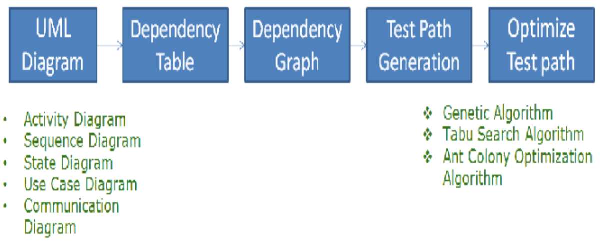


Fig. 2. System flow of test path generation.

Following are the steps of test case generation by using activity diagram among other UML diagram. The reasons that activity diagram is used as input are:

1) The concepts at a higher abstraction level compared to other diagrams like sequence diagrams, class diagrams, etc. are presented.
2) The results in path are the presence of loop and concurrent activities in the activity diagram.
3) It is difficult to consider all execution paths for testing [10].

The description of each step in Figure 2 will be illustrated as follows:

A. *Generation of ADT*

Activity diagram is used to automatically generate the Activity Dependency Table (ADT) with all the activities. These activities include decisions, loops and synchronization along with the entity performing the activity. ADT also consist of the input and the expected output values for each activity. Dependency of each activity on others is also shown clearly in ADT. The repeated activities in the diagram are grouped into one symbol only instead of having several symbols for the same activity.

B. *Generation of ADG*

The ADT is accomplished to automatically generate the Activity Dependency Graph (ADG). The symbols given for each activity are used to name the nodes in the ADG where each node represents an activity in the activity diagram. Since repeated activities are given the same symbol in the ADT, only one node is created for them no matter how many times they are used in the activity diagram. This will decrease the search space in the ADG. The transitions from one activity to another are represented by edges in the ADG. The presence of an edge from a node to another is determined by checking the dependency column in the ADT for the current node's symbol. Specifically, if it contains the previous node's symbol then an edge from the previous node to the current one is drawn in the ADG.

C. *Generation of Test Path*

The generated ADG applied to obtain all the possible test paths. A test path is composed of steps represented by successive symbols/nodes (representing the activities) forming a complete path from the start node in ADG to the end node separated by arrows. Details are then extracted from the ADT and added to each node in the test path to obtain all the final test cases. Each node in the test case is accompanied with its input and expected output. Besides, the whole test case will be accompanied with its initial input and final expected output [7].

D. *Generation of Best Test Path / Optimal Test Path Generation*

Many researchers have been successfully proposed test case generation for much software, using mainly search optimization techniques such as Genetic Algorithm, Ant Colony Optimization Algorithm, Tabu Search Algorithm *et al.*

IV. Architecture of Proposed System

The proposed system uses the activity diagram as an input for the automated algorithm of generating test paths. This model constructs three main modules as shown in Fig. 3.

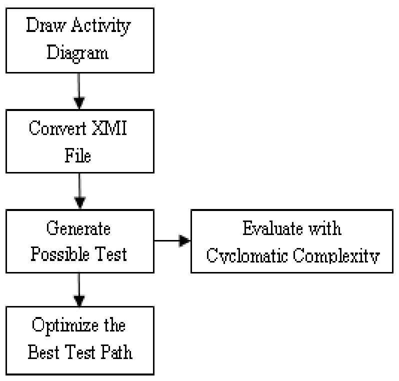

Draw Activity Diagram
Convert XMI File
Generate Possible Test
Evaluate with Cyclomatic Complexity
Optimize the Best Test Path

Fig. 3. Overall architecture of the proposed system.

The description of each module will be illustrated as follows:

A. Conversion to XMI File

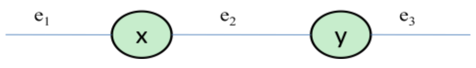
The Modelio activity diagram is converted to XMI file. The converted XMI file is the input to the proposed system. In order to generate test path in the next step, some important information must be detected and extracted as the followings:

1) The "packagedElement" is a tag that provides description of the activity diagram: diagram type, identity of diagram, and diagram name.
2) The "node" is a tag that provides description of a node in the activity diagram: node type, identity of node, node name, identity of incoming edges, and identity of outgoing edges. Inside incoming edge and outgoing edge, if it has more than one edge, each edge will be separated with the space character.
3) The "edge" is a tag that provides description of an edge in the activity diagram: edge type, identity of edge, edge name, identity of source node, and identity of target node.

The fundamental elements of the activity diagram are actions node and control nodes such as Initial node, Flow final node, Activity final node, Decision node, Merge node, Fork node, Join node. Edges connect the individual components of activity diagrams. These node and edge in the activity diagram are mapped with the extracted from the above information.

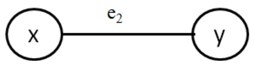
B. Generation of Possible Test Path

The extracted information is applied to obtain all the possible test paths. A test path is composed of steps represented by successive symbols/nodes (representing the activities) forming a complete path from the start node in the activity diagram to the end node separated by arrows. Details are then extracted from the XMI file and added to each node in the test path to obtain all the final test cases. This module generates all possible test paths by applying the following assumption;



$$\begin{aligned}&\forall x,y,e_1,e_2,e_3\ (\mathit{Target}(x,e_1)\wedge \mathit{Source}(x,e_2))\wedge\\&(\mathit{Target}(y,e_2)\wedge \mathit{Source}(y,e_3))\Leftrightarrow \mathit{Path}(x,y)\end{aligned}\qquad(1)$$

where *x*, *y* are nodes in the activity diagram and e_1, e_2, e_3 are edges in the activity diagram. Alternatively,



$$\forall x,y,e_2\ \mathit{Source}(x,e_2)\wedge \mathit{Target}(y,e_2)\Leftrightarrow \mathit{Path}(x,y)\qquad(2)$$

where *x*, *y* are nodes in the activity diagram and e_2 is an edge in the activity diagram.

C. Optimization of the Best Test Path

With the help of these test cases are generated, most prioritized test case are generated by applying the proposed assumption. The proposed assumption is as follows;

$$\begin{aligned}&\exists p\ \mathit{Has}(p,\mathit{MaxControlNode})\wedge\\&\mathit{Has}(p,\mathit{MaxTotalNode})\Leftrightarrow \mathit{BestPath}(p)\end{aligned}\qquad(3)$$

where *p* is a path in the activity diagram, MaxControlNode is maximum number of control node of the test path and MaxTotalNode is maximum number of total node of the test path.

If a test path has maximum total number of control nodes and maximum total number of nodes, it is the best test path for given activity diagram.

D. Validation of the Generated Test Path

The minimum numbers of test cases are computed that should be covered for each activity diagram using one of the following two ways:

1) Cyclomatic complexity is defined as:

$$V=E-N+2\qquad(4)$$

where *E*, is the number of edges; and N is the number of nodes.

2) Cyclomatic complexity is also defined as:

$$V=P+1\qquad(5)$$

where *P* is the number of predicate nodes contained in the diagram.

Branch coverage<=Cyclomatic complexity<=no. of paths

The generated test cases apply Branch coverage criteria, and the Cyclomatic complexity coverage. In some cases the generated test cases exceed the Cyclomatic complexity criterion which means that the proposed model's test cases apply the full-path coverage criteria. The proposed model applies the hybrid coverage criterion [11]-[14].

V. Case Study

This section presents a case study of Home Insurance Proposal management system. Problem statement: The system represents the process of making an insurance proposal. The system receives 'AcceptHomeInsurance Proposal'. One dealing with validation of the proposal details called 'ValidateProposalDetails' and the other with obtaining the underwriter's approval, 'ObtainUnderwriter Approval'. ValidateProposalDetails may take only half an hour, but ObtainUnderwriterApproval may take a day. It is only when both of these activities, with different time frames, are complete that the next activity can start. Either the insurance proposal is valid, or it is invalid and the cover is rejected. The activity diagram for the above problem is as shown in Fig. 4.

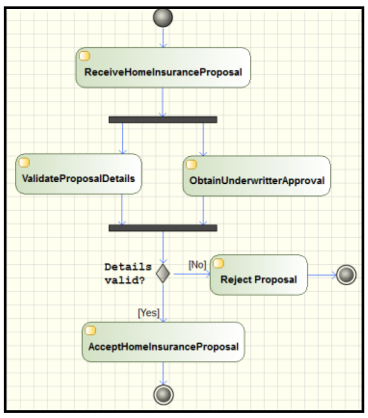
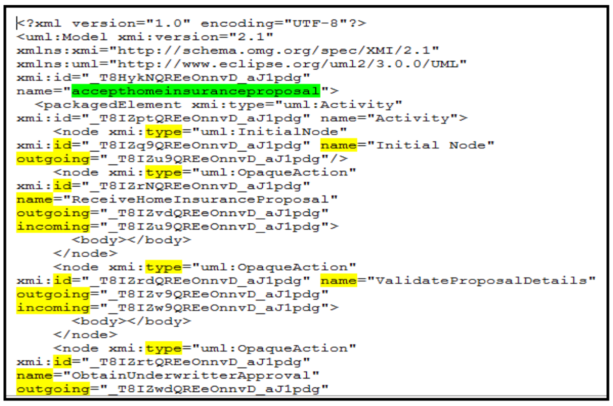

ReceiveHomeInsuranceProposal
ValidateProposalDetails
ObtainUnderwritterApproval
Details valid?
[No]
Reject Proposal
[Yes]
AcceptHomeInsuranceProposal

Fig. 4. Activity diagram for home insurance proposal management system.

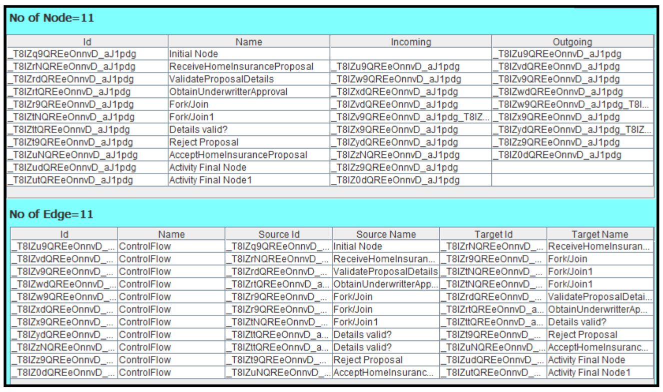
In Fig. 5, the nodes and edges information of the activity diagram is extracted from converted XMI file of the activity diagram.



```
<?xml version="1.0" encoding="UTF-8"?>
<uml:Model xmi:version="2.1"
xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
xmlns:uml="http://www.eclipse.org/uml2/3.0.0/UML"
xmi:id="_T8HykNQREeOnnvD_aJ1pdg"
name="accepthomeinsuranceproposal">
  <packagedElement xmi:type="uml:Activity"
xmi:id="_T8IZptQREeOnnvD_aJ1pdg" name="Activity">
    <node xmi:type="uml:InitialNode"
xmi:id="_T8IZq9QREeOnnvD_aJ1pdg" name="Initial Node"
outgoing="_T8IZu9QREeOnnvD_aJ1pdg"/>
    <node xmi:type="uml:OpaqueAction"
xmi:id="_T8IZrNQREeOnnvD_aJ1pdg"
name="ReceiveHomeInsuranceProposal"
outgoing="_T8IZvdQREeOnnvD_aJ1pdg"
incoming="_T8IZu9QREeOnnvD_aJ1pdg">
      <body></body>
    </node>
    <node xmi:type="uml:OpaqueAction"
xmi:id="_T8IZrdQREeOnnvD_aJ1pdg" name="ValidateProposalDetails"
outgoing="_T8IZv9QREeOnnvD_aJ1pdg"
incoming="_T8IZw9QREeOnnvD_aJ1pdg">
      <body></body>
    </node>
    <node xmi:type="uml:OpaqueAction"
xmi:id="_T8IZrtQREeOnnvD_aJ1pdg"
name="ObtainUnderwritterApproval"
outgoing="_T8IZwdQREeOnnvD_aJ1pdg"
```

Fig. 5. Converted XMI file from activity diagram.

The number of node and edge and detailed information are shown in Fig. 6.



No of Node=11

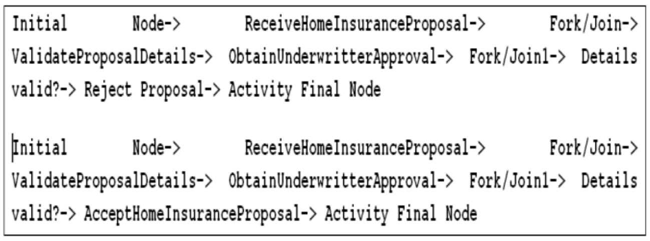
Id	Name	Incoming	Outgoing
_T8IZq9QREeOnnvD_aJ1pdg	Initial Node		_T8IZu9QREeOnnvD_aJ1pdg
_T8IZrNQREeOnnvD_aJ1pdg	ReceiveHomeInsuranceProposal	_T8IZu9QREeOnnvD_aJ1pdg	_T8IZvdQREeOnnvD_aJ1pdg
_T8IZrdQREeOnnvD_aJ1pdg	ValidateProposalDetails	_T8IZw9QREeOnnvD_aJ1pdg	_T8IZv9QREeOnnvD_aJ1pdg
_T8IZrtQREeOnnvD_aJ1pdg	ObtainUnderwritterApproval	_T8IZxdQREeOnnvD_aJ1pdg	_T8IZwdQREeOnnvD_aJ1pdg
_T8IZr9QREeOnnvD_aJ1pdg	Fork/Join	_T8IZvdQREeOnnvD_aJ1pdg	_T8IZw9QREeOnnvD_aJ1pdg_T8I...
_T8IZtNQREeOnnvD_aJ1pdg	Fork/Join1	_T8IZv9QREeOnnvD_aJ1pdg_T8IZ...	_T8IZx9QREeOnnvD_aJ1pdg
_T8IZttQREeOnnvD_aJ1pdg	Details valid?	_T8IZx9QREeOnnvD_aJ1pdg	_T8IZydQREeOnnvD_aJ1pdg_T8IZ...
_T8IZt9QREeOnnvD_aJ1pdg	Reject Proposal	_T8IZydQREeOnnvD_aJ1pdg	_T8IZz9QREeOnnvD_aJ1pdg
_T8IZuNQREeOnnvD_aJ1pdg	AcceptHomeInsuranceProposal	_T8IZzNQREeOnnvD_aJ1pdg	_T8IZ0dQREeOnnvD_aJ1pdg
_T8IZudQREeOnnvD_aJ1pdg	Activity Final Node	_T8IZz9QREeOnnvD_aJ1pdg	
_T8IZutQREeOnnvD_aJ1pdg	Activity Final Node1	_T8IZ0dQREeOnnvD_aJ1pdg	

No of Edge=11

Id	Name	Source Id	Source Name	Target Id	Target Name
T8IZu9QREeOnnvD...	ControlFlow	_T8IZq9QREeOnnvD_...	Initial Node	_T8IZrNQREeOnnvD_...	ReceiveHomeInsuran...
T8IZvdQREeOnnvD...	ControlFlow	_T8IZrNQREeOnnvD_...	ReceiveHomeInsuran...	_T8IZr9QREeOnnvD_...	Fork/Join
T8IZv9QREeOnnvD...	ControlFlow	_T8IZrdQREeOnnvD_...	ValidateProposalDetails	_T8IZtNQREeOnnvD_...	Fork/Join1
T8IZwdQREeOnnvD...	ControlFlow	_T8IZrtQREeOnnvD_a...	ObtainUnderwritterApp...	_T8IZtNQREeOnnvD_...	Fork/Join1
T8IZw9QREeOnnvD...	ControlFlow	_T8IZr9QREeOnnvD_...	Fork/Join	_T8IZrdQREeOnnvD_...	ValidateProposalDetai...
T8IZxdQREeOnnvD...	ControlFlow	_T8IZr9QREeOnnvD_...	Fork/Join	_T8IZrtQREeOnnvD_a...	ObtainUnderwritterAp...
T8IZx9QREeOnnvD...	ControlFlow	_T8IZtNQREeOnnvD_...	Fork/Join1	_T8IZttQREeOnnvD_a...	Details valid?
T8IZydQREeOnnvD...	ControlFlow	_T8IZttQREeOnnvD_a...	Details valid?	_T8IZt9QREeOnnvD_...	Reject Proposal
T8IZzNQREeOnnvD...	ControlFlow	_T8IZttQREeOnnvD_a...	Details valid?	_T8IZuNQREeOnnvD_...	AcceptHomeInsuranc...
T8IZz9QREeOnnvD...	ControlFlow	_T8IZt9QREeOnnvD_...	Reject Proposal	_T8IZudQREeOnnvD_...	Activity Final Node
T8IZ0dQREeOnnvD...	ControlFlow	_T8IZuNQREeOnnvD_...	AcceptHomeInsuranc...	_T8IZutQREeOnnvD_...	Activity Final Node1

Fig. 6. Extracted the information details of activity diagram.

Based on these extracted information, all possible test paths are generated directly from XMI file as mentioned in section 3, B. Fig. 7 shows these generated test paths.

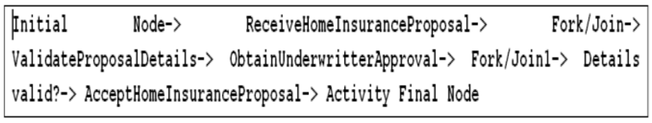


```
Initial Node-> ReceiveHomeInsuranceProposal-> Fork/Join->
ValidateProposalDetails-> ObtainUnderwritterApproval-> Fork/Join1-> Details
valid?-> Reject Proposal-> Activity Final Node

Initial Node-> ReceiveHomeInsuranceProposal-> Fork/Join->
ValidateProposalDetails-> ObtainUnderwritterApproval-> Fork/Join1-> Details
valid?-> AcceptHomeInsuranceProposal-> Activity Final Node
```

Fig. 7. All possible test paths.

And then the best test path is optimized depend on number of control nodes and number of total nodes of each test path according to Section III-C. The best test path is shown in Fig. 8.



```
Initial Node-> ReceiveHomeInsuranceProposal-> Fork/Join->
ValidateProposalDetails-> ObtainUnderwritterApproval-> Fork/Join1-> Details
valid?-> AcceptHomeInsuranceProposal-> Activity Final Node
```

Fig. 8. The best test path.

We can see that the number of predicates are two, edges E are 11 and nodes N are 11, then the will be as follows:

$$V = E - N + 2 = 11 - 11 + 2 = 2 \quad testpaths$$

Then the upper bound of test paths that ensures the full activity path coverage is 2 test paths which were produced by the proposed model. Since,

Branch coverage<=Cyclomatic complexity<=no. of paths

Therefore, the branch coverage criteria, and the Cyclomatic complexity coverage are applied by the generated test cases.

VI. Conclusion

Most of existing automation tool using MBT is needed to the fundamental steps such as constructing dependency table, creating dependency graph, generating possible paths. The proposed system can develop to be more accurate all possible test paths due to generate directly from XMI file instead of using the dependency table and graph. The generated test paths are validated with Cyclomatic Complexity. Moreover the proposed system saves time in choosing the best test path because other optimization algorithms spend time to calculate weight but the proposed approach does not need to evaluate weight. As the human effort (cost and time) in finding bugs and errors, and the steps at the evolution of generated test paths are reduced, the proposed system is more efficient and effective in the software development.

References

[1] A. Bertolino, "Chapter 5: Software Testing," *IEEE SWEBOK Trial Version 1.00*, May 2001.

[2] A. V. K. Shanthi and G. Mohan Kumar, "A heuristic technique for automated test cases generation from UML activity diagram," *Journal of Computer Science and Applications*, vol. 4, no. 2, pp. 75-86, 2012.

[3] S. Srivastava, S. Kumar, and A. K. Verma, "Optimal path sequencing in basis path testing," *International Journal of Advanced Computational Engineering and Networking*, ISSN (PRINT): 2320-2106, vol. 1, iss. 1, 2013.

[4] S. S. Priya and P. D. Sheba. "Test Case Generation from UML models-A survey," in *Proc. International Conference on Information Systems and Computing (ICISC-2013)*, INDIA, January 2013, vol. 3, special iss. 1.

[5] P. N. Boghdady, N. L. Badr, M. Hashem, and M. F. Tolba, "A proposed test case generation technique based on activity diagrams," *International Journal of Engineering & Technology IJET-IJENS*, vol. 11, no, 3.

[6] A. A. Kyaw and M. M. Min, "Model-Based automatic optimal test path generation via search optimization techniques: A critical review," in *Proc. the 12th International Conference on Computer Applications 2014*.

[7] R. K. Swain, V. Panthi, and P. K. Beher, "Generation of test cases using activity diagram," *International Journal of Computer Science and Informatics*, ISSN (PRINT): 2231 –5292, vol. 3, 2013.

[8] T. D. Trong: "A systematic procedure for testing UML designs," in *Proc. ISSRE*, 2003.

[9] A. V. K. Shanthi and G. MohanKumar, "A novel approach for automated test path generation using TABU search algorithm," *International Journal of Computer Applications (0975 – 888)*, vol. 48, no. 13, June 2012.

[10] V. M. Sumalatha and G. S. V. P. Raju, "A model based test case generation technique using genetic algorithms," *The International Journal of Computer Science & Applications (TIJCSA)*, vol. 1, no. 9, November 2012, ISSN: 2278-1080.

[11] P. N. Boghdady, N. L. Badr, M. Hashem, and M. F. Tolba, "A proposed test case generation technique based on activity diagrams," *International Journal of Engineering & Technology IJET-IJENS*, vol. 11.

[12] M. Chen, P. Mishra, and D. Kalita, "Coverage-driven automatic test generation for UML activity diagrams," in *Proc. the 18th ACM Great Lakes symposium on VLSI*, 2006, pp. 139–142.

[13] R. K. Swain, V. Panthi, and P. K. Behera, "Generation of test cases using activity diagram," *International Journal of Computer Science and Informatics*, ISSN (PRINT): 2231 –5292, vol. 3, iss. 2, 2013.

[14] S. Tahiliani and P. Pandit, "A survey of UML-Based approaches to testing," *International Journal of Computational Engineering Research*, vol. 2, iss. 5, September 2012.

[15] S. K. Swain and D. P. Mohapatra, "Test case generation from behavioral UML models," *International Journal of Computer Applications (0975 – 8887)*, vol. 6, no. 8, September 2010.

Myat Myat Min received the PhD in information technology from the University of Computer Studies, Yangon, Myanmar. She is an associate professor and the head of Computer Software Development and Technology Department at the University of Computer Studies, Mandalay. Her special fields of interest included software engineering, operating system and wireless network system.

Aye Aye Kyaw was born on March 3, 1985. She graduated from the University of Computer Studies, Mandalay, Myanmar and she studied her M.C.Sc at the University of Computer Studies, Mandalay. She is persuing Ph.D in University of Computer Studies, Mandalay. Her employment experience includes tutor in Computer University (Lashio). Her special fields of interest included software engineering, UML-based.