2013 International Conference on Communication Systems and Network Technologies

Testcases Formation using UML Activity Diagram

Puneet E. Patel
Dept. of Computer Engineering
R. C. Patel Institute of Technology,Shirpur
Dhule, India
1510.puneet@gmail.com

Nitin N. Patil
Dept. of Computer Engineering
R. C. Patel Institute of Technology,Shirpur
Dhule, India

Abstract— **Most of the software practitioners believe in Model-based test case generation. Advantages of this are the early detection of faults, reducing software development time etc. The most important part of the testing attempt is the test case generation. As a modeling language, Unified Modeling Language (UML) is generally used to describe design specification and analysis by both software practitioners. Therefore, UML becomes the sources of Test cases which are usually generated from the requirement, and UML activity diagram illustrates the sequential control flows of activities that make it possible to generate test cases for activity diagrams. This research paper focuses on implementation and comparison of two approaches to automatically generate test cases directly from UML activity diagram. The first approach is A novel Approach to Generate test Cases from UML Activity Diagrams (Debasish Kundu and Debasis Samanta, 2009) and second is Automatic Test Case Generation From UML Activity Diagram using Activity Path (Yasir Dawood Salman Almulham, 2010).Both the approaches make use of activity graph. Finally we have the statistical graph of generated number of test cases by both approaches. In implementation we have also used the specification to draw activity diagrams.**

Keywords — **Activity Diagram, Test Cases, Activity Graph, Activity Path.**

I. INTRODUCTION

The new paradigm to develop the software is to use a Model-driven approach. The advantage behind this is the increasing efficiency with its supports in many domains like solution, development, and business problems. In the development of the model-driven software, Unified Modeling Language (UML) has become the industry standard for object oriented software development, also are efficient enough to hold most of functioning phase. One of the biggest challenges in software testing is the test cases generation. It becomes especially complicated when a system contains simultaneously executing participant, since a system like that can show different response depending on the simultaneous occurrence condition. A UML activity diagram is suitable modeling language for describing interaction between system object given that an activity diagram can be conveniently used to capture business processes, workflow and interaction scenarios [12]. There are many programs and applications with different purposes and types which have been used everywhere. Test case generation is one of the most important elements for the testing efforts for programs and application [17].

This research paper focuses on test case generation using UML activity diagram by applying activity path approach, in which the test case generating will be automatically generated. The activity path is a method to calculate all the possible paths from the activity diagram, and by converting each number in the path to its path to its original data, which will generate the test cases. Through applying of the activity graph method for generating the test case study, enhancing and proving the algorithm will open the opportunity for the new automatic generation of the test cases. Also, it will open the way to full automatic tests. The same testing method will also be reusable by any other case study to generate it automatically.

The purpose of this research is to compares the algorithms of Kundu & Samantha [12] and Yasir Dawood Salman Almulham [25] in order to generate test cases automatically from activity diagrams and also to evaluate both algorithms in terms of its usability and reliability. Therefore, all the testing will be generated automatically to get riddance from all error of the manual test.

II. SCOPE OF RESEARCH

This research work of this paper can be classified in the area of model-based test case generation and software development. Java language is used to implement the algorithms; the algorithms are constricted to generate test cases from any activity diagram. The activity diagram specification will be inputted into the system manually, the system will generate test cases from the activity diagrams, and code reads the inputted specification only.

III. LITERATURE REVIEW

The L. C. Briand et al. [15] propose the TOTEM system for system level testing. In their approach, all possible invocation of use case sequences is captured in the form of an activity diagram. L. C. Briand et al. [15] consider sequence diagrams to represent use case scenarios. Further, they propose to derive various test information, test requirements, test cases, and test oracles from the detailed design description embedded in UML diagrams, and expressed in Object Constraint Language (OCL). In another work, J. Hartmann et al. [11] describe an approach of system testing using UML activity diagrams. Their approach takes the textual description of a use case as input, and converts it into an activity diagram semi-automatically to capture test cases. They also add test requirements to the test cases with the help of stereotypes. Test data are then generated applying category partition method. In an approach on scenario-based testing, Xiaoqing BAI et al. [24] consider hierarchies of activity diagrams where top level activity diagrams capture use case dependencies, and low level activity diagrams represent behavior of the use cases. First eliminate the hierarchy structure of the activity diagrams, and subsequently, they convert them into a flattened system level

978-0-7695-4958-3/13 $26.00 © 2013 IEEE
DOI 10.1109/CSNT.2013.191

activity diagram. Finally, it is converted into an activity graph by replacing conditional branches into its equivalent execution paths and concurrency into serial sequences. Such thin threads are further processed to generate test cases.

Activity diagrams are also used for gray-box testing and checking consistency between code and design. Wang Linzhang et al. propose an approach of gray-box testing using activity diagrams [18]. In gray-box testing approach, test cases are generated from high level design models, which represent the expected structure and behavior of software under testing. Test scenarios are generated from this activity diagram following basic path coverage criterion, which tells that a loop is to be executed at most once. Basic path coverage criterion helps to avoid path explosion in the presence of a loop. Test scenarios are further processed to derive gray-box test cases. Chen Mingsong et al. present an idea to obtain the reduced test suite for an implementation using activity diagrams [4]. The Authors [4] consider the random generation of test cases for Java programs. Finally, reduced test suite is obtained by comparing the simple paths with program execution traces. Simple path coverage criterion helps to avoid the path explosion due to the presence of loop and concurrency.

IV. UML Activity Diagram

The activity diagram extracts the centre idea from flowcharts. The activity diagram contains activity states, which are represented in the implemented of a statement in a process or the performance of an activity in a workflow [4].

UML activity diagrams describe the ordering of atomic pieces of behavior, called activities. The notation is inspired by flowcharts, state transition graph. An activity diagram can be used to model complex processes that have parallelism, loops and event driven behavior. They can also be used to model the behavior of some use cases or to specify the workflow or business process. The nodes represent processes or process control, including action states, activity states, decisions, swim lanes, forks, joins, objects, signal senders and receivers. The edges represent sequences of activities including the control flows, message flows and signal flows. Activity state and action state are denoted with round corner boxes. Transitions are shown as arrows. Branches are shown as diamonds with one incoming arrow and multiple exit arrows. Each arrow may be labeled with a Boolean expression to be satisfied to choose the branch. Forks and joins are represented by multiple arrows entering or leaving a synchronization bar. The following control nodes coordinate the flow in an activity:

Initial- The starting node of the activity.

Decision-A decision node has one incoming flow and multiple outgoing flows, of which only one will be taken. Guard condition should be made to insure that only one flow can be taken.

Merge - A control node that brings together multiple alternate flows. It is not used to synchronize concurrent flows, but to accept one among the several flows.

Fork - A control node that splits an incoming flow into multiple concurrent outgoing flows. Tokens arriving at a fork are duplicated across each outgoing flow.

Join - A join node is a control node that synchronizes multiple flows. If there is a token offered on all incoming flows, then a token is offered on the single outgoing flow.

Activity End - A node that stops all flow activity.

Flow Final -A node that terminates that flow.

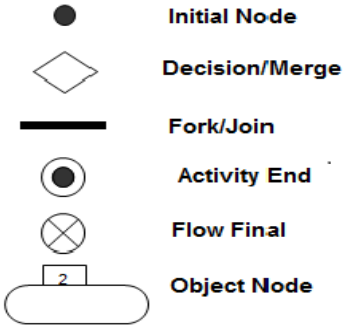

Fig. 1: Control Node of an Activity Diagram

An edge starts from the source node and terminates at the target. An edge is labeled with an expression of the form *e[c] =a*, where *e* is an event expression, *c* is a guard condition expression, and *a* is an action expression. Events are also called signals in UML. Each of these three components is optional. An edge with label *e[c]=a* has the following meaning: If the system is in the source state, the event e occurs, and the guard condition c evaluates to true, then the system transits out of the source state, the action a is performed, and the target state is entered. A guard expression is a conditional/Boolean expression that can refer to local variables in UML diagram. The local variables of diagram are Boolean, integers and strings.

An activity diagram models describes the expected behaviours of an operation. The incorrect implementation of the activity diagram will result in unexpected behaviours of the operation of the operation, if there is any defect in the implementations of the operation modeled in the activity diagram

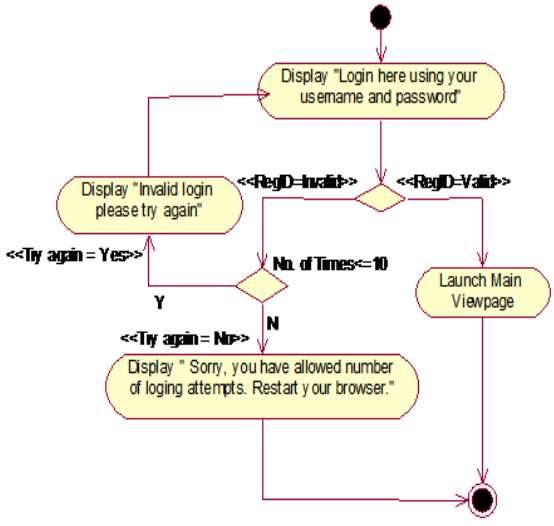

Fig. 2: Simple Activity Diagram for login screen

V. UML ACTIVITY GRAPH

One of the ways to generate the test cases is by using activity graphs. An activity graph is a directed graph while its construct is represented by each node in the activity graph (initial node, decision node, flow final node, guard condition, join node, fork node, etc.), and from that each border of the activity graph symbolizes the stream in the activity diagram. Furthermore, the activity graph symbolizes assemble of an activity diagram in an organized way which can be used for further automation [13].

Kundu and samata [13] proposed a set of rules for mapping construct of an activity diagram into nodes of an activity graph. These nodes have ten different types in activity graph: *S (start node), A (activity), E (flow final), O (object), M (merge), OS (object state), F (fork), D (decision), J (join) and C (condition).*

These nodes are used for test case generation due to their capability for detecting more faults in the synchronization and loop faults than the other approaches. In addition the ability to identify location of the faults will help to reduce testing afford and having model-based test case generation to improve and develop design quality. Furthermore, it has the possibility to built automatically prototype for the activity graph. The algorithm by Kundu & Samantha [13] has taken the activity graph as an input and the output from it will be the activity path. Furthermore, the generated paths will need some of rules to apply on the result paths to get all the possible paths that will be needed to generate the test cases [13].

These nodes are used for test case generation due to their capability for detecting more faults in the synchronization and loop faults than the other approaches. In addition the ability to identify location of the faults will help to reduce testing afford and having model-based test case generation to improve and develop design quality. Furthermore, it has the possibility to

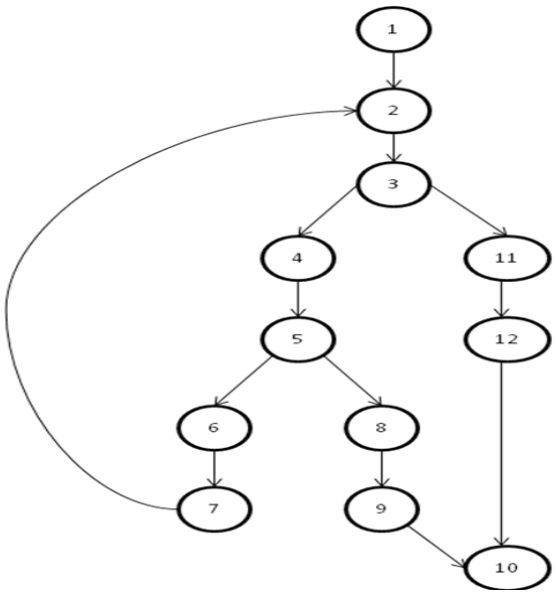

Fig. 3: Activity Graph for login screen.

built automatically prototype for the activity graph. The algorithm [13] has taken the activity graph as an input and the output from it will be the activity path. Furthermore, the generated paths will need some of rules to apply on the result paths to get all the possible paths that will be needed to generate the test cases.

VI. TEST COVERAGE CRITERIA

Test coverage indicates the extent to which a testing criterion such as path testing, branch testing or basis path testing is achieved. A popular test coverage criterion involves covering all the basis paths. Since basis path coverage is a stronger test coverage than transition or state coverage, it ensures that all edges visited at least once. So this will give us activity path coverage. These are finite as the numbers of basis paths are also finite.

A test coverage criterion [8] is a set of rules that guide to decide appropriate elements to be covered to make test case design adequate. We discuss the existing test case coverage criteria followed by our proposed criterion, which we consider as an improved test coverage criterion.

A. Basic path coverage criterion

First, we define basic path in activity graph. A basic path is a sequence of activities where an activity in that path occurs exactly once [18][4]. Note that a basic path considers a loop to be executed at most once.

Given a set of basic paths P_B obtained from an activity graph and a set of test cases T, for each basic path $P_i \in P_B$, there must be at least one test case $t \in T$ such that when system is executed with the test case t, p_i is exercised.

B. Simple path coverage criterion

A *simple path* is considered for activity diagrams that contain concurrent activities [4]. It is a representative path from a set of basic paths where each basic path has same set of activities, and activities of each basic path satisfy identical set of partial order relations among them. Note that partial order relation between two activities A_i and A_j, denoted as $A_i < A_j$ signifies that activity A_i has occurred before activity A_j.

Given a set of simple paths P_S for an activity graph which contains concurrent activities and a set of test cases T, for each simple path $p_i \in P_S$ there must be a test case $t \in T$ such that when system is executed with a test case t, p_i is exercised.

C. Activity path coverage criterion

Activity path coverage criterion, Activity Path Coverage is a sequence of non-concurrent activities (that is, activities which are not executed in parallel) from the start activity to an end activity, where each activity in the sequence has at most one occurrence except those activities that exist within a loop. Each activity in a loop may have at most two occurrences in the sequence.

The aim is to use this coverage criterion for both loop testing and concurrency among activities of activity diagrams. Before describing our new coverage criterion, we have mention about activity path and types of activity paths, namely (i) non-concurrent activity path, and (ii) concurrent activity path.

First, consider a precedence relation as given below. A precedence relation, denoted as '$\prec$', over a set of activities S_A in the activity diagram is defined as follows-

1. If an activity $A_i \in S_A$ precedes a fork F and $A_j \in S_A$ is the first activity that exists in any thread originated from the fork F, then $A_i \prec A_j$.
2. If an activity $A_j \in S_A$ follows next to a join J and $A_k \in S_A$ is the last activity in any thread joining with the join J, then $A_k \prec A_j$.
3. If $A_i \in S_A$ and $A_j \in S_A$ are two consecutive concurrent activities in a thread originated from a fork F where A_i exists before A_j in the thread, then $A_i \prec A_j$.

An activity path is a path in an activity graph that considers a loop at most two times and maintains precedence relations between concurrent and non-concurrent activities.

Non-concurrent activity path is a sequence of non-concurrent activities (that is, activities which are not executed in parallel) from the start activity to an end activity in an activity graph, where each activity in the sequence has at most one occurrence except those activities that exist within a loop. Each activity in a loop may have at most two occurrences in the sequence.

Concurrent activity path is a special case of non-concurrent activity path, which consists of both non-concurrent and concurrent activities satisfying precedence relation among them.

Next, define the activity path coverage criterion:

Given a set of activity paths P_A for an activity graph and a set of test cases T, for each activity path $p_i \in P_A$ there must be a test case $t \in T$ such that when system is executed with a test case t, p_i is exercised.

TESTING RESULTS

In this section I have considered both approaches to generate test cases from activity diagram using activity path and also have compared the result of both approaches. Algorithm 1: By Debasish Kundu and Debasis Samanta, [13]

This approach consists of the following three steps.

1. Augmenting the activity diagram with necessary test information.
2. Converting the activity diagram into an activity graph.
3. Generating test cases from the activity graph.

A guideline for modeling necessary test information into an activity diagram is described as-

1. For an activity Ai that changes the state of an object *O Bi* from state *Sa* to state *Sb*, we can show state Sa of object *O Bi* along *with O Bi* at input pin of the activity Ai and state *Sb* of the object *O Bi* along with *O Bi* at output pin of *Ai* [2][5].

2. For an activity Ai that creates an object *O Bi* during execution, we can show that object *O Bi* at output pin of the activity *Ai*.

3. We can replace a loop, decision block or fork-join block in any thread originated from a fork by an activity with higher abstraction level.

Note that we are not considering the details of each activity such as what are actions encapsulated in an activity and what is the input, output parameters of each action in order to preserve simplicity of activity diagrams.

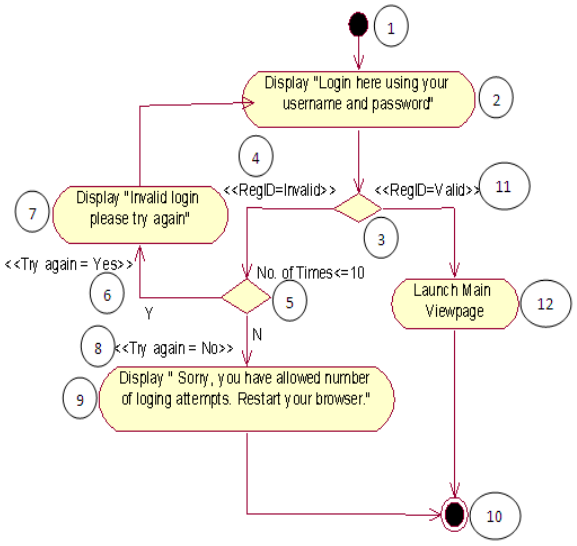

Fig:4. Activity Diagram for "Login" with node number.

The second algorithm by Yasir Dawood Salman Almulham, [25] also uses the activity diagram as input to generate test cases. It makes use of certain variables to find all possible paths.

To find the number of paths, the following are the variables used:

a. TDN-the number of node which has two directions.
b. RNN-the number of return node.
c. TDNI-the number of two directions inside each return node.
d. RNNI-the number of return node inside each return node.

Basic paths with the return paths number (BPR) =TDN+1
Basic paths number (BP) =BPR-RNN
Number of paths after return (PAR) =TDNI-RNNI
The number of all paths (AP) =BP+PAR(s)

The generated paths along with the saved specifications are used as test case for the given module.

To find the basic paths with the return paths, the following are the variables used:

a. Node - to put the nodes
b. LoopFlag - to keep track of the loop
c. N1 - for the first nest of the nodes
d. N2 - for the second nest of the nodes
e. Nodeflag - for keeping track of the visited nodes
f. End - for know the end of the path

Begin
LoopFlag = TDN, Node = first node, Nodeflag = 0, End = ture

While LoopFlag is not empty do
While End is true do
If (its the first node) then put the fisrt node in the path;
Put the node first nest in N1and its second next in N2;
If (N1 = 0) then put end to true;
If(NodeFlag = 0 and N2 = 0) then
put N1 for the next node in the path;
Else if (NodeFlag = 0 and N2! = 0) then
put N1 for the nest node in the path and
put NodeFlag = 1,
put NodeFlag = NodeFlag the number of two direction node before it;
Else if (NodeFlag = 1 and N2! = 0) then
put N2 for the nest node in the path and
put NodeFlag = 2;
Put the next node of the path as the current node;
End
Put the path in array;
End

To found all the (AP)

Duplicate the paths that have without End to the (TDNI + RNNI) number;

Put the return paths inside it to continue it to the number of paths inside the return path; Finish each one from the size basic paths;

From the saved information of each node, print out the details of path to the last one, what will give us the test case.

CONCLUSION

A. Findings and Results

The algorithms has been implementing as a prototype that is able to generate the test case automatically. The algorithms will generate all the possible paths from the UML activity diagram followed by refining each number of the activity path to its details to create test cases. This is because the sequences of the automatically generated test cases are complete and correct.

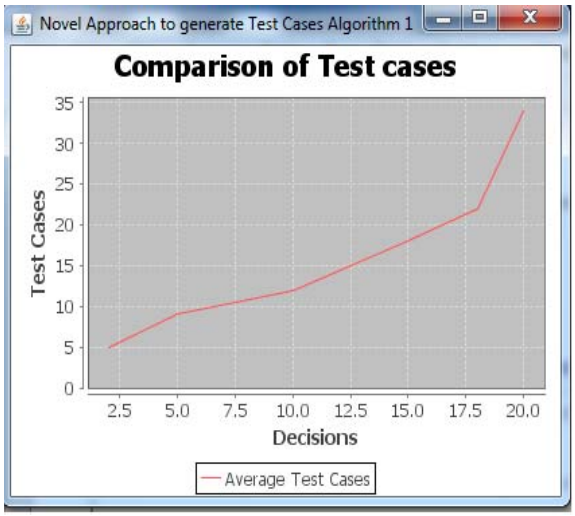

Fig:5. Graph for Algorithm1 number of Testacses Vs Decisions

B. Limitation

In this research the focus was only on the UML activity diagram with a single use case at the time. It has not considered the infinite loop if found in the diagram, as well as the decisions made in diagram are correct.

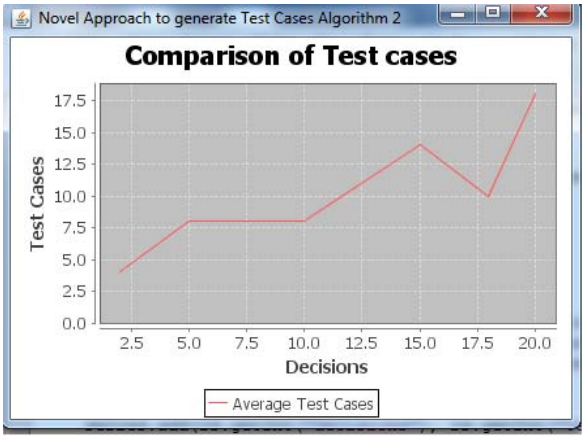

Fig:6. Graph for Algorithm2stating number of Testacses Vs. Decisions

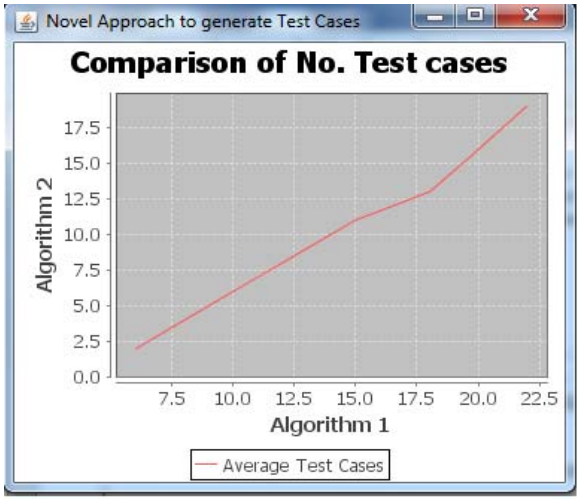

Fig:7. Comparison Graph of Testcase generation

C. Future Work

In the future, the algorithm to generate test cases can be generalized so that it can accommodate various test coverage criteria within the same test derivation framework. Also, the prototype developed has an ability to build the directed graph and parse test case from it.

D. Conclusion

In this research, I have compared the two approaches for generating test cases from activity diagram at use case scope. Also a test coverage criterion, called *activity path* coverage criterion has been implemented. In the present submission, I have focused only activity diagram of a single use case at a time. However, activity diagrams of multiple use cases which are related to each other by various relationships such as,

include, extend, generalization / specialization can be considered.

REFERENCES

[1] B. P. Douglass. Real Time UML: Advances in The UML for Real-Time Systems. Addison Wesley, Third Edition, February 2004.

[2] Cartaxo, E., Neto, F., & Machado, P. (2007). Test Case Generation by means of UML Sequence Diagram and Labeled Transition System. *IEEE,* 1292-1297.

[3] Chen, M., Qiu, X., Xu, W., Wang, L., Zhao, J., & Li, X. (2009). UML Activity Diagram-Based Automatic Test Case Generation for Java Program. *The Computer Journal,* 52 (5), 545-556.

[4] D. Pilone and N. Pitman. UML 2.0 in a Nutshell. O'Reilly, June 2005.

[5] France, R., Ghosh, S., Dinh-Trong, T., & Solberg, A. (2006). Model-driven development using UML 2.0: promises and pitfalls. *IEEE*, 39 (2), 59-66.

[6] Heumann, J. (2001). *Generating Test Cases From Use Cases.* Retrieved 5 July, 2010, from:

[7] http://www.ibm.com/developerworks/rational/library/content/RationalEdge/jun01/GeneratingTestCasesFromUseCasesJune01.pdf

[8] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. ACM Computing Surveys, 29(4):366-427, December 1997.

[9] Javed, A., Strooper, P., &Waston, G. (2007). Automated Generation of Test Cases Using Model-Driven Architecture. *IEEE*, 3-9.

[10] J. Z. Gao, H.-S. J. Tsao, and Y. Wu. Testing and Quality Assurance for Component-Based Software. Artech House Publishers, 2003.

[11] J. Hartmann, M. Vieira, H. Foster, and A. Ruder. A UML-based approach to system testing. Innovations in Systems and Software Engineering, 1(1):12-24, April 2005.

[12] Kim,H., Kang, S., Baik, J., & Ko, I. (2007). Test Cases Generating from UML Activity Diagrams. *IEEE,* 556-561.

[13] Kundu, D., & Samanta, D. (2009). A Novel Approach to Generate Test Cases from UML Activity Diagrams. *Journal of Object Technology,* 8 (3), 65-83.

[14] Kuo, F. (2009). An indepth study of mirror adaptive random testing. *IEEE,* 51-58.

[15] L.C. Briand and Y. Labiche. A UML-based approach to system testing. In 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools, pp. 194-208, 2001.

[16] Lilly, R., & G, U. (2010). Reliable Mining of Automatically Generated Test Cases from Software Requirements Specification. *International Journal of Computer Science Issues,* 87-91.

[17] Linzhang, W., jiesong, Y., Xiaofeng, Y., Jun, H., Xuandong, L., & Guoliang, Z. (2004). Generating Test Cases from UML Activity Diagram based on Gray-Box Method. *IEEE* 284-291.

[18] Mingsong, C., Xiaokang, Q., & Xuandong, L. (2006). Automatic Test Case Generation for UML Activity Diagrams. *ACM,* 2-8.

[19] Robert, A., Maksimchuk, R., Engle, M., Young, B., Conallen. J., & Houston, K. (2007). *Object-Oriented Analysis and Design with Applications.* New York: Random House.

[20] Ryser. J.,& Glinz, M. (2005). Using Dependency Charts to Improve Scenario-Based Testing. *Elsevier Science*, 85-97.

[21] R. V. Binder. Testing Object-Oriented Systems Models, Patterns, and Tools. Addison Wesley, Reading, Massachusetts, October 1999.

[22] Usman, M., & Nadeem, A. (2009). Automatic Generation of Java Code from UML Diagrams Using UJECTOR. *International Journal of Software Engineering and Its Applications, 21-38.*

[23] Vaishnavi, V., & Kuechler, W. (20 January, 2004). *Design Research in Information System.* Retrieved 16 August, 2009, from: http://desrist.org/design-research-in-information-systems.

[24] X. Bai, C. P. Lam, and H. Li. An approach to generate the thin-threads from the UML diagrams. In 28th Annual International Computer Software and Applications Conference (COMPSAC04), pp. 546-552, 2004.

[25] Yasir Dawood Salman Almulham. Automatic Test Case Generation from UML Activity Diagram Using Activity Path. Malaysia 2010.

[26] Z. Guangmei, C. Rui, L. Xiaowei, and H. Congying. The automatic generation of basis set of path for path testing. In 14th Asian Test Symposium (ATS 05), pp. 46-51, 2005.