ISSN: 2277-9655
Scientific Journal Impact Factor: 3.449
(ISRA), Impact Factor: 1.852

IJESRT
INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

Automatic Test case Generation from UML Activity Diagrams

V.Mary Sumalatha[*1], Dr G.S.V.P.Raju[2]
[1] Research Scholar, Gitam University, Visakhapatnam, Andhra Pradeshm, India
[*2] Professor,Andhra University, Visakhapatnam, Andhra Pradesh, India
vmsumalatha2002@gmail.com

Abstract

Test Case Generation is an important phase in software development. Nowadays much of the research is done on UML diagrams for generating test cases. Activity diagrams are different from flow diagrams in the fact that activity diagrams express parallel behavior which flow diagrams cannot express. This paper concentrates on UML 2.0 Activity Diagram for generating test cases. Fork and join pair in activity diagram are used to represent concurrent activities. A novel method is proposed to generate test case for concurrent and non concurrent activities. Proposed approach details about the importance of concurrent nodes and their execution order in path generation.

Keywords: UML Diagrams, Activity Diagrams, Concurrency, Partial Ordering, Precedence Diagramming.

Introduction

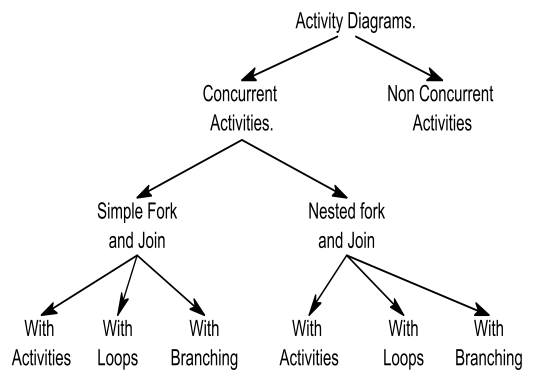
The growth of powerful execution made parallel programming became an essential nowadays. Activity diagrams are different from flow diagrams in the fact that activity diagrams express parallel behavior which flow diagrams cannot express. Fork and join pair in activity diagram are used to represent concurrent activities. To generate test paths fork and join pairs need to be explored. If a fork and join pair contain a set of activities it is easy to generate test paths. But the problem lies when there are decision, merge, and nested fork join contained in them. Fork node indicated by a synchronization bar initiates concurrent activities in an activity diagram where no sequential order is established. Join node indicated by another synchronization bar stops all concurrent activities.

Concurrency in Activity Diagrams

Many researchers proposed methods to verify UML diagrams using XML and XMI. Attribute grammar techniques are used to check the semantic consistency of UML diagrams in XML [Kotb and Katayama,2004]. Activity Diagram was presented in Human readable form called the Activity Diagram

Linear Form which was in Text format [Falter et al.,2007. Activity diagrams were presented using Action Description Languages (ADL) [Narkngam and Limpiyakon, 2012]. Enhancements were made to Action Description languages to verify Activity Diagrams to reduce defects and to consume fewer resources during software development [Kaewchinporn and Lilpiyakorn, 2013]. Activity diagrams are different from flow diagrams in the fact that activity diagrams express parallel behavior which flow diagrams cannot express. This paper concentrates on UML 2.0 Activity Diagram for generating test cases. Activity diagrams can be classified into two types based on concurrency, non concurrent activity diagrams, and concurrent activity diagrams. A fork and join pair in an activity diagram are used to process activities in parallel. Four categories of fork and join pairs are defined by [Xu et al., 2005] namely atomic, simple, nested, and branched fork and join pairs. These categories are further simplified basing on branches present in between fork and join pair.

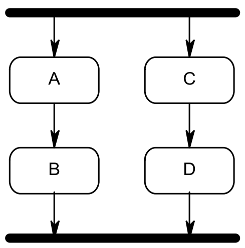
Figure 1: Categories of Fork and Join pairs



ISSN: 2277-9655
Scientific Journal Impact Factor: 3.449
(ISRA), Impact Factor: 1.852

Simple fork and join pair

Simple fork and join contains set activities that can be executed in parallel. Fork node may contain two or more outgoing edges from it and join node may contain two or more incoming edges to it. These are further categorized into simple fork and join with only activities, simple fork and join with decision and merge. Simple with decision and merge are further classified into simple fork and join with loops, Simple fork and join with branching. Simple fork and join with loops contains set activities that can be executed in parallel and a set of activities that loop for n number of times. simple fork and join pair with branching contains set activities that can be executed in parallel and a set of activities contain a decision and a merge node which are used to select alternate paths so that some paths are skipped while executing.



\Figure 2.a Simple fork and join

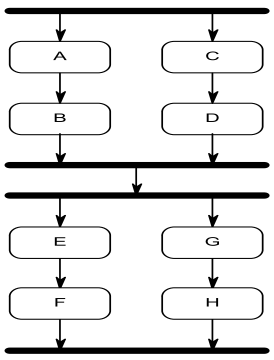


Figure 2.b Two separate fork and join pairs

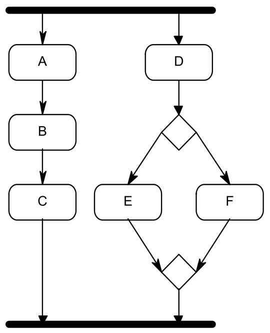


Figure 3.a Fork and join pair with two alternate paths

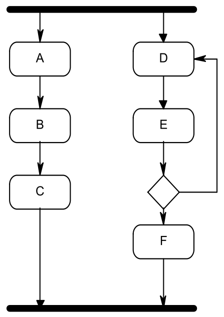


Figure 3.b Fork and join pair with a loop

Nested fork and join.

Nested fork and join pair contains another set of fork and join pair with set activities that can be executed in parallel. In this type the inner most fork join pair needs to be executed before the outer pair starts execution. These are further classified into nested fork and join with only activities, with decision and merge. Nested fork and join with decision and merge are classified into nested fork and join with loops, with branching. Nested fork and join with branching contains another set of fork and join pair with set activities that can be executed in parallel. Either of the fork and join pairs contains a decision and merge node to skip some activities. Nested fork and join with loops contains another set of fork and join pair with set activities that can be executed in parallel. Either of the fork and join pairs contains a loop which indicates that some activities need be executed for n number of times

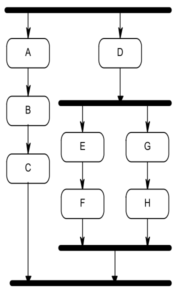


Figure 4.a Nested fork and join pair

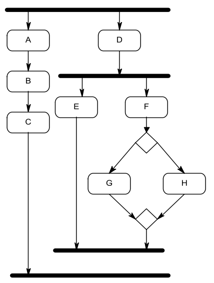


Figure 4.b Nested fork and join with alternate paths

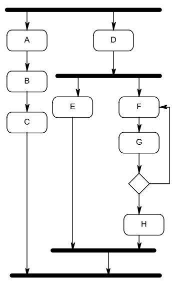


Figure 4.c Nested fork and join with a loop

Literature survey

Debasish Kundu and Debasis Samanta proposed a novel approach to generate test cases from Activity Diagrams which works for both concurrent and non concurrent paths [Debasish Kundu and Debasis Samanta, 2009]. They proposed Basic path, simple path, Activity path coverage criteria to generate test cases. Proposed approach defines two types of paths, non concurrent activity paths and concurrent activity paths. A precedence relation has been defined to find all relations that happened before fork, after join nodes between fork and join. To generate test paths DFS is applied for non concurrent and BFS is applied for concurrent activities. Instead of displaying set of concurrent paths, one path called as representative activity path is generated by applying BFS for the fork and join pair. Proposed work avoids loops or alternate paths between fork and join pair.

Chen Mingsong et al., proposed an automatic test case generation technique for Activity Diagrams with non concurrent paths based on execution traces [Chen Mingsong et al., 2006] To generate paths which include concurrent activities a slight change has been made to DFS algorithm to cover all nodes between fork and join pair. Proposed work mainly concentrates on non concurrent activities and avoids loops and concurrency.

Puneet Patel and Nitin N Patil proposed test case generation from Activity Diagrams which includes both concurrent and non concurrent activities [Puneet Patel and Nitin N Patil, 2012]. A novel test coverage criterion called activity path coverage criterion was developed which works for both loop testing and concurrent activities in Activity diagrams. A relationship called precedence relation was defined for concurrent activities.

a. **Partial ordering and precedence ordering in Activity Diagrams.**

In Mathematics a pair (X,P) is called partially ordered or a poset if X is a set and p is a reflexive, antisymmetric, and transitive binary relation on X. X is called as the ground set and P is a partial order on X. for the first time Lamport in 1978 defined Partial order between events using happened-before relation [Lamport 1978].

Definition of Happened-before-relation. A happened-before B if A and B are within the same process and A occurred before B. If A happened before B, B happened before C then A happened before C.

Activity diagram with concurrent activities includes the partial ordering of the activities. A partial ordering on a set of activities is denoted using ‘<’. A partial ordering between two activities A and B is denoted as A<B represents that activity A happened before activity B. in precedence diagramming method four types of precedence namely finish to start, start to finish, start to start and finish to finish are allowed.

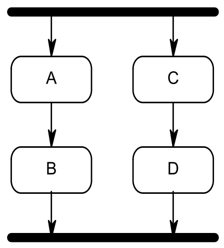
b. **Precedence diagramming method.**

Precedence diagramming method is similar to happened before relation in posets. Precedence diagramming method represents the precedence of nodes that start first and that finish later. Following are the types of precedence diagramming relations between nodes.

1. Finish-to-Start. In finish to start precedence relationship Activity B cannot start until Activity A has completed. In most schedules, all relationships will be finish-to-start.
2. Start-to-Finish. In Start-to-finish precedence relationship Activity A must start before Activity B can finish. This is a very rare relationship.
3. Start-to-Start. In start to start precedence relationship Activity A must start before Activity B can start.
4. Finish-to-Finish. In finish to finish precedence relationship Activity A must finish before Activity B can finish.

When these four precedence relationships are applied to Activity diagrams applicability of all four precedence relationship is not guaranteed.

Figure 5: Simple Fork and Join pair.



In the above figure Finish to Start precedence are A<B, C<D. Finish to Finish precedence are B<Join, D<Join. Start to Start precedence are Fork<A, Fork<C. combining all the above precedence the allowed paths are

1. Fork<A<B<C<D<Join.
2. Fork<A<C<B<D<Join.
3. Fork<A<C<D<B<Join.
4. Fork<C<D<A<B<Join.
5. Fork<C<A<B<D<Join.
6. Fork<C<A<D<B<Join.

While dealing with concurrency all nodes that happened before or the precedence relationships need to be determined. To find all happened before relations between join and fork pair we apply adjacency list representation of a graph which represents all happened before relations. Adjacency list for a directed graph with n nodes and e edges is a list that contains vertex j if there is an edge between vertex i and vertex j.

Proposed Method

The complexity generating test paths increase when concurrency, loops, nesting are present in the Activity Diagram. The numbers of paths to be generated increases drastically when there are more number of nodes between fork and join pair. The main aim of concurrency is to execute nodes independently. So instead of generating a set of concurrent paths we generate a single path in which loops and alternate paths are traversed between fork and join pairs. Complexity increases when more number of nodes is involved in between a fork and join pair. For a simple fork and join pair with 4 activities in between generates 6 paths, if one more activity is added with two outgoing nodes from fork node then the number of paths become 10. For 5 activities with 3 outgoing edges from fork node generates nearly 30 paths. So in the case of nested fork and join pair the complexity gets doubled. So to trace the paths between fork and join pair two well known strategies DFS and BFS are applied to generate paths. DFS produces the Fork<A<B<C<D<join and BFS produces Fork<A<C<B<D<Join. In simple fork and join pairs any of these techniques suites well. But the problem arises when there is a decision merge pair in fork and joins pair. Applying BFS to generate test paths may choose all alternate paths which violates the definition of decision node and is not proper to apply in all cases. By applying DFS produces different paths but violates the definition of parallel execution and limits it to sequential execution. The overall aim of parallel execution lies in the fact that activities can be executed at the same and every activity must be executed at least once except in the case of looping and skipping. So this paper concentrates on finding paths that execute all activities at least once.

So whether it is the case of simple fork and join pair or nested fork and join pair, one thing is guaranteed that every node must be executed at least once. For this, the proposed approach makes modifications to the activity diagram graph by allowing only one incoming and one outgoing nodes to both fork and join nodes. But according to OMG standard fork can have multiple outgoing and join can have multiple incoming nodes. The flow between the fork and join pairs is changed in such a way that the activities from the fork node get executed from left to right, and path by path. After traversing the first path using DFS the flow is connected to the second path and the entire procedure is repeated until join node is reached. By the end of this traversal fork and join nodes will have one incoming and one outgoing nodes. The same procedure is applied to nested fork and joins pair with loops and alternate paths. Consider the following activity diagrams. The simple fork and join pair is modified to generate a single path, nested fork and join pair is modified to generate a single path.

To make these modifications the number of outgoing nodes for fork and number of incoming nodes for join are stored in two separate arrays. Staring from the first outgoing node from fork node, DFS is applied and every node is added to the graph until join is reached. Once the join node is reached instead of pointing to the join node the flow continues to the second outgoing node of fork node. The process is repeated until all outgoing and incoming nodes are traversed for fork and join nodes. Then the last node is connected to join.

a. Pseudo code for dealing with concurrent nodes: Modified Activity Diagram graph.

```
Input: Edge, Node description tables, stack s.
Node description table contains 3 columns namely type of the node, node id, node name in the diagram.
Edge description table contains 2 columns namely source node id and target node id.
Output: Modified Activity Diagram Graph, Modified Edge description table.
Begin
For i:=1 to number of edges in NDT
Begin
If ntype == “ForkNode’ flag=1 else flag == 0; end If;
End for;
If flag==0 then exit
Else
Begin
For i:= 1 to number of elements in NDT
For j=1 to number of elements in EDT
If from[j] in EDT != “ForkNode” in NDT or to[j] in EDT != “JoinNode” in NDT copy the elements in EDT
Else If ntype == “ForkNode”
Begin Store all outgoing edges from frknode in frkfrom array let its length be m.
Push the forknode id onto stack. End;
Else If ntype == “JoinNode”
Begin Store all incoming edges to joinnode in frkto array let its length be n.
Pop element from stack and store it in forknode variable. End;
End for
End for
Add a row to EDT with from=forknode id, its to == frkfrom[1];
For k=1 to number of n -1
For l=2 to number of n,
Add a row to EDT with from = Frkto[k] and to =frkfrom[l];
End for
End for
Add a row to EDT with From = Frkto[n] and To = joinnode id;
End else
End.
```

Consider the following Activity Diagram and its equivalent activity diagram graph and modified Activity Diagram graph.

ISSN: 2277-9655
Scientific Journal Impact Factor: 3.449
(ISRA), Impact Factor: 1.852

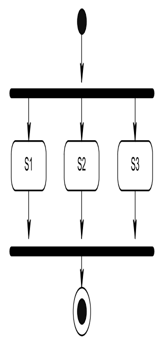


Figure 6.a Activity Diagram

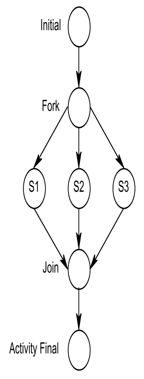


Figure 6.b Activity diagram graph

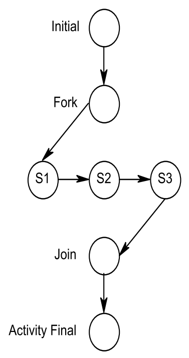


Figure 6.c Modified Activity Diagram graph in which DFS is applied for concurrent nodes.

Node description Table of the Activity Diagram is

Node type	Node ID	Node name
Initial Node	f8LtjEKGAqACAgRS	Initial Node
Fork Node	CUftjEKGAqACAgRl	F1
Activity Action	eNIdjEKGAqACAgT1	S1
Activity Action	Hg4djEKGAqACAgUM	S2
Activity Action	IMkdjEKGAqACAgUc	S3
Join Node	posdjEKGAqACAgUs	J1
Activity FinalNode	0rldjEKGAqACAgVt	Activity FinalNode

Edge Description Table is

Source ID	Destination ID
f8LtjEKGAqACAgRS	CUftjEKGAqACAgRl
CUftjEKGAqACAgRl	eNIdjEKGAqACAgT1
CUftjEKGAqACAgRl	Hg4djEKGAqACAgUM
CUftjEKGAqACAgRl	IMkdjEKGAqACAgUc
eNIdjEKGAqACAgT1	posdjEKGAqACAgUs
Hg4djEKGAqACAgUM	posdjEKGAqACAgUs
IMkdjEKGAqACAgUc	posdjEKGAqACAgUs
posdjEKGAqACAgUs	0rldjEKGAqACAgVt

Modified Edge Description table is

Source ID	Destination ID
f8LtjEKGAqACAgRS	CUftjEKGAqACAgRl
posdjEKGAqACAgUs	0rldjEKGAqACAgVt
CUftjEKGAqACAgRl	eNIdjEKGAqACAgT1
eNIdjEKGAqACAgT1	Hg4djEKGAqACAgUM
Hg4djEKGAqACAgUM	MkdjEKGAqACAgUc
IMkdjEKGAqACAgUc	posdjEKGAqACAgUs

Test Path printing

Concurrency and path printing are related to each other because if a n Activity Diagram contains fork join pair it needs to dealt separately and happened before relation defined in the previous section needs to be satisfied. Here in path printing we divide this step into two parts for convenience to print paths for activity diagrams without concurrent nodes and printing paths for activity diagrams with concurrent nodes.

1. Test path printing from Activity Diagram Graph without concurrent paths.

This section includes visiting the nodes of an Activity Diagram graph in a systematic order using a search algorithm. Basic search algorithms used t traverse graphs are Depth First search and Breadth First Search algorithms. To print all paths in an Activity Diagram Graph we combine Simple Graph Coloring and Depth First Search algorithm to traverse all nodes and edges at least once. DFS starts with the initial node and visiting node by node going away from the initial node to reach the end node and the same process is repeated to print all paths until all nodes become visited. Graph coloring is used to assign colors to each vertex of a graph. The only condition to be satisfied in graph coloring is that no two adjacent vertices should share the common color. In test path printing of Activity Diagrams every node is colored only if a node is visited twice if decision nodes are present else it remains uncolored.

2. Test path printing from Activity Diagram Graph with concurrent and non concurrent paths.

Test paths are generated from Activity Diagrams starting from Initial node to Activity Final node. Proposed work applies DFS for Activity Diagrams with concurrent and non concurrent nodes to generate a single activity path between fork and join pair which is one among the set of concurrent paths to be generated. The activity path generated covers all nodes between fork and join pair at least once. Proposed work generates activity paths for nested fork and joins pairs, fork and join pair with loops, simple fork and join pair without loops, fork and join pair with decision and merge pair in between to

ISSN: 2277-9655
Scientific Journal Impact Factor: 3.449
(ISRA), Impact Factor: 1.852

generate alternate paths. Loops are executed until all nodes are executed at least once.
If the Activity diagram contains no loops, concurrent activities, then simple Depth first Search algorithm is sufficient to print all paths. If loops, alternate paths, concurrent activities are present the DFS is modified slightly so that every loop gets executed at least once and every alternate path gets printed once. The modification done to generate paths is the number of times a node can get visited depends on the number of decision nodes present. If two decision nodes are present a node gets visited twice , if two decision nodes are present then a node gets a change of being visited for 4 times. Back tracking is allowed to print alternate paths by using adjacency matrix representation.

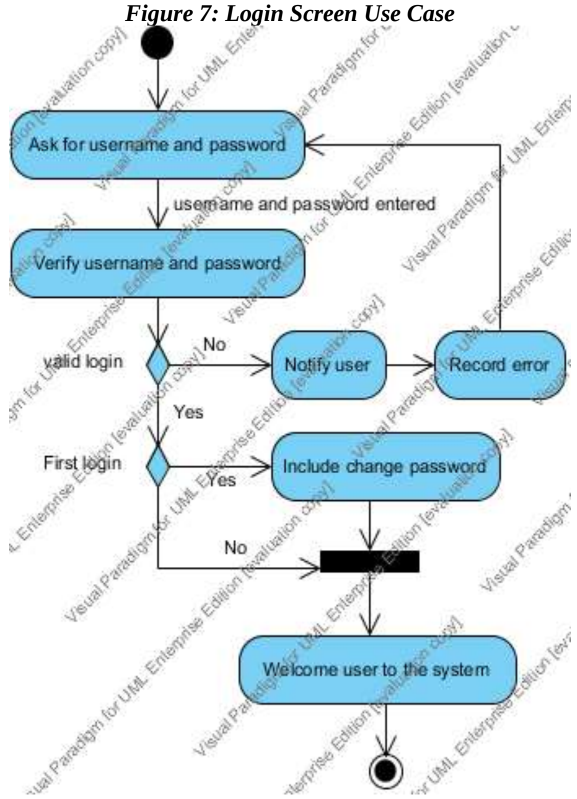
Pseudo code. Generate Activity Paths using Modified DFS Algorithm.

```
Input : Node, Edge description tables, srcpos, dstpos.
Output: set of activity paths.
Begin
Initialize an array al to empty to store all nodes traversed till now.
Generate an adjacency matrix using Edge Description table, which contains 1 if an edge is present between two node else 0 is assigned.
Traverse the activity diagram graph using DFS. For each node visited increase the number of counts. Store the node in the array al.
Start with Initial node's position from Node Description table as src. Add src to al. increment visited[src] by 1.
 If visited[src]==2 then color[src]=true else if visited[src]==1 then color[src]=false;
End
If src==dst then print all nodes stored in al
Else
Begin
Find the adjacent node from src by traverse the adjacency matrix with row number=src and find at least one column I with value 1 in it.
Find if that node is not colored then call DFS with src as I.
If more than one column has 1 in it then back track to the next value and continue DFS with that column value.
End.
```

Case study Login screen

Consider the case study Login Screen use case which waits for the user to enter user name and password. The entered username and password are verified. If the login is valid the user is given a chance to change password if it is first login. If the login is invalid an error is recorded and user is requested to enter new username and password. Login Screen use case, its adjacency list, and the test paths generated are as follows.

Figure 7: Login Screen Use Case


Ask for username and password
username and password entered
Verify username and password
valid login
No
Notify user
Record error
Yes
First login
Yes
Include change password
No
Welcome user to the system

Adjacency list.
mtYU9SKGAqACAgT : chb.8AKGAqACAQ0h
uB2.8AKGAqACAQzL : wtp.8AKGAqACAQzu
mtYU9SKGAqACAgT_
PLq.8AKGAqACAQy8 : 33c0aeac-096f-48
chb.8AKGAqACAQ0h : 54P.8AKGAqACAQ01
iX8.8AKGAqACAQye : 38S.8AKGAqACAQyq
uB2.8AKGAqACAQzL 33c0aeac-096f-48 :
pkI.8AKGAqACAQx6
ogU.8AKGAqACAQyM : iX8.8AKGAqACAQye
38S.8AKGAqACAQyq : PLq.8AKGAqACAQy8
E0Q.8AKGAqACAQxs : 33c0aeac-096f-48
wtp.8AKGAqACAQzu : mtYU9SKGAqACAgT_
pkI.8AKGAqACAQx6 : ogU.8AKGAqACAQyM

Test paths generated.

InitialNode MergeNode Ask-for-username-and-password Verify-username-and-password valid-login Notify-user Record-error MergeNode Ask-for-username-and-password Verify-username-and-password valid-login First-login Include-change-password JoinNode Welcome-user-to-the-system ActivityFinalNode

InitialNode MergeNode Ask-for-username-and-password Verify-username-and-password valid-login Notify-user Record-error MergeNode Ask-for-username-and-password Verify-username-and-password valid-login First-login JoinNode Welcome-user-to-the-system ActivityFinalNode
InitialNode MergeNode Ask-for-username-and-password Verify-username-and-password valid-login First-login Include-change-password JoinNode Welcome-user-to-the-system ActivityFinalNode
InitialNode MergeNode Ask-for-username-and-password Verify-username-and-password valid-login First-login JoinNode Welcome-user-to-the-system ActivityFinalNode

Conclusion

In this paper we proposed an approach for dealing with concurrent and non concurrent activities in Activity Diagrams. Proposed method is efficient as it deals with concurrency, loop testing and alternate paths. The approach can also be applied to nested fork and join pairs with loops and alternate paths. The main importance of the approach is that every node, every edge and every loop in the Activity Diagram gets executes at least once. This approach ensures in reducing cost of software development and improves the quality of the software.

References

[1] *Yasser Kotb, Katsuhiko Gondow and Takuya Katayama, , 2004, "Optimizing the Execution Time for Checking the Consistency of XML Documents", in Journal of Intelligent Information Systems (JIIS). Kluwer Academic Publishers. Vol. 22, No. 3, pp. 257-279*

[2] *David Flater , Philippe A. Martin, Michelle L. Crane, 2007, "Rendering UML Activity Diagrams as Human-Readable Text". National Institute of Standards and Technology.*

[3] *Narkngam, C., Limpiyakorn, Y., 2012 "Rendering UML Activity Diagrams as a Domain Specific Language - ADL." 24th International Conference on Software Engineering and Knowledge Engineering, San Francisco Bay, pp. 724–729.*

[4] *Chinnapat Kaewchinporn and Yachai Limpiyakorn, March, 2013 "Enhancement of Action Description Language for UML Activity Diagram Review" International Journal of Software Engineering and Its Applications Vol. 7, No. 2.*

[5] *Xu, D., Li, H., Lam, C.P., 2005 "Using Adaptive Agents to automatically Generate Test Scenarios from the UML Activity Diagrams", Proceedings of the 13th Asia-Pacific Software Engineering Conference.*

[6] *Debasish Kundu amd debases Samamta 2009, "A Novel Approach to generate Test Cases from UML Activity Diagrams", journal of Object technology, Vol 8, pp 65 – 85*

[7] *Chen mingsong, Qiu Xiaokang and Li Xuandog 2006, "Automatic Test Case Generation for UML Activity Diagrams", ACM, pp 2 – 8*

[8] *Chen mingsong, Qiu Xiaokang and Li Xuandog 2006, "Automatic Test Case Generation for UML Activity Diagrams", AST, National natural Science Foundation of China*

[9] *Puneet Patel and Nitin N Patil 2012, "Test Case formation using UML Activity Diagrams", World Journal of Science and Technology, vol 2, pp 57-62*

[10] *Leslie Lamport 1978, "Time, Clocks and the ordering of Events in a Distributed System". Communications of the ACM, 21(7), 558-565.*