Journal of Theoretical and Applied Information Technology
31st December 2014. Vol.70 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

AUTOMATED TEST CASE GENERATION USING UML USE CASE DIAGRAM AND ACTIVITY DIAGRAM

[1]ARUP ABHINNA ACHARYA, [2]PRATEEVA MAHALI, [3]DURGA PRASAD MOHAPATRA
[1,2]School of Computer Engineering, KIIT University, Bhubaneswar, India, 751024
[3]Department of Computer Science Engineering, National Institute of Technology,Rourkela, India, 769008
E-mail: [1]aacharyafcs@kiit.ac.in, [2]prateevamahali@gmail.com, [3]durga@nitrkl.ac.in

ABSTRACT

Testing plays a major role for improving the quality of a software product. Due to its iterative and incremental nature it needs special attention. Test case generation is one of the complex activities carried out during testing phase. Generating test cases in the early phases of development life cycle works like a catalyst for model based testing and at the same time efficiently manages time and resources. This paper describes a novel approach for test case generation from UML Activity Diagram (AD) and Use Case Diagram (UCD). At first UCD and AD are converted into Use Case Graph (UCG) and Activity Graph (AG) respectively. The AG and UCG are integrated to form a combined graph called Activity Use Case Graph (AUCG). The AUCG is further traversed to generate test cases. Test cases generated using the combined approach is capable of detecting more number of faults as compared to individual models while keeping intact the total coverage. The proposed approach also reveals faults like execution fault, operational fault and use case dependency fault.

Keywords: *Testing, AUCG, Test Case Generation, Dependency Fault, Operational Faults*

1. INTRODUCTION

According to IEEE, "Software Testing [1] is the process of analyzing software item to detect the difference between existing and required conditions (i.e bugs) and to evaluate the feature of the software item". For that reason testing of software is a time consuming activity which requires a proper planning and execution. One of the important criteria of testing is test case generation. A Test Case [7],[13] is defined as a set of conditions or variables which determine the level of correctness and quality of the product. The main issue is twofold i.e. generating test cases earlier in the development life cycle and attaining maximum coverage. Test cases are generated in two different scenarios i.e. code based testing and model based testing [11]. In code based testing the test cases are generated from the source code of the software and in model based testing the test cases are generated from models of the software. Today's software developers are using model based testing due to major issues in code based testing like non-availability of the source code of some components, delay in testing etc. Due to time consuming process code based testing is not the preferred testing strategy nowadays.

Model based testing on the other hand uses system models like Data Flow Diagram (DFD) [9], Entity Relationship Diagram (ERD) [2], and Unified Modelling Language (UML) [3] etc. Most of the industries are using UML 2.4 diagrams for modelling the system as it is a standard modeling language to visualize, specify, design and document requirements of the system. There are seventeen diagrams in UML 2.4 [4] but mainly nine diagrams are used for modelling purpose. In this paper we have used UCD and AD for test case generation. These diagrams can be constructed by using some tools like IBM Rational Software Architecture (RSA) [5], Star UML etc. This paper represents an approach for test case generation of use case diagram and activity diagram. Here the author explained the proposed methodology by using a case study of Hospital Management System (HMS). The test

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

case generated with this approach found to be detecting more number of faults due to integration of more than one UML diagrams.

The rest of the paper is organized as follows: Section 2 describes the related work. The proposed methodology is elaborated with a case study in Section 3. Section 4 discusses the comparison with related work and Section 5 concludes the paper along with the future works.

2. RELATED WORK

Dalai et al. [8] proposed an approach for test case generation for concurrent systems using combination of UML diagrams i.e sequence diagram and activity diagram. The authors have taken combination of diagrams for better coverage and fault detection capability. The author's approach can be summarized as:

(1) Construction of Sequence Diagram (SD) and Activity Diagram (AD).
(2) Maintaining a Sequence Table (ST) with different schema as Source Object (SO), Destination Object (DO), Message ID (MI) and Message Content (MC) by taking the information from SD.
(3) Then Sequence-Activity-Graph (SAG) is constructed by combining the features from SD and AD.
(4) SAG is traversed to generate test cases.

The generated test cases are capable of addressing the issues like test case explosion in concurrent system. The approach also depicts the dependency between the different activities in the SAG.

Sarma et al. [15] proposed an approach for test case generation from UML models i.e use case diagram and sequence diagram. As per author's approach, it consists of following steps.

(1) Convert the Use Case Diagram (UD) and Sequence Diagram (SD) to Use Case Diagram Graph (UDG) and Sequence Diagram Graph (SDG) respectively.
(2) The two graphs i.e UDG and SDG are integrated to form System Testing Graph (STG).
(3) To derive test cases required information is pre-stored into STG.
(4) Retrieve the information from the extended use case, class diagram and data dictionary are expressed in Object Constrained Language (OCL).
(5) Traverse the graph for generating the test cases. Test cases are generated by considering coverage criteria and fault model.

In this paper the author had used a case study of PIN Authentication of ATM System to describe the above steps. The main advantage of this proposal is that it describes test scenario generation process from sequence diagram and capable to detect operational fault.

Khandai et al. [10] presented an approach for test case generation from combination of UML models i.e Sequence Diagram (SD) and Activity Diagram (AD). In this proposal, first AD is converted into an intermediate format known as Activity Graph (AG). After that test sequences are generated from AG by applying Activity Path Coverage Criteria (APCC). Then SD is converted into Sequence Graph (SG) and the test sequences are generated by applying All Message Path Coverage Criterion (AMPCC). For having better coverage and high fault detection capability the author constructed a Activity Sequence Graph (ASG) which has the combine features of AG and SG. Finally the ASG is traversed to generate the test cases.

Swain et al. [12] proposed a method for generating test cases from combination of UML sequence diagram and activity diagram. The technique consists of following steps:

(1) First Message Flow Graph (MFG) is generated from activity diagram and sequence diagram.
(2) In the second phase test sequences from MFG corresponding to sequence diagram and activity diagram is generated.
(3) In third phase the MFG of the sequence diagram and the MFG of the activity diagram is traversed to generate the test cases.

In the first phase UML models are transformed into Message Flow Graph (MFG). The MFG can be represented as a quadruple (V, E, S, T) where each node $v \in V$ represents either a message or control predicate and an edge $e \in E$ represents a transition between the corresponding nodes. An edge $(m, n) \in E$ indicates the possible flow of control from the node m to the node n. Node S and T are unique nodes representing entry and exit of the diagram D. For obtaining the MFG a table called Object Method Association Table (OMAT) table is created for the sequence diagram which maintains information about

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

state change of an object when a message is passed between two objects. Another table is maintained for the activity diagram called Method Activity Table (MAT) which maintains the activities associated with the activity diagram. By referring to the tables the MFG for the activity and sequence diagram are created by taking each node and assigning an edge between them. The MFGs are next being traversed individually to generate the test cases.

Samanta et al. [14] had given a proposal for test case generation from UML Activity Diagram (AD). Test cases can be generated in three steps

(1) Augmenting the AD with necessary test information.

(2) Converting the AD into Activity Graph (AG).

(3) Test cases are generated from activity graph.

Here the author used Registration Cancellation as an example to describe the above three steps. Test cases are generated from activity diagram on the basis of path coverage criteria. They propose an algorithm called Generate Activity Path to generate all activity paths. In this algorithm Breadth-First-Search (BFS) and Depth-First-Search (DFS) are used for traversal of graph. The main advantages of this approach are

(1) It is capable to detect more faults like fault in loop and synchronization faults.

(2) It helps identify location of fault in the implementation so that the testing effort can be reduced.

(3) It improves design quality because the faults are detected in the early stage.

(4) It reduces software development time.

There are different approaches proposed by different author available for test case generation from UML diagrams. All these approaches are nearly similar to each other but the only difference is that different approach reveals different types of faults like scenario fault, operational fault, dependency fault etc.

3. PROPOSED APPROACH

In this paper the authors have proposed an approach for test case generation using combination of UML 2.4 diagrams i.e Use Case Diagram (UCD) and Activity Diagram (AD). Here the UCD and AD are converted into an intermediate formats called Use Case Graph (UCG) and Activity Graph (AG) respectively. Then AG and UCG are integrated to form combined graph called Activity Use Case Graph (AUCG). Finally test cases are generated by traversing the AUCG. The result shows that, fault detecting capability of this approach is more than the test cases which are generated from single UML diagram. To elaborate the proposed approach the author has considered the case study of a Hospital Management System (HMS). This approach also capable of detecting execution fault, operational fault and use case dependency fault. The fault occurs during the execution of system is called as execution fault and fault occurs during the operation time of the system is called as operational fault. Use Case Dependency Tree (UCDT) represented in Fig. 2 is used for detecting use case dependency fault.

3.1 Generating Use Case Graph from Use Case Diagram

Use Case Diagram (UCD) defines the client or user requirements of the system in terms of functions. The functions are represented by using actors and use cases. Actors represent the user or client who will use the system and performs the tasks or functions, which is represented as use cases. This diagram also shows the interrelationship between the actors and use cases. Basically it shows the overall behavior of a complete system. Therefore it is classified under behavioral diagram of UML. For example, UCD of Hospital Management System (HMS) (shown in Fig. 1) defines the complete management process of a hospital.

According to model based testing methodology, first the diagram is converted into an intermediate graph. Here also UCD is converted into an intermediate graph called Use Case Graph (UCG). UCG is generated by using Use Case Diagram Tree (UCDT). UCDT is nothing but the tree representation of UCD using linked list. Algorithms and node structures for construction of UCDT is followed from Acharya et al. [6]. By applying the proposed algorithm, UCDT of HMS is shown in Fig. 2. Due to limitation of space use case node structure of each actor is represented in Fig. 3, Fig. 4, Fig. 5, Fig. 6 and Fig. 7.

Journal of Theoretical and Applied Information Technology
31st December 2014. Vol.70 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

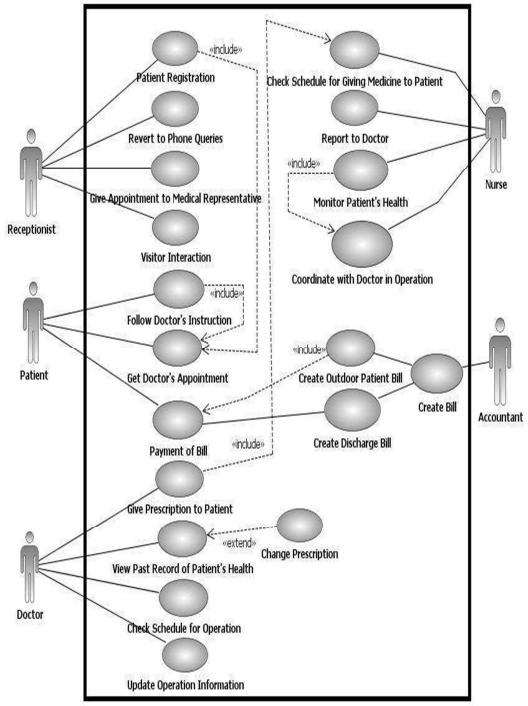

Figure 1: UCD of Hospital Management System (HMS)

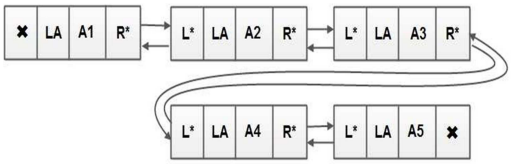

Figure 2: UCDT of Hospital Management System (HMS)

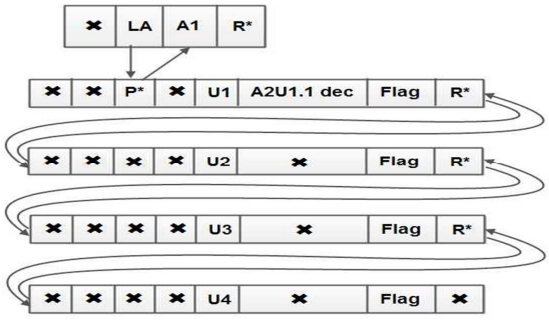

Figure 3: Use case node structure of A1

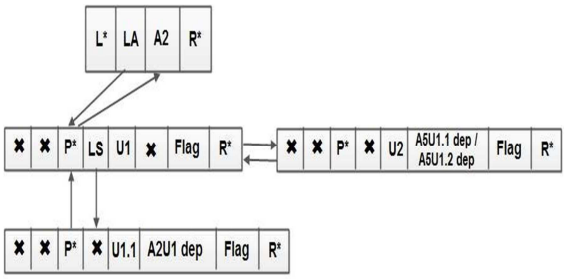

Figure 4: Use case node structure of A2

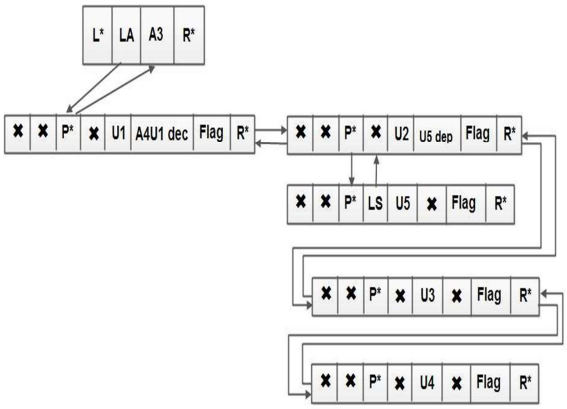

Figure 5: Use case node structure of A3

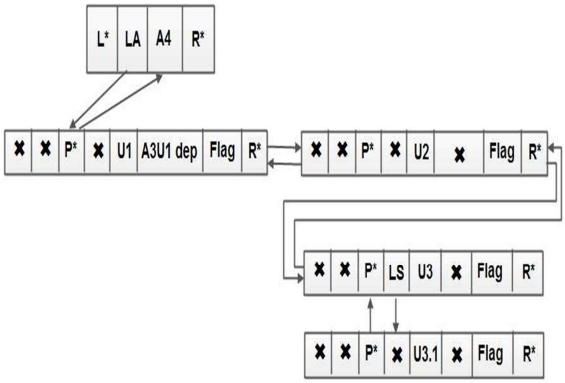

Figure 6: Use case node structure of A4

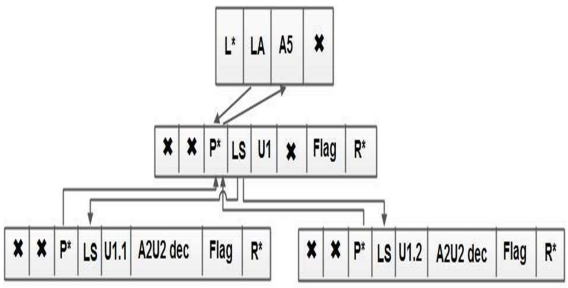

Figure 7: Use case node structure of A5

- L* represents the left pointer of the node pointing to the address of the preceding actor

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

or use case node in the sequence of actors or use cases in the linked list. Its value would be null provided L* belongs to the first actor or use case node in the sequence.

- LA stands for Left Actor field which a pointer pointing to the address of the node is representing the use case invoked by the actor.
- A represents the name of the actor.
- R* is the pointer field pointing to the address of the next actor node in the sequence. If the node is the last one in the sequence of the actor's linked list then R* field has to be assigned as NULL.
- RC is used for Redundancy Check. In case we have two or more actors invoking the same use case, we need to avoid traversing the use case node more than once i.e. the node need not be traversed and the use case need not be checked for each of the invoking actors once it has been traversed for any one of the actors out of the set of the actors invoking the use case.
- P* contains the address of the parent use case node which must be traversed and executed prior to the traversal and execution of the current child use case node.
- LS-Left Subordinate points to the child use case node to be executed only after the current parent node is traversed and executed.
- U No represents the use case number.
- RS-Right Subordinate is used for checking inter-set-use case dependency between two different actors. Naming convention for the RS field is denoted as:
  Actor_name | U No | dec/dep[deciding or dependent].
- Flag has value 0 or 1 depending if the use case has been executed at least once during test case generation.

From UCDT, we can know that A1 invokes four use cases, A2 invokes 3 use cases, A3 invokes 5 use cases, A4 invokes 4 use cases and A5 invokes 3 use cases. After generation of UCDT, UCG (shown in Fig. 8) is generated by considering use case number as node number in the graph. As per the algorithm, use case number of A1 in UCDT is defined as U1, U2, U3 & U4 and use cases number A2 is also defined as U1, U1.1 & U2. But in graph use cases number of A2 is defined as U5, U6 & U7 for better understanding of different use cases invoked by different actors. If we will represent like this manner then the developer will not face any confusion whether U1 is invoked by A1, A2, A3, A4 or A5. Each time it doesn't required defining the actor name for each use case. Hence Table 1 represents UCD and UCG mapping table by defining the use case names with respect to use case name and invoked actor name.

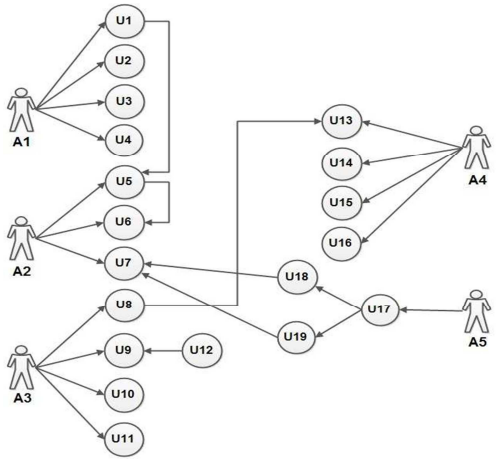

Figure 8: UCG of Hospital Management System (HMS)

Table 1: UCD and UCG Mapping Table

Use Case Number	Use Case Name	Actor Name
U1	Patient Registration	A1
U2	Revert to Phone Queries	A1
U3	Give Appointment to Medical Representative	A1
U4	Visitor Interaction	A1
U5	Follow Doctor's Instruction	A2
U6	Get Doctor's Appointment	A2
U7	Payment of Bill	A2
U8	Give Prescription to Patient	A3
U9	View Past Record of Patient's Health	A3
U10	Change Prescription	A3
U11	Change Prescription	A3
U12	Update Operation information	A3
U13	Check Schedule for giving Medicine to Patient	A4
U14	Report to Doctor	A4
U15	Monitor Patient's Health	A4
U16	Coordinate with Doctor in Operation	A4
U17	Create Bill	A5
U18	Create Outdoor Patient Bill	A5
U19	Create Discharge Bill	A5

3.2 Generating Test Cases from Activity Diagram

Activity Diagram (AD) [4] illustrates the dynamic nature of a system. Basically it is used for modelling business work flow or process and

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

internal operation. It shows a set of activities which are going to be executed for completion of a given task. The activity diagram of Hospital Management System (HMS) is shown in Fig. 9.

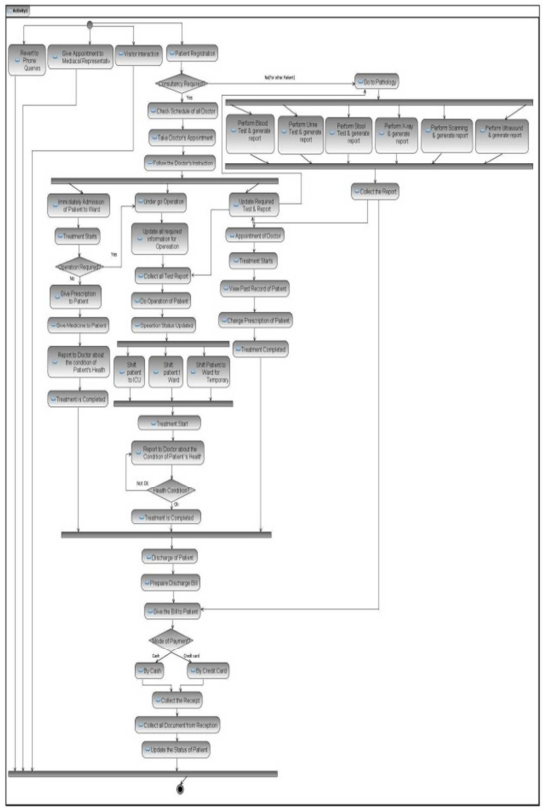

Figure 9: Activity Diagram of Hospital Management System (HMS)

In next step the AD is converted into an intermediate format called Activity Graph (AG). For generation of AG, the author has proposed a node traversing structure called as Activity Node Structure (ANS), shown in Fig. 10.

N*	LN*	CE	SP	TN	DN*	F*	J*	D

Figure 10: Activity Node Structure for generation of AG

- N* represents the address of the current node. To generate starting node i.e node 1, N* will store the address of starting activity. Address of starting activity can be found from AD.
- LN* represents the left node or predecessor node of current node. If predecessor node is not available the LN* will be NULL.
- CE is represented as Create Edge. If current node has any left node then it creates an edge between current node and left node. CE will be NULL for generation of starting node i.e node 1.
- SP is represented as sub-path. If the current node has any sub-path previously available then create an edge between sub-path and current node.
- TN represents type of node. After generating the node we have to check for node type. Node type may be a decision node or fork node or join node. It can be found from Table 2.
- DN* is represented as decision node. If the current node is decision node then address of node will be stored here otherwise it is NULL.
- F* is represented as fork node. If the current node is fork node then address of node will be stored here otherwise it is NULL.
- J* is represented as join node. If the current node is join node then address of node will be stored here otherwise it is NULL.
- D represents the dependency between the current node and left node. It may be of two type i.e intra-set dependency or inter set dependency. Dependency is same as use case dependency. If dependency exists then this field is 1, otherwise it is 0.

Table 2: Table for Node name & Node type

Name of Node	Type of Node
Initial node	S
End node	E
Decision node(where child nodes are the resultant of decision node)	DN
Control node (where parent node controls the child node i.e child node can be proceed after completion of parent node. Parent node is called as control node)	C
Merge node (which have single outgoing edge	M
Fork node (which have single incoming edge	F
Normal activity & activity associated with	A
Combined activity (Combination of more	CA
Join node (which have single outgoing edge	J

Test Case Generation - :

For generating the test cases Activity Diagram (AD) is converted into Activity Graph (AG) by using ANS. Here the author has proposed algorithms (defined in Algorithm 1 & Algorithm 2) for conversion of AD to AG and test case generation from AG. The general framework of working principle of Algorithm 1 & Algorithm 2 is shown in Fig. 12. First the AD is converted into

Journal of Theoretical and Applied Information Technology
31st December 2014. Vol.70 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

XMI code by using IBM RSA 7.1 tool and a parser file is generated according to ANS of AD. The parser file i.e Algorithm 2 represents the generation of linked list for each activity and that linked list generates the AG (shown in Fig. 11). By traversing that linked list, test cases are generated. In this way we obtain the following activity paths for each test case.

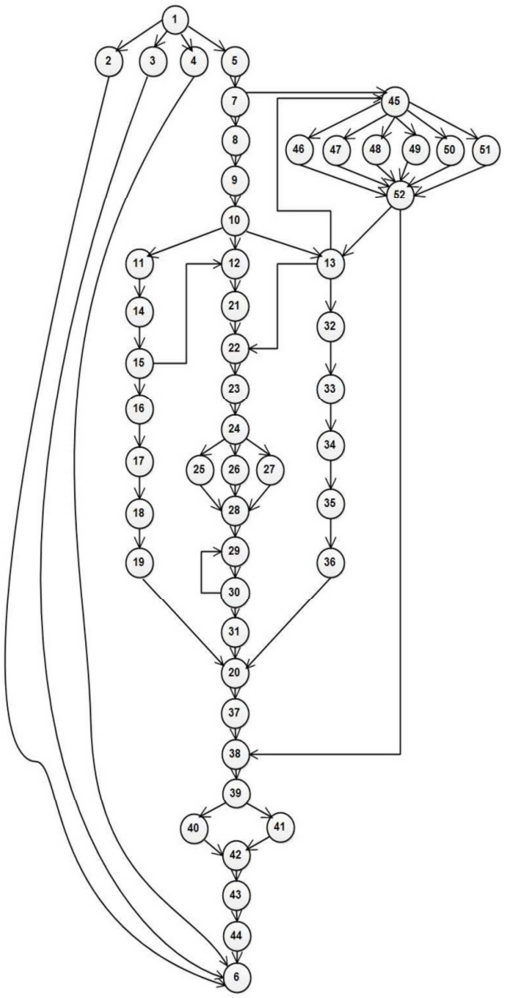

Figure 11: AG of Hospital Management System (HMS)

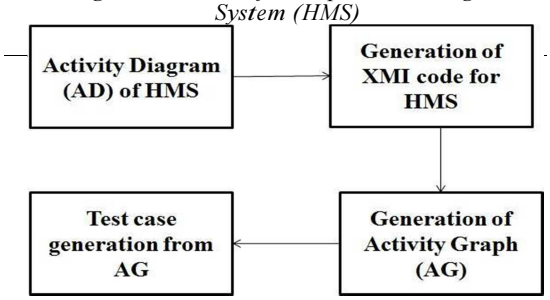

Figure 12: A model for test case generation for AD

Algorithm 1 Test case generation from Activity Diagram

Input: Activity Diagram (AD)
Output: Activity Graph (AG), Test cases
1: Start.
2: Convert the diagram into XMI code.
// IBM RSA 7.1 tool is used convert diagram into XMI code, shown in Fig. 13.
3: Generate_Test _Case();
// Algorithm 2 defines the generation & working principle of this function. This function takes the XMI code of the application as input.

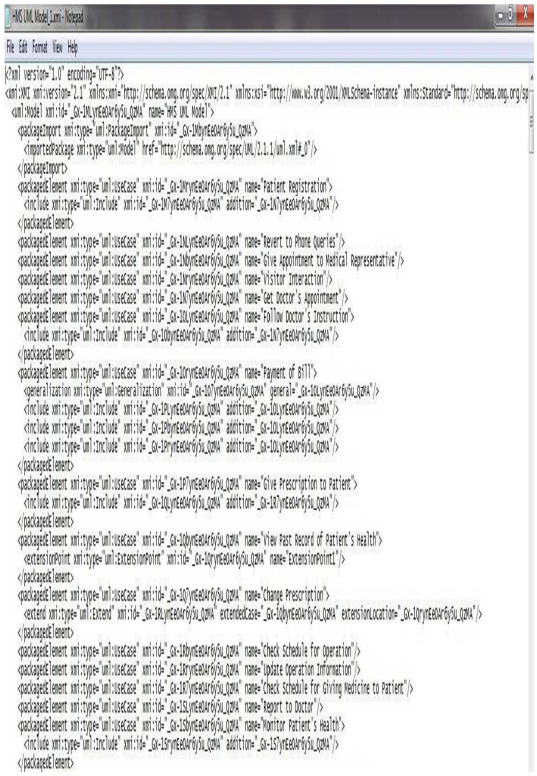

Figure 13: XMI code of HMS

Algorithm 2 Generation of Graph from XMI Code of HMS

Input: XMI Code
Output: Activity Graph (AG), Test cases
1: Generate_Test_Case()
{
2: Create an array N[ ]. // Node address of all activities will be stored in N[ ].
3: Create and initialize node for all activities.
// Nodes have 9 number of data fields as specified in Fig. 10.
4: Create a temporary pointer variable as ptr.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

5: **for** i = 1 to m **do** // m represents the total number of element in the array.
6: ptr=&nodei;
7: N[i]=ptr;
8: **end for**
9: Create another pointer variable for nodes i.e struct node_address *temp.
10: **if** xmitype==“Start” **then** // xmitype is an attribute of XMI code or file.
11: Set a pointer variable s as node type;
12: Address of “Start” is stored in s. // s represents the start of linked list.
13: set s=node1;
14: **end if**
15: **for** i = 1 to n **do** // n is the total number of nodes created for all activities.
16: // For node→ N* field
17: **if** nodei → N*==s **then** // if node i is the first node then value of other data field of first node is given below.
18: nodei → LN*=“NULL”;
19: nodei→CE=“F”;
20: nodei→SP=“F”;
21: nodei→TN=“A”;
22: nodei→DN*=“NULL”;
23: nodei→F*=“NULL”;
24: nodei→D=0;
25: **else**
26: nodei→ N*=nodei;
27: **end if**
28: temp=nodei→NEXT; // Address of next node is stored in temp variable.
29: // For node→LN* field
30: **for** j = temp to noden **do** // noden is the last node.
31: temp=node→ N*
32: node→ LN*=temp→PREV; // LN* field of current node is equal to address of previous node.
33: temp=temp→NEXT;
34: **end for**
35: // For node→CE field
36: **for** j = temp to noden **do**
37: **if** temp→LN*==temp→PREV **then**
38: node→CE=“T”;
39: **else**
40: node→CE=“F”;
41: **end if**
42: **end for**
43: // For node→TN field
44: **for** j = temp to noden **do**
45: **if** xmitype==“Forknode” **then**
46: node→TN=“T” and node→F*=nodej;
47: **else if** xmitype==“Decisionnode” **then**
48: node→TN=“T” and node!-->DN*=nodej;
49: **else if** xmitype==“Joinnode” **then**
50: node→TN=“T” and node→ J*=nodej;
51: **else**
52: node→TN=“A”; // A means normal activity.
53: **end if**
54: **end for**
55: // For node→SP field
56: **for** j = temp to noden **do**
57: **if** node→TN=“T” && (node→DN*!=“NULL” || node→ F*!=“NULL”) **then**
58: node→SP=“T”;
59: **else**
60: node→SP=“F”;
61: **end if**
62: **end for**
63: // For node→D field
64: **for** j = temp to noden **do**
65: **if** node→CE=“T” && node→TN=“T” **then**
66: node→D==1;
67: **else**
68: node→D==0;
69: **end if**
70: **end for**
71: **end for**
72: // Traverse the generated linked list to generate Activity Graph (AG)
73: **for** i = 1 to n **do**
74: **if** nodej→CE=“T” **then**
75: push(DN, nodei); // DN(Destination Node) is the stack name and nodei is the element.
76: **end if**
77: **end for**
78: // Creation of Source Node(SN)
79: **for** i = 1 to n do
80: key=nodei→ N*;
81: Create a stack named as Stemp
82: PUSH(Stemp,nodei→ LN*);
83: **for** j = 0 to top **do**
84: **if** nodei→CE=“T”&&nodei→N*== Stemp[top] **then**
85: PUSH(SN, Stemp[top]); // SN is the stack name and Stemp[top] is the element.
86: **end if**
87: **end for**
88: **end for**
89: // Creation of Sub Path Node(SPN)
90: **for** i = 1 to n do
91: **if** nodei→SP=“T”&& (nodei→F !=“NULL”|| nodei→DN!=“NULL” || nodei→j != “NULL”) then
92: PUSH(SPN, nodei(F or DN or J)) // SPN is the stack name and nodei(F or DN or J) means node name with tag F, DN or J.
93: **end if**
94: **end for**
95: // Creation of Complete Node(CN)
96: Push node1 into stack CN.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

97: **for** i = 1 to n **do**
98: **if** nodei→ LN*==node(i+1)→N* **then**
99: PUSH(CN, nodei); // CN is the stack name and nodei is the element.
100: **end if**
101: **end for**
102: //Traverse the graph to generate test cases.
103: **for** i = 1; i < SN.size() && i ≤ DN.size(); i + + **do**
104: **if**(SN.contains(CN.peek())==((DN.peek()== CN.peek()→Next)&(SPN.contains(DN.peek())))) **then**
105: Create an test sequence as Ttestcase=SN.get(i)+ "→"+ SPN.get(i); //Ttest case is the state name of test cases.
106: Tcount=Tcount+1; //Tcount represent number of test cases.
107: **end if**
108: **end for**
}

In this way 55 activity paths are obtained from the activity diagram. Due to lack of space we have not mentioned all the activity paths.

1. 1 → 2 → 6
2. 1→ 3 → 6
3. 1 → 4 → 6
4. 1 → 5 → 7 → 8 → 9 → 10 → 11 → 14 →15 → 16 → 17 → 18 →19 → 20 → 37 → 38 → 39 → 40 → 42 → 43 → 44 → 6
5. 1 → 5 → 7 → 8 → 9 → 10 → 11 → 14 →15 → 16 → 17 →18→19 → 20 → 37 → 38 → 39 → 41 → 42 → 43 → 44 → 6
6. 1 → 5 → 7 → 8 → 9 → 10 → 12→21 → 22 → 23→24 → 25 → 28 → 29 → 30 → 31→ 20 → 37→38 → 39 → 40 → 42 → 43 → 44 → 6
7. 1 → 5 → 7 → 8→9 → 10→12 → 21 → 22 →23 → 24→25 → 28 → 29→ 30 → 31 → 20 → 37 → 38→ 39→ 41 → 42 → 43 → 44 → 6

After generating the activity paths we now obtain the following test cases by obtaining the activity names from Table 3.

1. Start → Revert to Phone Queries → End
2. Start → Give Appointment to Medical Representative → End
3. Start → Visitor Interaction → End
4. Start → Patient Registration → Consultancy Required <?> → Check Schedule of all Doctor → Take Doctor's Appointment→ Follow the Doctor's Instruction → Immediately Admission of Patient to Ward → Treatment Starts→ Operation Required <?> → Give Prescription to Patient → Give Medicine to Patient → Report to Doctor about the Condition of Patient's Health → Treatment Completed → Discharge of Patient → Prepare Discharge Bill → Give the Bill to Patient → Mode of Payment <?> → By Cash → Collect the Receipt → Collect the Document from Reception → Update status of Patient → End
5. Start → Patient Registration → Consultancy Required <?> → Check Schedule of all Doctor → Take Doctor's Appointment → Follow the Doctor's Instruction →Immediately Admission of Patient to Ward → Treatment Starts → Operation Required <?> → Give Prescription to Patient → Give Medicine to Patient → Report to Doctor about the Condition of Patient's Health →Treatment Completed → Discharge of Patient → Prepare Discharge Bill→ Give the Bill to Patient → Mode of Payment <?> → By Credit Card → Collect the Receipt → Collect the Document from Reception → Update status of Patient → End
6. Start → Patient Registration→Consultancy Required <?> → Check Schedule of all Doctor → Take Doctor's Appointment → Follow the Doctor's Instruction → Under go Operation → Update all Required Information for Operation → Collect al Test Report → Do Operation of Patient → Operation Status Updated → Sift to ICU → Treatment Starts → Report to Doctor about the Condition of Patient's Health → Health Condition <?> → Treatment Completed → Discharge of Patient → Prepare Discharge Bill → Give the Bill to Patient → Mode of Payment <?> → By Cash → Collect the Receipt → Collect the Document from Reception → Update status of Patient → End
7. Start → Patient Registration → Consultancy Required <?> → Check Schedule of all Doctor → Take Doctor's Appointment → Follow the Doctor's Instruction → Under go Operation → Update all Required Information for Operation → Collect al Test Report → Do Operation of Patient → Operation Status Updated → Shift to ICU → Treatment Starts → Report to Doctor about the Condition of Patient's Health → Health Condition <?> → Treatment Completed → Discharge of Patient → Prepare Discharge Bill → Give the Bill to Patient → Mode of Payment <?> → By Credit Card → Collect the Receipt → Collect the Document from Reception → Update status of Patient → End

In this way the activity diagram generates 55 numbers of test cases. However we know that the activity diagram represents the information in abstract way. That means it represents only the sequence of activities but not the communication occurring between two different objects. So in the next step the Activity Graph (AG) with the Use Case Graph (UCG) are combined to generate a

Journal of Theoretical and Applied Information Technology
31st December 2014. Vol.70 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

graph called Activity Use Case Graph (AUCG) having the combined features of both the diagrams.

Table 3. Node Details of Activity Diagram (NDAD)

Node No.	Activity Associated	Node Type
1	Start	S
2	Revert to Phone Queries	A
3	Give Appointment to Medical Representative	A
4	Visitor Interaction	A
5	Patient Registration	A
6	End	E
7	Consultancy Required?	DN
8	Check Schedule of all Doctor	C
9	Take Doctor's Appointment	A
10	Follow the Doctor's Instruction	C
11	Immediately Admission of Patient to ward	A
12	Under go Operation	A
13	Update Required Test and Report	C
14	Treatment starts	A
15	Operation Required?	DN
16	Give Prescription to Patient	A
17	Give Medicine to Patient	A
18	Report to Doctor about the condition of Patient's Health	A
19	Treatment Completed	A
20	Discharge of Patient	J
21	Update all Required information for Operation	C
22	Collect All Test Report	A
23	Do Operation of Patient	A
24	Operation Status Updated	F
25	Shift to ICU	A
26	Shift to Ward	A
27	Shift to Ward for Temporary	A
28	Treatment Start	J
29	Report to Doctor about the condition of Patient's Health	A
30	Health Condition?	DN
31	Treatment Completed	A
32	Appointment of Doctor	A
33	Treatment starts	A
34	View Past Record of Patient	A
35	Change Prescription of Patient	A
36	Treatment Completed	A
37	Prepare Discharge Bill	A
38	Give the Bill to Patient	A
39	Mode of Payment?	DN
40	By Cash	A
41	By Credit Card	A
42	Collect the Receipt	A
43	Collect the document from Reception	A
44	Update Status of Patient	A
45	Go to Pathology	F
46	Preform Blood Test & generate report	A
47	Perform Urine Test & generate report	A
48	Perform Stool Test & generate report	A
49	Perform X-ray & generate report	A
50	Perform Scanning & generate report	A
51	Perform Ultrasound & generate report	A
52	Collect the Report	J

3.3 Generating Test Cases from Activity Use Case Graph (AUCG)

In order to generate test cases having better coverage and high fault detection capability, in this section we proposed an approach which combines the Activity Graph (AG) with Use Case Graph (UCG) and generates a graph called Activity Use Case Graph (AUCG). AUCG is generated by combining the features of both the diagrams. For generating the AUCG we propose an algorithm called Generate Activity Use Case Graph (AUCG) as explained in Algorithm 3 which will combine the AG and UCG and will generate a graph called Activity Use Case Graph (AUCG).

Algorithm 3 Generate Activity Use Case Graph (AUCG)
Input: Activity Graph (AG) and Use Case Graph (UCG)
Output: Activity Use Case Graph (AUCG)

1: Start.
2: Activity names and use case names for respective node can be extracted from Table 3 and Table 1. // Input all activity names into a stack called AN (Activity Names).
4: Create a stack AN; // AN is the stack name.
5: for i = 1 to m do // m represents the total number of activity names present in activity diagram.
6: push all the activity names into AN.
7: end for
8: // Input all use case names into a stack called UCN (Use Case Names).
9: Create a stack UCN; // UCN is the stack name.
10: for j = 1 to n do // n represents the total number of use case names present in use case diagram.
11: push all the use case names and actor names into UCN and use case names are defined like actor name, use case name.
12: end for
13: // Mapping of activity names with use case names.
14: Create a stack AUCN; // AUCN is a stack for Activity Use Case Name. It stores the activity use case names for combination of activities and use cases.
15: for k = 1 to k _ size(AN)&& k _ size(UCN) do

Journal of Theoretical and Applied Information Technology
31st December 2014. Vol.70 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

16: key = AN[top];
17: if key == UCN[top] then
18: push(AUCN,pop(UCN)); // First UCN[top] is deleted and then push it to AUCN stack. It means UCN[top] is stored in AUCN.
19: end if
20: end for
21: for i = 1 to size(AUCN) do
22: // size(AUCN) represents the number of element present in AUCN.
23: nodei = AUCN[peek];
24: peek = peek+1;
25: end for
26: Display node1;
27: for i = 2 to AUCN[top] do
28: Connect nodei with nodei-1;
29: if nodei-1 is Normal Activity then
30: Connect by single"↓" from nodei to nodei-1;
31: else if nodei-1 is Decision Node then
32: Connect by two"↓" from nodei;
33: else if nodei-1 is Fork Node then
34: Connect by multiple"↓" from nodei;
35: else
36: Do simple connection by "↓";
37: end if
38: end for
39: Connect the corresponding nodes to decision node and fork node. // The nodes can be obtain from Table 3 and AD.
40: Generate the graph called as Activity Use Case Graph (AUCG).

The pseudocode for generating AUCG is defined in Algorithm 3. As per the proposed algorithm, AG and UCG are used as input. First activities and use cases with respective activity names and use case names are stored in stacks named as AN and UCN. Then mapping of activities and use cases are done. If they have relationship with each other then combination of activity name and use case name are stored in another stack called AUCN (Activity Use Case Name). The combinations are defined like Activity Name,[use case name]. In this way all the activities with related use cases are stored in AUCN. Then nodes are created for each activity use case names and they are connected to each other according to proposed algorithm. Hence AUCG is generated by combining the AG and UCG. Applying this technique, the AUCG is generated for Hospital Management System (HMS) which is shown in Fig. 14.

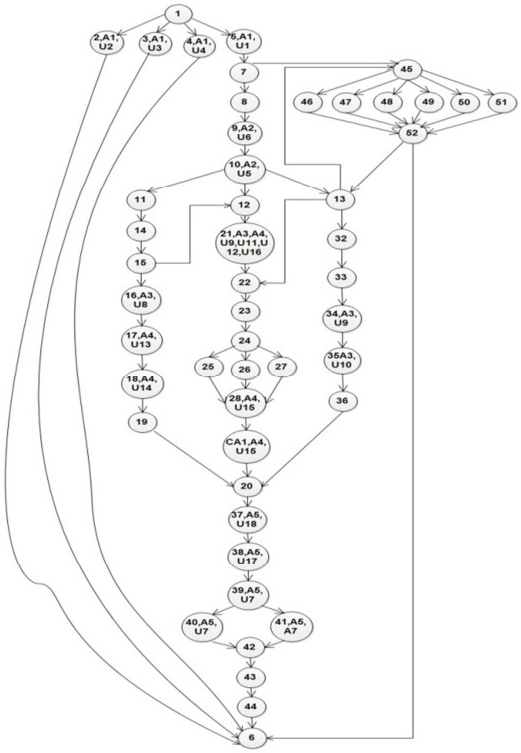

Figure 14: AUCG of Hospital Management System (HMS)

Test Case generation -:

Now the AUCG is followed for test case generation process. Test cases are generated by using ANS and Algorithm 2 defined in Section 3.2. ~~As per this algorithm, nodes are created using ANS~~ for activity use case names and that nodes are traversed for generating the test cases. For generating the test sequences from AUCG, activity path coverage criteria are used. By applying this technique we obtain the following activity paths.

1. 1 → 2,[A1,U2] → 6
2. 1→3,[A2,U3] → 6
3. 1 → 4,[A1,U4] → 6
4. 1→5,[A1,U1] → 7 →8 → 9,[A2,U6] → 10,[A2,U5] → 11 → 14 → 15 → 16,[A3,U8] → 17,[A4,U13] → 18,[A4,U14] → 19 –! 20 → 37,[A5,U17,U19] → 38,[A5,U18,U19] → 39,[A2,U7] → 40,[A2,U7] → 42 → 43 → 44 → 6
5. 1→ 5,[A1,U1] → 7 →8 → 9,[A2,U6] → 10,[A2,U5] → 11 → 14 → 15 → 16,[A3,U8] → 17,[A4,U13] → 18,[A4,U14] → 19 → 20 → 37,[A5,U17,U19] → 38,[A5,U18,U19] → 39,[A2,U7] → 41,[A2,U7] –→ 42 → 43 → 44 → 6

In this way 60 test cases are obtained from the activity use case graph.

After obtaining the activity sequences extract the use case names from Table 1 and activity names

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

from Table 3 to obtain the test cases. The use case and actor names are written in the square bracket to distinguish the use cases and actors from activity sequences. The test cases obtained from AUCG are as follows.

1. Start → Revert to Phone Queries,[Receptionist, Revert to Phone Queries] → End
2. Start → Give Appointment to Medical Representative,[Receptionist, Give Appointment to Medical Representative] → End
3. Start → Visitor Interaction,[Receptionist, Visitor Interaction] →End
4. Start → Patient Registration,[Receptionist, Patient Registration] → Consultancy Required <? > → Check Schedule of all Doctor Take Doctor's Appointment,[Patient, Get Doctor's Appointment] → Follow Doctor's Instruction,[Patient, Follow Doctor's Instruction] → Immediately Admission of Patient to Ward → Treatment Starts → Operation Required <?> → Give Prescription to patient,[Doctor, Give Prescription to patient] → Give Medicine to Patient, [Nurse, Give Medicine to Patient] → Report to Doctor about Condition of Patient's Health,[Nurse, Report to Doctor] → Treatment completed → Discharge of Patient → Prepare Discharge Bill,[Accountant, Create Bill, Create Discharge Bill] → Give the Bill to Patient,[Accountant, Create Outdoor Patient Bill, Create Discharge Bill] → Mode of Payment <?>,[Patient, Payment of Bill] → By Cash,[Patient, Payment of Bill] → Collect the Receipt → Collect the Document from Reception → Update the status of Patient → End
5. Start → Patient Registration,[Receptionist, Patient Registration] → Consultancy Required <? > → Check Schedule of all Doctor Take Doctor's Appointment,[Patient, Get Doctor's Appointment] → Follow Doctor's Instruction,[Patient, Follow Doctor's Instruction] → Immediately Admission of Patient to Ward → Treatment Starts → Operation Required <?> → Give Prescription to patient,[Doctor, Give Prescription to patient] → Give Medicine to Patient, [Nurse, Give Medicine to Patient] → Report to Doctor about Condition of Patient's Health,[Nurse, Report to Doctor] → Treatment completed → Discharge of Patient → Prepare Discharge Bill,[Accountant, Create Bill, Create Discharge Bill] → Give the Bill to Patient,[Accountant, Create Outdoor Patient Bill, Create Discharge Bill] → Mode of Payment <?>,[Patient, Payment of Bill] → By Credit Card,[Patient, Payment of Bill] → Collect the Receipt → Collect the Document from Reception → Update the status of Patient → End

In this way 60 activity use case paths are obtained from activity use case graph. The test cases generated are capable of detecting operational faults, use case dependency faults, execution faults. The coverage capability of combinational diagrams is more than the coverage capability of individual diagram.

4. COMPARISION WITH RELATED WORK

In this paper, we have proposed a technique for generating the test cases for object-oriented system using use case diagram and activity diagram. There are maximum number of papers which describes the test case generation process using single UML diagrams and combination of two UML diagrams. In those papers the authors have described about the different techniques to generate test cases from a model i.e any UML diagram. Table 5 describes the comparison of our approach with related work done by different researcher.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Table 5: Comparison with Related Work

Paper Name	UML Diagram	Synchronization Fault	Control Dependency	Operational Fault	Executional Fault	Data Dependency	Fault in loop
A novel approach to generate test cases from uml activity diagram [14]	Activity diagram	Yes	No	No	No	No	Yes
Automatic test case generation from uml models [15]	Use case & sequence	No	No	Yes	No	No	No
Test case generation for concurrent object-oriented system using combinational uml models [8]	Sequence & activity diagram	No	Yes	No	No	Yes	No
Test case generation for use case dependency fault detection [6]	Use case diagram	No	Yes	No	No	Yes	No
Our proposal	Use case & activity diagram	No	Yes	Yes	Yes	Yes	No

5. CONCLUSION AND FUTURE WORK

In this paper a detailed approach for test case generation for an object-oriented system by using use case diagram and activity diagram of a system is discussed. The diagrams are converted into an intermediate graph and then the test cases are generated from the graphs. This paper also presents test case generation process by integrating both use case graph and activity graph called as Activity Use Case Graph (AUCG). The proposed approach also included use case dependency fault detection and redundancy check. This approach can detect operational or executional fault, message dependency and control dependency between activities at any instance of time. By reducing the redundancy of nodes in each test case we can save the cost and effort required for software testing.

In a composite graph it is very difficult to detect errors due to redundant nodes, in each test case execution. So we can use dynamic slicing criteria (either forward slicing or backward slicing) for the detection of errors and the affected nodes due to the same error in each test case. This approach can further be used for optimization and prioritization of test cases in regression testing.

REFRENCES:

[1]"IEEE Glossary." www.ieeexplore.ieee.org/IEEE Glossary.

[2] "The Importance of Business Understanding in Requirements Structuring." http://www.umsl.edu/_cjtz4/umsl/erdiagrams.html.

[3] "Software Design Tutorials." www.smartdraw.com/resources/tutorials/Software Design Tutorials.

[4] "UML 2.4 Diagrams Overview." http://www.uml-diagrams.org/uml-24 diagrams.html.

[5] "IBM Rational Functional Tester." http://www.ibm.com/developerworks/downloads/r/rft/.

[6] G. Budha, N. Panda, and A. A. Acharya, "Test case generation for use case dependency fault detection", *3rd International Conference on Electronics Computer Technology (ICECT)*, Vol. 6, No. 6, September 2011, pp. 178–182.

[7] N. Chauhan, "Software Testing Principles & Practices", *Oxford University Press, New Delhi*, 2010.

[8] S. Dalai, A. A. Acharya, and D. P. Mohapatra, "Test case generation for concurrent object-oriented system using combinational uml models", *International Journal of Advance Computer Science and Applications*, Vol. 3, No. 5, 2011, pp. 97–102.

[9] R. Ibrahim, and S. Y. Yen, "Formalization of the data flow diagram for consistency check", *International Journal of Software Engineering*

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

& Applications (IJSEA), Vol. 1, No. 4, October 2010, pp. 95–111.

[10] M. Khandai, A. A. Acharya, and D. P. Mohapatra, "Test case generation for concurrent system using combinational diagram", *International Journal of Computer Science and Information Technologies (IJCSIT)*, Vol. 2, No. 3, 2011, pp. 1172–1181.

[11] B. Korel, and G. Koutsogiannakis, "Experimental comparison of code-based and model-based test prioritization", *IEEE International Conference on Software Testing Verification and Validation Workshops*, 2009, pp. 77–84.

[12] S. kumar Swain, and D. P. Mohapatra, "Test case generation from behavioural uml models", *International Journal of Computer Applications*, Vol. 6, No. 8, September 2010, pp. 5–11.

[13] R. Mall, "Fundamental of Software Engineering", *PHI Learning Private Limited, New Delhi*, 2009.

[14] D. Samanta, and D. Kundu, "A novel approach to generate test cases from uml activity diagram", *Journal of Object Technology*, Vol. 8, No. 3, May-June 2009, pp. 65–83.

[15] M. Sarma, and R. Mall, "Automatic test case generation from uml models", *10th International Conference on Information Technology*, 2007, pp. 196–201.