ISSN:0975-9646
Monalisha Khandai et al, / (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 2 (3) , 2011, 1172-1181

Test Case Generation for Concurrent System using UML Combinational Diagram

Monalisha Khandai[#1], Arup Abhinna Acharya[#2], Durga Prasad Mohapatra[*3]

[#] *School of Computer Engineering, KIIT University*
Bhubaneswar, India

[*]*Department of Computer Science & Engineering*
National Institute of Technology
Rourkela, India

***Abstract*— The unreasonable interference of concurrent threads makes the testing activity for concurrent systems a difficult task. Test case explosion is the major problem in concurrency testing and make an interruption in systematic testing of concurrent systems. In this paper we propose an approach of generating test cases from combinational UML models. In our approach Activity Diagram (AD) and Sequence Diagram (SD) are used to model a system. The AD has converted into a graph called Activity Graph (AG) and SD into a graph called Sequence Graph (SG). Finally AG and SG are combined to form a graph called Activity Sequence Graph (ASG). The ASG is traversed using a traversing algorithm to generate the test cases. After comparing the test cases generated from ASG with the test cases generated from AG and SG, it is found that the test cases generated from ASG gives a better coverage when compared with the test cases from single modelling graph. The test cases are generated by controlling the test case explosion and are useful for controlling synchronization fault, loop fault, as well as scenario faults and interaction faults.**

***Keywords*— *Testing*, Concurrency, Activity Sequence Graph (ASG), synchronization fault, loop fault, scenario faults.**

I. INTRODUCTION

According to IEEE testing is "the process of exercising or evaluating a system or system components by manual or automated means to verify that it satisfies specified requirements". In other words testing is the process of identifying the difference between the expected and actual results. If the software does not perform as required and expected then a software failure is said to be occurred. Testing effort consists of three things: i) test case generation or selection ii) test execution iii) test evaluation. Among the three, test cases generation problem is receiving highest attention. A test case is normally a triplet [I, S, O], where "I" is data input to the system "S" is the state of the system to which the data will input, and "O" is the expected output from the system. Testing is an important phase of software development which aims at producing highly reliable system and maintaining quality. The reliability and quality of the end product depend to a large extend on testing. Therefore more than 50% of software development effort is being spent on testing. A test case is said to be having good code coverage if it uncovers/detects maximum number of faults with minimum number of test cases. Combination of all the test case with which a given software product is to be tested is called test suite.

Depending on the testing method employed, Software testing can be implemented at any time in the development process. However, most of the test effort occurs after the requirements have been defined and the coding process has been completed. But Code based testing have the following disadvantages: i) certain aspects of behaviour of a system are difficult to extract from code but are easily obtained from design models, ii) test case generation process is delayed till the coding is over. An alternative approach is to generate test cases from the models representing the software, which has the added advantage of applying testing techniques through out the development process, on the basis of requirement, specification and design models.

Recent approach that has been taken by researchers is to use analysis design models like Unified Modelling Language (UML) for test case generation. UML models are very popular because when software engineering industry was in desperate need for standardization and utilization of design methodologies, UML came up as a solution. Other advantage of UML models is that it provide different diagram for representing different view of system models and it is easy to automate. Automated test case generation is advantageous when we have to generate the test cases for large system which is inherently complex. In such a case generating all the large number of test cases and carrying out the test cases is very time consuming and labour intensive. The automated test generating tool can be helpful in such a cases by saving the time and cost. There are different tool such as QTP, Rational Rose available for generating the test cases automatically. But the recent approaches can generate the test cases semiautomatically.

Though many work has been done for sequential testing a few work has been done for concurrency testing. Testing concurrent systems is a very crucial task since such a system

can exhibit different responses depending on the concurrency conditions. Due to concurrency there may be test explosion. Synchronization and deadlock create problems when concurrently running objects want to interact with each other. The UML Sequence Diagram, Activity Diagram and State Chart Diagram can be used for testing concurrency. However State Chart Diagram is useful for unit testing and results a large number of test cases, due to consideration of each and every state that an object undergoes during its operation, where as the Sequence Diagram can be useful for integration testing and results a less number of test cases. The Sequence Diagram is also useful in detecting scenario as well as interaction faults. The Activity Diagram is useful for representing complex sequence of parallel and conditional activities. The Activity Diagram is also useful in detecting faults in loop and synchronization faults present in concurrent systems, where the different concurrent processes need to be synchronized properly.

In this paper we have proposed a method to generate test cases from combinational UML models such as Sequence Diagram and Activity Diagram. In this approach we have converted the Activity Diagram (AD) into Activity Graph (AG) and the Sequence Diagram (SD) into Sequence Graph (SG) with the help of appropriate algorithms. Finally ASG is being constructed by combining AG and SG, which is being traversed to generate the test cases. The resultant shows that the test case generated from the ASG is having better code coverage with more number of faults detection capabilities.

The rest of the paper is organized as follows: Section II represents the related works, section III represents basic concepts, section IV proposed approach finally section V represents the conclusion and future work.

II. RELATED WORK

Kundu et al. [1] proposed an approach of generating Test Cases from Activity Diagram (AD). In their approach they have transformed the AD into an intermediate format called ***Activity Graph* (AG)**. The AG is traversed to generate Test cases. Sun [2] proposed an approach for generating test cases from AD, in their approach converted the AD into an intermediate format called ***Extended AND_OR Tree* (ET)** by applying a transformation rule on the fork, join, merge, branch activities. The **ET** is traversed to generate the test cases. Kim et al. [3] in their approach convert the AD into an intermediate format called ***I/O explicit Activity Diagram* (IOAD)**, by suppressing non-external input and output. **IOAD** is then traversed to generate the test cases. Sarma et al. [4] proposed a method for generating Test cases from UML Sequence Diagram (SD). The technique converts the SD into an intermediate format called ***Sequence Diagram Graph* (SDG)**, which is being traversed to generate the Test Cases. Since Sequence Diagram only is not enough to generate the test cases so OCL (Object Constraint Language) is used to store the pre and post conditions of each node. Samuel et al. [5] in their approach proposed a method to generate the Test Sequences from Sequence Diagram available in UML 2.0. The Sequence Diagram (SD) is converted into an intermediate format ***Sequence Dependency Graph* (SDG)** by combining the message sequences that are related to each other and representing it as a node in SDG. The **SDG** is traversed to generate the test cases.

Sarma et al. [6] in their approach proposed a method for generating test cases from combination of UML Sequence diagram and Usecase Diagram (UD). The technique converts the SD into SDG and the UD into UDG. Then the UDG and the SDG are combined to form a graph called System Testing Graph (STG). The STG is being traversed to generate the test cases. OCL is used to store the pre and post condition of each node. Sokenou [7] proposed an approach for generating Test Cases from UML Sequence and Statechart Diagram. In their approach the main information is extracted from Sequence Diagram, and the Statechart Diagram is used as a complementary for initializing sequences for the participating objects. Riebisch et al. [8] proposed a method for generating test cases from combination of Use case and Statechart diagram. Swain et al. [9] proposed a method for generating test cases from combination of Activity and Statechart Diagram. Swain et al. [10] proposed a method for generating test cases from combination of Activity and Sequence Diagram. In their approach the have converted the Activity Diagram and the Sequence Diagram into individual MFG (Message Flow Graph). Then a traversing technique has been used to generate the test cases from the MFGs.

III. BASIC CONCEPT

CONCURRENT SYSTEM: In case of Concurrent System *several threads runs concurrently*. The execution of concurrent threads begins from *fork* node (in case of Activity Diagram) or from par fragment (in case of Sequence Diagram). And the execution of concurrent threads /activity finishes on *join* node (in case of Activity Diagram) or exit from par fragment (in case of Sequence Diagram).

***SEQUENCE DIAGRAM*:** The UML Sequence Diagram consists of two basic elements i.e. the "*objects*" that participate in the interaction and the "*sequence of messages*" that are passed between the objects. Out of the thirteen models used in UML, only sequence diagrams show the messages exchanged between the objects.

ACTIVITY DIAHRAM: The UML Activity Diagram consists of two basic elements i.e. "*activity"* and "*transition"*. An *activity* can be represented as node is a state of doing something, which can be further classified into different type of activity such as Start Activity, End Activity, Branch Activity, Merge Activity, Fork Activity, Join Activity, so on. A *transition* can be represented as an edge connecting two different activities which can be control flow, message flow or object flow.

Activity Diagram can be used to describe the complex sequence of activities, with support for both conditional as well as parallel behaviour. Conditional behaviour can be represented by *branch* and a *merge*, and parallel behaviour can be represented by *fork* and *join*. A branch has a single

incoming transition and several guarded outgoing transitions. Parallel processes during the fork and join can choose the arbitrary order to execute [3].

Actually an UML Activity Diagram (AD) can be represented as a tuple < A, T, C >, where A is the collection of different types of activities in the AD; T: A x C → A, The C is the constraint condition which is an optional. If any constraint condition is there then that must be satisfied when the transition (T) happens [2].

FAULT IN LOOP: This type of fault may occur in the loop entry or terminating loop condition or increment operation or decrement operation [1]. For example in an ATM system, after inserting a card, the user enter the password and the password is incorrect then for the First time TryAgain = Yes, loop is executed for its 2nd iteration and say, at the end of 2nd iteration, after giving TryAgain = No, loop is not exiting rather it executes for its 3rd iteration.

SYNCHRONIZATION FAULT: This type of fault occurs when an activity start its execution before the groups of activities preceding activities have finishes their execution [1]. For an example in a library information system when ever the user issues a book then the library database and the user account must be updated simultaneously. Only after that the system will check wether the user want to issue any book further. Suppose in a case only the library database is updated and the user account is not updated then when ever the user will attempt to further issue any book then if the book is available then the user can issue that book even though the issue limit has crossed. This happens because the update library database and the update the user account these two parallel activities are not synchronized properly.

SCENARIO FAULT: This type of faults occur when a sequence of messages don't follow the desire path. A sequence diagram depicts several operation scenarios. Each scenario corresponds to a different sequence of message path in the sequence diagram. For a given operation scenario, sequence of message may not follow the desired path due to incorrect condition evaluation, abnormal termination etc [4].

INTERACTION FAULTS: These types of faults generally occur when messages are exchanged between objects. Several faults come under this type of faults such as incorrect response to a message, correct message passed to a wrong object etc [4].

ALL MESSAGE PATH COVERAGE CRITERIA: This criterion is used for generating test sequences from Sequence Diagram. Given a test set T and a Sequence Diagram D, then T must cause each sequence of message path to be exercised at least once [6].

ACTIVITY PATH COVERAGE CRITERION: This criterion is used for generating test sequences from Activity Diagram. The ACTIVITY PATH COVERAGE CRITERION maintains the precedence relationship (an activity can't begins its execution before the preceding activity or groups of activities finishes their execution) between the concurrent and non-concurrent activities and considers the loop at most two (all the activity will be consider exactly once except those activities which are in loop, the activity in the loop will be at most two times) [1]

IV. PROPOSED APPROACH

In this paper we have proposed a method of generating test cases for Concurrent systems using combinational UML models (i.e. Activity Diagram and Sequence Diagram). In our approach we have converted the Activity Diagram (AD) and the Sequence Diagram (SD) into intermediate formats called Activity Graph (AG) and Sequence Graph (SG) respectively. Finally we have combined the AG and the SG to form a combined graph called Activity Sequence Graph (ASG). While the ASG is traversed the resultant shows that the generated test cases are being capable of detecting more faults than test cases constructed from individual diagrams.

A) Generating Test Sequences from Sequence Diagram:

The Sequence Diagram, Activity Diagram, Statechart Diagram are very useful for representing the concurrent activities. The disadvantage of Statechart Diagram is that it results in state explosion since all the state for every objects through which the objects undergoes during the lifecycle are considered. Where as the Activity Diagram (AD) represents the sequence of activity flows and the Sequence Diagram (SD) represent the sequence of messages passed between the objects with out leading to test case explosion.

In our approach we have taken the SD available in UML 2.0. The UML Sequence Diagram, also known as interaction diagram, represents the scenarios as possible sequence of message exchange between the objects to specify the task. The Sequence Diagram available in UML 2.0 enables complex scenario to be specified in a single Sequence Diagram. UML 2.0 combines multiple scenarios by means of ***Combined Fragment (CF).*** A CF may contain another CF, this features allows complex scenarios to be specified in a single SD. A CF encloses one or more processing sequences in a frame which are executed under specific fragment operator [5].

There are 12 different type of fragment operator, but we will be discussing only those operators which we have used in our proposed works.

Combined Fragment (Par): Typically, the interaction fragment *par* denotes the parallel merge among the messages in the operands of a par fragment.

Combined Fragment Alt: The fragment *alt*, denote a choice of behaviours, which to be controlled by an interaction constraint.

Fig.1. represents a Sequence Diagram (SD) for a library management system. Here after the card value is passed to the session manager, the session manager checks the card weather it is valid one or not. After that alternatively two activities are carried out i) if the card is invalid one then eject message will be displayed ii) it will check for the Password if the card is a valid one. These two things being two alternate things are represented in an ALT fragment. Similarly all the message sequences are represented by suitable operational fragments.

We now convert the SD into an intermediate graph called Sequence Graph (SG) using the Algorithm 1.

Algorithm 1: *Generate Sequence Graph.*
Input: *Sequence Diagram (SD).*
Output: *Sequence Graph (SG).*

1) Start.
2) For each M_C
3) For $M_C! = M_E$ // M_C, M_E are the current and end message , //continue till current message is not the end message
4) $M_C = M_i$ // begin with the initial message M_i
5) For $M_C = M_{i+1}$ // there is message sequence M_i, M_{i+1}
6) Create two nodes M_i, M_{i+1} in the SG and assign a edge between them i.e. $M_i \rightarrow M_{i+1}$
7) End

In the algorithm 1 for constructing the SG from SD each message that has passed in the Sequence Diagram are considered. Here the message Id (M_c) of the SD is transformed into the node in the SG and the edge represents the connection between the messages. When ever a message is passed between two objects in the SD then two nodes are created in the SG, the nodes are named according to the message ID of the SD. And an edge is assigned between them to show the dependency among the messages. The SG of the library management system is represented in Fig.2.

We maintain a table called Message Details of the Sequence Diagram (MDSD) to store the Message ID along with the message associated with the Message ID which can be used further to generate test sequence from the message sequence path. The MDSD of the library management system is shown in TABLE I.

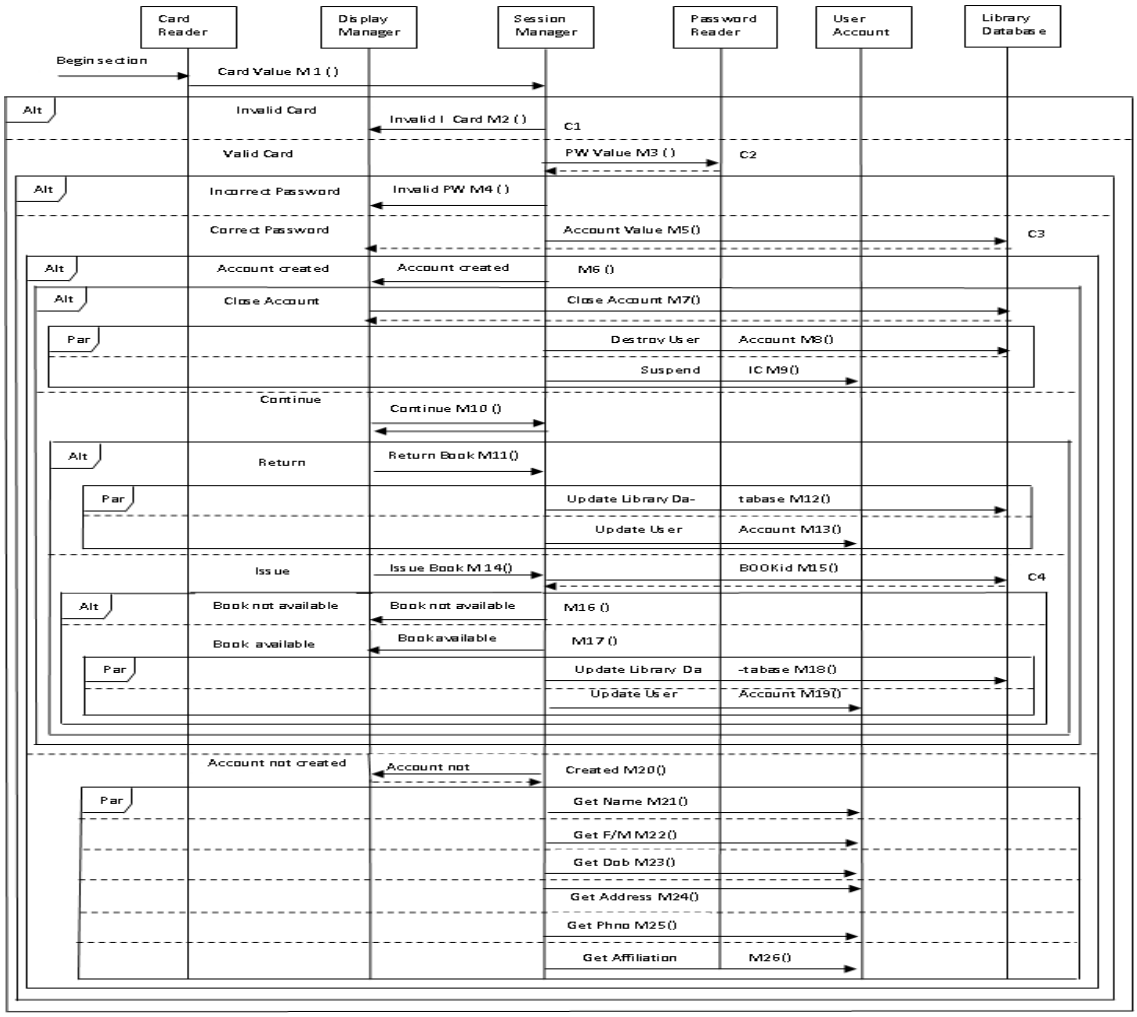

Fig. 1. Sequence Diagram of Library Management System

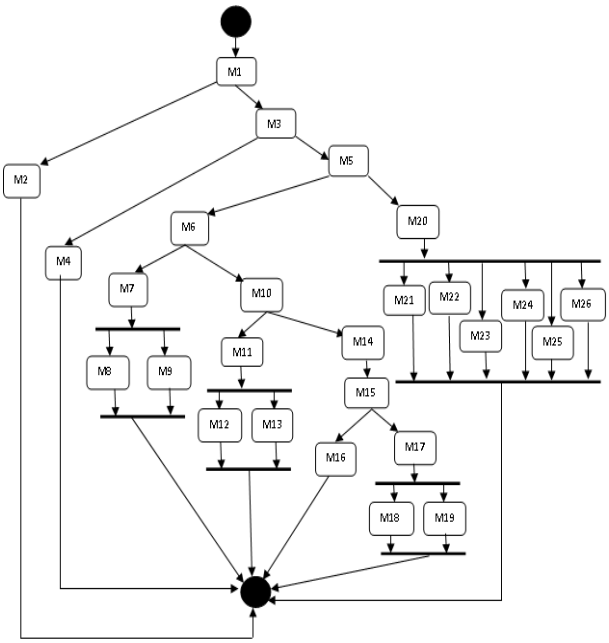

Fig. 2. Sequence Graph (SG) of Library Management System

TABLE.I.
MESSAGE DETAILS of SEQUENCE DIAGRAM

Message ID	Message Name	Message ID	Message Name
M1	Card Value	M14	Issue Book
M2	Invalid I_Card	M15	BOOKid
M3	PW Value	M16	Book not available
M4	Invalid PW	M17	Book available
M5	Account Value	M18	Update Library Database
M6	Account created	M19	Update User Account
M7	Close Account	M20	Account not created
M8	Destroy User Account	M21	Get Name
M9	Suspend IC	M22	Get F/M
M10	Continue	M23	Get Dob
M11	Return Book	M24	Get Address
M12	Update Library Database	M25	Get Phno
M13	Update User Account	M26	Get Affiliation

Test case generation:

For generating test sequences from the Sequence Graph ***All Message Path Coverage Criteria*** (explained in section III) is used.

We now propose a traversal algorithm called Generate Message Sequence Path which will traverse SG and generate the test case. The algorithm is a combination of Depth-First-Search (DFS) and Breath-First-Search (BFS). The BFS is used to traverse the concurrent nodes where as the DFS is used to traverse the rest nodes of the SG.

ALGORITHM 2: *Generate Message Sequence Path*
Input: *Sequence Graph (SG)*
Output: *Set of Message Sequence Paths (MSP)*

1) Start.
2) Traverse the MG.
3) Repeat the step 4 -7 until $(N_C)! = (N_E)$ // N_C, N_E being the current node end node respectively.
4) If $(N_C)!$ = Fork node.
5) Traverse the CCG using DFS (*Depth-First-Search*).
   a) Initialize all nodes to ready state.
   b) Push the starting node into the stack and changes its status to the waiting state.
   c) Repeat step d and e until stack is empty.
   d) Pop the top node n of stack. Process n and change the status of n to the processed state.
   e) Push on to the stack, all the neighbour of n that are in ready state, and change their status to the waiting state.
   f) Exit.
6) If NC = Fork node.
7) Traverse the subtree using BFS (*Breath-First-Search*).
   a) Initialize all nodes to ready state.
   b) Put the starting node in queue and change its status to the waiting state.
   c) Repeat step d and e until queue is empty.
   d) Remove the front node n of queue. Process n and change the status of n to the processed state.
   e) Add to the rear of the queue all the neighbours of n that are in ready state, and change their status to the waiting state.
   f) Exit.
8) End.

After applying the Algorithm 2 on the SG, we obtain the following Message Sequence Paths (MSP):

i) M1 – M2
ii) M1 – M3 – M4
iii) M1 – M3 – M5 – M6 – M7 – M8 – M9
iv) M1 – M3 – M5 – M6 – M10 – M11 – M12 – M13
v) M1 – M3 – M5 – M6 – M10 – M14 – M15 – M16
vi) M1 – M3 – M5 – M6 – M10 – M14 – M15 – M17 – M18 – M19
vii) M1 – M3 – M5 – M20 – M21 – M22 – M23 – M24 – M25 – M26

Now after getting the MSP the message name are obtained from the TABLE.I and applying it on the above message sequence path we obtain the following test sequences:

i) Card value – Invalid card
ii) Card value – Password value – Invalid password

iii) Card value – Password value – Account value – Account created – Close account – Destroy user account – Suspend IC
iv) Card value – Password value – Account value – Account created – Continue – Return book – Update library database – Update user account
v) Card value – Password value – Account value – Account created – Continue – Issue Book – Book Id – Book not available
vi) Card value – Password value – Account value – Account created – Continue – Issue Book – Book Id – Book available – Update library database – Update user account
vii) Card value – Password value – Account value – Account not created – Get Name – Get F/M – Get DOB – Get Address – Get PhNo – Get Affiliation

Though SD can be used to represents sequence of message that is passed between the objects and is useful for detecting *scenario faults*. The disadvantage is that a connector can not be used, suppose after filling of the form if the user wants to issue book then we are unable to connect the form filling thread to the book issue thread, where as this can be represented in Activity Diagram.

B) Generating Test Sequences from Activity Diagram

Activity Diagram shows set of activities which are to be executed to accomplish the task. As explained in section IV (A) if a user wants to issue a book after filling up the form then Sequence Diagram is able to connect the book issue thread with the form filling thread. But this is possible in case of Activity Diagram.

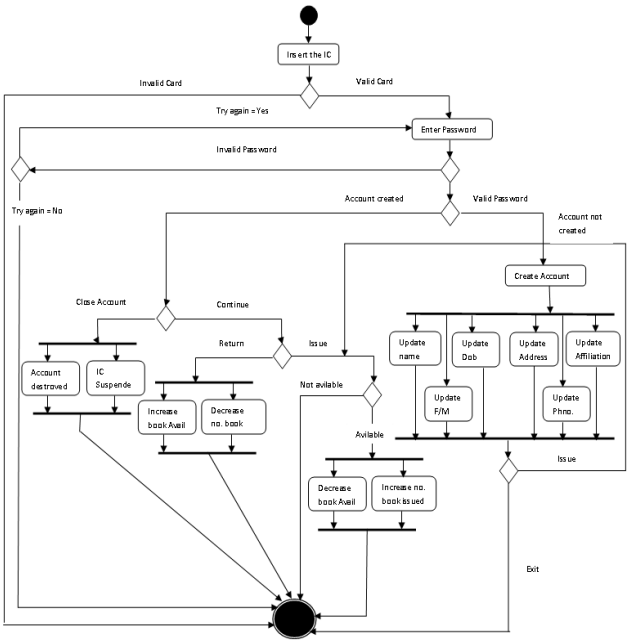

Fig. 3. Activity Diagram for Library Management System

Fig.3. shows an Activity Diagram (AD) for a library management system. In our next step the AD is converted into an intermediate format called Activity Graph (AG). AG being an intermediate format makes the test case generation process easier. We now propose a mapping algorithm which will convert the Activity Diagram (AD) into an Activity Graph (AG)

Algorithm 3: *Mapping Algorithm.*
Input: *Activity Diagram (AD).*
Output: *Activity Graph (AG).*

1) Start.
2) For each node participating in the AD do step 3 to11.
3) If the node is a initial node then map it into a node of type 'S'(start node) in the AG with pre-edge of 'S'=∅.
4) If the node is a end node then map it into node of type 'E'(end node) in the AG with post-edge of 'E'=∅.
5) If the node is a decision node then map it into node of type 'D' in the AG where the children are the resultant of the decision node.
6) If the node is the guard condition associated with the decision node then map in into node of type 'C' in the AG which is associated with the condition string and its parent node is of type 'D'
7) If the node is merge node then map it into node of 'M' in the AG which is having single outgoing edge and more than one incoming edge
8) If the is a fork node then map it into node of type 'F' in the AG with single incoming edge and more than one outgoing edge.
9) If the node is a activity associated with fork node then map it into node of type 'A' in the AG and its parent node is of type 'F'
10) If the node is a join node then map it into node of type 'J' in the AG with having one outgoing.
11) If the node is a normal activity node then map it into node of type 'A' in AG which is associated with the name of the activity associated with that node.
12) End

Where S is start activity, A is normal activity, E is end activity, D is decision node, C is the condition node, M is merge node, F is the fork node and J is join node,
After applying the mapping algorithm on the Activity Diagram the AG of the library management system is shown in Fig. 4.

Test case generation:

For generating test sequences from the Activity Graph ***Activity Path Coverage Criterion*** (as explained in section III) is used.

The AG is being traversed using the Algorithm 2. While traversing the nodes when ever there is a fork node we will go for a Breath First Search (BFS) otherwise the rest node are traversed using Depth First Search (DFS). However it is impossible to predict the type of nodes in the intermediate

format, because from the AG we can find that both fork and decision node are having multiple outgoing edges and both the merge and join node is having multiple incoming edges. So here we maintain a table called Node Details of Activity Diagram (NDAD) which will store the node number along with the activity associated and the type of activities. When ever we encounter a node with multiple outgoing edges we check the table to find out weather it is a decision node or fork node. The NDAD of the activity diagram is shown in TABLE.II.

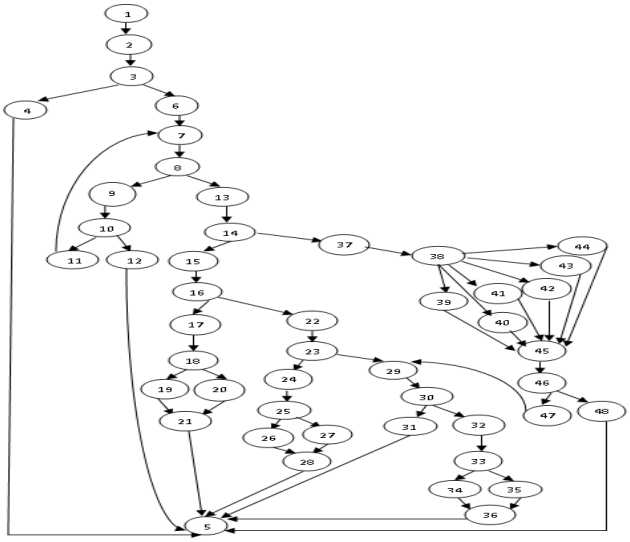

Fig. 4. Activity Graph of Library Management System

TABLE II.
NODE DETAILS of ACTIVITY DIAGRAM (NDAD)

Node Id	Activity associated	Type	Node	Activity associated	Type
1	Start	S	26	Increase book availability	A
2	Insert IC	A	27	Decrease no of book issued	A
3	Verify IC	D	28		J
4	Invalid IC	C	29	Issue	A
5	Eject card	A	30	Check availability	D
6	Valid card	C	31	Not available	C
7	Enter password	A	32	Available	C
8	Verify password	D	33		F
9	Invalid password	C	34	Decrease book availability	A
10	Try again	D	35	Increase no of book issued	A
11	Yes	C	36		J
12	No	C	37	Account not created	C
13	Valid password	C	38		F
14	Check acc. Created/ not	D	39	Update Name	A
15	Account created	D	40	Update F/M	A
16	Check close/ continue	C	41	Update DOB	A
17	Close	D	42	Update Address	A
18		F	43	Update Phno.	A
19	Account destroyed	A	44	Update Affiliation	A
20	IC suspended	A	45		J
21		J	46	Check issue/exit	D
22	Continue	C	47	Issue	C
23	Issue/ Return	D	48	Exit	C
24	Return	C	49	End	E
25		F			

After traversing the AG we obtain following of Activity Path.

i) 1 → 2 → 3 → 4 → 5 → 49
ii) 1 → 2 → 3 → 6 → 7 → 8 → 9 → 10 →12 → 5 → 49
iii) 1 → 2 → 3 → 6 → 7 → 8 → 9 → 10 → 11 →7 → 8 → 9 → 10 →12 → 5 → 49
iv) 1 → 2 → 3 → 6 → 7 → 8 → 13 → 14 → 15 → 16 → 17 → 18 → 19 →20 → 21 → 5 → 49
v) 1 → 2 → 3 → 6 → 7 → 8 → 13 → 14 → 15 → 16 → 22 → 23 → 24 →25 → 26 → 27 → 28 → 5 → 49
vi) 1 → 2 → 3 → 6 → 7 → 8 → 13 → 14 → 15 → 16 → 22 → 23 → 29 →30 → 31→ 5 → 49
vii) 1 → 2 → 3 → 6 → 7 → 8 → 13 → 14 → 15 → 16 → 22 → 23 → 29 →30 → 32→ 33 → 34 → 35 → 36 → 5 → 49
viii) 1 → 2 → 3 → 6 → 7 → 8 → 13 → 14 → 37 → 38 → 39 → 40 → 41 → 42 → 43 → 44 → 45 → 46 → 48 → 5 → 49
ix) 1 → 2 → 3 → 6 → 7 → 8 → 13 → 14 → 37 → 38 → 39 → 40 → 41 → 42 → 43 → 44 → 45 → 46 → 47 → 29 →30 → 31→ 5 → 49
x) 1 → 2 → 3 → 6 → 7 → 8 → 13 → 14 → 37 → 38 → 39 → 40 → 41 → 42 → 43 → 44 → 45 → 46 → 47 → 29 →30 → 32→ 33 → 34 → 35 → 36 →5 → 49
xi) 1 → 2 → 3 → 6 → 7 → 8 → 9 → 10 → 11 →7 → 8 →13 → 14 → 15 → 16 → 17 → 18 → 19 →20 → 21 → 5 → 49
xii) 1 → 2 → 3 → 6 → 7 → 8 → 9 → 10 → 11 →7 → 8 → 13 → 14 → 15 → 16 → 22 → 23 → 24 →25 → 26 → 27 → 28 → 5 → 49
xiii) 1 → 2 → 3 → 6 → 7 → 8 → 9 → 10 → 11 →7 → 8 → 13 → 14 → 15 → 16 → 22 → 23 → 29 →30 → 31→ 5 → 49

xiv) 1 → 2 → 3 → 6 → 7 → 8 → 9 → 10 → 11 →7 → 8 → 13 → 14 → 15 → 16 → 22 → 23 → 29 →30 → 32→ 33 → 34 → 35 → 36 →5 → 49

xv) 1 → 2 → 3 → 6 → 7 → 8 → 9 → 10 → 11 →7 → 8 → 13 → 14 → 37 → 38 → 39 → 40 → 41 → 42 → 43 → 44 → 45 → 46 → 48 → 5→ 49

xvi) 1 → 2 → 3 → 6 → 7 → 8 → 9 → 10 → 11 →7 → 8 → 13 → 14 → 37 → 38 → 39 → 40 → 41 → 42 → 43 → 44 → 45 → 46 → 47 → 29 →30 → 31→ 5 → 49

xvii) 1 → 2 → 3 → 6 → 7 → 8 → 9 → 10 → 11 →7 → 8 → 13 → 14 → 37 → 38 → 39 → 40 → 41 → 42 → 43 → 44 → 45 → 46 → 47 → 29 →30 → 32→ 33 → 34 → 35 → 36 →5 → 49

After getting the activity path we extract out the activities names from TABLE.II. and obtain the following test sequences. Due to space complexity here we have represented the first two and last one test sequences. The rest of the test sequences can be obtained in the same manner.

i) Start → Insert IC → Verify IC → Invalid IC → Eject card → End.

ii) Start → Insert IC → Verify IC → Valid IC → Enter password → Verify password → Invalid password → Try again → No → Eject card → End.

xviii) Start → Insert IC → Verify IC → Valid IC → Enter password → Verify password → Invalid password → Try again → Yes → Enter password → Verify password → Valid password → Check acc. Created/ not → Account not created → Update Name → Update F/M → Update DOB → Update Address → Update Phno.→ Update Affiliation → Check issue/exit → Issue → Check availability → Available → Decrease book availability → Increase no of book issued → Eject card → End.

As explained in IV (B), an Activity Diagram represents the information in an abstract way. It is useful for representing only sequence of activities but not how the communications happens between objects. Since combinational UML models are being capable of detecting more faults than compared to single UML models [7, 8, 9, 10]. So we propose an approach which combines the AG with the SG and will generate a graph called ASG. The test sequences will be generated from ASG, having the combined features of both the diagrams.

C) Generating Test Sequences from Activity Sequence Graph

In this section we propose an algorithm called Generate ActivitySequence Graph, which will combine the Activity Diagram (AD) and the Sequence Diagram (SD) to form a combinational diagram called Activity Sequence Graph (ASG). The algorithm for generating the ASG is explained in Algorithm 4

ALGORITHM 4: *Generate Activity Sequence Graph (ASG)*
INPUT: *Activity Graph (AG) and Sequence Graph (SG)*
OUTPUT: *Activity Sequence Graph (ASG)*

1) Start.
2) Traverse the AG
3) For $(N_C)! = (N_E)$ // N_C, N_E being the current node end node respectively.
4) $N_C = N_x$ // start with the node N_x
5) For $N_c = N_y$ Do the following // move to the next node
6) While $N_x \rightarrow N_y$ // there is a transition from N_x to N_y
7) Create two nodes N_x and N_y in the ASG, assign a Edge between them
8) Traverse SG to find out the message (M suppose) responsible for $N_x \rightarrow N_y$
   // Traverse SG to find out the message responsible for the transition between N_x and N_y
9) Assign the message M to the edge connecting the node.// i.e . N_x - M→ N_y
10) End.

In algorithm 4 we propose a technique for generating ASG by combining AG and SG. In this approach we first traverse the AG to find the transition from one node to another. When ever a transition is there we then take the two nodes in ASG and assign an edge between them. We then traverse the SG to find out the corresponding message which is responsible for the transition, when the message is found we then assign the message to the edge connecting the two nodes. Applying this technique the ASG is generated for the library management system which is shown in Fig.5.

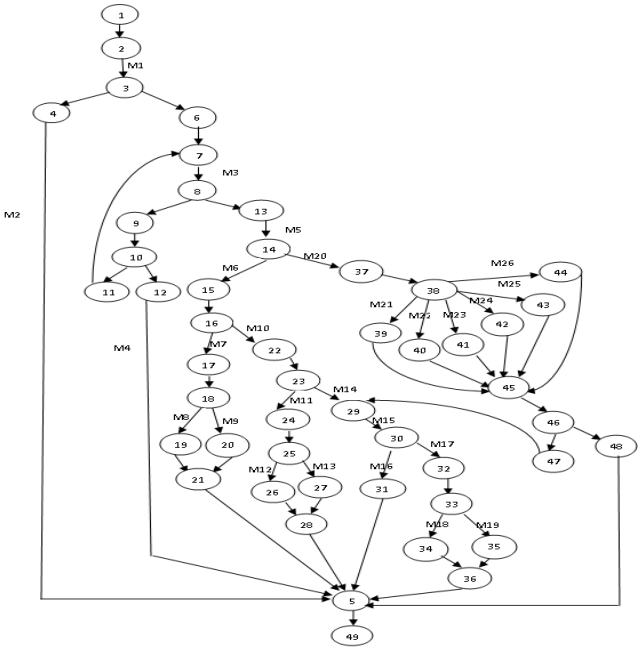

Fig. 5. Activity Sequence graph of Library Management system

Test case generation:

For generating test sequences from the Activity Sequence Graph ***Activity Path Coverage Criterion*** (explained in section III) is used.

The ASG is being traversed using Algorithm 2 to ASG to generate the activity path. However when a transition happens from one node to another node the message assigned to the edge will be considered in the Activity Sequence Path generation process

After applying the algorithm we obtain the following Activity Sequence Path:

i) 1 → 2 → M1 → 3 → 4 → M2 →5 → 49
ii) 1 → 2 → M1 → 3 → 6 → 7 → M3 → 8 → 9 → 10 →12 → M4 → 5 → 49
iii) 1 → 2 → M1 → 3 → 6 → 7 → M3 → 8 → 9 → 10 →11 →7 → 8 → 9 → 10 →12 → M4 → 5 → 49
iv) 1 → 2 → M1 → 3 → 6 → 7 → M3 → 8 → 13 → M5 → 14 → M6 →15 → 16 → M7 → 17 → 18 → M8 →19 →M9 →20 → 21 → 5 → 49
v) 1 → 2 → M1 → 3 → 6 → 7 → M3 → 8 → 13 → M5 → 14 → M6 →15 → 16 →M10 → 22 → 23 → M11 → 24 →25 → M12 → 26 → M13 →27 → 28 → 5 → 49
vi) 1 → 2 → M1 → 3 → 6 → 7 → M3 → 8 → 13 → M5 → 14 → M6 →15 → 16 →M10 → 22 → 23 → M14 → 29 → M15 →30 →M16 → 31→ 5 → 49
vii) 1 → 2 → M1 → 3 → 6 → 7 → M3 → 8 → 13 → M5 → 14 → M6 →15 → 16 →M10 → 22 → 23 → M14 → 29 → M15 →30 → M17 → 32→ 33 →M18 → 34 → M19 → 35 → 36 → 5 → 49
viii) 1 → 2 → M1 → 3 → 6 → 7 → M3 → 8 → 13 → M5 → 14 → M20 → 37 → 38 → M21 → 39 → M22 → 40 → M23 → 41 → M24 → 42 → M25 → 43 → M26 → 44 → 45 → 46 → 48 → 5 → 49
ix) 1 → 2 → M1 → 3 → 6 → 7 → M3 → 8 → 13 → M5 → 14 → M20 → 37 → 38 → M21 → 39 → M22 → 40 → M23 → 41 → M24 → 42 → M25 → 43 → M26 → 44 → 45 → 46 → 47 → 29 → M15 →30 →M16 → 31→ 5 → 49
x) 1 → 2 → M1 → 3 → 6 → 7 → M3 → 8 → 13 → M5 → 14 → M20 → 37 → 38 → M21 → 39 → M22 → 40 → M23 → 41 → M24 → 42 → M25 → 43 → M26 → 44 → 45 → 46 → 47 → 29 → M15 →30 → M17 → 32→ 33 →M18 → 34 → M19 → 35 → 36 → 5 → 49
xi) 1 → 2 → M1 → 3 → 6 → 7 → M3 → 8 → 9 → 10 →11 →7 → 8 →13 → M5 → 14 → M6 →15 → 16 → M7 → 17 → 18 → M8 →19 →M9 →20 → 21 → 5 → 49
xii) 1 → 2 → M1 → 3 → 6 → 7 → M3 → 8 → 9 → 10 →11 →7 → 8 →13 → M5 → 14 → M6 →15 → 16 →M10 → 22 → 23 → M11 → 24 →25 → M12 → 26 → M13 →27 → 28 → 5 → 49
xiii) 1 → 2 → M1 → 3 → 6 → 7 → M3 → 8 → 9 → 10 →11 →7 → 8 →13 → M5 → 14 → M6 →15 → 16 →M10 → 22 → 23 → M14 → 29 → M15 →30 →M16 → 31→ 5 → 49
xiv) 1 → 2 → M1 → 3 → 6 → 7 → M3 → 8 → 9 → 10 →11 →7 → 8 →13 → M5 → 14 → M6 →15 → 16 →M10 → 22 → 23 → M14 → 29 → M15 →30 → M17 → 32→ 33 →M18 → 34 → M19 → 35 → 36 → 5 → 49
xv) 1 → 2 → 3 → 6 → 7 → 8 → 9 → 10 → 11 →7 → 8 → 13 → 14 → 37 → 38 → 39 → 40 → 41 → 42 → 43 → 44 → 45 → 46 → 48 → 5→ 49
xvi) 1 → 2 → M1 → 3 → 6 → 7 → M3 → 8 → 9 → 10 →11 →7 → 8 →13 → M5 → 14 → M20 → 37 → 38 → M21 → 39 → M22 → 40 → M23 → 41 → M24 → 42 → M25 → 43 → M26 → 44 → 45 → 46 → 48 → 5 → 49
xvii) 1 → 2 → M1 → 3 → 6 → 7 → M3 → 8 → 9 → 10 →11 →7 → 8 →13 → M5 → 14 → M20 → 37 → 38 → M21 → 39 → M22 → 40 → M23 → 41 → M24 → 42 → M25 → 43 → M26 → 44 → 45 → 46 → 47 → 29 → M15 →30 → M17 → 32→ 33 →M18 → 34 → M19 → 35 → 36 → 5 → 49

After obtaining the Activity Message Sequence extract message name from TABLE.I and TABLE.II to obtain Test sequences. Due to space complexity here we have represented the first two and last one test sequences. The rest of the test sequences can be obtained in the same manner.

i) Start → Insert IC → [Card Value] → Verify IC =Invalid IC → [Display Invalid I_Card] → Eject card → End
ii) Start → Insert IC → [Card Value] → Verify IC =Valid IC → Enter password → [PW Value] → Verify password = Invalid password → Try again = No → [Display Invalid PW] → Eject card → End
xviii) Start → Insert IC → [Card Value] → Verify IC =Valid IC → Enter password → [PW Value] → Verify password = Invalid password → Try again =Yes Enter password → PW Value → Verify password = Valid password → [Account Value] → Check acc. Created/ not = [Account not created] → [Get Name] → Update Name → [Get F/M] → Update F/M → [Get DOB] → Update DOB → [Get Address]→ Update Address → [Get Phno] → Update Phno. → [Get affiliation] → Update Affiliation → Check Issue/exit = \Issue → [BOOKid] → Check availability = [Available] → [Update Library Database] → Decrease book availability → [Update User Account] → Increase no of book issued → Eject card → End

V. CONCLUSION AND FUTURE WORK

In this paper we have proposed a method for generating test cases from UML combinational diagram i.e. Activity Diagram (AD) and Sequence Diagram (SD). We have

converted the AD and the SD into intermediate formats called Activity Graph (AG), Sequence Graph (SG) respectfully. Finally we have combined the AG and the SG to form a combined graph called Activity Sequence Graph (ASG) and traversed the ASG to generate the test cases. The resultant test cases show that the test cases generated from the ASG is having more fault detection capabilities than the single modelling graphs.

Consider the last test sequence obtained from the Sequence Diagram. Here we can only find information about the message passed between the objects, but can't get any information about how the activity flow occurs. So this test sequence will be capable of detecting faults associated with message sequencing, and will not able to detect fault associated with decision node or loop faults. For example when ever we insert a card then suppose the last test sequence is obtained, then it is a valid test sequence for valid card and password. So we are unable to detect the faults associated with decision. Now consider the last test sequence obtained from Activity Diagram. Here we can find out the faults associated with decision as well as faults. For example when ever the card is verified then out put of the decision node invalid IC or valid IC is represented in the decision thread. But we are not able to find out the messages passed between the objects. On the other hand consider that the last test sequence of the Activity Sequence Graph, here the activities as well as message sequence is considered. So we will be able to detect more number of faults.

In future we are planning to apply Genetic Algorithm on the combined modelling graph ASG, so that we will obtain the optimal prioritized test suites which will be capable of detecting more faults in less effort and time.

References

[1] D. Kundu, D. Samanta "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.

[2] C. a. Sun "A Transformation-based Approach to Generating Scenario-oriented Test Cases from UML Activity Diagram for Concurrent Application" Annual IEE International Computer Software and Applications Conference, IEEE 2008, pp 160 – 167.

[3] H. Kim, S. Kang, J. Baik, I. Ko "Test Case Generation from UML Activity Diagram" Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing IEEE 2007, pp 556 – 561.

[4] M. Sarma, D. Kundu, R. Mall "Automatic Test Cases Generation from UML Sequence Diagram" 15th International Conference on Advanced Computing and Communications, IEEE, 2007, pp 60 – 65.

[5] P. Samuel, A. T. Joseph "Test Sequence Generation from UML Sequence Diagram", ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing IEEE 2008, pp.879 – 887.

[6] M. Sarma, R. Mall "Automatic Test Cases Generation from UML Models", 10th International Conference on Information Technology, IEEE, 2007, pp.196 – 201.

[7] D. Sokenou, "Generating Test Sequences from UML Sequence diagram and State Diagrams" pp 236 – 240.

[8] M. Riebisch, I. Philippow, M. Götze "UML-Based Statistical Test Case Generation"

[9] S. K. Swain, D. P. Mohapatra, Rajib Mall "Test Case Generation Based on State and Activity Models", Journal of Object Technology, pp. 1 – 27.

[10] S. K. Swain, D. P. Mohapatra, "Test Case Generation from Behavioral UML Models", International Journal of Computer Applications, Volume 6– No.8, September 2010, pp. 5 – 11.