Test Case Generation from Behavioral UML Models

Santosh Kumar Swain
School of Computer Engineering
KIIT University, Bhubaneswar
Orissa, India

Durga Prasad Mohapatra
Department of Computer Science & Engineering
National Institute of Technology, Rourkela
Orissa, India

ABSTRACT

We propose an integrated approach to generate test cases from UML sequence and activity diagrams. We first transform these UML diagrams into a graph. Then, we propose an algorithm to generate test scenarios from the constructed graph. Next, the necessary information for test case generation, such as method-activity sequence, associated objects, and constraint conditions are extracted from test scenario. Our approach reduces the number of test cases and still achieves adequate test coverage. We achieve message-activity path coverage and category partitioning method for each predicate conditions found in the specific path of the design model.

Keywords

Software Testing, UML Models, Sequence diagram, Activity diagram, Model Flow Graph, Test Sequence.

1. INTRODUCTION

Thorough *software testing* is necessary to produce highly reliable systems. Quality of the end product and effective reuse of software depend to a large extent on testing [1, 2]. Unless we can find more efficient ways to perform effective testing, the percentage of development costs devoted to testing will increase significantly. Testing requires executing a program on a set of test cases and comparing the actual results with the expected results [3]. Large systems are inherently complex to test and require large number of test cases to be designed. Creation of test cases is possibly the most difficult step in testing. Developers therefore spend considerable time and effort to achieve thorough testing. Designing a large number of test cases and carrying out the tests turn out to be very labor-intensive and time consuming. To reduce the testing cost and effort and to achieve better quality software, automatic testing has become an urgent necessity. This is especially true since program sizes and complexities are rapidly increasing. Automatic *test case* generation can reduce development cost by eliminating costly manual *test case* design efforts and at the same time help increase reliability through increased test coverage. A test adequacy criterion defines the extent to which a property that must be tested [4, 5]. Tests that are adequate with respect to a criterion, covers all the elements in the domain determined by the criterion.
Generating test data form high level design notations has several advantages over code-based test case design [6]. Testing based on design models has the advantage that the test cases remain valid even when the code changes a little bit. Design models can be used as a basis for *test case* generation, significantly reducing the costs of testing [7]. The process of generating test cases from design will help to discover problems early in the development process and thus it saves time and resources during development of the system. However, selection of test cases from UML model is one of the most challenging tasks [8].

We report our work concerning automatic test case generation based on UML sequence and activity diagrams. UML has now become the de facto standard for object oriented modeling and design [9]. *UML models* are an important source of information for test case design, which if satisfactorily exploited, can go a long way in reducing testing cost and effort and at the same time improve software quality [10]. UML-based automatic test generation is a practically important and theoretically challenging topic and is receiving increasing attention from researchers. Traditionally there have been lots of efforts to generate test cases from UML diagrams using heuristic based techniques such as statement-coverage, branch-coverage, message sequence coverage etc.

In UML, the behavior of a use case can be represented by using interaction, activity and state machine diagrams. *Sequence diagrams* capture the exchange of messages between objects during execution of a use case. It focuses on the order in which the messages are sent. *Activity diagrams*, on the other hand, focus upon control flow as well as the activity-based relationships among objects. These are very useful for visualizing the way several objects collaborate to get a job done. These are very useful for describing the procedural flow of control through many objects.

In our approach, we transform the sequence and *activity diagrams* to an intermediate graph. From the constructed graph, we generate different test sequences, which represent different scenarios. From the generated test sequences, test cases are generated, which satisfy the message-sequence test path adequacy criteria. We focus on generating tests from design description, as it represents a significant opportunity for testing in a form that can easily be manipulated by automated means.

The rest of the paper is organized as follows: The next section discusses review of relevant UML diagrams. The third section describes analysis of related works. Section 4 defines the different testing criteria. Section 5 presents concepts, notations and terminologies with intermediate representations of UML diagram. Section 6 describes our approach for generating test cases from sequence and *activity diagram*. Section 7 illustrates a case study to measure the effectiveness of our approach. Section 8 reports the comparison with related works. The paper concludes with section 9.

2. RELEVANT UML DIAGRAMS

A *sequence diagram* shows system events for a use case. It is said to implement a use case [11]. A *sequence diagram* shows the messages that are exchanged among several objects, as well as certain control-flow information (e.g. the order in which messages are sent and the conditions that guard the messages). It shows the dynamic collaborations between a certain number of objects, highlighting the way in which a particular scenario is realized using the interactions of a (sub)set of these objects [3]. *Sequence diagram*s include flows of events during interactions, with primary flows and alternative flows. Alternative flows represent conditional branches in the processing. In UML, a *message* is a request for a service from one object to another; these are typically implemented as *method calls*. Each *sequence diagram* represents a complete trace of messages during the execution of a user-level operation. Such diagrams capture important aspects of object interactions, and can be naturally used to define testing goals that must be achieved during testing. When a message is sent to an object, it invokes an operation of that object. Once a message is received, the operation that has been invoked begins to execute.

An UML *activity diagram* describes the sequential or concurrent control flow between activities. *Activity diagram* can be used to model the dynamic behavior of a group of objects. *Activity diagram*s emphasize the activities of the object or a group of objects, so it is the perfect one to describe the realization of the operation in the design phase and to describe the sequence of the activities among the involving objects in the control flow during the implementation of an operation. It also describes the relationship between the activity and the object in the message flow, the state change of object in the object flow at the time of execution of activity [12]. Use cases are often supplemented with *activity diagram*s if the control structure of the use case includes loops or branches. The use of *activity diagram*s allows defining a coverage criterion to ensure a particular degree of completeness of the test scenarios. This diagram is able to reflect all possible scenarios for one use case.

3. RELATED WORK

Trung D.Trong [13] proposed a systematic approach to testing design models described by UML class diagrams, *sequence diagram*s and *activity diagram*s and also *test adequacy criteria* for those diagrams. These criteria are presented as a general discussion and not explicitly defined. They suggested all edge criterions which require every activity edge of an *activity diagram* to be covered during testing.

Pilskalns et al. [14] propose a graph-based approach to combine the information from class diagrams and *sequence diagram*s. In this approach, each *sequence diagram* is transformed into an Object-Method Directed Acyclic Graph (OMDAG). The OMDAG can be used to derive test execution paths and their corresponding conditions, which are recorded in a table called the Object-Method Execution Table (OMET).

Ghose et al [4] propose a graph-based approach to combine the information from class and *sequence diagram*s. In their approach, the relevant information are integrated into a Variable Assignment Graph (VAG). The VAG is used to derive test input that satisfies the *test adequacy criteria*.

Linzhang et al. [12] propose a method to automatically generate test cases from UML *activity diagram*s using a gray-box method. In their method they generate test cases directly from UML *activity diagram*s. Their proposed method exploits the advantage of black box testing to analyze the expected external behavior and white box testing to cover the internal structure of the *activity diagram* of the system under test to generate test cases. Basanieri and Bertolino [3] define a testing approach that considers all message sequences in a *sequence diagram* and apply the category-partition method to choose appropriate test data for exercising these sequences.They characterize a test case as a combination of all suitable choices of the involved settings and interactions in a sequence of messages. In another work, Basanieri et al. describe the CowSuite approach [15] which provides a method to derive the test suites and a strategy for test case prioritization and selection. CowSuite is mainly based on the analysis of the use case and sequence diagrams. From these two diagrams they construct a graph structure which is a mapping of the project architecture and this graph is traversed using a modified version of the depth-first search algorithm. They use category partition method for generating tests. Their test procedure consists of a sequence of messages, and the associated parameters.

Fraikin and Leonhardt [16] describe the SeDiTeC tool for testing based on *sequence diagram*s. Their approach achieves coverage of all possible sequences of messages in a set of related *sequence diagram*s. The diagrams are augmented with information about expected input and output values for method invocations and these values are checked during test execution.

4. BASIC CONCEPT

In this section, we define different testing Criteria.

1. Test Criteria based on Sequence Diagram

We have adopted message path *test adequacy criteria* for *sequence diagram.* They are described as follows:

1) *Message Sequence Path criterion*: For each *sequence diagram*, there must be at least one test case T such that when the software is executed using T , the software that implements the message sequence path of the *sequence diagram* must be executed. The message sequence path coverage criterion is used to generate tests from the *sequence diagram*s. For each *sequence diagram* in the specification, a test case is generated for each normal and for each alternative message sequence.

2. Test criteria based on Activity Diagram

The *test adequacy criteria* proposed in the literature based on *activity diagram* are as follows:

1) *All Basic Path Coverage Criterion*: A basic path is a complete path through an *activity diagram* where each loop is exercised either zero or one times. This ensures that all iterations in an *activity diagram* are exercised.

2) *All Activity Path Coverage*: Given at test set T and *Activity Diagram* AD, T must cause each possible activity path in AD to be taken at least once. An Activity Path is any sequence of activities from the initial activity into the terminal activity in the *activity diagram*.

3. Coverage Criteria based on both Sequence and Activity Diagram

In this section, we describe a few criteria that can be defined by considering both Sequence and *activity diagram* together. A message path represents the flow of message from the start message to the last in a sequence diagram. The message invokes a method call. All the activities to execute the method can be shown through activity diagram. Using sequence diagram, we can show only message paths. But if we use sequence and *activity diagram* we can cover message as well as activity path which is called message-activity-path. So errors uncovered in message-activity-path can not be uncovered by message-path. But the reverse is possible. Thus message-activity-path coverage which is the super set ensures message-path coverage which is subset.

Theorem1: *Message Path Coverage is a stronger testing technique compared to message coverage.*

Proof: A message path in a MFG is a path from the root node to any other node in the MFG. All message paths in the MFG implies that there is no any message in the MFG, which is not covered by some message path(s). Hence, if a test suit achieves message path coverage, then it essentially covers all messages. Therefore, message path coverage ensures message coverage. So, message path coverage is a stronger testing technique compared to message coverage.

All Message-Activity Path Coverage: Given a test set T and a *sequence diagram* SD and an *activity diagram* AD, T must cause each possible message path in SD with corresponding activity path in AD to be taken at least once.

Theorem2: *Message-Activity path Coverage is a stronger testing technique compared to message path coverage.*

To prove above theorem2, we need to prove that

i) Message-path coverage does not ensure Message-Activity-Path coverage.

ii) Message-Activity-Path coverage ensures Message-Path coverage.

The proof is given in [17].

4. Category Partition Method

The category-partition method [16, 18] is a specification-based testing strategy that uses an informal functional specification to produce formal test specifications. The category-partition method offers a general procedure for creating test specifications. The key job is to develop categories, which are defined to be the major characteristics of the input domain of the function under test, and to partition each category into equivalence classes of inputs called choices. By definition, choices in each category must be disjoint, and together the choices in each category must cover the input domain. The category partitioning approach utilizes a program's specification to 1) identify separately testable functional units 2) categorize the inputs for each functional unit and 3) partition the input categories into equivalence classes. The category-partition method identifies behavioral equivalence of classes within the structure of a system under test. A category or partition is defined by specifying all possible data choices that it can represent.

5. INTERMEDIATE REPRESENTATION

To generate test data, it is first required to transform the diagram into a suitable intermediate representation. Both the UML sequence and *activity diagram*s represent behavioral aspects of the design phase. Thus we have proposed a method to integrate both the diagrams into intermediate graph structure called *Model Flow Graph* (MFG) and generate test sequence from the graph. The algorithm for generating MFG from the UML diagrams is described in the next section. In this section we present a few basic concepts, notations, and terminologies with intermediate graph representation (MFG).

Definition 1 (*Model Flow Graph*): *A Model Flow Graph (MFG) G of a diagram D is a flow graph 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. Nodes S and T are unique nodes representing entry and exit of the diagram D, respectively.*

Note that the existence of an edge (x, y) in the MFG does not mean that control must transfer from x to y during execution of the diagram. Fig. 1 shows a *sequence diagram* and Fig.2 is its corresponding MFG. In Fig. 2, we have labeled the nodes using message sequence numbers.

A *Model Flow Graph* (MFG) generated from *sequence diagram* represents possible message/method sequences in an interaction. We generate test cases from MFG. A MFG can be viewed as a graph G= (V, E), where V is a set of nodes of G, and E is a set of edges. The nodes of G represent messages and edges represent transition between two nodes exists, if the corresponding messages in the *sequence diagram* occur one after the other. The message that initiates the interaction is made the root of the graph. In the *activity diagram* all transitions are labeled with a guard condition. The conditional predicate of a guard might trivially be an empty predicate which is always true. In case of an *activity diagram*, the nodes of G represent conditional predicates and edges represent transition between two nodes, if the corresponding predicate in the *activity diagram* occur one after the other. In order to handle hierarchical states we consider the *activity diagram* as a tree of sub graphs. The initial predicate is represented as the start node of the graph.

An event in a Message/Activity Set is denoted by a tuple, maEvent: <messageName; SendObject; ReceiveObject [/guard]> where, messageName is the name of the message with its signature, SendObject is the sender object of the message and ReceiveObject is the receiver of the message and the optional part /guard is the guard condition subject to which the maEvent will take place.

6. OUR APPROACH TO GENERATE TEST CASES

The test generation process is divided into three main phases. The first phase is to generate MFG from sequence and *activity diagram* separately. The second phase is to generate test sequences from MFG corresponding to sequence and *activity diagram*s. The test sequences are a set of theoretical paths starting from initialization to end, while taking conditions (pre-condition and post-condition) into consideration. Each generated test sequence corresponds to a particular scenario of the considered use case. The third phase is to generate test case from the generated sequences satisfying the message-activity path *test adequacy criteria*.

A. *Algorithm for Generating MFG from UML diagrams*

The MFG for *sequence diagram* is created by 1. Associating methods in the sequence diagram with their originating objects, 2. Traversing the sequence diagram from beginning to end, showing choices and condition for method execution.

1. Associating methods with their objects:

a) For each method call m originating from object O, create an entry into the object method association (OMA) table and labeled as 'A'. For example O:m().

b) Each explicit return form any object has also an entry into the object method association table. For example O:return().

Create the MFG by analyzing the OMAT (Object Method Association table). The edges are directed to reflect the ordering of the method calls. The edges are labeled with a sequence number. Concentric circles denote method call of another object (transfer of control). The MFG for *activity diagram* is created by traversing the *activity diagram* from beginning to end, showing choices, conditions, concurrent executions, loop statements.

a) For each conditional statement create an entry into the Method Activity Table (MAT). Then traversing the MAT, create nodes in the MFG.

b) The loop statements are transformed into conditional statements, listed in the MAT.

c) For each concurrent execution statements an entry is made into the MAT for each execution path and in turn is represented by different execution paths in MFG.

The edges are labeled according to the following rule:

a) Each call must be assigned a positive integer value based upon the sequence, thus each new call will require being incremented by one.

b) A ‘dot’ notation indicates nesting within a calling sequence. Each time a nested call is made an additional number is appended to the sequence number. For example 1.1a, 1.2b where a, b represent conditions.
c) The conditions use letters to denote the options.
d) For each loop activity, transform the loop into series of conditional activity. This is possible because the max and min values are specified.
e) For concurrent statements use capital letters as prefix to denote concurrent execution paths. For example A1.1a, A1.1b.

B. *Test Sequence Generation Process*

From the MFG different control flow sequence are identified by traversing the MFG by depth first traversal algorithm. During traversal, we look for conditional predicates on each of the transitions. For each conditional predicate, we apply category partitioning method to identify the partitions which will be listed in the test sequence. The test sequence consists of the edge label of the current message or activity, the name of the activity, the object associated and the corresponding category for each guard condition. In case of no guard condition, NULL is used.

C. *Test Case Generation Process*

To generate test cases that satisfy the message-activity path criteria, we first enumerate all possible basic paths from the start node to a final node in the MFG of sequence and activity diagrams. Each path then is visited to generate test cases. During visit, we look for conditional predicates on each of the transitions for execution of corresponding message and activity. For each conditional predicate, we apply category partitioning method to identify, to which partition the guard condition belongs to and the partitions are listed in the test sequence.

To generate test cases that satisfy the message-activity path coverage criterion, we propose an algorithm called *TestCaseGeneration*. *TestcaseGeneration* starts by enumerating all basic paths in the SDG, from the start node to the final node. Each basic path then is visited to generate test cases. Steps 2 to 21 are iterated for each path in the MFG. A basic path essentially corresponds to a scenario. Step 4 determines the initial pre-condition of the scenario from the start node S. For each considered path, Steps 6 to 17 determine the various pre-conditions, input, output and post-conditions for each interaction of the considered test sequence. This gives the test cases for finding out interaction faults if any. Finally, Step 20 gives the test case corresponding to the test sequence as a whole. The algorithm *TestCaseGeneration* to generate test cases satisfying the coverage criterion is presented below.

```
Algorithm TestCaseGeneration
Input: MFG of Sequence and Activity diagram
Output: Test suite T
Steps:
1. P[ ] = EnumerateAllBasicPaths(MFG)   //Enumerate all paths P
   ={P[1],P[2] ,...,P[n]} from the start node to a final node in the MFG.
2. For each path P[i] ∈ P  do
3.      curr_node=S                        // current node is assigned
   with start node S
4.      preCi = FindPreCond (S)
5.      ti←Φ                          // the test case for the path i
6. while (curr_node≠final_node of path P[i]) do
7.   event_curr_node= FindEvent (curr_node)// Event contains method or
   activity, parameters
                         and condition C. The event corresponding to
   the node curr_node
8.   If C≠G                          //If there is no guard condition
   G
9.     t ={preC, I(a1 ,a2 ,...,al),O(d1,d2 ,...,dm ), postC}
        // preC = precondition of the method or activity
       // I( a1 ,a2 ,...,al) = set of input values for the method or activity in
   sendObject
      // O( d1 ,d2 ,...,dm) = set of resultant values in the receiveObject
   when the method or activity is executed
      // postC = the postcondition of the method or activity
10.  End-if
11.  If (C== G ) then   //method or activity  is under guard condition
12.      C(val)= (C1 ,C2 ,...,Cl)                 // The set of value of
   clauses on the path P[i]
13.      t={preC, I(a1 ,a2 ,...,al),O(d1 ,d2 ,..., dm ), C(val), postC}
14.  End-if
15.  ti  = ti  ∪ t   // Add t to the test case set ti
16. curr_node=next_node // Move to the next node on the path P[i].
17. End-while
18. Determine the final output Oi and postCi for the final_node of P[i]
19.  t = {preCi , Ii ,Oi , postCi}
20.  ti ←ti ∪ t
21. T ←T ∪ ti
22.  End-for
23. Return (T)
24. Stop
```

7. CASE STUDY

We give a case study of technique. The first section provides the overview of the problem, the *sequence diagram* and *activity diagram* of the design models. The next sections describe the process of the test case generation from the diagrams.

A. *The problem and the model of the solution*

The model described in this paper is a design solution to an ATM (Automated Teller Machine) System. The bank system has an account for each customer. The customer of the system enter an amount to the ATM machine and the system checks whether the enter amount can be withdrawn from the corresponding account by comparing it with the current account balance. The Fig. 1 shows the *sequence diagram* of a simple ATM withdraw operation. The users of the system enter the amount. The ATM system then sends the amount and the account number to the bank system. The bank system retrieves the current balance of the corresponding account and compares it with the entered amount. If balance amount is greater than the entered amount then the amount can be withdrawn and the bank system returns true otherwise it checks for credit limit if the entered amount is less than the total amount (current balance + credit amount) then return true otherwise return false. Depending on the return value the ATM machine dispenses the cash and prints receipts or displays the failure message.

B. *Sequence Diagram for ATM Machine and Object Method Association Table*

The *sequence diagram* of the ATM system is first transformed into MFG in Fig. 2. In the MFG construction each message in the *sequence diagram* is represented by a node in the MFG. The timing ordering of the diagram is maintained in the system. The conditional message in the diagram is represented by a node and two outward nodes. Whether the condition is true or false one of the edges is covered.

Whenever the withdraw method is called it is represented by a concentric circle because by calling the withdraw method a message is passed to the account object. The OMAT (Object Method Association Table) is created by analyzing the *sequence diagram*. The MFG is created by analyzing the OMAT is shown in Fig 3. For each label in the

OMAT a node is created in MFG. The sequence of message flow in the OMAT is maintained in the MFG.

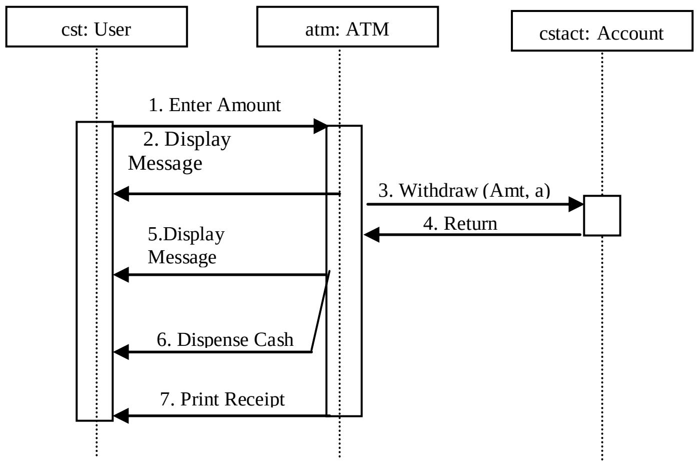

Figure 1 sequence diagram for ATM withdraw use case

TABLE 1: Object Method Association Table (OMAT) for the Fig. 1

SYMBOL	OBJECT-METHOD ASSOCIATION
A	atm:Validate Amount
B	cstact: Withdraw
C	atm: Display Message
D	cstact: Return
E	atm: Dispense Cash
F	atm: Print Receipt
G	atm: Return

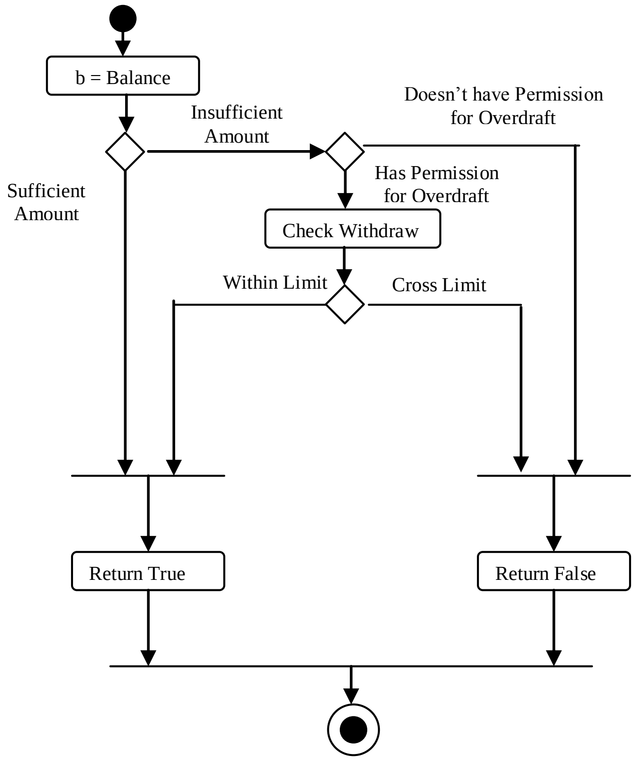

Figure.2 *Activity diagram* for ATM withdraw method

C. *Activity Diagram for method Withdraw() and Method Activity Table*

Fig 2 shows the *activity diagram* that describes the flow of activities inside the account object. There are different conditional predicates are associated with the *activity diagram.* The MFG is created directly from the *activity diagram*. Fig. 4 shows the corresponding MFG generated by analyzing the CPT (Conditional Predicate Table) shown in table 2. Then we have to consider the pre and post conditions and main scenario for generating the test case.

TABLE 2: Method Activity Table (MAT) for the Fig, 2

SYMBOL	ACTIVITY
A	Amount < Balance
B	Update Balance
C	Check for Overdraft
D	Check WithDraw Limit
E	Does not have Permission for Overdraft
F	With in Limit
G	Beyond Limit
H	Return True
I	Return False
J	Return

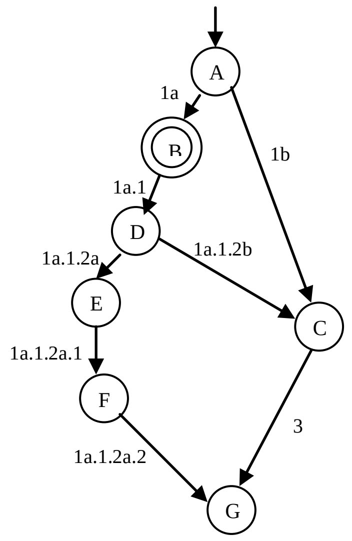

Figure 3 MFG for *sequence diagram* of withdraw use case

Conditions for accounts withdraw method are:

Pre Condition:

1) Account does have some minimum balance.
2) Amount entered should be a valid amount.

Post Condition

1) Withdraw successful.
2) Withdraw unsuccessful.

Main Scenario

1) The user invokes the withdraw method.
2) The user enter amount to be drawn.
3) The user successfully withdraw from the ATM.

Alternate course of action

The user can not withdraw the enter amount. The system notifies the user.

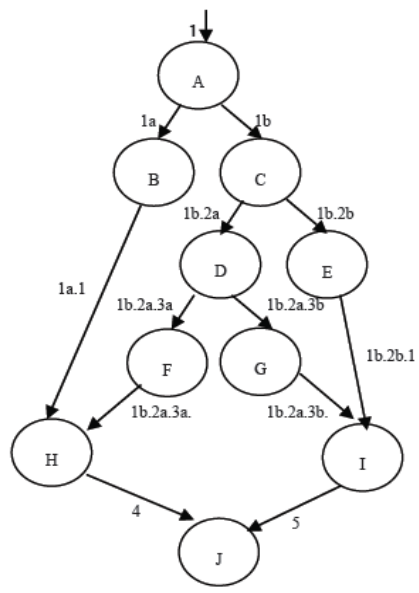

Figure. 4 MFG for *activity diagram of withdraw method*

D. *Test Sequence Generation*

From the MFG we identify five control flow sequence by traversing the MFG by depth first traversal algorithm. During traversal, we look for conditional predicates on each of the transitions. For each conditional predicate we apply category partitioning method to identify, to which partition the guard condition belongs to and the partitions are listed in the test sequence. The test sequence 1 lists customer object, transaction object and customer account object. The test sequence for scenario (customer enter invalid amount) consists of the edge label of the current message or activity, the name of the activity, the object associated and the corresponding category partitioning value for each guard condition. In case of no guard condition NULL is used.

1) *Test Sequence 1*

Objects: {cst: Customer, txn: Transaction, cstact: Customer Account}
Link: {cst-txn,txn-cst}

TABLE 3 Test Sequence 1

EDGE LABEL	MESSAGE/ACTIVITIES	OBJECT	CATEGORY PARTITION
1	Check Amount	ATM	Amount<0
1b	Display Message	ATM	NULL
3	Return	ATM	NULL

2) *Test Sequence 2*

Objects: {cst: Customer, txn: Transaction, cstact: Customer Account, crd: CreditCard, ctg: Category}
Link : {cst-txn, txn-cstact, cstact, txn-cst}

TABLE 4 Test Sequence 3

EDGE LABEL	MESSAGE/ACTIVITIES	OBJECT	CATEGORY PARTITION
1	Check Amount	ATM	Amount>0 && Amount mod 100 == 0)
1a	Withdraw ( )	Account	NULL
Withdraw ( )			
1a	Check Amount	Account	Amount>Balance
1b.2a	Check For Overdraft	Credit Card	TRUE
1b.2a.3a	Check for Withdraw Limit	Category	Amount<=Balance+ Credit Limit
1b.2a.3a.1	return= TRUE	Account	NULL
4	Return	Account	NULL
1a.1	Check Return	ATM	COR==TRUE
1a.1.2a	Cash Dispense	ATM	NULL
1a.1.2a.1	Print Receipt	ATM	NULL
1a.1.2a.2	Return	ATM	NULL

E. *Test Case Generation*

The test cases following the message-activity paths are given below.

Test case 1 {Input : Invalid amount, Output : Return Card with failure message}

Test case 2 {Input : Valid amount , Output : Dispense Cash}

Test case 3 {Input : Amount > Balance within Credit limit, Output : Dispense Cash}

Test case 4 {Input : Amount > Balance with no Credit facilities , Output: Return the card with failure message}

Test case 5:{Input : Amount > Balance + Credit limit, Output : Return card with failure message}

8. COMPARISON WITH RELATED WORK

Many of the related works reported in the literature attempt to use UML state chart diagrams for testing [3, 15, 16, 19]. State chart diagrams are appropriate in the context of class level testing as these do not represent message and activity sequences and communications. Our approach uses sequence and *activity diagram*s and targets cluster level testing involving interaction among objects. Further, we use exactly the same UML diagrams developed for analysis and design, without requiring any additional formalism or effort specifically made for testing purposes. Many reported methods require augmenting the UML specifications with specific annotations to facilitate the test derivation, or an additional formalism that the methods can process [19, 20, 21].

On the other hand, many of the UML related testing works reported. They require manual methods for test data generation [6, 12, 18, 20]. At the same time many were successful up to the extent of, automatically generating valid test requirements. Their test requirements [10] are specific things like possible execution sequences of use cases, messages, transitions, etc., that must be satisfied or covered during testing. We consider these test requirements as a part of test case, but a test case should also specify the values of test data for which a particular test sequence (or a path) will be executed together with the expected output.

From the *sequence diagram*, it is evident that covering all paths from the start node to a final node would eventually cover all interactions as well as all message sequence paths. An *activity diagram* shows control and data flow of activities in an operation. It depicts the detail of logic of procedurally complex operations. Several faults such as incorrect response to a message, message/method invocation with improper or incorrect arguments, incorrect sequencing of messages, inappropriate control flows and missing flows etc. may occur in an interaction [9].We follow the message-activity path coverage criterion to derive the test set. In our approach, no redundant test cases are generated. Further, as we use *Model Flow Graph*, path selection is simple and the number of paths is bounded. We select predicates through a depth first traversal of the MFG which can reduce the number of execution steps as well as helps to achieve message-activity paths coverage.

9. CONCLUSION

We have presented a method to generate test cases automatically from UML sequence and *activity diagram*s. We convert the models into an intermediate representation called *Model Flow Graph* (MFG), which is an integration of intermediate representation of sequence and *activity diagram*s. Integration of these representations is helpful for the following reasons. Our approach covers three important faults, which usually occur in a system: message sequence faults, operation consistency faults and activity synchronization faults. The first two category of faults can be covered from the *sequence diagram*, whereas the later from the *activity diagram*. Our approach is meant for cluster level testing where object interactions are tested. It may be noted that MFG models the operational details of a use case. The integration will help us to guide whether a test driver needs to apply a specific test suite or not. Another, important reason of integrating is that test data those are necessary for test case are mined once and used in different level such as, *sequence diagram* (message sequence faults within the logic of one operation), *activity diagram* (to test activity synchronization fault) etc. Integration of models also uncovers new sequence of message-activity faults.

In our approach, Tests are intended to exercise behavioral paths determined by conditions and uncover faults related to interactions between objects. Our approach can also work on executable forms of UML as well as code implementing a design. But the test cases generated are to be optimized to reduce unduly increasing number of test cases.

10. REFERENCES

[1]. R.V.Binder, Testing object-oriented software: a survey. *Software Testing* Verification and Reliability, 6(3/4): 125-252, 1996.

[2]. R. Mall. Fundamentals of Software Engineering. Prentice Hall, 2nd edition, 2003.

[3]. Bertolino, F. Basanieri, A practical approach to UML-based derivation of integration tests, in: Proceedings of the Fourth International Software Quality Week Europe and International Internet Quality Week Europe (QWE), Brussels, Belgium, 2000.

[4]. S. Ghose, R. France, C. Braganza, N. Kawane, A. Andrews, O. Pilskalns: "Test adequacy assessment for UML design model testing" In: Proceeding of the International Symposium on Software Reliability Engineering, Denver, CO , pp. 332-343, 2003.

[5]. N. Kawanw: "Fault Detection Effectiveness of UML Design Model *Test Adequacy Criteria* ". ISSRE (2003).

[6]. A. Abdurazik, J. Offutt: Using UML collaboration diagrams for static checking and test generation. In: 3rd International Conference on the UML. 383-395, 2000.

[7]. P. Samuel and R. Mall: "Boundary Value Testing based on *UML Models*". In: Proceedings of the 14th Asian Test Symposium (ATS '05), 2005.

[8]. J. Offutt, A. Abdurazik: "Generating tests from UML specifications", In Proceedings of 2nd International Conference on the UML, pp. 416-429,

[9]. Object Management Group: The Unified Modeling Language UML 1.5 Technical Report formal/03-03-01, The Object Management Group (OMG) , 2003.

[10]. T. Dinh Trong: "A Systematic Procedure for Testing UML Designs". ISSRE(2003).

[11]. Booch, J. Rumbaugh and I. Jacobson: "Unified Modeling Language User Guide". Addition-Wesley, 1999.

[12]. W.Linzhang,Y.Jiesong et al.: "Generating Test Cases from UML *Activity Diagram* based on Gray-Box Method". In Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC04), pages 284-291. IEEE,2004.

[13]. T. Dinh Trong: "A Systematic Approach to Testing Design Models" In Doctoral Symposium, 7th International Conference on the Unified Modeling Language, Lisbon, Portugal, 10-15, October 2004.

[14]. O. Pilskalns, A. Andrews, S. Ghose, Robert France: "Rigorous testing by merging structural and behavioral uml representations. " In: Proceeding of the 6th International Conference on the Unified Modeling Language, San Francisco, CA , pp. 234-248, 2003.

[15]. Basanieri, A. Bertolino, E. Marchetti, The cow suit approach to planning and deriving test suites in UMLprojectsProceedings of the Fifth International Conference on the UML, LNCS, 2460, Springer-Verlag GmbH, Dresden, Germany, pp. 383–397, 2000.

[16]. Fraikin, T. Leonhardt, SEDITEC-testing based on *sequence diagram*s, in: Proceedings 17th IEEE International Conference on Automated Software Engineering, IEEE Computer Society, pp. 261–266, 2000.

[17]. S. K. Swain. UML-based Testing of Software System, Technical Report, KIIT, 2005.

[18]. Andrew, R. France, S. Ghose, G. Craig: "Test Adequacy Criteria for UML Design Models". Journal of Software Testing, Verification and Reliability 13 : 95-127, 2003

[19]. Cavarra, C. Crichton and j. Davies: "A method for the automatic generation of test suites from object models. Information and Software Technology, 46(5): 309-314, 2004.

[20]. Hartmann, C. Imoberdorf, M. Meisinger, UML-based integration testing, in: ACM SIGSOFT Software Engineering Notes, Proceedings of International Symposium on *Software testing* and analysis, 2000.

[21]. T. Dinh Trong "Rules for Generating Code From UML Collaboration diagram and *Activity Diagram*s" Master's Thesis, Colorado State University, Fort Collins, Colorado,2003.

[22]. A. Abdurazik, J. Offutt, A. Baldini: "A Controlled Experimental Evaluation of Test Cases Generated from UML Diagrams", Technical report, George Mason University, Department of Information and Software Engineering, May, 2004.

11. AUTHORS PROFILE

Santosh Kumar Swain is presently working as faculty in School of Computer Engineering, KIIT University, Bhubaneswar, Orissa, India. He has acquired his M.Tech degree from Utkal University, Bhubaneswar. He has contributed more than seven papers to international Journals and Proceedings. He is having 18 years of teaching experience. He has written one book on "Fundamentals of Computer and Programming in C". His special fields of interest include Software Engineering, Object-Oriented Systems, Sensor Network and Compiler Design etc.

Durga Prasad Mohapatra studied his M.Tech at National Institute of Technology, Rourkela, India. He has received his Ph.D from Indian Institute of Technology, Kharagpur, India. He is currently working as associate Professor at National Institute of Technology, Rourkela. His special fields of interest include Software Engineering, Discrete Mathematical Structure, Program Slicing and Distributed Computing. He is a member of IEEE.