Software Testing is one among the major phases of Software Development Life Cycle (SDLC). Testing is a process carried out to ensure that the developed system is working as per the given requirements and is error-free Test case generation is a major step in software testing. Software can be tested at any phase of SDLC. Earlier the system is tested, it becomes easier to reveal the faults and there by greatly reducing the cost and time required to develop the system. This paper aims at generating test cases from artefacts developed from design phase of the Software Development Life Cycle. UML Activity Diagram is selected as the design artefact to generate the test cases. The proposed approach generates optimized set of paths from the Activity Diagram. The main idea behind this is the efficiency of the testing process. Generating all possible test cases and then optimizing them is impractical. It will unnecessary create either invalid test cases or redundant test cases and eventually makes the entire process of test case generation inefficient. This approach mainly aims at improving the efficiency of the system by automating the whole process and by generating only the optimized set of test cases 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. Testing plays a major role in the development of a system. The testing process verifies whether the designed software works as per the user requirements. The testing process ingests nearly 80 percent of the cost of the software development. Testing can be done during any phase of the software development life cycle. The usual practice is to test the system after the implementation phase In case of any defects at this phase, the entire system needs to be corrected to fix the defect. Therefore this process will increase the cost involved for the system development. This paper mainly aims at model based testing. In model based testing design models are used to generate the test cases. With model based testing, it helps us to identify the faults at the early stages of the software development life cycle there by reducing the cost and time involved in the system development. Model is a graphical representation of the system functionality. In this paper, UML Activity Diagram is used as the design model to generate the test cases. UML is a Unified Modelling Language used to model the systems. UML has different notations and symbols used for representing different system functions and components.

The main purpose of this work is to analyze and automatically generate the effective set of test cases from a given activity diagram. A XMI parser is used to process the XML file being obtained from the UML activity diagram and maps it to its corresponding adjacency list. Then the test scenarios are generated by applying test data generation technique or by list traversal using AI The existing approaches provide a semi-automated way of test case generation from an UML activity diagram. That is, the given diagram is translated to some intermediate representation such as activity dependency graph, data flow graph, control flow graph or the activity dependency table. This intermediate representation is then given as input to the further phases. Even though it seems to be fully automated, the procedure for converting the diagram to its internal representation has not been addressed clearly in any of the existing approaches. The output of the system is optimized set of test suites and test cases. Without optimization the system would have generated all possible test suites out of which there could be even redundant or irrelevant test suites. Some test suites will have higher priority than others. And in case of Fork-Join activities the order of interleaving has to be maintained. Hence by taking care of these concepts, a Fork-Join coverage criterion has been applied to maintain the order of concurrent activities. The redundant paths are optimized using a Weighted Prioritized and Fork-Join coverage technique. 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 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 . 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 . 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 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 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 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 . *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 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 diagram*s 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 diagram*s, 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 diagram*s 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 messagesequence 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.

**BACKGROUND**

Activity Diagram is one of the behavioural diagrams in UML which describes the flow of activities in the system. Each activity can be further subdivided into smaller activities or it can be an atomic activity. The syntax of Activity Diagram is

very similar to that of flowcharts. Activity diagram mainly describes the way in which each functionality is executed in the system. It never explains “what the system does” rather explains “how the system does”. Activity Diagram uses variety of symbols to represent different scenarios. Every diagram will have one initial state and can have more than one final states. The start state in an Activity Diagram is denoted using a filled circle. Start state of an activity diagram will have no incoming edges. Activities are denoted using lozenge shape. If-Else scenario can be easily modelled by considering a decision construct which contains branches to represent true/false conditions. Diamond shaped symbol is used for decision making. UML also provides a mechanism to model concurrent activities of the system. When an activity needs to be run concurrently, we use Fork construct to model it. Fork is represented using horizontal thick line. Fork node contains one incoming edge which is split into multiple outgoing edges. The activities which are forked will execute concurrently. All the activities that are forked are joined later using join node. Join node is always used along with a Fork node. For an activity to occur in parallel it has to first fork and later once all the parallel activities are executed, they need to be joined together before starting with the next activity. End state of Activity Diagram is denoted using filled circle with an outer circle. End state will have no outgoing edge.

Fig. 1 shows a sample example of UAD consisting of seven activities. The various activities involved are A1, A2, A3, A4, A5, A6 and A7. Every Activity diagram contains only one start state and any number of end states Start state in the Activity Diagram is denoted using filled circle. End state is shown using a filled circle with an outer circle .In the above example, after completion of activity A1, there is a fork node represented using thick horizontal line. The activities following fork node namely A2, A3, A4, A5 and A6 execute concurrently. The order of execution needs to be maintained among forked activities i.e. an activity can execute only when the execution of previous activity is complete. The If-Else construct is modelled using diamond symbol. Every decision node in an Activity diagram will have one incoming edge and can have two or more number of leaving edges. The decision symbol in Fig. 1 has two leaving transitions and each of them has a label referred as the guard expression In Fig. 1 after executing the activity ‘A2’, if the condition of the branch node is evaluated to true, then ‘A5’ will be executed next, else the control moves on to activity ‘A4’. Merge nodes are also depicted in the same way as decision nodes but it acts as a source node to combine multiple flows. As per the example, activity A7 can begin only when all activities which were forked previously has completed their execution. A transition which is depicted as directed edge, describes the order of execution or control-flow between two activities. Consider two activities A and B, if there is a transition from A to B it means, Activity A has to be completed before Activity B starts.

A *sequence diagram* shows system events for a use case. It is said to implement a use case A *sequence diagram* shows the messages that are exchanged among several objects, as well as certain controlflow 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*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 . 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.

****

**Figure 1.** Sample Activity Diagram

**METHODOLOGY**

Fig. 2, shows an overview of the proposed methodology. In the first step, activity diagram is designed using standard UML tool called StarUML. StarUML provides a workspace for the user to model different views of the system using Unified Modelling Language. It is not possible to perform any operations using this diagram, until and unless we have its internal representation. Hence the Activity diagram needs to be pre-processed. To achieve this, in the next step the diagram is exported to its corresponding XMI representation using StarUML. UML Activity diagram in its XML format consists of various XML tags and its attributes which are essential to produce the desired result. XMI file contains details about each of the activity and transitions in the activity diagram. Once the relevant details are obtained, the next step is to store this information into intermediate data structure (i.e. Adjacency list) for further processing. Adjacency list is traversed using AI techniques to obtain the optimized set of test scenarios. In the last step, test cases are generated from each of the scenario.

Several UML tools are available to model UML diagrams. Some of the widely used tools are MagicDraw, RationalRose, StarUML etc. MagicDraw is a good suite for UML modelling. The tool manages a true model representation which it stores as one large XML flat file. The main reason for not using MagicDraw in the proposed approach is, the XML file generated by the tool for the diagram is very large and takes longer time for processing. RationalRose is an object-oriented Unified Modelling Language (UML) software design tool intended for visual modelling and component construction of enterprise-level software applications. Rational Rose allows designers to take advantage of iterative development because the new application can be created in stages with the output of one iteration becoming the input to the next. The RationalRose tool doesn’t support option for exporting the diagram to its XMI/XML representation. Hence further processing of the diagram cannot be done using RationalRose. Both the tools mentioned above are proprietary software. StarUML is an open source tool which overcomes the above mentioned limitations and hence is preferred for the proposed approach.

**Proposed methodology consists of following steps:**

Step 1: Input to the system is an Activity diagram, which is modelled using StarUML tool. The Activity diagram considered for this project supports any number of connected activities, branching constructs and set of concurrent activities. By default, the diagram modelled using StarUML is saved in its .mdl format.

Step 2: Activity diagram in its .mdl format is converted to its XMI representation. XMI is a standard used to store UML models.

Step 3: The XMI file obtained in Step 2 is parsed to extract the required data. At this stage an algorithm is designed to parse the XMI file. Algorithm 1 explains the method used to extract the data from XMI file.



**Figure 2.** Overview of the methodology

**Proposed methodology consists of following steps:**

Step 1: Input to the system is an Activity diagram, which is modelled using StarUML tool. The Activity diagram considered for this project supports any number of connected activities, branching constructs and set of concurrent activities. By default, the diagram modelled using StarUML is saved in its .mdl format.

Step 2: Activity diagram in its .mdl format is converted to its XMI representation. XMI is a standard used to store UML models.

Step 3: The XMI file obtained in Step 2 is parsed to extract the required data. At this stage an algorithm is designed to parse the XMI file. Algorithm 1 explains the method used to extract the data from XMI file.

*Step 4*: The XMI generated in Step 3 contains information about States and Transitions of Activity Diagram. The above algorithm makes use of two separate functions namely, parseXMLStates and parseXMLTransitions to parse states and its respective transitions. If the Activity diagram is modelled using Swimlanes, then the swimlane details will be enclosed within ActivityGraph.Partition element. The Activity state information is present within the CompositeState.subvertex element. Hence the file is traversed to get the descendants of CompositeState.subvertex element. In the first step, for each of the elements in CompositeState.subvertex, a state object is created, where all the attributes of state are initialized. For each of the element extract its id, name and kind. In the next step, the incoming transition is checked.

one incoming transition in the Activity diagram then it is separated by a space in the XMI file which is later split based on space and stored in an array. The same procedure is then repeated for outgoing transitions. More than one outgoing transitions are split based on space and is stored in another separate array. In the third step, all the states of Activity diagram are added to the stateList. Once all the states of the Activity diagram have been parsed, next step is to parse the transitions of the diagram. Transition details are contained in UML:StateMachine.transition.

*Step 5:* Data retrieved in step 4 is populated into a data structure for further processing. For convenience, Adjacency matrix is of size NXN is used; where ‘N’ refers to number of activities in an Activity diagram. The entry A [i,j] in Adjacency matrix is set to transition name when there is a transition from Activity i to Activity j. The main reason for choosing adjacency list as a data structure is- each node in the list can be easily traversed to generate the set of test suites. Or every element in the above mentioned tag, a transition object is created. If the transition has any guard expression then the guard expressions attributes needs to be extracted. Once all the transitions of Activity diagram has been parsed, the transitions are added to transList.

Consider the Activity diagram in Fig. 3. The adjacency list representation for the same is shown in Fig. 4



**Figure 3.** Example Activity Diagram



**Figure 4.** Adjacency list representation

*Step 5:* Once the data has been populated in Adjacency list, the list is traversed using AI technique called as Best First Search method (BFS). The BFS method is not used as it is; instead it is modified as per the need of the project. Best-first search in its most general form is a simple heuristic search algorithm. “Heuristic” here refers to a general problem-solving rule or set of rules that do not guarantee the best solution or even any solution, but serves as a useful guide for problem-solving. Best-first search is a graph-based search algorithm meaning that the search space can be represented as a series of nodes connected by paths.

The main objective here is to traverse the list and generate all possible paths from a Start activity node to a Final Activity node. The modified heuristic approach makes use of two lists namely Open List and Closed List. Open list is a list which contains the path without the Final node in it. Closed list is a list which contains the path with Final state. Every time a node is visited, it is first en-queued to the open path list. The last node in the list is then checked to see it it’s a final node. If it is a final node, then the whole path will be dequeued and will be added to Closed Path list. In other words, every closed path has been an open path in its lifetime. The same is explained using a pseudocode in Algorithm 2.

The Automated Teller Machine will serve one customer at a time. A customer needs to first insert an ATM card and enter the four digit PIN code. Once the PIN is entered, the system will verify if the entered PIN is valid or invalid. If it is valid one, then it will proceed to the next step. If the entered PIN is an invalid one, then it will allow the user to enter the PIN again. For the sake of simplicity, the number of trails on an invalid input is set to one. (i.e. if the entered input is invalid, user will be given only once chance to re-enter the data). The ATM Management System supports. Once the user entered PIN is valid, in the next step system will ask the user to enter the choice of option. User can select one among the following choices:

• Withdrawal

• Balance Inquiry

• Mini Statement of the previous transactions

When the user clicks on Withdrawal Option, the system will prompt the user to enter the amount to be withdrawn. Once the user enters, the amount entered will be verified to check if it is less than 500. If yes, the system will give a message saying insufficient balance and then asks the user to re-enter amount. Later, the amount value will be deducted from users account and cash will be dispensed to the user along with the receipt. When the user clicks on Balance Enquiry, the system will ask the user to confirm if he needs a printed receipt or no. If the user selects yes, then the system will output the balance details on a receipt and exits. If user selects no, then the system will output balance details on the screen and then exits. Mini Statement feature helps the user to know his past ten transaction details prior to current transaction. The Use case diagram for above mentioned case study is shown in Fig. 6 and Activity diagram for *Withdraw Money* use case is shown in Fig. 7.

Use Case diagram is one of the behavioral diagrams in UML. It is used to represent various users of the system and its various functionalities. In this ATM management system, there are two users namely Customer and Admin. Customer can withdraw money, enquire the balance or get the previous transaction details. Admin is the one who updates the ATM machine by periodically filling cash into it. The proposed

methodology is tested on two functionalities namely Withdraw Money and Enquire Balance of the system. As a first step, Activity diagram is modelled for these two functionalities.



**Figure 6.** Use Case Diagram for ATM case study



**Figure 7.** Activity Diagram for *Withdraw Money* use case.

Software Testing is a time consuming and costly process in software development life cycle. Automation of this phase may lead to overcome the above problems and also reduces the human effort in other ways it also helps in detecting the human intended errors and logical errors as well. Automation of testing will not be that much productive in terms of time consuming and cost, if we have to wait till the end of the SDLC stage i.e. if we follow the white box testing methodology of testing. If any errors will be detected in this stage, we have to go for that part of the code and design document as well. We have to follow up strict verification of both code and design document from beginning to short out the error. So only one solution to this problem is to, start the testing process from early stage of SDLC i.e. from requirement specification stage through design phase up to the last phase.

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. *4. Category Partition Method*

The category-partition method 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.