Chapter 4

Test Case Generation from UML Activity Diagram

The structural models of UML have traditionally been used for testing of object-oriented programs at the class level. On the other hand, the behavioral models of UML, have been recognized to be useful to test the object-oriented software at cluster and system level. The behavior of a use case can be represented by using activity, sequence, collaboration and state chart diagrams. An activity diagram is used to represent the scenarios related to a use case and used by stakeholders of the system to understand its functionality [64, 68, 75]. A path of activity diagram represents the set of activities from the start node to the end node. The path contains condition, fork-join, branch-merge nodes to show the control and procedural flow of activities. As an UML activity diagram captures the key system behavior, it is well suited for the cluster and system level testing of object-oriented software.

In this thesis, we have considered the synthesis of test sequences from UML activity diagrams for cluster level testing. An activity diagram is used to represent all possible flows of executions in a use case. As activity diagram describes the flow of activities inside the programs, so it is helpful to test the software in cluster level. Faults associated with activity diagram can be captured by the activity path coverage criterion. Possibly, UML activity diagram is the only design artifact which is good at describing the flow of control in an object-oriented system. Hence, activity diagrams are treated as one of the most important design artifacts among several UML models. An example of an activity diagram for online claim of refund is shown in Figure 4.1.
In this chapter, we propose a model to generate an Activity Flow Table (AFT) from an Activity Diagram and then convert it to Activity Flow Graph (AFG). By using activity coverage criterion, we traverse the AFG and the test paths are generated. From the graph different control flow sequences are identified by traversing the AFG by a depth first traversal technique which guarantees that all possible nodes of the AFG are visited during traversal. Next, an algorithm is proposed to generate all activity paths. Finally, test cases are generated using activity path
coverage criteria. The model is implemented on a case study of ATM Withdrawal system.

We start the chapter by presenting an overview of the activity diagram and introducing some basic concepts and definitions in Section 4.1. Section 4.2 discusses the test coverage criteria. Section 4.3 describes the proposed overall methodology to generate test cases from an activity diagram and our proposed algorithm. Section 4.4 shows one case study to illustrate the working of the proposed algorithm. Section 4.5 explains the experimental results. In Section 4.6, we provide a comparison with the existing related work. Section 4.7 concludes the chapter.

4.1 Basic concepts and Definitions

In this section, we elaborate the basic concepts and definitions which will be used in this chapter for generation of the test cases. After that, we introduce some notations and definitions that will be used in our proposed approach.

4.1.1 Activity Diagram

A UML activity diagram is used to model the flow of a particular use case. Activity diagrams describe the ordering of behavior, called activities. An activity diagram can be used to model complex processes that have parallelism, loops and event driven behavior. An activity diagram consists of a set of nodes and edges. The nodes represent the processes or process control, including action states, activity states, decisions, swimlanes, forks, joins, objects, signal senders and receivers. The edges represent sequences of activities including the control flows, message flows and signal flows. Activity state and action state are denoted with round corner boxes. All transitions are shown as arrows. Branches are shown as diamonds with one incoming arrow and multiple exit arrows. Each arrow may be labeled with a boolean expression to be satisfied to choose the branch. Forks and joins are represented by multiple arrows entering or leaving a synchronization bar. In a wait state, the system waits for the occurrence of an event, e.g. some deadline occurs or a customer sends some additional information. A wait state is also used for synchronization of a thread with other parallel threads. In the wait state, the system waits for the completion of the other parallel threads. In a compound activity state, a different activity diagram can be executed. This different activity diagram is started when the compound state is entered. When the activity diagram finishes,
the compound activity state is exited.

An edge starts from the source node and terminates at the destination. An edge is labeled with an expression of the form $e[c]/a$, where $e$ is an event, $c$ is a guard condition, and $a$ is an action. Each of these three components is optional. An edge with label $e[c]/a$ has the following meaning: If the system is in the source state, the event $e$ occurs, and the guard condition $c$ evaluates to true, then the system transits out of the source state, the action $a$ is performed, and the destination state is entered. It describes the flow of activities in different objects.

Activity diagram shows how objects behave or how they collaborate. Diagrams are read from top to bottom and have branches and forks to describe the condition and parallel activities. The edges represent sequences of activities including the control flows and message flows. The flows may be sequential, branched or concurrent. The flows are controlled by forks and joins.

The different activities/actions are denoted by rounded corners. Flow/transitions are represented by arrowheads. The decision is shown as a diamond shaped box with one incoming flow and multiple labeled outgoing flows. Forks are denoted by a synchronization bar with one incoming flow and multiple outgoing flows. Similarly, join is denoted by multiple incoming flows entering to a synchronization bar with one outgoing flow. Table 2.2 describes the control nodes and their actions used in the activity diagram.

**Definition 4.1** An activity diagram is a nine-tuple $AD = (A, B, C, F, J, M, T, a_0, a_f)$, where

- $A = \{a_1, a_2, \ldots, a_n\}$ is a finite set of activities.
- $B = \{b_1, b_2, \ldots, b_n\}$ is a finite set of branches.
- $C = \{c_1, c_2, \ldots, c_n\}$ is a finite set of guard conditions, and $c_i$ is in correspondence with $t_i$, $\text{Cond}$ is a mapping from $t_i$ to $c_i$ so that $\text{Cond}(t_i) = c_i$.
- $F = \{f_1, f_2, \ldots, f_n\}$ is a finite set of forks.
- $J = \{j_1, j_2, \ldots, j_n\}$ is a finite set of joins.
- $M = \{m_1, m_2, \ldots, m_n\}$ is a finite set of merges.
- $T = \{t_1, t_2, \ldots, t_n\}$ is a finite set of completed transitions.
4.2 Test Coverage Criteria

- \( a_0 \in A \) is the unique initial activity, and \( a_f \in A \) is the final activity.

**Definition 4.2** A Test scenario, \( ts \in TS \), in an activity diagram \( AD \), can be defined as a path from the initial activity to the final activity consisting of all activities and transitions along the path, i.e. \( \forall ts \in TS \), the test scenario can be written as
\[
    ts = a_0 \rightarrow t_0 \rightarrow a_1 \rightarrow t_1 \rightarrow \ldots \rightarrow t_n \rightarrow a_m
\]
where \( a_i \in A \), \( t_i \in T \), \( a_0 \) is the initial state, \( a_m \) is the final state and \( TS \) is the set of test scenarios.

4.1.2 Fault Model

Different methodologies used by researchers to target different types of faults are called the fault models [12]. By using the following fault models, we generate the test cases.

- Fault on loop: This type of fault generally occurs while control enters into the loop, exiting from the loop or at the time of increment /decrement.

- Fault in the decision: This type of fault occurs when a decision is to be taken in an activity diagram. Due to this fault, wrong / invalid outputs are produced for valid decisions.

- Fault while synchronization: This type of fault occurs when activity starts executing prior to the completion of execution of group of all preceded activities.

4.2 Test Coverage Criteria

A coverage criterion is a set of rules that impose test requirements on a test set. Test coverage criteria are used to decide which test inputs to be used as we cannot test with all inputs. It is a set of rules that decides appropriate elements to be covered to make test case design adequate. The test developers will easily find the faults by effectively using the coverage criteria. Test coverage analysis helps to determine the thoroughness of testing achieved. Major test coverage criteria used today are graph coverage, which includes structural coverage, data flow coverage, source code coverage, design elements coverage, specifications coverage, use case and logic coverage [7]. As we are confined to the activity diagram in this chapter, the test coverage criteria used are i) All Basic Path Coverage Criterion ii)Simple
Path Coverage Criterion iii) All Activity Path Coverage Criterion and iv) Transition Coverage Criterion. Below, we discuss each of these coverage criteria in brief.

4.2.1 All Basic Path Coverage Criterion

A basic path is a sequence of activities where an activity in that path occurs exactly once [20, 26]. Note that a basic path considers a loop to be executed at most once. Given a set of basic paths $BP$ obtained from an activity graph and a set of test cases $TC$, for each basic path $bp_i \in BP$, there must be at least one test case $tc \in TC$ such that when the system is executed with the test case $tc$, $bp_i$ is exercised.

A basic path is a complete path through an activity diagram where each loop is exercised either zero or once.

4.2.2 Simple Path Coverage Criterion

A simple path is considered for activity diagrams that may contain concurrent activities [1]. It is a representative path from a set of basic paths where each basic path has the same set of activities, and activities of each basic path satisfy an identical set of partial ordering relations among them. Note that the partial ordering relation between two activities $A_i$ and $A_j$, denoted as $A_i < A_j$ signifies that activity $A_i$ has occurred prior to activity $A_j$.

4.2.3 All Activity Path Coverage Criterion

Given a 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 to the terminal activity in the activity diagram. Given a set of activity paths $PA$ for an activity graph and a set of test cases $T$, for each activity path $p_i \in PA$ there must be a test case $t \in T$ such that when the system is executed with a test case $t$, $p_i$ is exercised.

4.2.4 Transition Coverage Criterion

It requires that all transitions in the activity diagram be covered. For any test case $t \in T$, we can get the program execution trace (PET). If for the PET, there exists a corresponding transition that is not marked in the activity diagram, we mark all the corresponding unmarked transitions for PET and record the test case $t$. The value of transition coverage is the ratio of the marked transitions to all transitions in the activity diagram.
4.3 Proposed Framework

In this section, we present our work to generate test cases from the specification document using a UML activity diagram. UML model is used as a de-facto standard in both industry and in academics [16]. It is widely used as a modeling language and is designed to specify, construct and document the software artifacts with support to special aspects of software such as dynamic and structural aspects [112]. Figure 4.2 represents the proposed framework of the system.

The proposed test case generation approach consists of the following steps.

- Step 1: Construct the Activity Diagram (AD) of the system.
- Step 2: Construct the Activity Flow Table (AFT) from AD.
- Step 3: Generate a graph called Activity Flow Graph (AFG) \( G = (V, E) \), where \( G \) represents the graph, \( V \) represents the set of nodes (activities) and \( E \) represents the set of connecting edges (workflow) using AFT.

Figure 4.2: Proposed Framework
• Step 4: Generate the possible test paths using Depth First Search (DFS) traversal that guarantees visiting all the possible nodes of the AFG, while removing the redundant ones. The generated test paths cover all the branches and conditions. Apply dominance to generate the test cases.

• Step 5: Generate test cases.

• Step 6: Print the test cases, conditions, activity coverage and errors.

Now, we elaborate the details of the steps in the following section.

4.3.1 Construct the activity diagram of the system

Here, we describe the way of modeling the necessary test information of a system into an activity diagram. First of all, we create the use cases of the system. Each use case of the system can be represented by one or more activity diagrams. Activity diagram represents the scenarios related to a use case. By using different basic nodes, we construct the activity diagram of the system under test.

4.3.2 Construct the activity flow table

In this subsection, we construct the Activity Flow Table (AFT) for the activity diagram. The AFT describes the symbols by numbers given to each activity. The table has two columns, namely, Activity Name which describes the name of the activity performed and Symbol Name i.e. the symbol given for the activity. Here, we have used numerals to describe the activities.

4.3.3 Construct the activity flow graph

In this subsection, we try to convert the AFT to an Activity Flow Graph (AFG) using the symbols mentioned in AFT. An activity flow graph is a directed graph while its construct is represented by each node in the AD (initial node, decision node, flow final node, guard condition, join node, fork node, etc.), and from that each border of the activity flow graph symbolizes the stream in the activity diagram.

4.3.4 Generate the test paths using dominance

Different control flow sequences have been identified by traversing the AFG using a depth first traversal algorithm. We generate test paths following the activity path coverage criterion. To do this, we obtain all activity paths from start node of type
4.3 Proposed Framework

Start to the end node of type End in the activity flow graph.

To generate test paths that satisfy the activity path criterion, we first enumerate all possible paths from the start node to an end node in the AFG. During the visit, we look for conditional predicates on each of the transitions for execution of corresponding flow and activity. For each conditional predicate, based on the activity path coverage criteria and the guard condition, we generate the test paths.

**Algorithm 1 ActivityTestPathGeneration**

**Input:** Activity Flow Graph (AFG)  \(\triangleright\) Input the AFG

**Output:** Test Paths (TP)  \(\triangleright\) The list of test paths

1: \(TP_i[] \leftarrow \phi\)
2: \(node_j[] \leftarrow \phi\)
3: \(condition.node_k[] \leftarrow \phi\)
4: \(TP_i[] \leftarrow AFG.start.node\)
5: if \((AFG.start.node = AFG.node)\) then
6:   while \((AFG.node \neq AFG.end.node)\) do
7:     \(TP_i[] = AFG.start.node_j[]\)
8:     if \((AFG.node = condition.node_k[])\) then
9:       if \((condition.node_k[] = valid)\) then
10:          \(TP_i[] = condition.node_k[]\)
11:     end if
12:     continue
13:   end if
14:   if \(TP_i[] = AFG.node\) then
15:     exit(0)
16: end if
17: end while
18: end if
19: Display \(TP_i[]\)
20: Stop

We propose an algorithm called *ActivityTestPathGeneration*, shown in Algorithm 1 to generate all the possible activity test paths. In this algorithm, we use the path enumeration algorithms. We traverse the activity graph by depth-first search. In case of no guard condition, NULL is used. The algorithm *ActivityTest-
PathGeneration starts by enumerating all basic paths in the AFG, from the Start node to the End node. Each basic path is then visited to generate the test paths. A basic path essentially corresponds to a scenario.

Initially all test paths (TP) are empty. A test path contains many intermediate nodes. The algorithm runs with the start node. When the first intermediate node is reached, it will be marked as visited and will be added to path $i$, where $i=1$ to $n$ (number of paths). After visiting one node in $i^{th}$ path, the next connecting node will be visited. Again this node will be marked and added to $i^{th}$ test path. This process will be repeated till the pointer reaches the end node. If there will be a condition node, two separate paths will be created by copying the previously visited nodes along the path to each test path and all other nodes in the truth part of the condition and the nodes in the false part till the end node. This process will be repeated till each node in AFG is marked.

**Proof of correctness of ActivityTestPathGeneration algorithm:** Here, we sketch the proof of correctness and working of our proposed algorithm.

**Theorem 4.1.** ActivityTestPathGeneration algorithm correctly generates the test paths.

**Proof:** Activity flow graph (AFG) is a graph, having $n$ number of paths. We assume that all statements work as expected and that comparisons such as (AFG.node $\neq$ AFG.end.node) are appropriately carried out. Initially $i=1$, where $i$ varies from 1 to $n$ (where $n$ is the number of paths in the AFG), all nodes are unmarked and TP is empty. Every time one node will be read and checked for the end node. If it is an end node then it will continue for the next path till all nodes are marked. When a node is visited and it is unmarked then it will be added to TP. The algorithm terminates successfully, if all the nodes are marked. Since the number of nodes in the given graph is always finite, the loop is exited and hence the algorithm terminates. Then, it will display the test path (TP). Similarly all possible activity paths will be generated. Our algorithm finitely terminates. So, our algorithm correctly generates the test paths. $\square$

### 4.3.5 Generate the test cases from the test paths

After the generation of the test paths, we apply dominance concept to generate test cases. We have defined our test cases as $<input>, <transition1>, \ldots, <$
4.4 Case Study

In this section, we explain the working of our approach with the help of a case study of the ATM withdrawal system. First, we provide an overview of the problem and the corresponding activity diagram. Then, we describe the process of the test path and test case generation for this system using UML activity diagram.

4.4.1 The model of the case study using an Activity Diagram

In our approach, we considered the Activity Diagram for ATM withdrawal system. We use an activity diagram due to the dynamic behavior of its modeling. The dynamic aspects of the object can be constructed, visualised, specified and documented by the activity diagram. An activity diagram models the behavior by specifying the sequence of actions and the conditions for co-ordinating actions. Figure 4.3 represents the Activity Diagram for the ATM withdrawal system.

In this case study of ATM system, the customer can withdraw, deposit, check the balance, transfer the funds, change the PIN and also print the mini statement. In our approach, we are confined to the cash withdrawal only. When a customer approaches the ATM counter, an ATM machine generally shows the welcome message and the initial state is idle. The complete system will perform as per the followings.

Precondition: ATM displays an idle welcome message on the screen. The customer uses a valid card to withdraw the funds from an ATM.

Description:

1. ATM card is inserted by the customer into the ATM machine.

2. If the system can recognize the card, it will read the card number.

3. Immediately the system prompts the customer to insert the PIN.
4. PIN is entered by the customer.

5. The system checks for the authentication, i.e. whether the PIN entered by the customer matches the card PIN or not?

6. If the PIN matches, then the system checks out what accounts the card can access.

7. System displays customer accounts and waits for the customer to choose a type of transaction he/she wants, out of the three types of transaction (Withdrawal of Funds, Get Balance and Transfer Funds).

8. Customer chooses Withdrawal of funds and enters the amount.
9. System checks that enough fund is available in the account, daily limit has not been exceeded and the ATM has enough funds.

10. If all the three conditions are satisfied, then the system dispenses cash.

11. System prints the receipt.

12. The system ejects the card.

13. The transaction is successful.

14. The system again comes to its idle welcome message state.

4.4.2 Activity Flow Table (AFT) for the Activity Diagram

We construct the Activity Flow Table (AFT) for the above activity diagram. The AFT describes the symbols by numbers given to each activity. The table has two columns, namely, Activity Name which describes the name of the activity performed and Symbol Name i.e. the symbol given for the activity. Here, we have taken numerals. Table 4.1 shows the AFT corresponding to the activity diagram of the ATM withdrawal System.

4.4.3 Activity Flow Graph (AFG) using the AFT

We convert the AFT to an Activity Flow Graph using the symbols mentioned in the AFT shown in Table 4.1. An Activity Flow Graph is a directed graph while its construct is represented by each node in the activity diagram (initial node, decision node, flow final node, guard condition, join node, fork node, etc.), and from that each border of the activity flow graph symbolizes the stream in the activity diagram. Figure 4.4 represents the Activity Flow Graph for the ATM withdrawal system. By using the concept of a control flow graph, we construct the AFG from AFT.

4.4.4 Generation of Test Paths

All possible test paths are generated before the generation of the test cases from the AFG. Our approach traverses the graph using Depth First Search (DFS) method which guarantees visiting all possible nodes of AFG. The dominance tree of the AFG is given in the Figure 4.5.

All possible test paths generated are as follows:
Test Path 1 : 1 → 2 → 3 → 2 → 14 → 15 → 16
Table 4.1: The AFT table of an ATM withdrawing Activity Diagram

<table>
<thead>
<tr>
<th>Sl. No.</th>
<th>Activity Name</th>
<th>Symbol Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Insert Card</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Check Pin</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Authenticate</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Enter Amount</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>Check balance</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>Decision 1</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>Update Account</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>Fork 1</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>Receive Cash</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>Take Receipt</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>Join 1</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>Merge 1</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>Show Balance</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>Merge 2</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>Eject Card</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>Stop</td>
<td>16</td>
</tr>
</tbody>
</table>

Test Path 2 : 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 11 → 12 → 13 → 14 → 15 → 16  
Test Path 3 : 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 10 → 11 → 12 → 13 → 14 → 15 → 16  
Test Path 4 : 1 → 2 → 3 → 14 → 15 → 16  
Test Path 5 : 1 → 2 → 3 → 4 → 5 → 6 → 12 → 13 → 14 → 15 → 16

By using the dominance tree, the dominator leaf nodes are generated. Now the leaf nodes age given by

\[
\text{Dom (14)} = 1, 2, 3, 2, 3, 14 \\
\text{Dom (11)} = 1, 2, 3, 4, 5, 6, 7, 8, 9, 11 \\
\text{Dom (12)} = 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12 \\
\text{Dom (13)} = 1, 2, 3, 4, 5, 6, 12, 13 \\
\text{Dom (16)} = 1, 2, 3, 14, 15, 16
\]
4.4.5 Generation of Test Cases

After the creation of all the test paths, we develop the test cases from the paths. For the generated test paths, we generate test cases on the basis of covering the activities. Our test cases achieved all activity path coverage criterion. The test cases are shown in Table 4.2. Nine different test cases are generated, whose ids are shown in Column 1. The PIN to be entered is shown in Column 2, the amount to be entered by the user is shown in Column 3. Balance in the account and ATM are dealt with the bank and are shown in Column 4 and Column 5 respectively. The expected result and actual results are shown in Column 6 and Column 7 respectively. The test cases generated are described below.

Test Case 1 {Input: Invalid PIN, Output: Return ATM card with Invalid Pin message}
Test Case 2 {Input: Valid PIN, Valid Amount, Output: Dispense Cash, Update Account, Print Receipt}
Test Case 3 {Input: Valid PIN, Valid Amount, Output: Return card with failure message}
Test Case 4 {Input: Valid PIN, Valid Amount, Output: Return card with failure
message 'Insufficient Funds'}
Test Case 5 {Input: Valid PIN, Invalid Amount, Output: Enter Amount } 
Test Case 6 {Input: Valid PIN, Valid Amount, Output: Error, Unable to Dispense } 
Test Case 7 {Input: Invalid PIN, Output: Return ATM card with Invalid Pin message} 
Test Case 8 {Input: Valid PIN, Valid Amount, Output: Dispense Cash, Update Account, Print Receipt } 
Test Case 9 {Input: Invalid Card, Output: Invalid card }
4.5 Implementation and Results

In this Section, we discuss the results obtained by implementing our proposed approach. We have implemented the complete approach using Rational Software Architect (RSA) and JAVA Swing. We have implemented our approach by taking

Table 4.2: The observed test cases

<table>
<thead>
<tr>
<th>Test case id</th>
<th>Pin No.</th>
<th>Amount Entered</th>
<th>Amount in Account</th>
<th>Expected Result</th>
<th>Actual Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>2104</td>
<td>NA</td>
<td>50000</td>
<td>Invalid Pin / Unsuccessful</td>
<td>Invalid Pin / Unsuccessful / Welcome message</td>
</tr>
<tr>
<td>TC2</td>
<td>2014</td>
<td>5000</td>
<td>15000</td>
<td>Success / Dispense Cash /Print Receipt /Update Account</td>
<td>Success / Dispense Cash /Print Receipt /Update Account</td>
</tr>
<tr>
<td>TC3</td>
<td>2014</td>
<td>3000</td>
<td>20000</td>
<td>Warning Message Show Balance</td>
<td>Show Balance</td>
</tr>
<tr>
<td>TC4</td>
<td>2014</td>
<td>2000</td>
<td>1000</td>
<td>Insufficient Funds Show Balance</td>
<td>Unsuccessful / Show Balance</td>
</tr>
<tr>
<td>TC5</td>
<td>2014</td>
<td>1050</td>
<td>5000</td>
<td>Enter Amount</td>
<td>Enter Amount in Multiples of 100</td>
</tr>
<tr>
<td>TC6</td>
<td>2014</td>
<td>2000</td>
<td>5500</td>
<td>Success / Print Receipt / Dispense Cash</td>
<td>Machine Error</td>
</tr>
<tr>
<td>TC7</td>
<td>4532</td>
<td>na</td>
<td>9000</td>
<td>Invalid Pin</td>
<td>Invalid Pin / Unsuccessful / Welcome message</td>
</tr>
<tr>
<td>TC8</td>
<td>2343</td>
<td>6000</td>
<td>85000</td>
<td>Success/Dispense Cash/ Print Receipt / Update Account</td>
<td>Success /Dispense Cash/ Print Receipt / Update Account</td>
</tr>
<tr>
<td>TC9</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>Invalid Card</td>
<td>Invalid Card</td>
</tr>
</tbody>
</table>
ATM (Automated Teller Machine) as the case study. After constructing the AD of ATM withdrawal system in RSA, the XMI code is generated whose snapshot is given in Figure 4.6. On execution of these XMI codes, we obtain the coverage and test paths (sequence) as output, which is shown in Figure 4.7. We conducted experiments based on the model and as per the algorithm proposed. Table 4.3 lists the faults detected in test sequences which are obtained from five activity paths. Here, test cases 1, 4, 5 cover fault in loop (minimal loop testing), whereas test sequences 2, 3 cover fault in decision.

![Figure 4.6: Snapshot of the XMI code generated from the activity diagram](image)

We also applied our proposed approach to other four different case studies from the real-life situations and observed the different types of faults detected in the systems. Table 4.3 shows the types of faults detected in the different case studies. Figure 4.8 shows the graph of the types of faults and its number in different case studies.

### 4.6 Comparison with related work

Many researchers used UML state chart diagrams for testing object-oriented software [10, 55, 71]. In our research work, we used activity diagrams and targeted for cluster level testing involving interaction among objects. Moreover, the same UML
4.6 Comparison with related work

Figure 4.7: Snapshot of Activity Flow Table and Test Paths

diagrams were used for the analysis and design, without requiring any additional formalism or effort specifically made for testing the software. Offut et al. [81] presented an idea of using state chart diagrams, to generate test cases at unit level. Abdurazik et al. [3] used collaboration diagram for generating test cases. On the other hand, our work derives activity flow graph which is an important part of the test plan. By using activity flow graph, we have studied the detailed plan and derived the test cases.

On the contrary, many of the UML related testing work are reported in which the researchers used manual methods to generate the test data and test cases [3, 68]. At the same time, many were successful up to the extent of automatically generating valid test requirements. Their test requirements [16] are specific things like possible execution sequences of use cases, messages, transitions, etc., that must be satisfied or covered during testing. We considered these test requirements as a part of a test
Table 4.3: Types of fault detected in different case studies

<table>
<thead>
<tr>
<th>Sl. No.</th>
<th>Name of the case study</th>
<th>Number of Classes</th>
<th>Number of test cases</th>
<th>Fault in Loop</th>
<th>Fault in decision</th>
<th>Synchronization Fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ATM Withdraw System (ATMWS)</td>
<td>7</td>
<td>26</td>
<td>5</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Book Issue of Library Information System (BILIS)</td>
<td>12</td>
<td>63</td>
<td>8</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>User Login System (ULS)</td>
<td>6</td>
<td>24</td>
<td>3</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Online Shopping System (OSS)</td>
<td>18</td>
<td>86</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Online Ticket Reservation System (OTRS)</td>
<td>11</td>
<td>47</td>
<td>3</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

case. 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, which we have considered in our work.

In comparison to the related work [71, 89, 115], the advantage of our work is that it explores interactions, conditions, asynchronous activities and concurrent paths. An activity diagram shows the data and control flow of activities in a system. It represents the detail logic of procedurally complex operations. Several faults such as incorrect response to an activity, improper method invocation with incorrect arguments, incorrect sequencing of activities, inappropriate control flows and missing flows, etc. may occur in an interaction [68]. We follow the activity path coverage criterion to derive the test cases. In our approach, no redundant test cases are generated. A number of test cases are generated in our approach which can test more meta conditions used at the integration level.

Mingsong et al. [75] proposed an approach for generating test cases from activity diagrams. The authors used basic paths for handling the loops. Based on a partial order relation they sequence the activities of an activity diagram. They
define a simple path which selects a representative path for handling concurrency. These simple paths were then matched with the execution trace of the corresponding program by mapping functions to activities. In this way, they found out coverage of simple paths in an activity diagram. But, in our work, we achieve activity coverage, basic path coverage, etc. directly from the graph obtained from the activity diagram. In this way, we obtain better coverage and less time complexity.

In contrast, with the approaches [68, 75], we generate test cases semi-automatically from activity diagrams. State based faults are difficult to be detected from the software code. State chart diagram and state changes are difficult to implement in code. Activity diagram presents concepts at a higher abstraction level of the system. Activity diagrams are also used for gray-box testing and checking consistency between code and design nearer to the code. Presence of the loops and concurrent activities in the activity diagram may result in path explosion.

Linzhang et al. [68] suggested an approach for generating test cases from activity diagrams and presented a tool UMLTGF. The tool UMLTGF supports automation in test case generation. The authors restricted that the loops will be executed at most once and considered the basic path coverage criterion for generating test cases. The approach relies on the assumption that any fork node can only have two outgoing edges and they generate two scenarios. The approach in [68] is a very long process. In comparison, our approach has less complexity in time consumption, as it traverses the AFG and generates test cases from the AFG.

4.7 Conclusion

In this chapter, we have proposed a technique to generate test cases from an activity diagram. Initially, we have constructed an activity flow table (AFT) from the activity diagram. Then, we constructed activity flow graph (AFG) by using the AFT. We proposed a new algorithm to generate test cases from UML activity diagram and illustrated our approach by considering the ATM case study. The graph was traversed by using a depth first search method. We finally, enumerated all possible paths from the start node to the end node of the AFG. We use activity coverage criteria in our approach. In our work, we have used the UML 2.0 model features like fork and join. The proposed model covers maximum activity nodes in the graph. It is applied in many different activity diagrams of different domains. Furthermore,
the test cases are also reduced. Our approach is capable of detecting loop, decision and synchronization faults. The approach is not fully automated. So, a complete automated tool can be developed for the proposed approach.

Different models are used to capture different types of faults in a system. By combining two models, it will be more efficient and certainly achieve better coverage and will give a stronger level of testing than using a single UML model. It will also detect more types of faults. The proposed approach in this chapter does not address the generation of test cases by using a combination of UML models. So, we aim at considering the combined UML models to generate test cases to uncover more types of faults in our next work.