Test Case Generation From UML Interaction Overview Diagram and Sequence Diagram

Thesis submitted in partial fulfillment of the requirements for the degree of

Master of Technology

in

Computer Science and Engineering

(Specialization: Software Engineering)

by

Deepak Kumar Meena

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela, Odisha, 769008, India

June 2013

Test case generation from UML Interaction Overview Diagram and sequence Diagram

Thesis submitted in partial fulfillment of the requirements for the degree of

Master of Technology

in

Computer Science and Engineering

(Specialization: Software Engineering)

by

Deepak Kumar Meena

(Roll- 211CS3072)

Under the guidance of

Prof. Ramesh Kumar Mohapatra

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela, Odisha, 769008, India

June 2013

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769008, Odisha, India.

Certificate

This is to certify that the work in the thesis entitled "***Test Case Generation from UML Interaction Overview Diagram and Sequence Diagram*** " submitted by ***Deepak Kumar Meena*** is a record of an original research work carried out by him under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of Master of Technology in Computer Science and Engineering with the specialization of Software Engineering in the department of Computer Science and Engineering, National Institute of Technology Rourkela. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Place: NIT Rourkela
Date:June 2013

Prof. Ramesh Kumar Mohapatra
Dept. of Computer Science and Engineering
National Institute of Technology, Rourkela
Odisha-769008

Acknowledgment

I am grateful to numerous local and global peers who have contributed towards shaping this thesis. At the outset, I would like to express my sincere thanks to Prof. Ramesh Kumar Mohapatra for his advice during my thesis work. As my supervisor, he has constantly encouraged me to remain focused on achieving my goal. His observations and comments helped me to establish the overall direction of the research and to move forward with investigation in depth. She has helped me greatly and been a source of knowledge.

I am very much indebted to Prof. Ashok Kumar Turuk, Head-CSE, for his continuous encouragement and support. He is always ready to help with a smile. I am also thankful to all the professors of the department for their support.

I am really thankful to my all friends and Research Scholars. My sincere thanks to everyone who has provided me with kind words, a welcome ear, new ideas, useful criticism, or their invaluable time, I am truly indebted.

I must acknowledge the academic resources that I have got from NIT Rourkela. I would like to thank administrative and technical staff members of the Department who have been kind enough to advise and help in their respective roles.

Last, but not the least, I would like to dedicate this thesis to my family, for their love, patience, and understanding.

Deepak Kumar Meena

Email : deepakmeena.iiita@gmail.com

Abstract

The most important part of the testing attempts is the test case generation. Unified modeling language(UML) is the most generally used to describe design specification and analysis by both academic and industry. UML models offer a lot of information that should not be ignored in testing.

Testing of software is a time-consuming activity which requires a great deal of planning and resources. In scenario-based testing, test scenarios are used for generating test cases, test drivers etc. By combining different UML components, different views of the program under test are used. UML provides the capability to enhance (explore) the static structure and dynamic behavior of a software system. Different UML strategies and techniques are implemented during the whole software development life cycle. Therefore UML becomes the source of test case generation. The main problems in testing object oriented programs is test case selection, it is impossible to stimulate the program with all data of the input domain.

A pragmatic approach is to concentrate on typical message sequences as modeled using the sequence diagram. Testing based on sequence diagrams seems to be intuitive. Each sequence diagram specifies one test case or set of test cases.

We proposed a method generate test cases using Interaction Overview diagram and sequence diagram. Our work considers interaction operators of UML 2.0 Sequence diagram like alt , loop par to generate test cases. First we construct the SD and Interaction Overview diagram for the given problem .After this we generate XMI code for these diagram using magic draw software ,its generate ID's of all nodes and all paths. Then we developed an intermediate graph, named UML interaction graph(UIG) and message dependency graph of sequence diagram. From the generated UIG, we generate different case, for represent different scenarios. The generated test cases achieve message path coverage.

Keywords: UML models, sequence diagram, interaction overview diagram, combined fragment, UML interaction graph, test cases.

Contents

List of Figures

List of Tables

List of Abbreviations

UML	Unified Modeling Language
IOD	Interaction Overview Diagram
SD	Sequence Diagram
TS	Test Case
UIG	UML Interaction Graph
ATM	Automated Teller Machine
STC	Set of Test Cases
STCID	Set of Test Cases from Interaction Diagrams

Chapter 1

Introduction

Motivation and Objectives
Problem Definition
Thesis Organization

Chapter 1

Introduction

Software testing is an important and an expensive process when developing a system. Software testing is a process in which defects are identified and subjected for rectification and finally make sure that all the defects are rectified, and ensure that the product is a quality product. Creation of test cases is possibly the most difficult step in testing. Test cases are commonly designed based on program source code. Therefore, designing a large number of test cases and carrying out the tests turn out to be very labor-intensive and time consuming. To reduce testing cost, test case generation from design documents has the added advantage of allowing test cases to be available early in the software development cycle, thereby making test planning more effective. The Unified Modeling Language (UML) is a general-purpose visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system.

In this thesis, we propose a test case generation method using interaction overview diagram and sequence diagram (SD). Sequence diagram are used for modeling the dynamic aspects of the system. SD represents the behavioral aspects of a system because they show how the messages are exchanged among objects. Interaction overview diagram can picture a control flow with nodes which contain visualizing of a sequence of activities. It shows dependency between the important sequences of a system, which can be presented by an activity diagram.

We use sequence diagram and interaction diagram as a source of test case generation. Our generated test suite aims to cover various interaction faults as well as scenario faults. For generating test data, sequence diagram alone may not be enough to decide the different components, i.e input, expected output pre condition and post condition of a test case. We propose to collect this information from the use case template, class diagram and data dictionary. These are associated with the use case for which the sequence diagram is considered in our work.

UML 2.0 SD combines multiple scenarios by the help of combined (interaction) fragments. There are different types of combined fragments in UML 2.x such as loop, alt and par. Execution of combined fragment is controlled by an operator called interaction operator. Interaction overview diagram can picture a control flow with nodes which contain visualizing of a sequence of activities. It shows dependency between the important sequences of a system, which can be presented by an activity diagram. In our approach, we transform sequence diagram and interaction overview diagram into an intermediate graph using XMI Code. This intermediate representation is called UML Interaction Graph (UIG) and message dependency graph of sequence diagram. From the intermediate graph, we generate different test cases, which represent different scenarios. In this thesis, we use UML diagrams to represent concurrent activities and generate test scenarios.

1.1 Motivation

Today's world software systems and their designs grow larger and more complex and this trend likely to continue in the near future. The only constant factor is that high quality of a software system is required. There is a relation between quality of a design and the quality of a system. If the software is bigger and complex then we need check that it works for all inputs and satisfys all the condition in software , that's why we need software testing.

Software testing is an important and an expensive process when developing a system. Software testing is a process in which defects are identified and subjected for rectification and finally make sure that all the defects are rectified, and ensure that the product is a quality product. Creation of test cases is possibly the most difficult step in testing. Test cases are commonly designed based on program source code. In our thesis we generate test case from UML interaction overview diagram and Sequence diagram. Where sequence diagram provides the dynamic view of a system and interaction overview diagram describes control flow among nodes.

1.2 Problem Definition

This thesis work focuses generating test cases for ATM problem. First we draw UML interaction overview diagram and sequence diagram for ATM problem. From IOD and SD we draw control flow graph and XMI code. With the help of XMI code and control flow graph we generate java code. From this java code we find all possible valid path. After generated all valid path we generated test cases for these valid paths.

1.3 Thesis Organization

The rest of the thesis is organized as follows: A literature survey is being carried out in **chapter 2**. We have discussed UML models in **chapter 3**. Basic concept and test case adequacy have been discussed in **chapter 4**. Proposed approach have been discussed in **chapter 5**. In **chapter 6** we gave conclusions , scope for the future work an also the limitations are discussed.

Chapter 2

Related Work

Chapter 2

Related Work

Many testing techniques are developed to generate tests from different UML diagrams [4] [6] [8] [10] [13]. Abdurazik and Offut et al. [7] developed a technique to generate test cases from UML state diagrams, which helps performing class-level testing but do not address test case generation. Kim et al. [15] us present a approach to generate test cases from UML activity diagram. Sharma et al. [1] have used SD (Sequence Diagram) for generating the test cases. In their work they have converted the SD into sequence diagram graph (SDG), and then traversed the SDG to generate the test cases. Swain et al. [3] have proposed a method for generating the test cases using sequence diagram and activity diagram. First they have converted the sequence diagram and activity diagram into model flow graph (MFG) to show the behavioral aspects. Then, they have generated the test cases from the MFG. This is a very complex task. Nayak et al. [9] in their proposed method have converted the SD into SCG (Structure Composite Graph) for test case generation. Samuel et al. [2] in their work they proposed a method for generating test cases from SD by transforming into Sequence Dependency Graph (SDG). Then, the SDG is traversed to generate test cases. Li Bao-Lin et al. [5] have proposed a method to generate test cases with the combination of SD and OCL expression. In this method, first they have constructed a tree for representing the SD, then traversed the SD for generating the test cases. In [10], sequence diagrams are combined with pre conditions and post conditions for the referenced methods. The paper proposes three different techniques for initializing objects, but all of these involve initialization from outside, e.g. based on a database of objects. Linzhang et al. [11] proposed a method to automatically generate test cases from UML activity diagrams using a gray-box method. In their method they generated test cases directly from UML activity diagrams. Their proposed method described 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. Briand and Labiche [6] generated functional system test requirements from UML analysis artifacts such as use cases, their corresponding sequence and collaboration diagrams, class diagrams and OCL used in all these artifacts. But, then did not generate test cases. In Gnesi et al. [16] an approach for formal test case generation from UML state charts is proposed. In contrast with the above discussed approaches, we generate actual test cases from combination of sequence diagram and interaction overview diagrams. As we use interaction diagrams therefore it detects interaction faults. In our approach, no redundant test cases are generated. As compare to other related work, in our approach, we have proposed a method for generating test cases for concurrent systems.

Chapter 3

UML Models

UML 2.0 Sequence Diagram

Interaction overview Diagram

Chapter 3

UML Models

3.1 UML 2.0 Sequence Diagram

Sequence diagram provides the dynamic view of a system. An important characteristic of a sequence diagram is that time passes from top to bottom, the interaction starts near the top of the diagram and ends at the bottom. A sequence diagram shows object interactions arranged in time sequence. It represents various possible interactions among different objects during an operation. There are 12 types of combined fragments in UML 2.0. A combined fragment has one or more processing sequence. Each combined fragment has one operator called interaction operator, one or more operands called interaction operands and zero or more guard conditions. Depending on the guard condition, the decision is made on what all operands need to be processed. UML sequence diagrams represent concurrency through the par combined fragment.

Loop (minint , maxint , [guard])-The loop operand will be repeated a number of times.

minint-the interaction must loop at least this number of times.

maxint (optional)-the interaction may not loop more than this number of times.

[guard] (optional)-after the first minint iterations, the condition is tested before each additional loop iteration , if the condition is false, then the loop is abandoned.

alt : The interaction operator alt means that the combined fragment represents a choice or alternatives of behavior. It selects one interaction to be executed from a set of interactions ,the selected interaction follows a true [guard] condition or an

[else] condition if none of the guard conditions are true (think of it as the case of if statements);guards are specified over the lifeline of the object which eventually contains the evaluated variable(s).

par : The interaction operator par defines potentially parallel execution of behaviors of the operands. Parameters are optional (they can be used if the diagram behaves in different ways) when they referred in different occurrences.

Sequence diagrams are sometimes called event diagrams, event scenarios, and timing diagram. A sequence diagram shows, as parallel vertical lines (lifelines), different processes or objects that live simultaneously, and as horizontal arrows, the messages exchanged between them, in the order in which they occur. This allows the specification of simple run time scenarios in a graphical manner.

3.2 Interaction overview Diagram

Interaction overview diagram describes control flow among nodes. It can show dependency between the important sequence of system, which can be presented by activity diagram. Interaction overview diagram can be defined by tuple

$IOD =< n0, nf, F, D, I, E >$ where

N: initial node

N: nf_1,nf_2....... nf_n set of final nodes

F:Set of fork nodes

D:Set of decision node

I:Set of interaction node

E:Edge connecting the IOD nodes

Chapter 4

BASIC CONCEPT AND TEST ADEQUACY CRITERION

Test case (TS)

Message path coverage criterion

Basic Interaction Coverage Criterion

Cyclomatic Complexity

Chapter 4

BASIC CONCEPT AND TEST ADEQUACY CRITERION

4.1 Test case (TS)

A test case is the triplet T [I,D,O] where I is the initial state of the system at which the test data is input, D is the test data input to the system, and O is the expected output of the system. [12]

A perfect test data must satisfy all specifications i.e. namely it must satisfy all test adequacy criterion. A test adequacy criterion helps to decide whether a set of test cases are sufficient, for software under testing. A testing criterion is a rule or a collection of rules which tells us the requirements that a set of test cases should satisfy.

4.2 Message path coverage criterion

A message sequence path represents the behavior to be tested and describes the interactions among the objects necessary to realize the corresponding functionality. Given a test set TS and a sequence diagram SD, in order to satisfy the message path criterion, it is required that TS must cause each possible message path in SD to be taken at least once [5]. A message sequence path begins at the entry of the tree and ends at the exit node. A message path is an overall process of message sending.

4.3 Basic Interaction Coverage Criterion

A basic path is a complete path through an interaction overview diagram where each loop is exercised either zero or one time. This ensures that all iterations in an interaction overview diagram are exercised.

4.4 Cyclomatic Complexity

Cyclomatic complexity [14] defines the number of independent paths in the basic set of a program. Cyclomatic complexity (or conditional complexity) is a software metric (measurement). It was developed by Thomas J. Mccabe Sr. in 1976 and is used to indicate the complexity of a program. It directly measures the number of linearly independent paths through a program source code. Cyclomatic complexity is computed using the control flow graph of the program: the nodes of the graph correspond to indivisible groups of commands of a program, and a directed edge connects two nodes if the second command might be executed immediately after the first command. Cyclomatic complexity may also be applied to individual functions, modules, methods or classes within a program.

M=E-N+2P

Where

M=Complexity

E=The number of edges of the graph

N=The number of nodes in the graph

P=The number of connected components (exit nodes).

Chapter 5

Proposed Approach

UML Interaction Graph

Chapter 5

Proposed Approach

In our proposed approach, we have taken UML 2.0 sequence diagram and interaction overview diagram as input, because SD allows the complex scenarios to represent in one sequence diagram and IOD describes control flow among nodes. First, we draw the sequence diagram (SD) and interaction overview diagram (IOD), after this we generate XMI code for these diagram. From XMI code we get IDs of all objects and sequence(path). After that we transform these diagrams into a graphical representation named UML Interaction Graph (UIG) using XMI ID's. Next, we calculate all possible paths from graph. Now , we create JAVA code for ATM problem using XMI ID'S. Then we generate the test cases . After that we insert all input values and conditions in our code and check that are satisfied or not.

5.1 UML Interaction Graph

An UIG is generated from the sequence diagram and interaction overview diagram which represents the possible control flow and message dependency between nodes. For construction of UIG, we use two tables to show message dependency and control flow between nodes. Then, we generate test cases using UIG.

Figure 1 shows the sequence diagram for the ATM (Automated Teller Machine) withdraws use case. Using an ATM, customers can access their bank accounts in order to make cash withdrawals, credit card cash advances, and check their account balances. The valid card users can access the ATM system. After validation of card, PIN verification is done. 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 returns false. Depending on the return value, the ATM machine dispenses the cash and prints receipts or displays the failure message.

	Methods between objects
A1	card Info( )
A2	Ask for PIN( )
A3	Display Invalid Card ( )
A4	Request PIN( )
A5	Display Invalid PIN( )
A6	PIN verified()
A7	Request for enter amount( )
A8	Enter Amount( )
A9	Display Error Message( )
A10	Withdraw process( )
A11	Display Error Message( )
A12	Dispense Cash( )
A13	Dispense Cash( )
A14	Print Receipt( )
A15	End

Table 5.1: Extracted message dependency from the sequence diagram

After that we generate XMI code from sequence diagram and show it in Figure 2. XMI code gives the IDs of all object and sequence(path). From XMI code we find all possible paths in sequence diagram and after that we construct the UIG for sequence diagram. With the help of XMI code we create java code , from this code we can check our test cases.

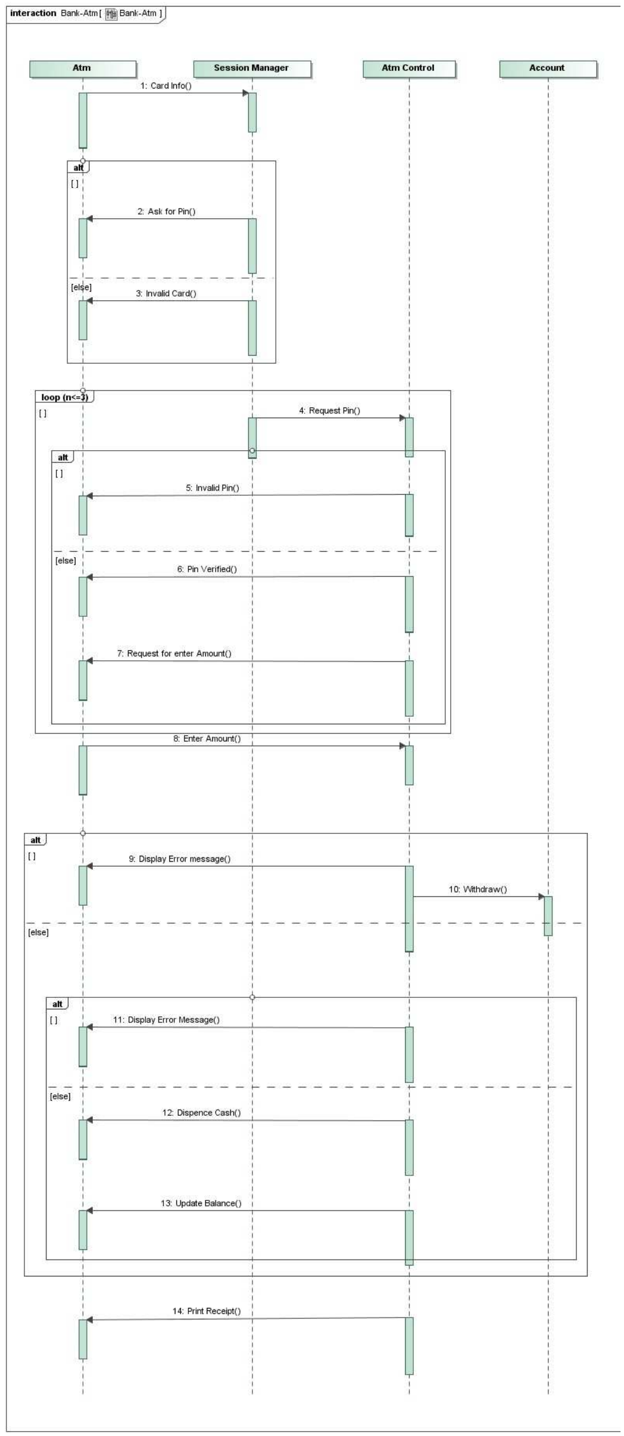

Figure 5.1: UML 2.0 sequence diagram for ATM withdraw use case

```xml
  <ownedAttribute name="Atm" xmi:id="_17_0_3_1_5c801ab_1366135998266_82860_2883" xmi:type="uml:Property" visibility="private"/>
  <ownedAttribute name="Session Manager" xmi:id="_17_0_3_1_5c801ab_1366136003835_3597_2896" xmi:type="uml:Property" visibility="private"/>
  <ownedAttribute name="Atm Control" xmi:id="_17_0_3_1_5c801ab_1366136006283_909648_2909" xmi:type="uml:Property" visibility="private"/>
  <ownedAttribute name="Account" xmi:id="_17_0_3_1_5c801ab_1366136008256_371614_2922" xmi:type="uml:Property" visibility="private"/>
  <ownedAttribute xmi:id="_17_0_3_1_5c801ab_1366136097918_832742_2935" xmi:type="uml:Property" visibility="private"/>
- <fragment xmi:id="_17_0_3_1_5c801ab_1366136097942_453887_2937" xmi:type="uml:MessageOccurrenceSpecification" visibility="public"
  message="_17_0_3_1_5c801ab_1366136097927_583386_2936">
    <covered xmi:idref="_17_0_3_1_5c801ab_1366135998261_806520_2882"/>
  </fragment>
- <fragment xmi:id="_17_0_3_1_5c801ab_1366136097945_349646_2938" xmi:type="uml:MessageOccurrenceSpecification" visibility="public"
  message="_17_0_3_1_5c801ab_1366136097927_583386_2936">
    <covered xmi:idref="_17_0_3_1_5c801ab_1366136097899_483423_2934"/>
  </fragment>
- <fragment xmi:id="_17_0_3_1_5c801ab_1366136318572_504179_2962" xmi:type="uml:MessageOccurrenceSpecification" visibility="public"
  message="_17_0_3_1_5c801ab_1366136318570_519282_2961">
    <covered xmi:idref="_17_0_3_1_5c801ab_1366135998261_806520_2882"/>
  </fragment>
- <fragment xmi:id="_17_0_3_1_5c801ab_1366136318573_397983_2963" xmi:type="uml:MessageOccurrenceSpecification" visibility="public"
  message="_17_0_3_1_5c801ab_1366136318570_519282_2961">
    <covered xmi:idref="_17_0_3_1_5c801ab_1366136003832_141874_2895"/>
  </fragment>
- <fragment xmi:id="_17_0_3_1_5c801ab_1366136508347_575354_3029" xmi:type="uml:CombinedFragment" visibility="public"
  interactionOperator="alt">
    <covered xmi:idref="_17_0_3_1_5c801ab_1366135998261_806520_2882"/>
    <cfragmentGate xmi:id="_17_0_3_1_5c801ab_1366136601921_696553_3070" xmi:type="uml:Gate" visibility="public"
      message="_17_0_3_1_5c801ab_1366136601919_362502_3069"/>
  - <operand xmi:id="_17_0_3_1_5c801ab_1366136508348_704782_3030" xmi:type="uml:InteractionOperand" visibility="public">
    - <fragment xmi:id="_17_0_3_1_5c801ab_1366136601922_794686_3071" xmi:type="uml:MessageOccurrenceSpecification" visibility="public"
```


Figure 5.2: Snapshot of XMI code of sequence diagram

With the help of Table 1 and XMI code , we generate message dependency graph for the sequence diagram of Figure 1. Figure 3 shows the above graph.

Figure 4 shows the interaction overview diagram for ATM withdraws activity. It represents the possible control flow between nodes.

Table 2 illustrates the control flow between the nodes extracted from the interaction overview diagram.

After that we generate XMI code from interaction overview diagram and show it in figure 5. XMI code gives the IDs of all object and sequence (path) . From XMI code we find all possible paths in sequence diagram and after that we construct the UIG for sequence diagram. With the help of XMI code we create java code , from this code we can check our test cases.

With the help of Table 2 and XMI code , we generate control flow graph for the interaction overview diagram of Figure 4. Figure 6 shows the above graph.

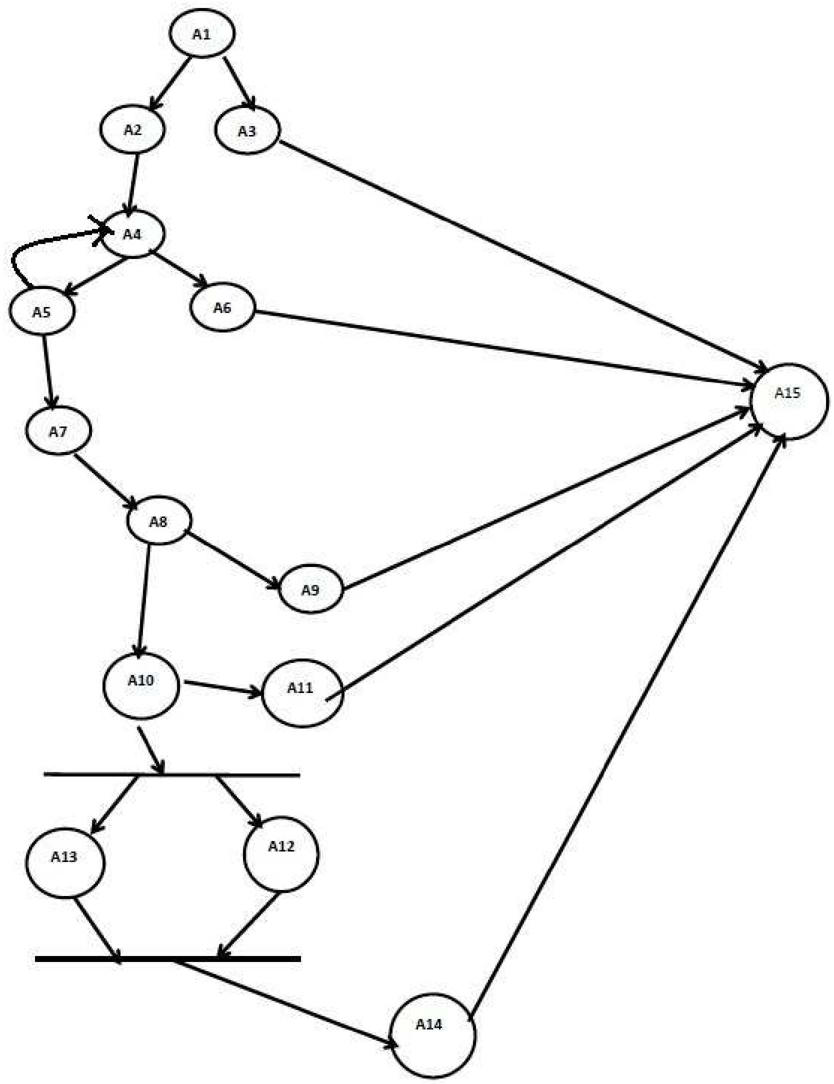

Figure 5.3: Message Dependency Graph for Sequence Diagram

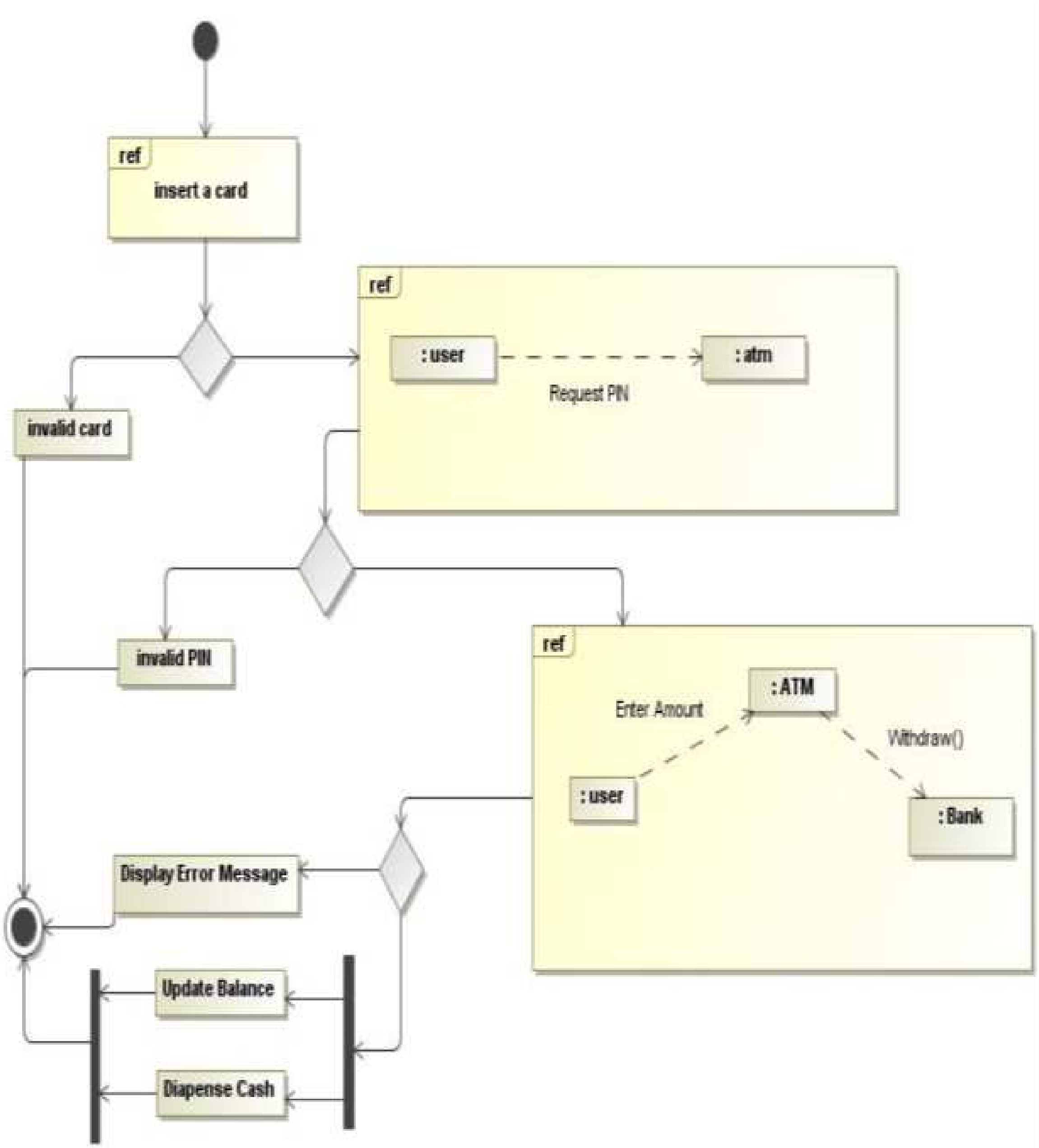

Figure 5.4: Interaction Overview Diagram for ATM withdraw activity

Unique Identity Number	Interaction Nodes
B1	Insert a card
B2	Invalid Card
B3	Request PIN ()
B4	Invalid PIN
B5	Enter Amount
B6	Validate Amount
B7	Display Error message
B8	Update balance
B9	Dispense Cash
B10	End

Table 5.2: Extracted control flow from the Interaction Overview diagram


```
<region visibility="public" xmi:id="_16_5_4_24400562_1351744592362_459219_994" xmi:type="uml:Region">
    <subvertex visibility="public" xmi:id="_16_5_4_24400562_1351744600700_818516_1015" xmi:type="uml:Pseudostate"/>
    <subvertex visibility="public" name="INSERT CARD" xmi:id="_16_5_4_24400562_1351744603882_613420_1019" xmi:type="uml:State"/>
    <subvertex visibility="public" name="VALIDATE CARD" xmi:id="_16_5_4_24400562_1351744627980_682506_1036" xmi:type="uml:Pseudostate"
        kind="choice"/>
    <subvertex visibility="public" name="ENTER PIN" xmi:id="_16_5_4_24400562_1351744652386_177374_1044" xmi:type="uml:State"/>
    <subvertex visibility="public" name="PIN" xmi:id="_16_5_4_24400562_1351744686594_291196_1076" xmi:type="uml:State"/>
    <subvertex visibility="public" name="ENTER AMOUNT" xmi:id="_16_5_4_24400562_1351744700082_528604_1093" xmi:type="uml:State"/>
    <subvertex visibility="public" name="VERIFY PIN" xmi:id="_16_5_4_24400562_1351744713796_477590_1107" xmi:type="uml:Pseudostate"
        kind="choice"/>
    <subvertex visibility="public" name="AMOUNT" xmi:id="_16_5_4_24400562_1351744813890_975058_1130" xmi:type="uml:State"/>
    <subvertex visibility="public" name="CHECK BALANCE" xmi:id="_16_5_4_24400562_1351744829996_141001_1147" xmi:type="uml:Pseudostate"
        kind="choice"/>
    <subvertex visibility="public" name="TAKE CASH" xmi:id="_16_5_4_24400562_1351744850426_198948_1155" xmi:type="uml:State"/>
    <subvertex visibility="public" name="CASH" xmi:id="_16_5_4_24400562_1351744872139_356296_1174" xmi:type="uml:State"/>
    <subvertex visibility="public" name="END" xmi:id="_16_5_4_24400562_1351744884378_591018_1191" xmi:type="uml:State"/>
    <subvertex visibility="public" xmi:id="_16_5_4_24400562_1351744924508_124004_1223" xmi:type="uml:FinalState"/>
    <transition visibility="public" xmi:id="_16_5_4_24400562_1351744615170_574555_1033" xmi:type="uml:Transition"
        target="_16_5_4_24400562_1351744603882_613420_1019" source="_16_5_4_24400562_1351744600700_818516_1015"/>
    <transition visibility="public" xmi:id="_16_5_4_24400562_1351744642762_77630_1041" xmi:type="uml:Transition"
        target="_16_5_4_24400562_1351744627980_682506_1036" source="_16_5_4_24400562_1351744603882_613420_1019"/>
  - <transition visibility="public" xmi:id="_16_5_4_24400562_1351744663266_692370_1058" xmi:type="uml:Transition"
    target="_16_5_4_24400562_1351744652386_177374_1044" source="_16_5_4_24400562_1351744627980_682506_1036">
        <trigger visibility="public" xmi:id="_16_5_4_24400562_1351744680971_764641_1069" xmi:type="uml:Trigger"
            event="_16_5_4_24400562_1351744680971_943077_1068"/>
    </transition>
```


Figure 5.5: Snapshot of XMI code of Interation Overview Diagram

5.2 Intermediate Representation

With the help of message dependency graph and control flow graph, we construct the UML Interaction Graph (UIG).

5.3 Test case Generation

Testing requires executing a program on a set of test cases and comparing the actual results with the expected results. A test case consists of a test input value, its expected output and the constraints, i.e. the pre condition and post condition for that input value [12]. In this section, we propose an algorithm to generate test cases. After constructing the UML Interaction Graph (UIG), we traverse the UIG

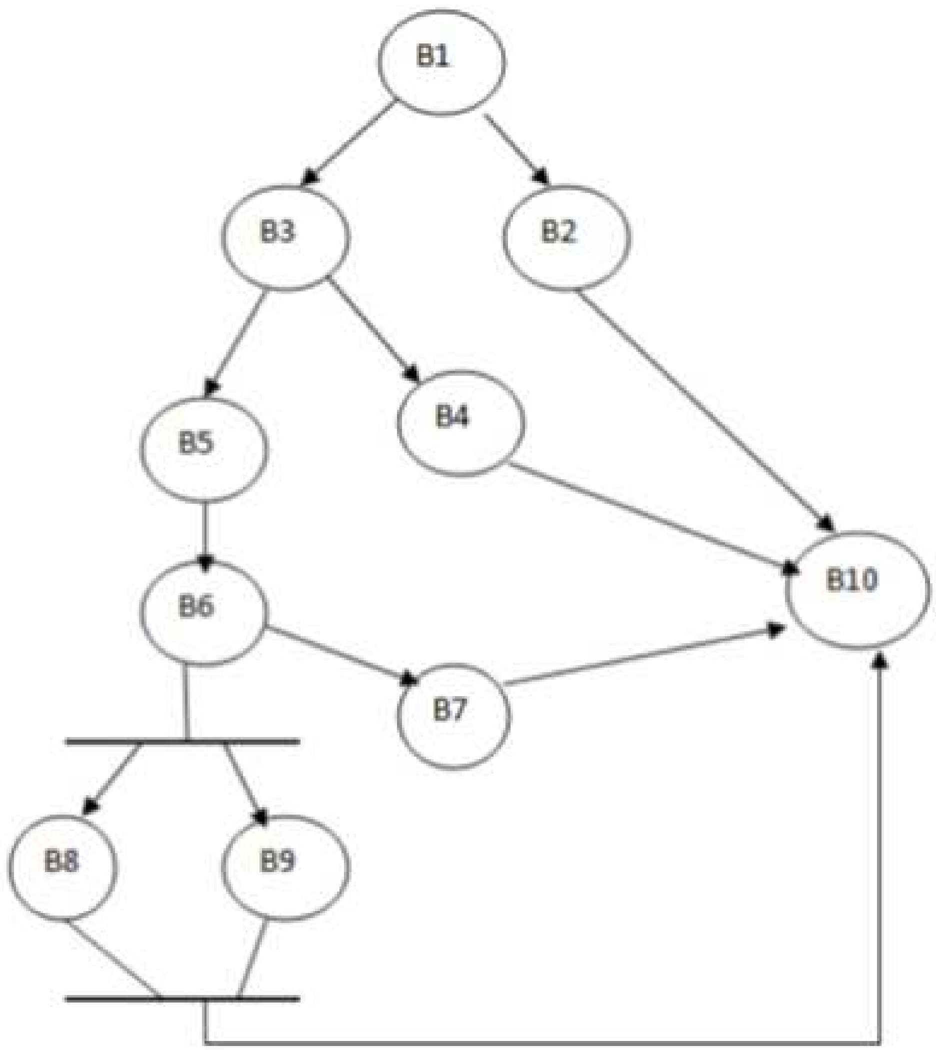

Figure 5.6: Control Flow Graph for Interaction Overview Diagram

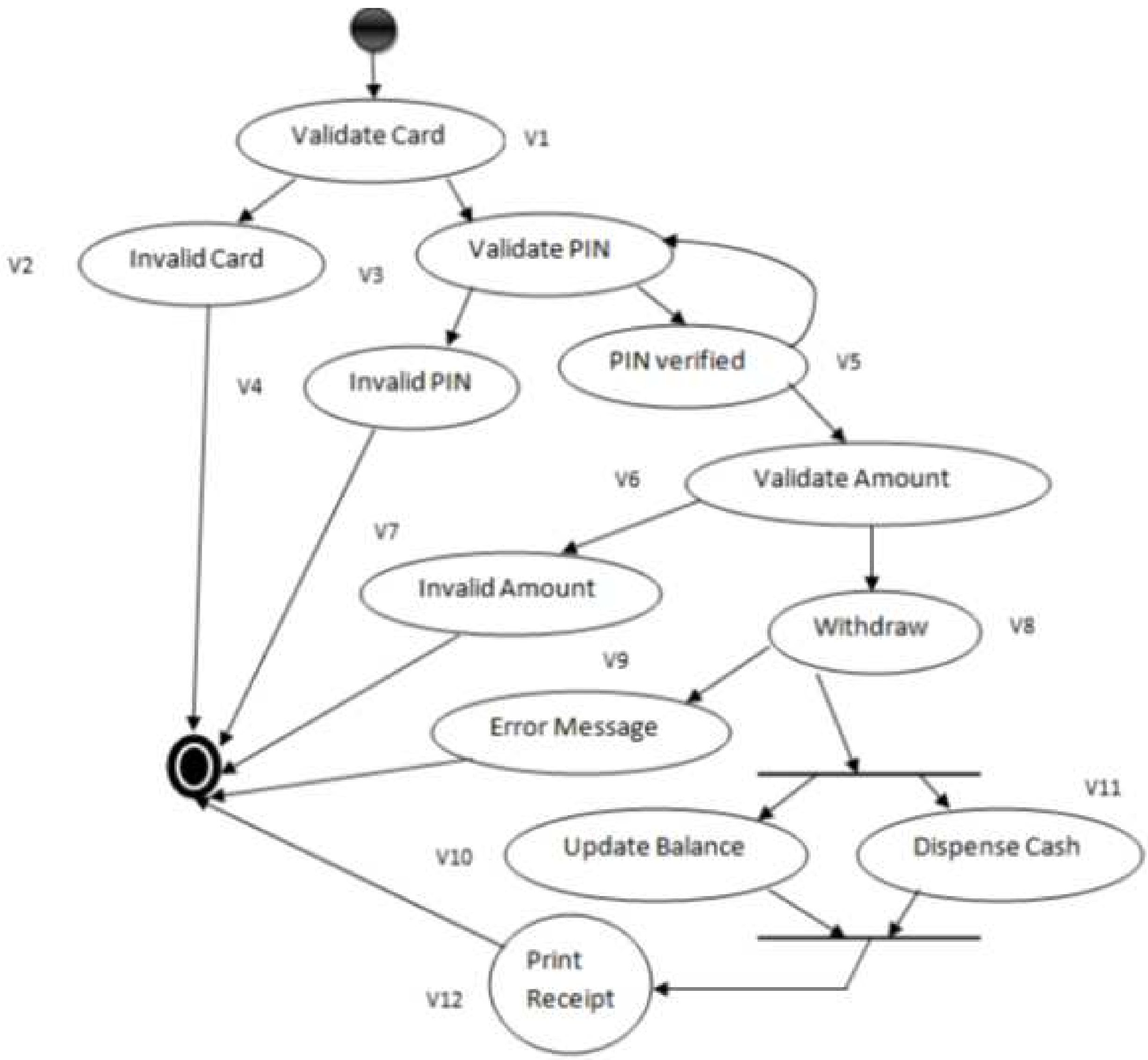

Figure 5.7: UML Interaction Graph (UIG) for ATM withdraw use case

for generation of the test cases. Now we present our proposed methodology for test case generation, in pseudo code form.

5.3.1 Algorithm

Algorithm 1: Set of Test Cases from Interaction Diagrams (STCID) Input : UML Interaction Graph (UIG)

Output : Set of test cases (STC) Steps

1. Itemize nodes of the graph UIG
2. Create initial node i.e. (In) = current node (Cn)
3. While (In! =Ln) where Ln =last node
4. Evaluate TCi = < TID, PreCond, I/P , O/P, PostCond >
    - i. Get test case ID (TID)
    - ii. Add pre condition value for $state_i$
    - iii. Add input values for $state_i$
    - iv. Add output values for $state_i$
    - v. Add post condition value for $state_i$
5. If parallel nodes is present, then

Call ConcurrentTestCaseGeneration(CTG)

6. End If
7. STC = combination of TCi
8. In= Sn where Sn = Succeeding node
9. If (In= =Ln) then

UpdatePath of UIG

10. EndWhile
11. Return the SetofTestCases(STC)
12. Exit

Algorithm 2: ConcurrentTestCaseGeneration(CTG)

Input: Concurrent Graph (CG) of UIG

Priority []: priority of activities Output: TestSet (TS)

Steps:

1. For each node Ni do
2. Push Ni into queue
3. While (Ni!= Null)
4. If priority (Ni) > priority (all nodes) then
5. Evaluate TCi= < TID, PreCond, I/P , O/P, PostCond >
   i. Get test case ID (TID)
   ii. Add pre condition value for $state_i$
   iii. Add input values for $state_i$
   iv. Add output values for $state_i$
   v. Add post condition value for $state_i$
6. Remove Ni from queue
7. TS = combination of TCi
8. EndIf
9. EndWhile
10. EndFor
11. Return TS
12. Exit

5.4 WORKING OF OUR ALGORITHM

Figure 1 shows the sequence diagram of a simple ATM withdraw use case and Figure 4 shows the interaction overview diagram for ATM withdraw activity. In Figure 7, (i=1..12) denotes a state corresponding to the sequence diagram and interaction overview diagram. In Algorithm 1, first we itemize all the paths, and then traverse all the paths of UIG graph for test case generation. Then, Steps 3 to 10 are iterated for each path in the UIG. Step 4 determines test cases for each iteration which consists pre conditions, input, output and post conditions. Algorithm 2 is called when concurrent node is encountered. In algorithm 2, the priority queue contains the concurrent nodes from the fork node. Priority of each concurrent node is stored in priority queue. Here, we set priority for concurrent process. A priority queue contains the combination of concurrent activities from the fork node. If the priority of is greater than the priority of, then is a legal path. In our example, we set the priority of (Update Balance) is greater than the priority of (Dispense Cash) in Figure 7. So, is a legal path. By applying proposed algorithms, we enumerate 5 different paths from the UIG graph and generate 5 test cases. Paths are given in Table 3.

Path ID	Valid Paths
P1	Start->v1->v2->End
P2	Start->v1->v3->v4->End
P3	Start->v1->v3->v5->v6->v7->End
P4	Start->v1->v3->v5->v6->v8->v9->End
P5	Start->v1->v3->v5->v6->v8->->(v10,v11)->v12->End

Table 5.3: Valid Paths

Then we generated the test cases corresponding to each path according our algorithms. The test cases are shown in Table 4.

Test case ID	Pre-condition	Input	Output	Post-condition
TID1	ATM is idle and displaying a welcome screen	Insert a card	Not an ATM card, Eject Card	Displays a welcome screen
TID2	ATM is idle and displaying a welcome screen	Insert a card, Enter PIN	Valid ATM card, Display message (Invalid PIN), Eject Card	Displays a welcome screen
TID3	ATM is idle and displaying a welcome scree.n	Insert a card, Enter PIN, Enter amount	Valid ATM card, Valid PIN, Display error message (Amount should be multiple of 100), Eject Card	Displays a welcome screen
TID4	ATM is idle and displaying a welcome screen	Insert a card, Enter PIN, Enter valid Amount, Amount > balance	Valid ATM card, Valid PIN, Display error message (Insufficient Balance), Eject Card	Displays a welcome screen
TID5	ATM is idle and displaying a welcome screen	Insert a card, Enter PIN, Enter valid Amount, Amount < balance	Valid ATM card, Valid PIN, Display message (Please Collect your money), Update Balance, Eject Card	Displays a welcome screen

Table 5.4: Generated Test cases corresponding to the Paths

Implementation of our approach

We used StarUML to produce UML 2.x SD design artifact and MagicDraw v. 16.5 to produce IOD design artifact. We implemented our approach using Java programming language in windows. A snapshot of sample output on a run over the example of ATM withdraw use case is shown in Figure. 6

```
run:
withdraw money 30000
After withdraw Your balance 71000
Reprint your Transaction Reciept
done (Idle, InsertCard, Reading_Card)
done (Reading_Card, CardReadSuccessfully, Choosing_Transaction)
done (Choosing_Transaction, SpecificEnter, SendingToBank)
done (SendingToBank, ApprovedPin, Performing_Transaction)
withdraw money 30000
After withdraw Your balance 71000
done (Performing_Transaction, CheckingForTransaction, WithdrawMoney)
done (WithdrawMoney, Print, PrintingReciept)
Reprint your Transaction Reciept
done (PrintingReciept, AskForAnotherPrint, PrintingReciept)
done (PrintingReciept, ExitAfterRPrint, Exit)
done Forced reset(true)
done (Idle, InsertCard, Reading_Card)
done Buildgraph reset(false)
done (Idle, InsertCard, Reading_Card)
done (Reading_Card, CardReadSuccessfully, Choosing_Transaction)
done (Choosing_Transaction, SpecificEnter, SendingToBank)
done (SendingToBank, ApprovedPin, Performing_Transaction)
withdraw money 30000
After withdraw Your balance 71000
done (Performing_Transaction, CheckingForTransaction, WithdrawMoney)
done (WithdrawMoney, Print, PrintingReciept)
done (PrintingReciept, ExitAfterRPrint, Exit)
done Forced reset(false)
done (Idle, InsertCard, Reading_Card)
done (Reading_Card, CardReadSuccessfully, Choosing_Transaction)
done Buildgraph reset(true)
State Coverage=8/8
transition coverage was 8/8
Transition pair coverage = 8/9
Action coverage = 8/15
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 5.8: A snapshot of program

Chapter 6

Conclusion and Scope of Future Work

Chapter 6

Conclusion and Scope of Future Work

We have proposed an approach to generate test cases from UML sequence diagram and interaction overview diagram. Our approach first transformed the sequence diagram and interaction overview diagram to an intermediate form called UML interaction graph (UIG) using XMI code. Then, our approach has traversed the graph to find the all valid path then we generate the test cases. Here, the message dependency is identified using feature of UML 2.0 sequence diagrams called the combined fragment. The information which are required for the test cases i.e. input, output and states are recovered from the UIG. The test cases are suitable for system testing and to detect interaction faults. In future, we want to optimize the test cases for getting an appropriate amount of test cases by eliminating redundant test cases.

Bibliography

Bibliography

[1] M. Sarma, D. Kundu and R. Mall, "Automatic Test Case Generation from UML Sequence Diagrams", *In IEEE 15th International Conference on Advanced Computing and Communications*, pp. 60-65, 2007.

[2] P. Samuel and A. T. Joseph, "Test Sequence Generation from UML Sequence Diagrams", , *In IEEE Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*,pp. 879-887, 2008.

[3] S. K. Swain and D. P. Mohapatra , "Test Case Generation from Behavioral UML Models ", *International Journal of Computer Appli-cations*,Volume-6, pp. 5-11, September 2010.

[4] Emanuela G. Cartaxo, Francisco G. O. Neto and Patricia D. L. Machado, "Test Case Generation by means of UML Sequence Diagrams and Labeled Transition Systems ", *International Journal of Computer Applications*,pp. 1292-1297, 2007.

[5] Li Bao-Lin, Li Zhi-shu, Li Qing and Chen Yan Hong, "Test Case automatic Generation from UML Sequence Diagrams and OCL expression ", *In IEEE International Conference on Computational Intelligence and Security.*,pp. 1048-1051, 2007.

[6] L. Briand and Y. Labiche, "A UML-Based Approach to System Testing ", *Journal of Software and Systems Modeling, Springer Verlag*, Volume-1, pp. 10-42, 2002.

[7] A. Abdurazik and J. Ofut, "Generating Tests from UML Specifications ", *In the Proceedings of 2nd International Conference on Unified Modeling Language (UML), Fort Collins, CO,* 1999.

[8] S. K. Swain, D. P. Mohapatra and R. Mall , "Test case generation on use case and sequence diagram ", *International Journal of Software Engineering*, Volume-3, pp. 2152, 2010.

[9] A. Nayak and D. Samanta , "Automatic test data synthesis using UML sequence diagram ", *Journal of Object Technology*,Volume-09, pp. 75-104, 2010. .

[10] F. Fraikin, and T. Leonhardt, "SEDITEC-testing based on sequence diagrams", *In Proceedings 17th IEEE International Conference on ASE*,pp. 261266, 2002.

[11]] 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)*,pp. 284-291, 2004.

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

[13] "Object Constraint Language 2.0 is available from Object Mangement Groups web site (http://www.omg.org/) ",

[14] Ayan Nigam, Bhawna Nigam , Devendra Kumar Vatsa , "Generating all Navigational Test Cases using Cyclomatic Complexity from Design Documents for Mobile Application ", *International Journal of Computer Applications (0975 8887)*, Volume 40 No.12, February 2012.

[15] H. Kim, S. Kang, J. Baik, and I. Ko. , "Test cases generation from uml activity diagrams Application ", *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pages 556 561, 2007.

[16] S. Gnesi, D. Latella, M. Massink, and C. ISTI. , "Formal test-case generation for uml statecharts", *Proceedings of the Ninth IEEE International Conference on Engineering Complex Computer Systems*, pages 7584, Jan 2004.