An Investigation of Test Cases Generation from Activity Diagram

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

Master of Engineering
in
Software Engineering

Submitted By
Jeena Gupta
(Roll No. 801231014)

Under the supervision of:
Dr. Ajay Kumar
Assistant Professor,
Computer Science and Engineering Department

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

June 2014

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, "*An Investigation of Test Cases Generation from Activity Diagram*", in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Software Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Dr. Ajay Kumar* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

Jeena

(Jeena Gupta)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(Dr. Ajay Kumar)
Assistant Professor
Thapar University
Patiala

Countersigned by

(Dr. Deepak Garg)
Head
Computer Science and Engineering Department
Thapar University
Patiala

(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

ACKNOWLEDGEMENT

No volume of words is enough to express my gratitude towards my guide, **Dr. Ajay Kumar.** I thank my supervisor for his time, patience, discussions and valuable comments. He has been very concerned and have aided for all the material essential for the preparation of this thesis report. His enthusiasm and optimism made this experience both rewarding and enjoyable.

I am equally grateful to **Dr. Deepak Garg**, Head of Computer Science and Engineering Department and **Ms. Damandeep Kaur**, P.G Coordinator, for motivation and inspiration that triggered me for the thesis work.

I also want to express my gratitude to **Dr. S. K. Mohapatra**, Dean of Academic Affairs, for making provisions of infrastructure such as library facilities, computer labs equipped with net facilities, immensely useful for the learners to equip themselves with the latest in the field.

I am also thankful to the entire faculty and staff members of Computer Science and Engineering Department for their help, cooperation and affection, which made my stay at Thapar University memorable.

Last but not the least, I would like to thank my parents and Almighty for showing me the right direction, without their blessings none of this would have been possible.

Jeena Gupta
(801231014)

ABSTRACT

Generation of test cases is the most important issue in the software testing. Thus test cases need to be carefully designed. Test cases can be generated manually or automatically. Activity diagrams are even more challenging for creating test cases. An activity diagram consists of flowchart of various activities with transition among them. It states the internal behaviour of operations of the system. An activity diagram is used for modelling the dynamic aspects of the system. It helps in visualizing the sequence of activities involved in a control flow. The proposed approach is to automate the generation of test scenarios from activity diagram using DFS method. By using this method, the useless test paths are eliminated which in turn reduces the time complexity. This approach also helps in synchronization between various activities. It generates a directed graph called Activity Flow Graph (AFG) and an intermediate table called Activity Dependency Table (ADT) automatically for each activity diagram. The ADT considers all the dependencies in the form of parent child nodes. Automated test paths are generated from AFG. The test cases are generated to cover all the test paths created by AFG. This approach was found to be effective in reducing the time complexity.

TABLE of CONTENTS

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

This chapter introduces the concept of testing, its objectives, testing techniques, importance of testing and overview of UML.

1.1. Software Testing

Testing is one of the crucial phases in the software development process. Testing is the process of finding errors in the program. In software testing, tester tests the program with well designed inputs with the intent of noticing failures. Software testing is a practice of determining faults in the software [20]. Testing increases the quality of software in terms of correctness, reliability, usability and maintainability. It begins from the requirements specification phase, continue to the deployment phase, and so on. Preparation for testing should begin as soon as possible when software product is defined. The objectives of testing are as following [28]:

i. Testing is the process of exercising a program in order to find bugs in the programs.
ii. A good test is one that has a probability of discovering errors.
iii. Testing should aim at suggesting changes or modification if required, thus adding value to the entire process.
iv. Testing is the process of determining whether the output comes according to a given input.

Abstract view of the flow of testing:

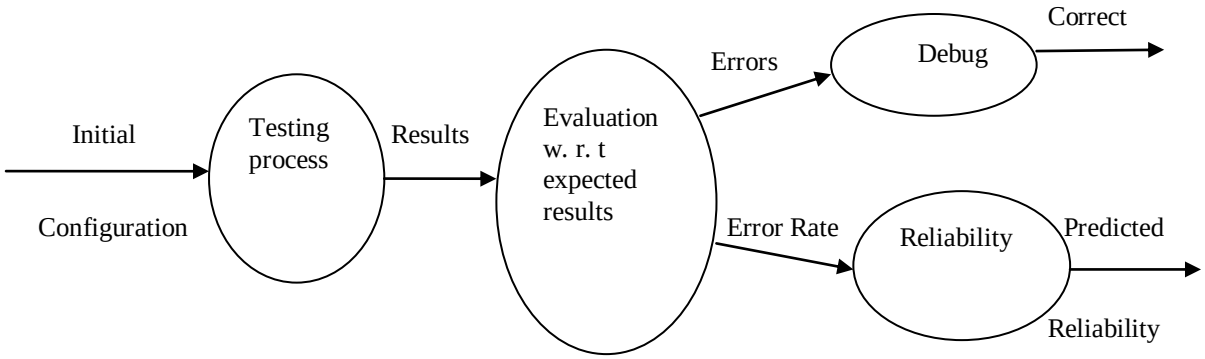


Figure 1.1: Testing Information Flow [28].

1.2. Software Testing Techniques

Software testing techniques are of two types.

1.2.1. Static Testing

It involves manual review of the code, design document without executing the code. It is done with respect to a set of inputs. It is conducted during the initial stages of software lifecycle in order to early detection of errors [5]. Static testing is performed at the verification stage. It includes code review, inspection, audits and walkthrough etc.

1.2.2. Dynamic Testing

It involves execution of the application with valid inputs to check the expected output. This is used for testing the functional behavior of software [3]. This testing is done with respect to a specific input. Dynamic testing is performed at the validation stage. It includes unit testing, system testing, integration testing and acceptance testing.

1.3. Testing Approaches

Software testing is a process of exercising the software in order to ensure the software is error free. It is a process of verifying and validating the developed programs. Following are the major testing approaches used [33].

1.3.1. White Box Testing

White Box testing helps in the creation of test cases with proper knowledge of internal working of the code. This testing uses the internal structure of the program to generate test data. It takes into account the program code, internal structure and design flow. This testing is commonly known as transparent testing or glass box testing. The main focus is on the internal structure of the software.

Using white box testing, tester can derive a test case that guarantees:

i. All independent paths must be exercised at least once.
ii. All logical decisions must be exercised on its true and false sides.
iii. All loops should be executed at its boundaries.
iv. To access internal data structures

1.3.2. Black Box Testing

Black Box testing helps in the creation of test cases without having any knowledge of

the functionality of the program. The programmer needs to provide input and corresponding output is checked. This testing is based on requirements given by the user.

This testing is also known as functional testing. It involves evaluation of the correctness of the program which is free from the internal structure of the code. Various Testing Techniques include in the Black Box Testing are Equivalence Partitioning, Boundary Value Analysis and Comparison Testing. The main focus of attraction is the external behavior of software.

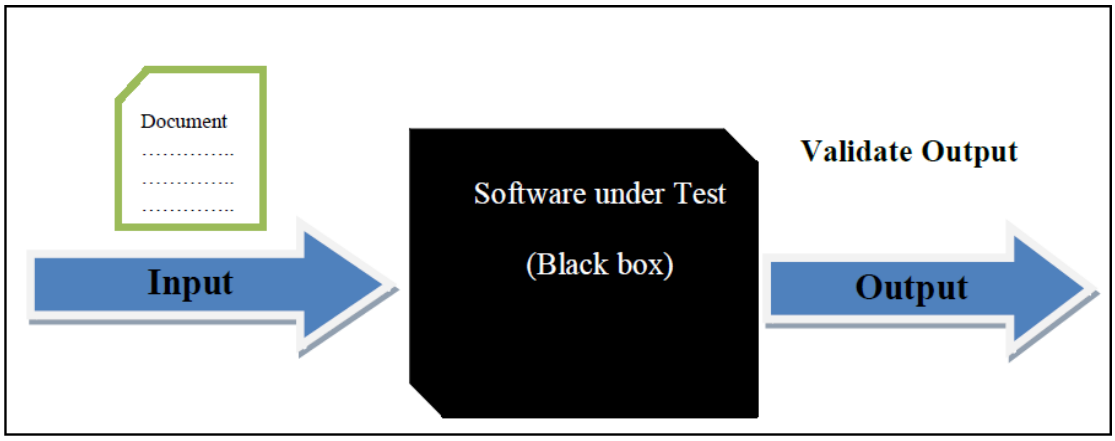


Figure 1.2: Black Box Testing [33].

1.3.3. Grey Box Testing

This testing includes the features of both white box testing and black box testing. The tester needs to have partial knowledge of the internal structure which includes documentation of the internal data structures and the algorithms used. It is used for searching the defects if there, due to improper structure or incorrect usage of the application.

1.4. Importance of Testing

Testing takes 50% cost of the total software development cost. Testing is used in various safety critical applications such as medical, avionics, complex and other control systems. In such application, highly reliable software is required. Testing is an important part of quality assurance in the software development process [31]. The testing has come into high demand for object-oriented programming and developing Fourth Generation Languages (4GL) from last few years. The feature such as encapsulation, polymorphism, inheritance and others associated with object oriented technology increases the complexity of software and highly complex software is a challenge for testing.

1.5. Unified Modeling Language (UML)

In the last few years, object-oriented analysis and design (OOAD) has come into existence, it has found widespread acceptance in the industry as well as in academics [28]. The main reason for popularity of OOAD is

i. Reusability of code.
ii. Increased productivity.
iii. Maintenance and ease of testing.
iv. Better design understandability.

UML was released by Object Management Group (OMG) in 1997. UML is a language used for designing the system. It is also used for constructing, documenting and modeling the artifacts of a software system. UML provides a way to visualize the behavior of real life objects. UML helps to visualizing the software at early stage of development lifecycle which in turn increases the confidence of developers and end users. UML also helps in creating documentation of software which maintains consistency between specifications and design document. UML is used to build large, complex and constructs various different diagrams representing different views of software model. With the growing demand of UML in modeling object oriented software, researchers have begun investigating in how UML can be useful in testing phase in software development process. As a result, many UML- based approaches to software testing have been proposed. In these approaches, test requirements and coverage criteria are derived from UML diagrams.

1.6. UML Diagrams Overview

Fourteen diagrams of UML 2.2 [30] have been divided into two categories. Seven diagrams represent structural information and the other seven represent the general types of behaviour, including four that represent different aspects of interactions. These fourteen diagrams can be categorized hierarchically as shown in the following class diagram as shown in figure 1.2.

UML does not restrict UML element types to a certain diagram type. In general, every UML element may appear on almost all types of diagrams; this flexibility has been partially restricted in UML 2.0.

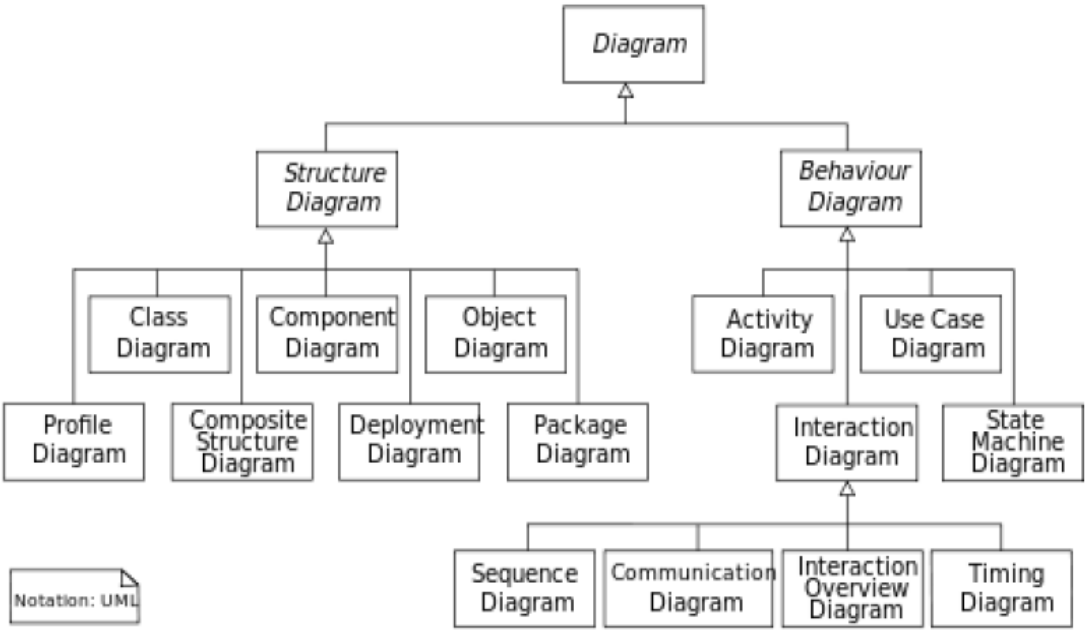


Figure 1.3: UML Diagrams Overview [28].

1.6.1. Structure Diagrams

Structure diagrams emphasize the objects that are present in the system to understand its working and implementation. Structural diagrams come under static view model since the structure of the system remains constant. It includes the following diagrams [30]:

i. **Class Diagram**

This diagram describes the system structure by showing systems classes, their attributes and relationships among classes.

ii. **Component Diagram**

This diagram shows the various components of the system and describes the dependencies among components.

iii. **Object Diagram**

An object diagram describes a complete or partial view of the system modeled at a specified time.

iv. **Deployment Diagram**

A deployment diagram describes the implementation details of the system i.e. what kind of hardware is deployed on the system.

v. **Composite Structure Diagram**

This diagram describes the internal structure of a class and the collaborations

possible using this structure.

vi. **Package Diagram**

Package diagram allows organizing model elements into groups by showing the dependencies among these groups.

vii. **Profile Diagram**

This diagram operates at the meta-model level to show stereotypes as classes. It includes stereotype and profiles as packages. The extension relation indicates what meta-model element a given stereotype is extending.

1.6.2. Behavior Diagrams

Behavior diagrams emphasize on the behavior of the software system i.e. how various objects interact with each other. It includes the following diagrams [28, 30]:

i. **Activity Diagram**

Activity diagram consists of flowchart of various activities with transition among them. An activity states the internal behaviour of an operation of the system. An activity diagram is used for modeling the dynamic aspects of the system. It emphasizes the sequential or concurrent flow path from activity to activity. Both conditional and parallel activities can be represented by an activity diagram [33].

ii. **State Machine Diagram**

State machine diagram captures various states and their transitions occur in the system. The state describes the operation of the system.

iii. **Use Case Diagram**

A use case diagram shows the functionality of the system in terms of actors, use cases and dependencies between use cases.

1.6.3. Interaction Diagrams

An interaction diagram is a subset of behavior diagram which shows the flow of control among objects in the system. It includes the following diagrams [28]:

i. **Communication Diagram**

A communication diagram represents the interaction between objects in terms of the sequence of messages. It describes the structural as well as behavioral relationships among objects.

ii. Sequence Diagram

A sequence diagram shows communication between the objects takes place with the sequence of messages transferred between them. It indicates the lifespan of objects relative to those messages.

iii. Interaction Overview Diagram

Interaction overview diagram provides an overview of the nodes and their respective communication diagrams.

1.7. UML Modeling Tools

IBM Rational Rose is the most well-known UML modeling tool. Other tools are Visual Paradigm for UML, Enterprise Architect, Magic Draw UML, Modelio, Rational Rhapsody, Power Designer, Rational software architect, StarUML and many more. The popular development environments that support UML modeling tools are Eclipse, Net Beans and Visual Studio [30].

CHAPTER 2

TESTING TERMS

2.1. Test Case Generation

A test case consists of a set of inputs, condition and the expected output. A test case is input values which a tester can apply to check if the application produces a correct value or not.

Generation of test cases is the most important issue in the software testing. Thus test cases need to be carefully designed. Test cases can be generated manually or automatically. Manual designing of test cases are very time consuming and laborious. Thus design of test cases needs to be automated. Major problem is that the software applications become larger and complex, thus generation of automated test cases is an important issue. In this way, the testing costs can be minimized by avoiding creation of irrelevant test cases hence increases the maintenance cost and time required for doing testing and debugging. Automated test case generation process eliminates the cost of manual test case design efforts [33].

2.2. Techniques for Test Case Generation

Following are the various techniques that can be used for the generation of test cases:

2.2.1. Specification-based Test Case Generation Techniques

These techniques help to derive testing information from a specification of the software under test, rather than from the implementation. The specifications provided by user helps in deriving the test scenarios. Specifications play an important part in the testing in providing inputs and check corresponding outputs [19]. An important application of specification in testing is to provide test oracles.

Advantages of specification based test case generation techniques:

i. The specifications define significant aspects of the software.
ii. It supports a simple, organized and logical approach for the development of functional tests.
iii. The specifications of software can be used as a guide for designing functional tests for the product.

2.2.2. Model-Based Test Case Generation Techniques

Model Based Testing is a black-box testing technique where common testing tasks such as test case generation and test result evaluation are automated based on a model of the application under test. It supports automation of the complete test cases design and the construction of traceability matrix, which connects a link between specifications and generated test cases. The test designer generates an abstract model of the system under test instead of writing hundred of test cases. A set of test cases are generated by using model based testing tool [8].

Following are the advantages of model based test case generation techniques:

i. **Fault Detection**

   Model based test generation finds faults at the initial stage of software development lifecycle. Hence it reduces the testing time.

ii. **Enhance Test Quality**

   It supports automatic and systematic creation of test suites that satisfy coverage criterion of the test model which enhances the test quality. Model-based testing however uses an automated test generator based on algorithms and heuristics to choose the test cases from the model, which makes the design process systematic and repeatable.

iii. **Specification Defect Detection**

   In model based testing, the first step is the creation of abstract model of system SUT, which helps in finding the requirement issues. It helps in finding faults and inconsistencies within the requirements. This is a major benefit of model based testing because requirements problems are a major source of system problems.

iv. **Reduces Maintenance Cost**

   The testing begins at early stages of the software development lifecycle and it allows improvement in the design Therefore maintenance cost can be minimized.

v. **Availability of Test Cases**

   Test cases in the model based technique are available early in the SDLC and it makes the test planning more effective.

vi. **Implementation Independent**

   The test data obtained from this technique is independent from the implemen-

tation architecture.

2.2.3. Code Based Test Case Generation Techniques

For constructing the code, data paths and test cases, it uses code based analysis techniques by traversing these paths. In white box testing, the main concern is code based test case generation techniques. In the testing process, first locate the area in the code that is not executed by the current test case set and for increasing the coverage, additional test case creation is followed [20].

Advantages of the code based test case generation:

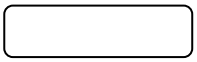
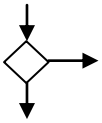
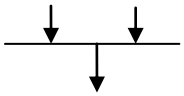
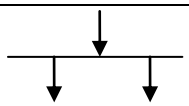
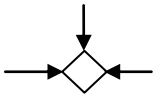
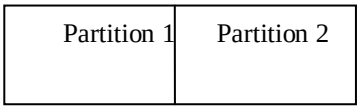
i. Code based testing helps in creating an additional set of test cases for increasing code coverage.
ii. It is useful in identifying the areas of a program that are not exercised by a set of test cases.

2.3. Activity Diagram

Activity diagram is one of the UML behavior diagram. An activity diagram consists of flowchart for various activities with transition among them [19]. It states the internal behavior of an operation of the system. An activity diagram is used for modeling the dynamic aspects of the system. Activity diagram is a kind of procedural flow chart. It helps in visualizing the sequence of activities involved in a control flow. An activity diagram emphasize on the sequential or concurrent flow path from activity to activity. It discusses the ordering of the activities. An activity diagram consists of atomic and compound actions which consist of further sub activities. The sub activities denote a separate flow of control within the activity diagram.

Both conditional and parallel activities can be represented by an activity diagram. An activity diagram is having branch and fork construct to describe condition and concurrent activities. The edges represent the control flow between various activities. The flow can be sequential, branched or concurrent which is represented by different elements such as fork and join purpose. The fork is used to show concurrent activities flowing through synchronization bar and join is used to combine multiple activities into a single path. Activity diagrams are helpful in considering the issues of synchronization and concurrency for test case generation purpose.

Table 2.1: Control Nodes Used in the Activity Diagram [19].

Names	Symbols Used	Description
Initial node		It is filled circle indicates the start of the flow.
Activity		Rounded rectangle shows the activity to be performed.
Edge		The direction of flow
Decision node		Diamond shaped box with one incoming flow and two outgoing flows.
Join		A synchronization bar is having multiple incoming flows and one outgoing flow.
Fork		A synchronization bar is having one incoming flow and multiple outgoing flows.
Merge		Diamond shaped box with multiple incoming flows.
Swim lanes		Describes the concurrent flow of activities and by whom these are performed.
Final node		Filled circle with outline denotes end of the flow.

An Illustration of an Activity Diagram:

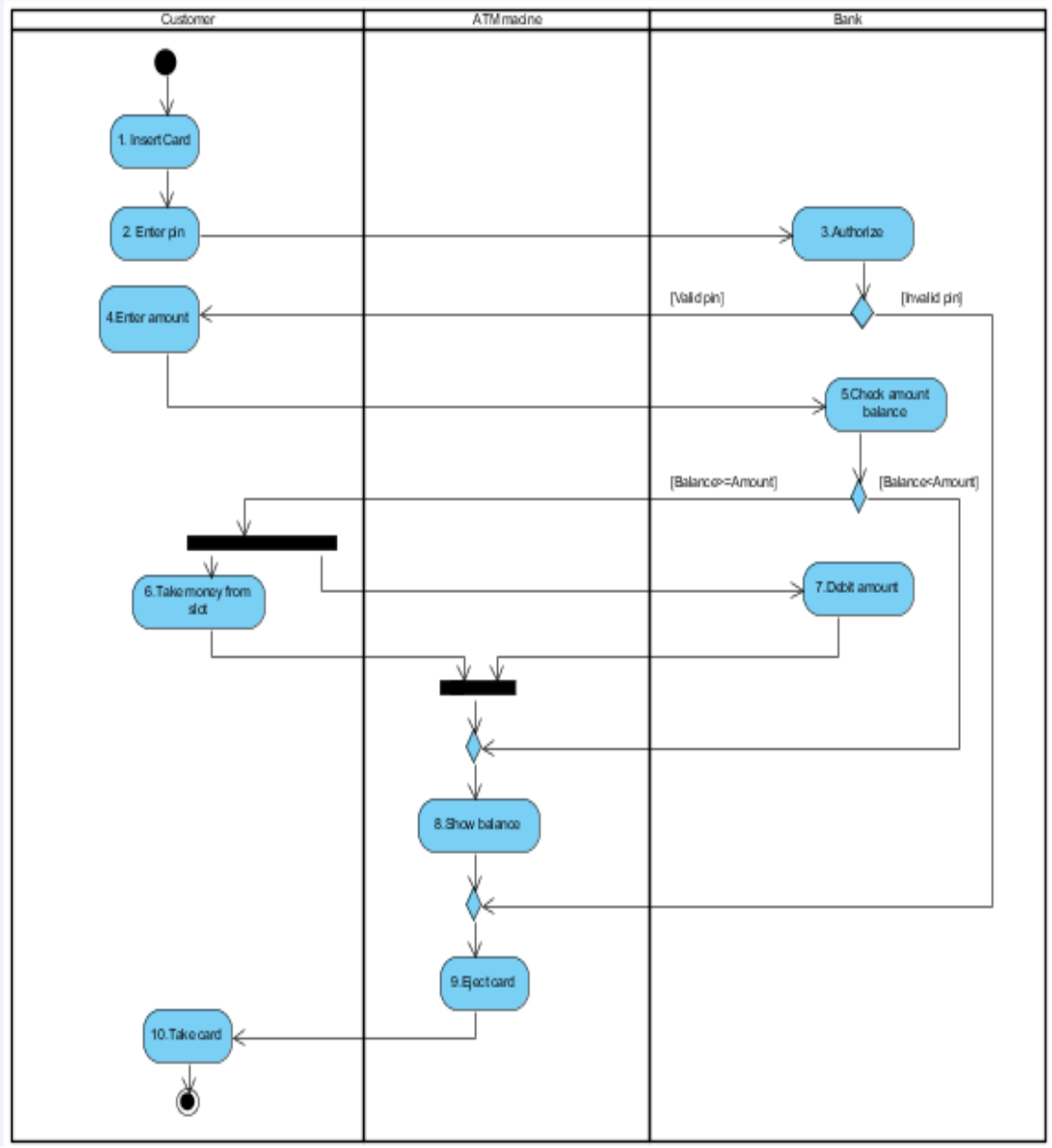


Figure 2.2: Activity Diagram for ATM Cash Withdrawal [23].

2.4. Test Coverage Criteria

As the specifications play a key role to obtain an accurate result. In the same way, the adequacy criteria keep importance as it specifies where to stop the testing process. Therefore before going to follow the testing procedure, the adequacy criteria need to be specified. For generation of test scenarios from the activity diagram, the following adequacy criteria need to be considered [25].

i. Activity Coverage

Test scenarios guarantee that every activity state needs to be covered at least

once. An activity coverage criterion follows any sequence of activities from the initial node to the terminal node.

ii. **Path Coverage**

Test scenarios need to cover all the possible paths at least once in the flow graph. It is not a possible approach because many flow graphs contain an infinite number of paths. The solution is to remove redundant edges or to remove redundant nodes.

iii. **Predicate Coverage**

A test set T satisfies full predicate coverage criteria if and only if there exist t_1 in T which evaluates clause c in condition to TRUE and there exist t_2 in T which evaluates cause c to FALSE. This condition requires that all the predicates in the condition are checked.

iv. **Iteration Coverage**

Given a test set T such that T must cause the loop to be traversed at least once. This criterion requires that all the iterations should be covered once, twice and K times.

2.5. Structure of the Thesis

The rest of the thesis is organized as follow:

Chapter 3 – This chapter discusses the detailed analysis of literature review of existing techniques/algorithms for test case generation using activity diagram.

Chapter 4 - This chapter describes the problem definition.

Chapter 5 - This chapter discusses in detail the solution of the problem with the help of block diagram and proposed algorithm. It also focuses on implementation details and experimental results obtain from the algorithm that has been developed for reducing the total time for generating test cases.

Chapter 6 - This chapter describes the conclusion drawn from the work done and future directions possible.

CHAPTER 3
LITERATURE SURVEY

This chapter discusses the existing algorithms and various techniques comprising of activity diagrams which can be used for generating test cases.

3.1. Test Case Generation using Grey- Box Method

Linzhang *et al.* [1] presented a technique that helps in generating test cases from UML activity diagram directly by using gray-box method. They are used in reducing the cost of test model by creating a reused design. In this approach, a method was proposed to generate test cases along with the implementation of code. It helps testers to make use of test resources optimally. The test cases modeling an operation were directly generated from the activity diagram. Then whole data gathered for generating test cases such as input/output sequence, various parameters, the constraint conditions and so on is obtained from every test case. Finally, by implementing the method of category-partition, the potential values of all the parameters such as input/output were generated. The inconsistency between the implementation and the design can also be unraveled by the generated test suites. UMLTGF is a prototype tool constructed by the authors to support the overall process. The limitations of white box method were overcome by Grey box testing which was used to find all the potential paths from the design model. The expected behavior of an operation can be described by these paths. Then test cases are generated which satisfied the path coverage conditions constructed by black box method. This approach was able to find the problems which were neglected by both black and white method by applying the combination of both the approaches.

3.2. Test Case Generation for Java Programs

Chen *et al.* [18] presented an automatic technique for generation of test cases for Java programs by keeping test adequacy criteria with respect to UML activity diagrams. This approach considers an indirect approach for constructing test cases. First, Java program is instrumented against the activity diagram specifications and then random test cases get created for the program. After this, a code corresponding to produce test cases and the execution traces of program are obtained. In last, compare a respective activity diagram with the execution traces, a minimum set of test cases can be

obtained according to test adequacy criteria. The consistency between behavior of UML activity diagram and program execution traces can be checked by this approach.

3.3. Test Case Generation from IOAD Model

Kim *et al.* [17] presented a methodology using I/O explicit Activity Diagram (IOAD) model for creation of test cases. This method has been proposed using an ordinary activity diagram which can be converted into a directed graph. The graph leads to creation of test cases for initial activity diagram. This transformation uses single stimulus principle. This method helps in optimization of the number of test cases keeping all required test cases. It reduces time and cost of the software development process without compromising quality.

3.4. Test Case Generation using Prototype Tool

Xu *et al.* [3] dealt with UML Activity diagrams and complex structures inside them like exception handlers. They improved the approach of deriving test scenarios (TSs) from the activity diagram. They described some systematic test coverage criteria. Their approach is fully automatic and accompanied by prototype tool, which results in fewer gaps between high level analysis and test.

3.5. Test Case Generation for Concurrent Activities

Sapna *et al.* [2] described the method for generating test scenarios in case of concurrent activities in the activity diagram. These scenarios are generated automatically. The main problem addressed by them is due to fork-join nodes as there is exponential growth in test scenarios due to them. In order to solve this problem they limited the increase of test scenarios by observing and controlling domain dependency among concurrent activities. This method defines dependencies among the activities inside fork-join pairs and after that the same information is used for generating feasible and required test scenarios. This information also helps in the design of specifications.

Nanda *et al.* [16] presented a technique for generation of test scenarios directly by the use of activity diagram, where the design is reused. One of great benefits of this approach is that reuse of design model helps in reducing the cost of building test models and defects. These defects can then be eliminated as early as possible. This

approach constructs test scenarios containing fork join pairs in activity diagram and deals with the complicacy of fork join pairs.

3.6. Test Case Generation using Sub-Activity Diagram

Fan *et al.* [4] presented a method for generating test cases. The whole activity diagram and its sub-activity are considered with test cases hierarchically. The sub-activity diagrams were generated as well. Their activity diagrams consisted of flowchart of various activities. In activity diagram, the sub-activity state decomposed the activity diagram into two parts namely atomic activity diagram and compound activity diagram. For test case generation purpose, dealing with the compound activity diagram is an important issue. In this research test cases from sub-activity diagram to compound activity diagram were generated hierarchically. It implements this method by considering the following factors:

i. functional decomposition
ii. integration testing using a bottom-up strategy
iii. Round-robin strategy.

The results showed that this technique generated lesser number of test cases than the method following complete combination approach. This approach also had good coverage of all the test cases. This paper presented the detailed approach for generating the integrated test cases by combining the whole activity diagram from sub activity diagram. This technique consists of two steps. With the help of activity diagram composition tree (ADCT) test cases are generated and by applying round-robin strategy.

3.7. Test Case Generation using TSGen Tool

Sun *et al.* [5] has implemented a tool named as TSGen (model driven testing tool) using which test scenarios for UML activity diagram which can be generated automatically. Various issues related with designing and implementations of this tool were also examined by them. Different test scenarios were generated by this tool when demanded. This tool also supports various kinds of concurrency coverage criteria. Type of concurrency coverage criterion affects the generation of test scenarios. Using this tool better schedule for testing can be planned in the early stages which further save the efforts. TSGen describes which business scenario should be tested in the implemented system.

3.8. Test Case Generation using ITM Approach

Nayak *et al.* [7] presented a new approach with respect to a given activity diagram. In this approach, test scenarios that are dependent on the control flow graph analysis are automatically generated. Firstly, an activity diagram classification of control flow constructs is introduced. Secondly, a new approach named transformation approach is introduced which converts activity model having the flat structure into structure having a well formed hierarchy called as Intermediate Testable Model (ITM). This model considers combinations of nested structure into account. ITM maintains a simple control flow and the test scenario which can be systematically generated from ITM. The test cases constructed are precise and complete. The actual test cases are made by enhancing each scenario with test data. The target system is tested by the test values provided in the above scenario. This approach has taken the activity diagram which is produced from use cases to model behavior of the system. It may be useful to add sequence diagrams for showing each and every activity in the use case for knowing accuracy and detailing.

3.9. Test Case Generation using Dynamic Slicing

Samuel *et al.* [12] proposed a novel test case generation approach based on dynamic slicing of UML activity diagrams. The flow dependency graph created the activity diagrams as intermediate graph for generating dynamic slices. An FDG has constructed statically which needs to be constructed once. For given inputs, dynamic slices can be created corresponding to the nodes. The dynamic slices are created using edge marking slicing algorithm. A dynamic slice will consider the dependencies among activities that occur during run time. Slices are created corresponding to conditional predicate at each activity edge from FDG of respective activity diagram. This approach automates the test case generation process with respect to each slice, and it achieves high-path coverage criterion.

3.10. Test Case Generation using Conditional Slicing

Ray *et al.* [10] have presented a novel approach of test case generation with the use of conditioned slicing known as a slicing framework. This framework was used for test case generation of underlying the activity diagram. This method applies conditioned slicing on a flow dependency graph for generating test cases. The number of test cases gets reduced while generation of all practically required tests cases. This approach

satisfies high-path coverage criterion. This paper has the objective which is to minimize test cases that cover the maximum paths.

3.11. Test Case Generation using DFS Method

Boghdady *et al.* [21] proposed model as an automated algorithm for generation of test cases from the activity diagrams. This model constructs the Activity Dependency Table (ADT) as an intermediate table. The Activity Dependency Graph (ADG) is a directed graph that can automatically be generated from a produced ADT table. Then it is accessed by applying the Depth First Search (DFS) to obtain complete test paths. The branch coverage criterion should be satisfied by the generated test cases. Cyclomatic complexity technique is used as a measure for validation of generated test cases. The combination of branch coverage criterion, condition coverage criterion, the cyclomatic complexity coverage criterion and the full path coverage criterion called hybrid coverage criterion have been satisfied by the proposed model. The introduced model enhances the quality of generated test cases by saving time and effort. This model has applied on three different systems for evaluation purpose to show its effectiveness.

Boghdady *et al.* [6] introduced an enhanced XML-based approach that helped in automated test cases generation from activity diagrams. The introduced methodology generates a table known as Activity Dependency Table (ADT) corresponding to every XML file. It includes all functionalities and dealing with the loops, decisions, fork, join, merge, object as well as conditional threads in the activity diagram but a reduced manner. Then Activity Dependency Graph (ADG) was constructed in addition with the ADT to obtain resulting test cases. To ensure satisfying a hybrid coverage criterion, the generated test paths were validated by the proposed model. The proposed model has been implemented on 30 different sizes of activity diagrams in different fields. For the proposed model, a qualitative and quantitative evaluation had also introduced. Thus the enhanced automated approach minimizes the search space and reduces the time spent in generation of the test cases which enhance the quality of generated test cases. Hence the overall performance of the testing process was optimized with respect to saving time and effort. In additional, the generated test cases can be implemented on a system under test using any automatic test execution tool.

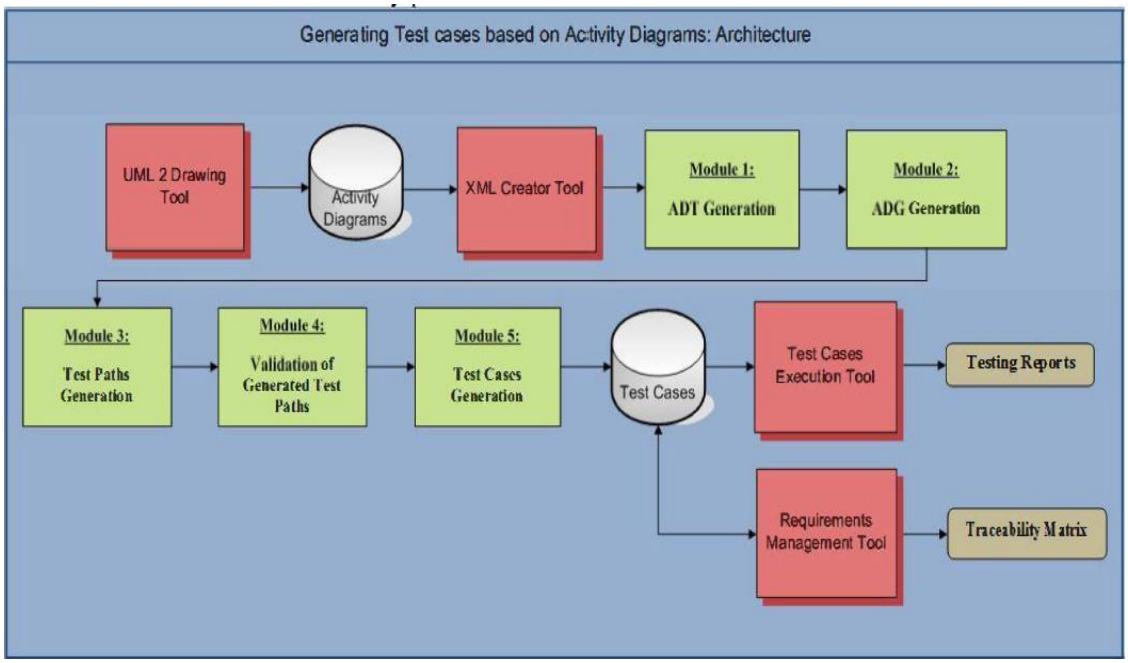


Figure 3.1: Boghdady et al. Model Architecture [6].

3.12. Test Case Generation using CQS Algorithm

Shirole *et al.* [14] presented an approach for generation of test cases for concurrency from UML diagrams such as sequence and activity diagrams. In their work, they have transformed sequence diagram into activity diagram. Then they implemented Concurrent Queue Search (CQS) algorithm to generate test sequences. CQS handles randomly concurrent tasks. The CQS algorithm generates test scenarios which can reveal data safety error and casual ordering errors in the presence of concurrency. Test scenario produced by this algorithm is better in comparision to DFS and BFS algorithm approach.

3.13. Test Case Optimization using Genetic Algorithm

Jena *et al.* [9] proposed a genetic algorithm for the purpose of test case generation from UML activity diagram. This approach helped in early fault detection and surely reduced in time, cost and effort. Activity Flow Table was generated from the model and then transformed into Activity Flow Graph (AFG). Then it used the activity coverage criteria for traversal of AFG and traversing test paths. This approach first used the Depth First Search (DFS) for the creation of test paths from AFG. These tests paths are used in creating test cases. The simple genetic algorithm used in this

approach optimizes the test cases. The model was only applied on ATM withdrawal system case study.

C. Sun *et al.* [5] proposed a transformation based approach for generation of scenario-oriented test cases for testing concurrent applications modeled by UML Activity Diagrams. The approach first converted a UML activity diagram specification into an intermediate representation with the help of transformation rules. Then a set of test scenarios were constructed from the intermediate representation according to the given concurrence coverage criteria. Finally, a set of test cases was derived from the constructed test scenarios. The approach implements the transformation to solve the nonstructural problem with activity diagrams and generated test cases in a controlled manner.

3.14. Test Case Generation using Model Based Testing

Chen *et al.* [18] proposed a methodology for automated generation of test cases based on different model checking techniques. It needs three steps: In the first step, automatically translate the activity diagram to formal modes based on coverage driven mapping rules. Second, the procedure for automatic generation is presented according to error models. Finally, various models based checking technique for efficient generation of test cases was applied. The results revealed that this approach reduces the validation effort drastically by minimizing the test case generation time and required number of test cases to fulfill functional coverage criteria.

3.15. Test Case Generation using Evolutionary Algorithm

Shirole *et al.* [22] came out with an idea of the evolutionary algorithm (EA) for the purpose of test case generation for UML activity diagram. Firstly, an Extended Control Flow Graph (ECFG), as an intermediate model was generated from an activity diagram. This intermediate model was used to generate test data and corresponding test cases. The EA algorithm was guided by an objective function to generate divergent test scenarios concerning issues of guard constraints and concurrency. The result revealed that this approach produced feasible test cases to satisfy transition and concurrency coverage. This approach was well suited for mission critical applications where concurrency is necessary.

3.16. Test Case Generation using a Combination of Use Case and Activity Diagram

Vieira *et al.* [24] presented an approach to combine existing techniques for data and graph coverage. To present data into UML model, it used the category partition method. UML Use Case and Activity Diagram were used to describe which functionality needs to be tested and how to test them. This combination had the capability to produce huge number of test cases. For managing number of test cases, there are two ways which offer this approach. Firstly, custom annotations and guards used the category partition data which permits the designer to control over possible and impossible paths. Secondly, automation gave different configurations for data and graph coverage. The objective of this approach was to improve automation on testing. This approach was evaluated for testing a Graphical User Interface (GUI).

3.17. Test Case Generation using Regression Testing

Ye *et al.* [29] presented an automatic approach for regression test case generation based on activity diagram. This approach had chosen path coverage as criteria for regression testing. The execution of software with certain set of test cases got execution traces. Based on the execution traces and the old activity diagram, a new activity diagram reflecting the behavior of modified software was constructed automatically. This helped in identification of affected paths as well as new paths. Finally, regression test cases were generated for these paths by applying an execution based approach. These test cases used to make up the regression testing suite.

3.18. Test Case Generation using Post Optimization Algorithm

Nguyen *et al.* [25] proposed a novel technique that integrates model based and combinatorial testing for generation of effective test cases from a model. This approach started with finite state model and employed model based testing for generation of test paths that represent sequence of events to be performed against the system under test. Then transform the test paths to classification trees. Using t-way combinatorial criteria, finally executable test cases were generated. Then post optimization algorithm was applied to minimize the number of test cases while satisfying the combinatorial coverage criteria. Test combinations were finally converted to an executable format. Example- Junit.

3.19. Test Case Generation using a Combination of Activity Diagrams and Communication Diagrams

Swain *et al.* [26] presented an idea of prioritization technique for cluster level test scenarios that were generated from UML Communication diagram and Activity diagram. Initially a tree representation was build from both the communication and activity diagrams. Afterwards an intermediate representation is constructed and is named as COMMACT tree. Conditional predicates are selected from the COMMACT tree using post order traversal. This tree is further used to generate test scenarios. Method activity sequence, associated objects and other needed information is obtained from these test scenarios. After that, those test scenarios were taken that satisfy message activity path adequacy criteria. Redundant test scenarios and adequacy test coverage is the main output of this approach.

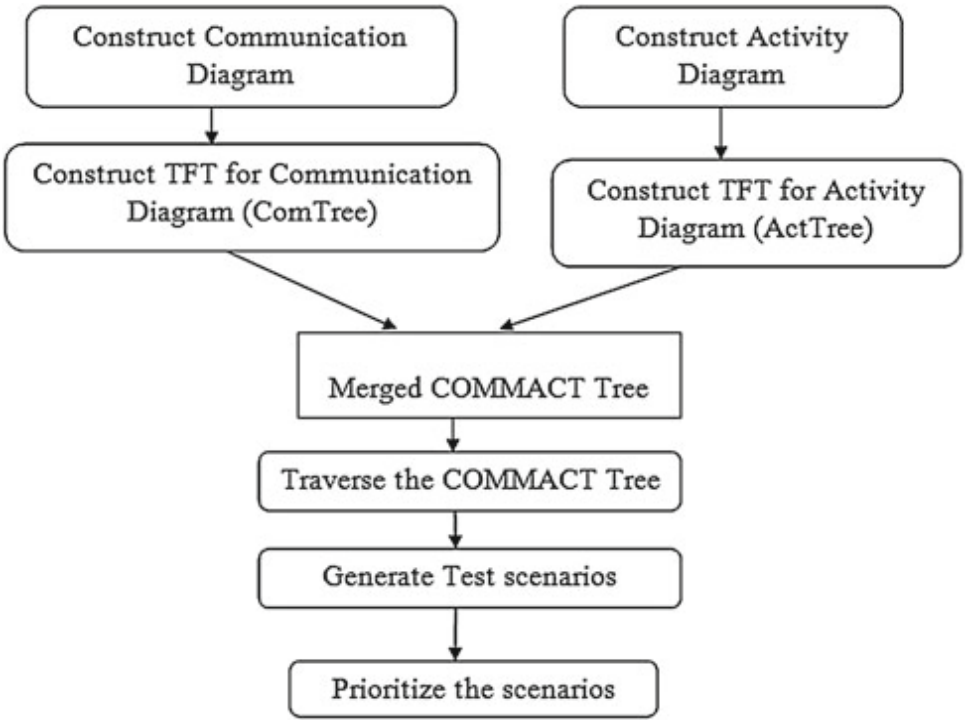


Figure 3.2: Swain et al. PRITECOMMACT [26].

3.20. Test Case Generation using Function Minimization Methodology

Samuel *et al.* [27] presented a method for generation of cluster level test cases for a given UML Communication diagram. In this approach, first build a tree representation for communication diagram. Then apply post-order traversal for selection of conditional predicates from communication diagram. Next, apply the

function minimization method for generation of test data. The generated test scenarios must satisfy message adequacy path coverage criteria as well as branch coverage criteria. This technique has been successfully implemented on various examples.

3.21. Test Case Generation using a Combination of Activity Diagrams and Sequence Diagrams

Sumulatha *et al.* [36] presented a novel approach for generation of test cases by combining UML diagrams. Mainly activity and sequence diagrams are considered in this approach. Sequence of messages exchanged between objects in order to perform the functionality of the system is described by sequence diagram. On the other hand activity diagram represent the flow of various activities without exchanging any messages. Both the diagrams are essential for representing the dynamic behavior. Proposed approach works in two steps. Activity and sequence diagrams are firstly converted into activity diagram graph and sequence diagram graph respectively. In second step both the graphs are combined to generate an activity-sequence graph. After that the Breadth first search (BFS) is applied on activity sequence graph to generate test cases and to create all probable paths. Using this approach, the complex software can be represented with clarity and in an efficient manner, which in turn improves the efficiency, accuracy and quality.

3.22. Test Case Generation using Priority Technique

Fernandez *et al.* [35] had proposed a generation of complete set of functional test scenarios from UML activity diagram. Software risk information is considered as main parameter for assigning priority to generated test scenarios. In order to get the risk information all the factors associated with risk and affect of these parameters on risk is taken into account. Test effort can be rearranged with respect to risk assumed by end user on the basis of risk factor.

3.23. Test Case Generation using Dynamic Slicing

Samuel *et al.* [27] used the concept of slicing for generating test cases from UML sequence diagram. Guard conditions are identified and dynamic slices are generated with respect to each conditional predicate. Test cases are generated automatically which are further used in automatic testing of program. Slice coverage criteria is used as a test adequacy criteria for generating test cases from sequence diagram.

3.24. Test Case Generation using a Combination of Use Case Diagrams and Sequence Diagrams

Swain *et al.* [31] had proposed an approach merging the information of use case and sequence diagram for integration testing. UML diagrams are taken as input in this approach and sequence diagram is considered in order to generate test cases. While deriving test plan, use case sequence is derived as its part. After this sequence diagrams with the sequence of use cases is used for getting details of message sequences which helps in generating test cases. Interaction faults, message sequence faults, operation faults as well as synchronization faults are the main target which are uncover by this approach. Generated test cases were then used in software testing for all the use case scenarios together.

3.25. Test Case Optimization Based on Heuristic Rule

Sapna *et al.* [15] presented an automated strategy for selecting test scenarios created from activity diagram basis on Levenshtein distance. It measures the dissimilarity between test scenarios. The objective of this paper is to select less similar test scenarios and provide maximum coverage. This work is different from previous work where random selection was done. This paper has taken criteria of minimum Levenshtein distance for picking the scenario pair and chooses one of them based on some heuristics such as random, priority or assumption based. This approach is continued until threshold being met. Heuristic applied for selection process offers advantage in producing better results.

Biswal *et al.* [28] had taken the UML activity diagram for generating test scenarios for object oriented programs. First step is to generate the activity diagram and then the test cases are generated randomly. Afterwards the program is run according to generated test cases and the corresponding program execution traces can be obtained. Now, these traces are compared with the constructed activity diagram. Heuristic rule is used in this approach for reducing the number of test cases so that it can satisfy the test case adequacy criteria. Simple path coverage is considered as test adequacy criteria for generation of automated test scenarios. Activity, sequence and class diagram are taken into account for generating test cases. Advantage of using this approach is that complicity of nested fork-join pair is handled in an efficient way. Transition coverage criteria achieved is another advantage of this approach.

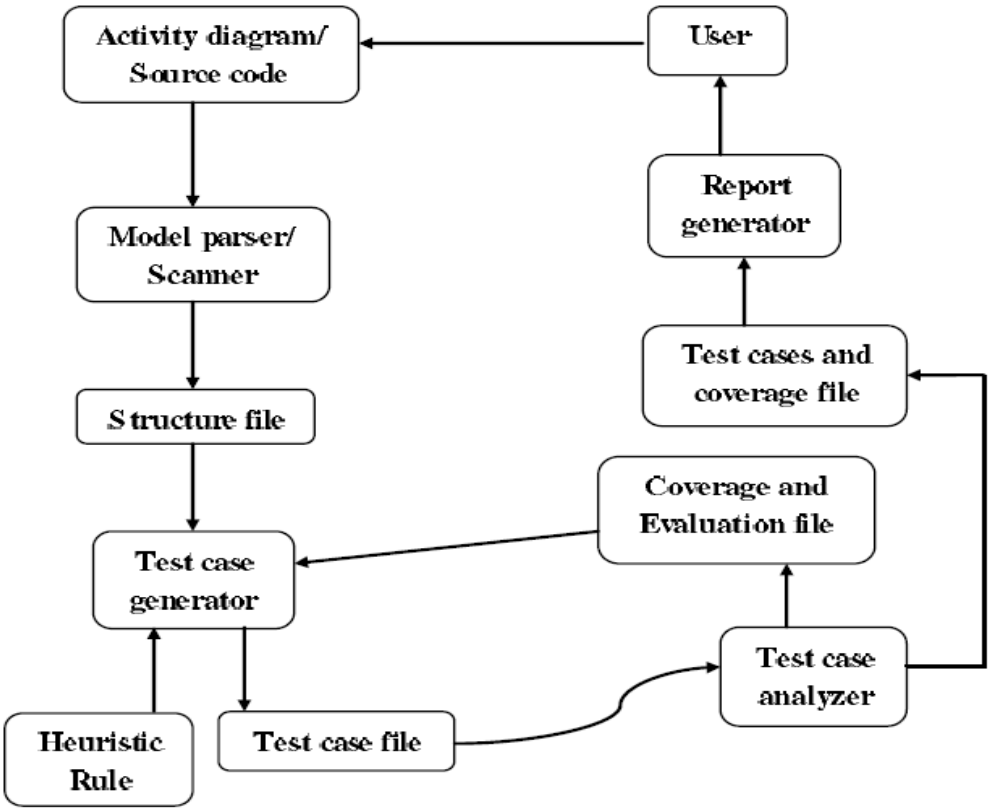


Figure 3.3: Biswal et al. System Model [28].

CHAPTER 4
PROBLEM STATEMENT

In the previous researches, test paths have been generated by using two approaches: Depth First Search (DFS) and Breadth First Search (BFS). The test paths generated by BFS algorithm are exponential including useful and useless test paths. The proposed work is to automated generation of test scenarios from activity diagram using DFS method. This approach helps in eliminating useless test cases and reduces the time complexity. This approach also helps in synchronization between activities. This approach is implemented on JDK 7 version and used the Net beans framework of 7.4 version. Following are the objectives of the thesis:

i. To analysis various techniques used for test generation in activity diagram, communication diagram, use case diagram and sequence diagram.
ii. To propose an approach for test case generation of test case generation of activity diagram using depth first search method.
iii. To implement the proposed approach by generating automated test paths from the activity flow graph using depth first search method. The approach is proposed that uses TCAFG algorithm which compares the string and then according to this, automated test paths are generated. Then test cases are created which cover all the test paths.
iv. The proposed approach reduces the total time required for executing test paths. Thus, it improves performance of test case generation.

CHAPTER 5

PROPOSED WORK

In this chapter, the proposed approach has been discussed to generate the test case from activity diagram. This chapter has been focused on analysis of gaps during the literature review of test case generation algorithms.

5.1. Gap Analysis

Test paths can be generated by applying Depth First Search (DFS) method or Breadth First Search (BFS) method. In DFS, the path is generated by starting with the root node and covering all the branches before backtracking. In BFS, the path is generated by exploring all the nodes at one level completely and then moves into the next level and so on. This is also called level order traversal. The limitation of this algorithm is synchronization between activities and it leads to generate an exponential number of test cases which increases the time complexity. The paths to be tested is coming out in exponential, out of which very few are useful. There is a need of tracing only useful ones.

With this problem, it needs to put focus on efficient test path generation from the activity diagram using depth first search method. By using this approach, the useless test paths are eliminated which in turn reduces the time complexity.

5.2. Proposed Technique

In the proposed technique, an algorithm Test Case for Activity Flow Graph (TCAFG) for generating test cases has been developed under this research. TCAFG significantly reduces time complexity. It includes following steps:

5.2.1. UML Activity Diagram

An activity diagram consists of flowchart of various activities with transition among them [19]. An activity diagram is used for modeling the dynamic aspects of the system. There are various tools such as Visual Paradigm, Rational Rose and Magic Draw etc for generation of UML diagram. In this approach, Visual Paradigm tool is used for generating UML diagram.

5.2.2. XMI Generation

UML diagram cannot be in readable format. For taking it as an input form, convert

the activity diagram into XML Metadata Interchange (XMI) format which is in readable format and can be taken as an input for further processing. For doing the conversion, there is an inbuilt functionality in Visual Paradigm tool. XMI is a standard that enables to express objects using Extensible Markup Language (XML), the. XMI specifies universal format for representing data on the World Wide Web how to produce XML schemas from UML models XMI provides an infrastructure for advanced features such as extensions to data, cross-file linking and identifying objects in various ways. One of the great benefits of XMI is that you can use XMI software without becoming an XML expert.

5.2.3. XMI parsing using SAX parser

Two common APIs that are used for reading XMI file are the Document Object Model (DOM) and the Simple API for XML (SAX). The only drawback to using these APIs with XMI documents is that you need to ensure that the documents you produce are valid XMI documents, and you need to know about the format of XMI documents in order to get the data from them.SAX to read an XMI document and use the data it contains to create corresponding instances of Java classes and set their fields. By using SAX, it can be decide whether the data from the XML file will be put into a data structure in memory or not. Therefore, the applications that read XML files may use less memory by using SAX than usage of DOM. SAX parse uses the parse method to create an instance of class that helps in parsing the file.

5.2.4. Generation of Activity Flow Graph

Information about the node generated by parser will be stored into separate .txt file. Pass the text file obtained from step 3 to graphics class which is included into swing component. Swing provides GUI for Java program. The class calls paint () to draw nodes and define relationship between them.

5.2.5. Generation of Activity Dependence Table

Activity Dependency Table (ADT), an intermediate table is generated from Activity Flow Graph (AFG). In ADT, all the dependencies are considered in the form of parent and child nodes. Also the independent probabilistic weights are assigned to child nodes for determining the probability of each child node.

5.2.6. Test Path creation

The test paths are generated using the TCAFG algorithm. The approach use DFS method for traversing activity flow graph and generating test paths which guarantees visiting all nodes of activity flow graph.

5.2.7. Test Cases

After generating test paths, test cases are created from the paths. The test cases are generated on the basis of covering test paths.

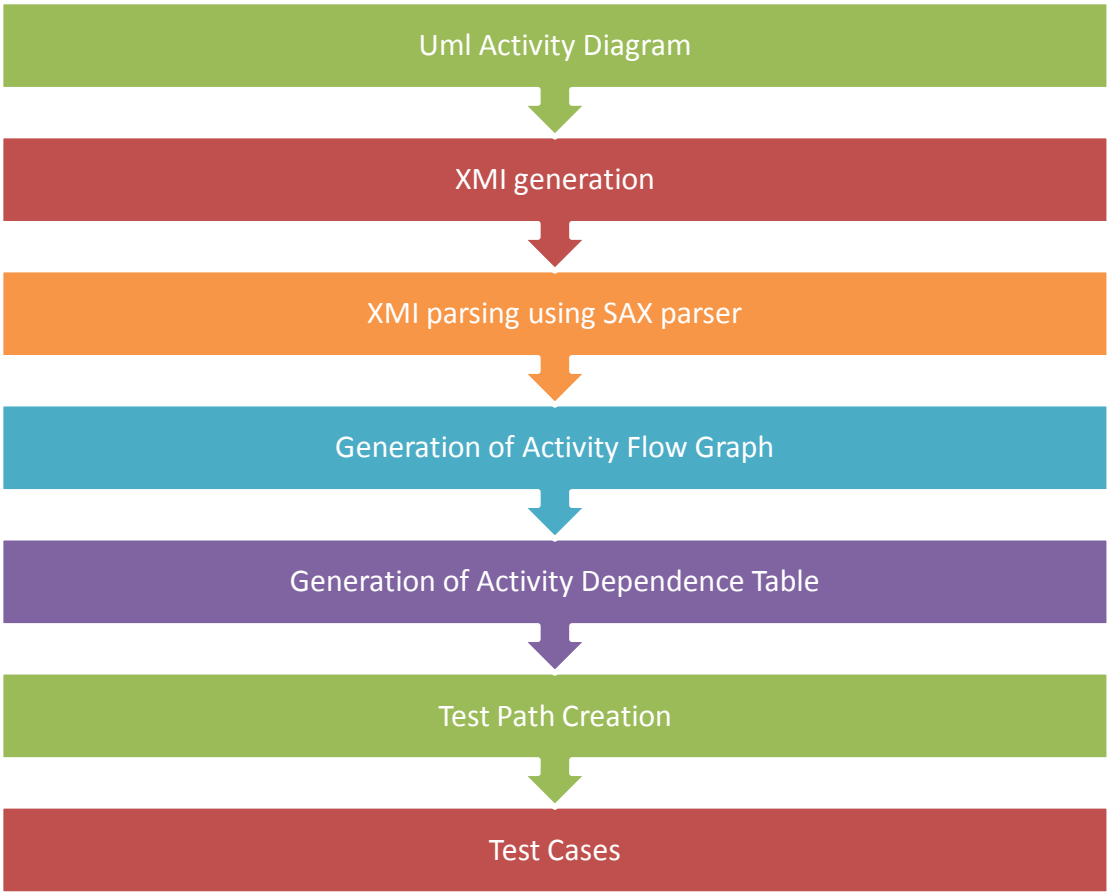


Figure 5.1.: Block Diagram for Proposed Approach.

5.3. Pseudo Code for TCAFG Algorithm

This section explains how this algorithm actually works. Here, a pseudo code is provided which explains the working of this algorithm.

Proposed Algorithm:

Step 1: Initially, a XMI file is taken and converted into text (string) file using SAX parser.

Step 2: After that two String Arrayist is used, first Arraylist stores information of parent nodes while second Arraylist stores information of child nodes.

Step 3: Make Flow graph using paint method in JAVA.

Step 4: For generating automated test paths, there is made use of 15 Arraylist, where each Arraylist contain parents and its child information.

Step 5: Apply the DFS method for generating test paths.

Step 6: Test cases are generated on the basis of coverage of test paths.

Step 7: Apply the linear search algorithm on proposed approach and find less execution time in comparison with depth first search method.

5.4. Implementation

The approach is implemented on JDK 7 version and used the Net beans framework of 7.4 versions. The implementation part has been divided into various steps.

The first step is to draw a UML diagram and transform it into XMI file. This can be done by making use of tools such as Magic draw, Rational Rose, Visual Paradigm for UML [32] etc. Here, Visual Paradigm for UML 10.0 version has been used to draw an activity diagram. Figure 5.2 shows the example of library issue book activity diagram.

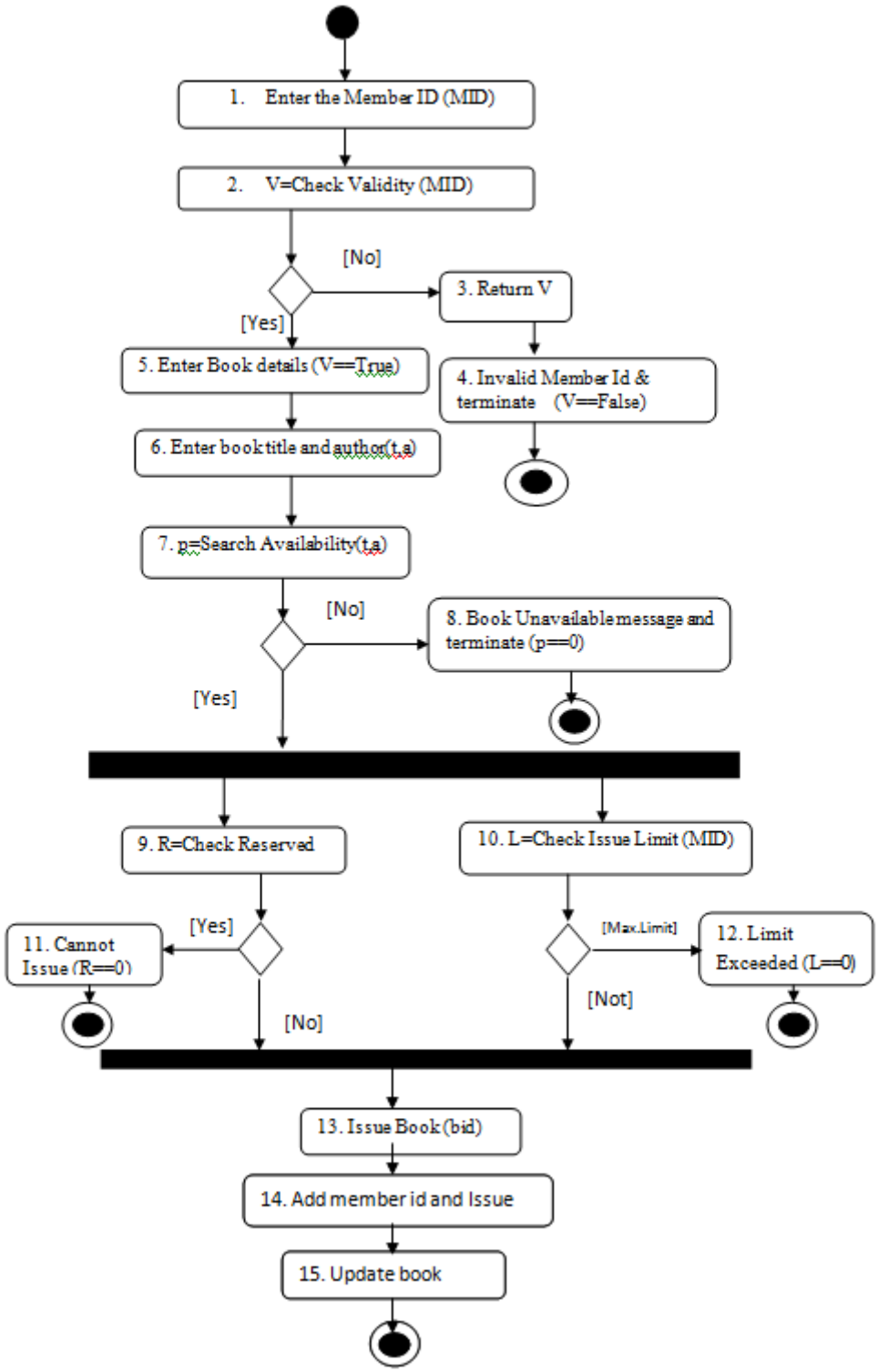
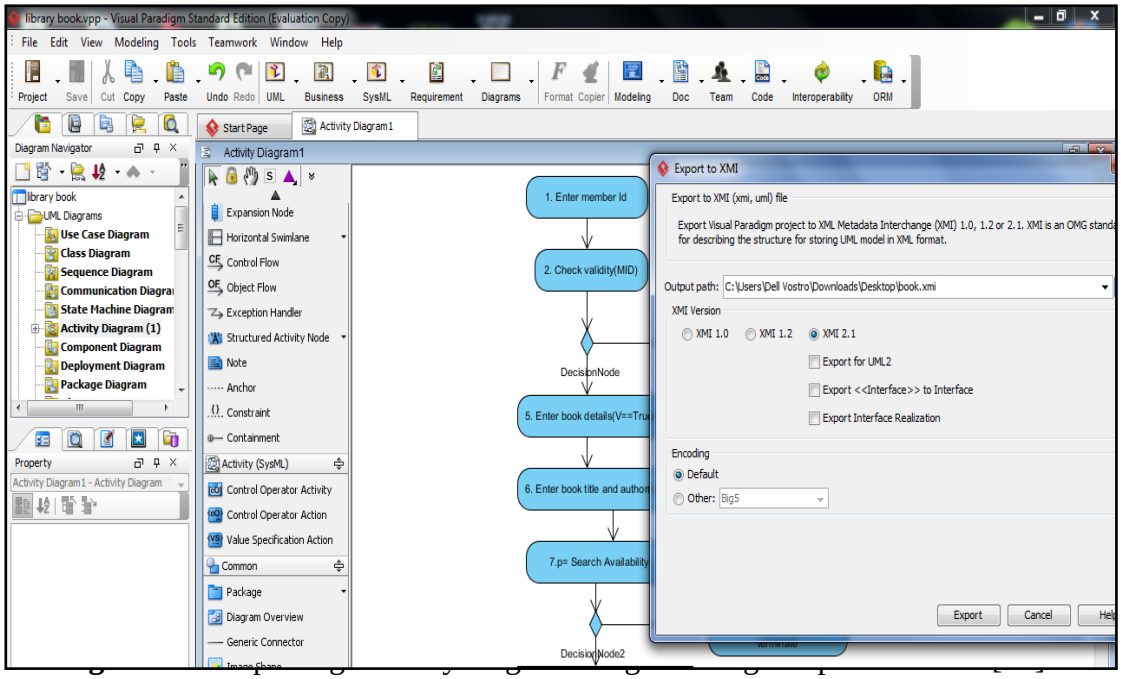


Figure 5.2: Activity Diagram of Library Issue Book [32].

UML activity diagram (helps in visualizing the sequence of activities involved in the control flow) is describing the scenario. The visual paradigm tool can also be used for conversion of the activity diagram into XMI format as shown in figure 5.3.



Visual Paradigm tool will generate the XMI format in the form as shown in Figure 5.4. It depicts XMI contains information about activity diagram such as node name with distinct ids, types and value etc. The purpose of converting UML diagram into XMI file is to obtain a readable format and provide platform independency.


```
<?xml version="1.0" encoding="UTF-8"?><xmi:XMI xmi:version="2.1" xmlns:uml="http://schema.omg.org/spec/UML/2.0" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
<xmi:Documentation exporter="Visual Paradigm" exporterVersion="7.0.2"> <xmi:Extension extender="Visual Paradigm">
<nickname exportedFromDifferentName="false"/> </xmi:Extension> </xmi:Documentation> <xmi:Extension extender="Visual Paradigm">
<projectProperties> <projectProperty name="author" value="DELL'S PC"/> </projectProperties> <ProjectOptions>
<IDPositionRotation> <Positions> <Position id="BPTask"
value="\u0004"/> <Position id="BPSubProcess" value="\u0004"/> <Position
id="BPGateway" value="\u0004"/> <Position id="BPPool" value="\u0002"/>
<Position id="BPLane" value="\u000b"/> <Position id="BPDataObject" value="\u0008"/>
<Position id="BPStartEvent" value="\u0004"/> <Position id="BPIntermediateEvent" value="\u0004"/>
<Position id="BPEndEvent" value="\u0004"/> <Position id="BPGroup" value="\u0004"/>
<Position id="BPChoreographyTask" value="\u0004"/> <Position
id="BPChoreographySubProcess" value="\u0004"/> <Position id="BPCallActivity" value="\u0004"/>
<Position id="BPCallChoreographyActivity" value="\u0004"/> <Position id="BPTextAnnotation" value="\u0004"/>
<Position id="BPSequenceFlow" value="\u000e"/> <Position id="BPMessageFlow"
value="\u000e"/> <Position id="BPAssociation" value="\u000e"/> <Position
id="BPDataAssociation" value="\u000e"/> </Positions> <Positions>
<Rotation id="BPTask" value="\u0000"/> <Rotation id="BPSubProcess" value="\u0000"/>
<Rotation id="BPGateway" value="\u0000"/> <Rotation id="BPPool" value="\u0000"/>
<Rotation id="BPLane" value="\u0000"/> <Rotation id="BPDataObject" value="\u0000"/>
<Rotation id="BPStartEvent" value="\u0000"/> <Rotation id="BPIntermediateEvent" value="\u0000"/>
<Rotation id="BPEndEvent" value="\u0000"/> <Rotation id="BPGroup"
value="\u0000"/> <Rotation id="BPChoreographyTask" value="\u0000"/>
<Rotation id="BPChoreographySubProcess" value="\u0000"/> <Rotation id="BPCallActivity" value="\u0000"/>
<Rotation id="BPCallChoreographyActivity" value="\u0000"/> <Rotation id="BPTextAnnotation"
value="\u0000"/> <Rotation id="BPSequenceFlow" value="\u0004"/> <Rotation
id="BPMessageFlow" value="\u0004"/> <Rotation id="BPAssociation" value="\u0004"/>
<Rotation id="BPDataAssociation" value="\u0004"/> </Positions> </IDPositionRotation>
<CodeSyncOptions addImportInsteadOfQualifiedName="true" autoRealizeInterface="false" braceOnAnnotationTypeDeclaration="\u0000"
braceOnClassDeclaration="\u0000" braceOnConstructorDeclaration="\u0000" braceOnEnumDeclaration="\u0000" braceOnMethodDeclaration="\u0000"
collectionType="\u0001" commitCodeSynchronizationPathsSetting="false" dataTypeForAssociationCollectionType="java.util.Collection" defaultAttributeType="int"
defaultOperationReturnType="void" defaultParameterType="int" generateAnnotationInCodeConvention="true" generateAnnotationOn="\u0000"
generatePreAndPostCondition="\u0000" getterTemplate="return this.{attribute};" importQualifiedNameForReferenceType="false" indentationPolicy="\u0000"
```


Figure 5.4: XMI File of Activity Diagram.

The interface for selecting file is shown as in figure 5.5.

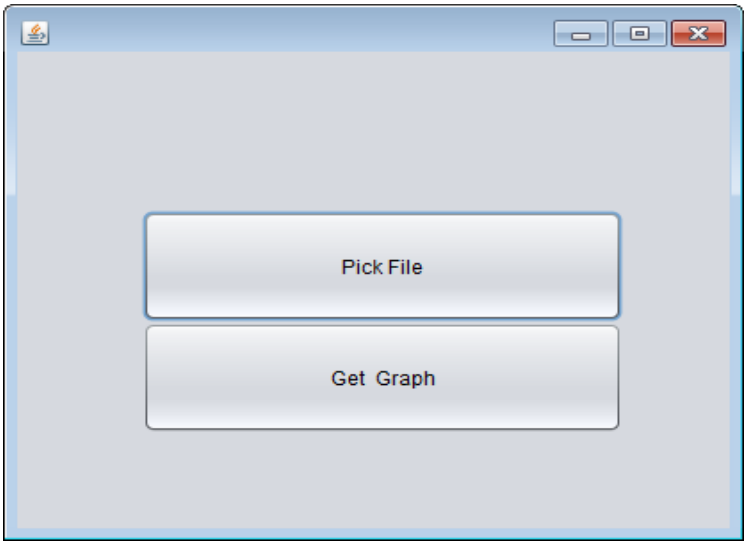


Figure 5.5: Interface

The next step is a selection of XMI file where it is located to get an Activity Flow Graph (AFG) is shown in figure 5.6.

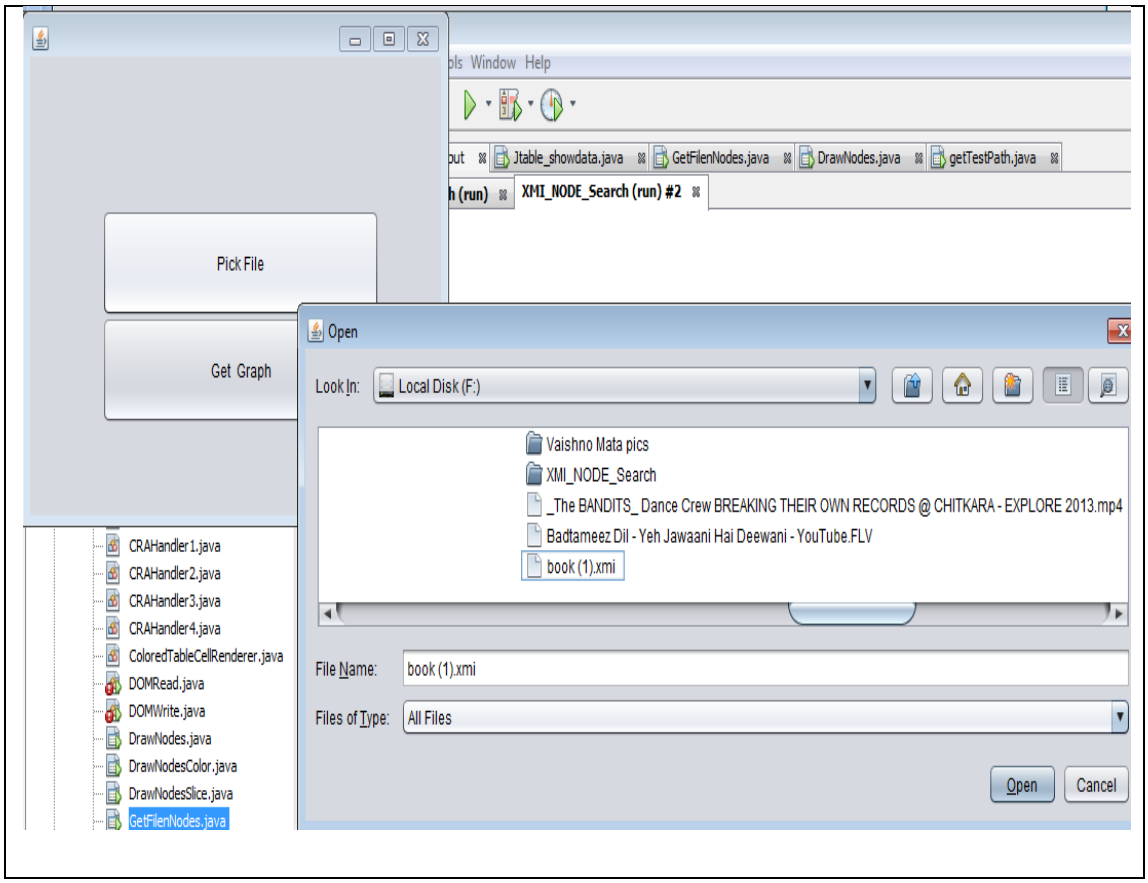


Figure 5.6: Selection of the XMI File.

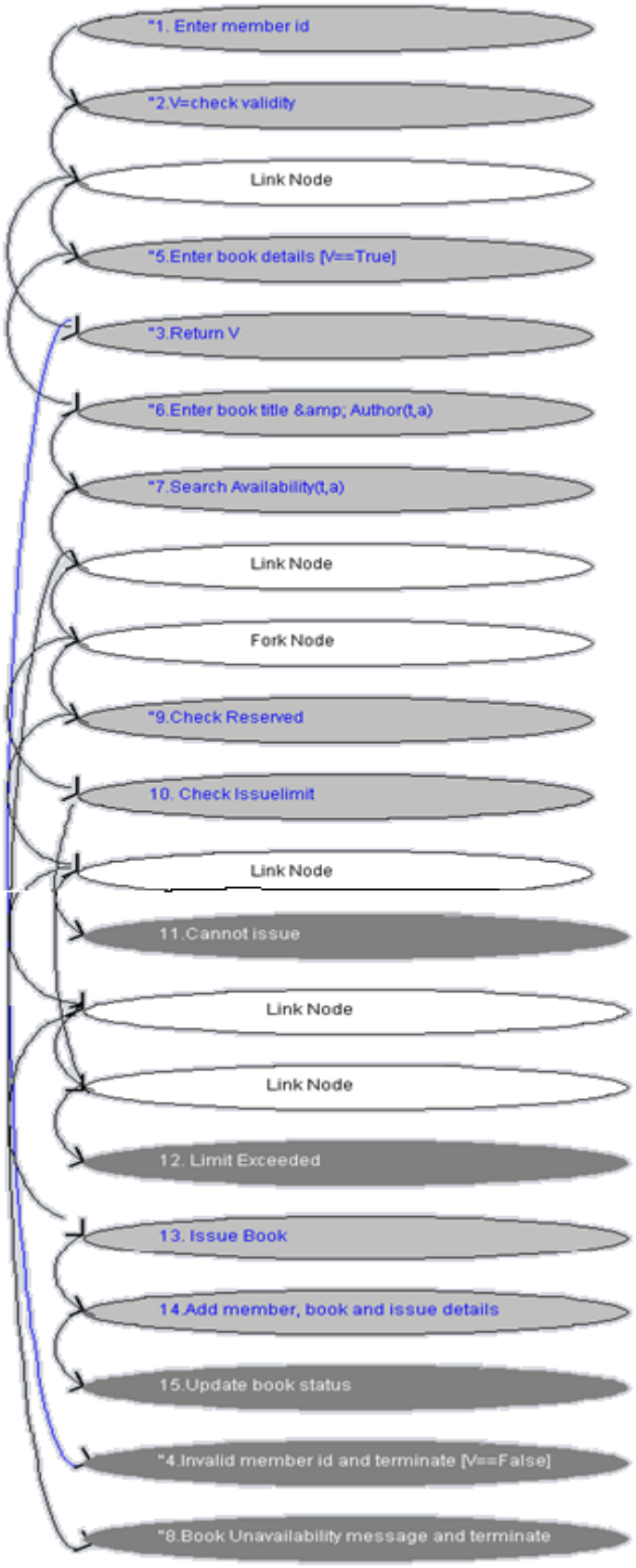


Figure 5.7: Activity Flow Graph.

.

Next step is parsing the XMI file to fetch the relevant information. The java SAX API parser is used to parse the XMI file. The parser helps in identifying the nodes and their relations. SAX parser will generate a text file that contains information about the nodes and their relation. All the information of nodes generated by parser will be stored in a separate .txt file.

Pass the text file into a class which uses graphics included in a swing component. Swing provides a framework of Graphical User Interface (GUI) for Java program. As shown in figure 5.7, the Activity Flow Graph is constructed which contains nodes and defined relationships.

Parent Node	Child Node	Weight
"1. Enter member id	.	
	"2.V=check validity	1.0
"2.V=check validity	.	
	"5.Enter book details [V==True]	0.5
	"3.Return V	0.5
"5.Enter book details [V==True]	.	
	"6.Enter book title & Author(t,a)	1.0
"3.Return V	.	
	"4.Invalid member id and terminate [V==False]	1.0
"6.Enter book title & Author(t,a)	.	
	"7.Search Availability(t,a)	1.0
"7.Search Availability(t,a)	.	
	"9.Check Reserved	0.3333333333333334
	10. Check Issuelimit	0.3333333333333334
	"8.Book Unavailability message and terminate	0.3333333333333334
"9.Check Reserved	.	
	11.Cannot issue	0.5
	13. Issue Book	0.5
10. Check Issuelimit		
	12. Limit Exceeded	0.5
	13. Issue Book	0.5
13. Issue Book	.	
	14.Add member, book and issue details	1.0
14.Add member, book and issue details	.	
	15.Update book status	1.0

Figure 5.8: Activity Dependency Table.

The Activity Flow Graph gets converted into Activity Dependence Table (ADT) consisting parent child relationships. The probabilistic weights are assigned to child nodes. The activities are given independent probabilities as shown in figure 5.8.

Next step is of generation of test scenarios from Activity Flow Graph. Activity Flow Graph is traversed by using depth first search method. During traversal test paths are generated which guarantees coverage of all nodes in the activity flow graph. All the possible generated test paths are shown in figure 5.9.

Test Path 1

"1. Enter member id --> "2.V=check validity --> "5.Enter book details [V==True] --> "6.Enter book title & Author(t,a) --> "7.Search Availability(t,a) --> "9.Check Reserved --> 11.Cannot issue

Test Path 2

"1. Enter member id --> "2.V=check validity --> "3.Return V --> "4.Invalid member id and terminate [V==False]

Test Path 3

"1. Enter member id --> "2.V=check validity --> "5.Enter book details [V==True] --> "6.Enter book title & Author(t,a) --> "7.Search Availability(t,a) --> 10. Check Issuelimit --> 12. Limit Exceeded

Test Path 4

"1. Enter member id --> "2.V=check validity --> "5.Enter book details [V==True] --> "6.Enter book title & Author(t,a) --> "7.Search Availability(t,a) --> "8.Book Unavailability message and terminate

Test Path 5

"1. Enter member id --> "2.V=check validity --> "5.Enter book details [V==True] --> "6.Enter book title & Author(t,a) --> "7.Search Availability(t,a) --> "9.Check Reserved --> 13. Issue Book -->

14.Add member, book and issue details --> 15.Update book status

Test Path 6

"1. Enter member id --> "2.V=check validity --> "5.Enter book details [V==True] --> "6.Enter book title & Author(t,a) --> "7.Search Availability(t,a) --> 10. Check Issuelimit --> 13. Issue Book -->

14.Add member, book and issue details --> 15.Update book status

Figure 5.9: Generated Test Paths.

After generating test paths, test cases are created from the paths. In test case, the certain inputs are provided and check whether output comes according to input. The test cases are generated on the basis of covering test paths as shown in table 5.1.

Table 5.1: Observed Test Cases.

Observed Test Cases						
Test Case:	Enter Member Id:	Available Bo...	Limit	Enter Book Detail:	Expected Result:	Actual Result:
1	224	500	10	N/A	Invalid member id	Invalid Member id
2	123	0	08	Engineering Mathematics	UnAvailable book	Unabailable and terminate
3	345	500	10	Yashwant Kanetkar	Book Issued	Book Reseved and terminate
4	323	500	0	Balaguruswami	Book Issued	Limit Exceeded
5	234	500	10	Chemistry	Book Issued	Update book status
6	143	500	09	Physics	Book Issued	Update book status

The execution time for depth first search time and linear search applied on the proposed approach is shown in table 5.2.

Table 5.2: Results Obtained from Approach.

Comparison	Linear Search	Depth First Search
No. of test cases	6	6
Time consumed (in sec.)	0.82	0.6

The comparison between Linear Search and Depth First Search on the basis of time consumed is shown in figure 5.10.

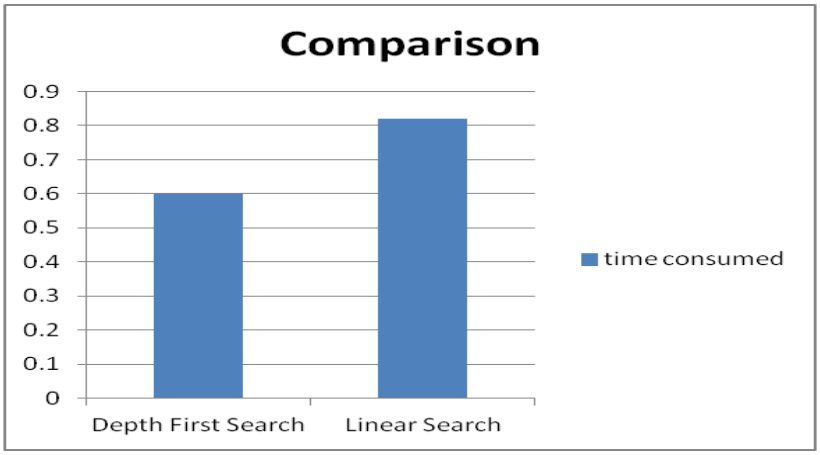


Figure 5.10: Comparison between Linear Search and Depth First Search.

CHAPTER 7
CONCLUSION AND FUTURE SCOPE

8.1. Conclusion

The proposed approach can generate efficient test cases with lesser effort. This helps in saving time and increases the quality of generated test cases. The overall testing process performance can be improved using this approach. The proposed model uses only activity diagrams as input. It generates a directed graph called Activity Flow Graph (AFG) and an intermediate table called Activity Dependency Table (ADT) automatically for each activity diagram. The ADT considers all the dependencies in the form of parent child nodes and independent probabilistic weights are assigned.

TCAFG algorithm is used for traversing AFG in order to extract all the possible test paths. The proposed algorithm is based on DFS method. It helps in reducing time, effort, and cost consumption. The test paths are finally used to generate the final test cases. Then the linear search algorithm was applied and studied against proposed approach for comparison. There is a clear improvement in execution time for proposed approach against linear algorithm.

8.2. Future work

The proposed technique has focused on automated generation of test paths from activity diagram but still there are the following points that can be explored further.

i. The execution time for generation of test paths can further be reduced. The proposed approach can be applied to other algorithms like binary search, bubble sort etc. which can further reduce the execution time for test paths in these algorithms.

ii. The concurrency issue can be worked upon in future work.

iii. Probabilities that are assigned in activity dependence table can be used for test case generation purpose.

REFERENCES

[1] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong and Z. Guoliang," Generating Test Cases from UML Activity Diagram based on Gray-Box Method, *In Proceedings of the 11th Asia-Pacific Software Engineering Conference(APSEC)*, IEEE, Busan, Korea, 1-8, 2004.

[2] P.G.Sapna and H. Mohanty, "Automated Scenario Generation based on UML Activity Diagrams," *In International Conference on Information Technology*, IEEE, Bhubaneswar, India, 209-214, 2008.

[3] D. Xu, W. Liu, Z. Liu and Philbert, "Tool Support to Deriving Test Scenarios from UML Activity Diagrams," *In International Symposium on Information Science and Engineering*, IEEE, Shanghai, China, 73-76, 2008.

[4] X. Fan, J. Shu, L. Liu, Q. Liang, "Test Case Generation from UML Subactivity and Activity Diagram," *In Second International Symposium on Electronic Commerce and Security*, IEEE, Nanchang, China, 244-248, 2009.

[5] C. Sun, B. Zhang, J. Li, "TSGen: A UML Activity Diagram-based Test Scenario Generation Tool," *In International Conference on Computational Science and Engineering*, IEEE, Vancouver, Canada, 853-858, 2009.

[6] P. N. Boghdady, N. L. Badr, M. A. Hashim and M. F. Tolba, "An Enhanced Test Case Generation Technique Based on Activity Diagrams," *In International Conference on Computer Engineering & Systems (ICCES),* IEEE, Cairo, Egypt, 289-294, 2011.

[7] A. Nayak ·and D. Samanta, "Synthesis of test scenarios using UML activity diagrams," *Software & Systems Modeling,* Springer-Verlag, London, 10(1), 63-89, 2009.

[8] Y. M. Malik, **"**Model Based Testing: An Evaluation**,**" Master of Science, Thesis, Blekinge Institute of Technology, 2010.

[9] A. K. Jena, S. K. Swain and D. P. Mohapatra, "A Novel Approach for Test Case Generation from UML Activity Diagram," *In International Conference on Issues*

and Challenges in Intelligent Computing Techniques (ICICT), IEEE, Ghaziabad, India, 621-629, 2014.

[10] M. Ray, S. S. Barpanda, D. P. Mohapatra, "Test Case Design Using Conditioned Slicing of Activity Diagram*," International Journal of Recent Trends in Engineering*, 1(2), 117-120, 2009.

[11] Karambir and K. Kaur, **"** Survey of Software Test Case Generation Techniques," *In International Journal of Advanced Research in Computer Science and Software Engineering* , 3(6), 937-942, 2013.

[12] P.Samuel and Rajib Mall, "Slicing-Based Test Case Generation from UML Activity Diagrams," *In ACM SIGSOFT Software Engineering Notes*, 34(6), pp. 1-14, 2009.

[13] C. Mingsong, Q. Xiaokang and L. Xuandong, "Automatic Test Case Generation for UML Activity Diagrams," *In Proceedings of the international workshop on Automation of software test*, ACM, Shanghai, China, 2-8, 2006.

[14] M. Shirole and R. Kumar, "Testing for Concurrency in UML Diagrams," *In ACM SIGSOFT Software Engineering Notes*, 37(5), 1-8, 2012.

[15] P.G.Sapna and H. Mohanty, "Automated Test Scenario Selection Based on Levenshtein Distance," *Distributed Computing and Internet Technology Lecture Notes in Computer Science*, Bhubaneswar, India, 5966, 255-266, 2010.

[16] P. Nanda, D. P. Mohapatra and S. K. Swain, "Generation of Test Scenarios Using Activity Diagram," *In Proceedings of SPIT-IEEE Colloquium and International Conference*, Mumbai, India, 4, 69-73, 2008.

[17] H. Kim, S. Kang, J. Baik, I. Ko, "Test Cases Generation from UML Activity Diagrams," *In Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, IEEE, Qingdao, China, 556-561, 2007.

[18] M. Chen, X. Qiu, W. Xu, L. Wang, J. Zhao and X. Li, "UML Activity Diagram Based Automatic Test Case Generation For Java Programs*," Computer Journal, Oxford University Press*, 52(5), pp. 545-556, 2007.

[19] A. Abdurazik and J. Offutt, "Generating test cases from UML specifications," *In Proceedings of the Second International Conference Fort Collins, George Mason University*, USA, 1723, 416-429 1999.

[20] M. Prasanna, S. N. Sivanandam, R. Venkatstesan and R. Sundarrajan, "A Survey ON Automatic Test Case Generation," *Academic Open Internet Journal*, 15(6), 2005, Available at http://www.acadjournal.com.

[21] P.N. Boghdady, N. L. Badr, M. A. Hashim and M. F. Tolba, "A Proposed Test Case Generation Technique Based on Activity Diagrams," *International Journal of Engineering & Technology*, 11(3), 35-52, 2011.

[22] M. Shirole, M. Kommuri and R. Kumar, "Transition Sequence Exploration of UML Activity Diagram using Evolutionary Algorithm**,"** *In Proceedings of the 5th India Software Engineering Conference, ISEC,* ACM, Kanpur, India, 97-100, 2012.

[23] M. Shirole, A. Suthar and R. Kumar, "Generation of Improved Test Cases from UML State Diagram Using Genetic Algorithm," *In Proceedings of the 4th India Software Engineering Conference*, Thiruvananthapuram, India, 125-134, 2011.

[24] M. Vieira, J. Leduc, B. Hasling, R. Subramanyan and J. Kazmeier, "Automation of GUI Testing Using a Model-driven Approach," *In Proceedings of the international workshop on Automation of software test, AST,* ACM Press, *Shanghai, China,* 9-14, 2006.

[25] C. D. Nguyen, A. Marchetto and P. Tonella, **"**Combining Model-Based and Combinatorial Testing for Effective Test Case Generation," *In Proceedings of the International Symposium on Software Testing and Analysis, ISSTA,* ACM, 100-110, 2012.

[26] R. K. Swain ,V. Panthi ,D. P. Mohapatra and P. K. Behera, "Prioritizing test scenarios from UML communication and activity diagrams," *Innovations in Systems and Software Engineering,* Springer-Verlag London, 2013.

[27] P. Samuel, R. Mall and S. Sahoo, "UML Sequence Diagram Based Testing Using Slicing," *In Indicon Conference,* IEEE, Chennai, India, 176-178, 2005.

[28] B. N. Biswal and D. P. Mohapatra, "Test Case Generation and Optimization of Object-Oriented Software using UML Behavioral Model," *MTech, Thesis, CSED, NIT, Rourkela,*1-15, 2010.

[29] N. Ye, X. Chen, W. Ding, P. Jiang, L. Bu and X. Li, "Regression Test Cases Generation Based on Automatic Model Revision," *In IEEE Sixth International Symposium on Theoretical Aspects of Software Engineering,* Beijing, 127-134, 2012.

[30] R. Singh and V. Arora, "A practical approach for model based slicing," *IOSR Journal of Computer Engineering*, 12(4), 18-26, 2013.

[31] S. K. Swain, D. P. Mohapatra, and R. Mall, "Test Case Generation Based on Use case and Sequence Diagram," *International Journal of Software Engineering,* 3(2), 21-33, 2010.

[32] Tool Visual Paradigm for UML, Available at: http://www.visualparadigm.com /download/vpuml.jsp [As Accessed on 28th January 2014].

[33] R. S. Pressman, "Software Engineering – A Practitioner's Approach," McGraw Hill Education Asia, 2005.

[34] L. Fernandez-Sanz and S. Misra, " PRACTICAL APPLICATION OF UML ACTIVITY DIAGRAMS FOR THE GENERATION OF TEST CASES," *In PROCEEDINGS OF THE ROMANIAN ACADEMY, Series A,* 13(3), 251–260, 2012.

[35] V.M. Sumalatha and G. S. V. P. Raju, "UML based Automated Test Case Generation technique using Activity-Sequence diagram", *The International Journal of Computer Science & Application (TIJCSA),* ISSN– 2278-1080, 1(9), 2012.

LIST OF PUBLICATIONS

[1] J. Gupta, " An Effective Approach for Test Case Generation from Activity Diagram", *International Journal of Advanced Research in Computer Science and Software Engineering*, 4(7), 2014. (Communicated)