TEST CASE GENERATION TECHNIQUE FOR CONCURRENCY IN ACTIVITY DIAGRAM

NUR SYAFIQAH ZAHIRAH BINTI ABDUL RAHMAN

A thesis submitted in fulfilment of the
requirements for the award of the degree of
Master of Philosophy

School of Computing
Faculty of Engineering
Universiti Teknologi Malaysia

NOVEMBER 2018

Dedicated to:

Myself

Through struggling, stressing, crying and almost ~~quitting,~~

Alhamdullilah, I made it!

Mak, Abah & My Siblings

Thank you for your endless affection, love, encouragement, prayers, moral & financial support to ensure the completion of my research with flying colors!

My Supportive Friend

Thank you for your willingness to hear all my problems & the cheerful word

"Sikit je lagi pika, dh nk abis tu"

Thank you!

ACKNOWLEDGEMENT

In The Name of Allah swt, The Most Gracious and The Most Merciful.

First and foremost, all praise to The Almighty, All praise to Allah swt for bestowed me with a knowledge, good health, courage and strength to complete my master study. Alhamdullilah for His endless blessings throughout my entire research process.

My sincere appreciation to my main supervisor, Associate Professor Dr Dayang Norhayati Abang Jawawi and my co-supervisor Associate Professor Dr Azurati Ahmad@Salleh for their supervision, guidance, countless hours in sharing understanding and patience throughout my research journey. I am thankful for their insightful comments, criticisms and advices during my learning process. I will always look up them as my academic role model to achieve my ambitions.

I would also like to express my gratitude to my parents Abdul Rahman bin Mohd Darom and Zam binti Mahasan, my siblings Abg long, Kak long, Kakak, Abg Ude and Abg Chik who means most to me, for their prayers of day and night, understanding, moral and financial support to complete my master study. My close friends, Irfan, Rooster, Ayu, Wani, Kak Sen, Kimah, Saidah and Murniyanti should also be recognized for their kindness, assistance and support through thick and thin.

Last but not least, I am also indebted to Jabatan Perkhidmatan Awam (JPA) and Research Management Center (RMC) for the sponsorship during my master study. It was invaluable for me to undertake this endeavor.

ABSTRACT

Presently, the application of Model-Based Testing (MBT) using Unified Modelling Language (UML) has attracted the attention of many practitioners to use UML diagrams for generation of test cases. By using this technique, early detection of faults can be achieved at the design phase. However, some UML diagrams have limitations in generating test cases such as the need for a loop combination fragment to describe looping, iteration process and combination fragment with the par operator to interpret concurrency activities. To overcome these issues, a feature analysis was conducted to observe the outcome of test case generation using similar cases but, by using different techniques and UML diagrams. Based on the results, a guideline for selecting UML diagrams in the generation of test cases based on the different features of software system in the cases was developed. However, system design of concurrent software is complex, leading to issues in system testing such as synchronization, non-deterministic, path explosion and deadlock. In this research, an enhancement of the generate-activity-paths algorithm as a test case generation technique was developed to solve the non-deterministic problem of concurrent system. As the test cases are generated in a random order, a prioritization technique using genetic algorithm was applied to find the critical path that must be tested first from the test paths produced. The technique was implemented on the Conference Management System case study and evaluated using cyclomatic complexity, branch coverage, mutation analysis and average percentage of fault detected (APFD) to measure the effectiveness and quality of the test cases in comparison to those using the original technique. Results indicated that the technique achieved 100% basis path and branch coverage criteria similar to the original technique. Moreover, it is also capable of revealing non-deterministic faults by injecting concurrency coverage criteria into the test paths, which was not possible using the original technique. Additionally, prioritization of test paths yielded an APFD value of 43% which is better and higher than the non-prioritized test paths (22%). This result signified that the usage of prioritization technique leads to an improve detection rate of severe faults as compared to applying random order.

ABSTRAK

Pada masa kini, Ujian Berasaskan Model dengan pengunaan Bahasa Pemodelan Bersatu (UML) telah mendapat perhatian daripada ramai pengamal untuk menggunakan gambar rajah UML dalam penghasilan kes ujian. Dengan menggunakan teknik ini, pengesanan kesalahan akan dapat dicapai dengan lebih awal pada fasa reka bentuk. Namun begitu, setiap jenis gamba rajah UML mempunyai kekangan dalam menjana kes ujian seperti memerlukan cebisan gabungan gelung untuk menggambarkan proses penggelungan, lelaran dan cebisan gabungan pengendali par untuk menafsirkan proses serentak. Bagi mengatasi masalah ini, satu analisis ciri telah dijalankan untuk melihat hasil kes ujian menggunakan kes yang serupa tetapi dengan teknik dan gambar rajah UML yang berbeza. Berdasarkan hasilnya, satu garis panduan dalam pemilihan gambar rajah UML untuk menjana kes ujian berdasarkan ciri-ciri berbeza sistem perisian yang terdapat dalam kajian kes telah dibangunkan. Walaubagaimana pun, reka bentuk sistem perisian serentak yang kompleks seperti penyelarasan, tidak berketentuan, ledakan laluan dan kebuntuan menimbulkan isu dalam melaksanakan ujian sistem. Dalam kajian ini, penambahbaikan algoritma penjanaan laluan aktiviti untuk teknik menjana kes ujian telah dibangunkan bagi menyelesaikan masalah tidak berketentuan dalam sistem serentak. Disebabkan kes ujian dijana secara rawak, teknik pengutamaan menggunakan algoritma genetik telah digunakan untuk mencari laluan kritikal yang perlu diuji terlebih dahulu daripada laluan lain yang dihasilkan. Teknik ini telah dilaksanakan ke atas Sistem Pengurusan Persidangan dan dinilai menggunakan kerumitan siklomatik, liputan cabang, analisis mutasi dan purata peratusan pengesan kesalahan (APFD) untuk mengukur keberkesanan dan kualiti kes ujian yang dijana berbanding penggunaan teknik asal. Dapatan hasil penilaian menunjukkan teknik yang telah ditambah baik mencapai 100% laluan dasar dan liputan cabang sama seperti menggunakan teknik asal. Tambahan pula, teknik yang telah ditambah baik ini juga berkebolehan mendedahkan kesalahan tidak berketentuan yang tidak mungkin dapat dilaksanakan menggunakan teknik asal. Selain itu, teknik keutamaan laluan ujian menghasilkan pengesanan kesalahan yang lebih baik, iaitu dengan nilai APFD 43% lebih tinggi berbanding kes ujian yang tidak diutamakan (22%). Hasil kajian menunjukkan penggunaan teknik pengutamaan membawa kepada peningkatan kadar pengesanan kesalahan yang teruk berbanding menggunakan tertib rawak.

TABLE OF CONTENTS

CHAPTER	TITLE	PAGE
	DECLARATION	ii
	DEDICATION	iii
	ACKNOWLEDGEMENTS	iv
	ABSTRACT	v
	ABSTRAK	vi
	TABLE OF CONTENTS	vii
	LIST OF TABLES	xiii
	LIST OF FIGURES	xvi
	LIST OF ABBREVIATIONS	xix
	LIST OF APPENDICES	xxi
1	**INTRODUCTION**	**1**
	1.1 Overview	1
	1.2 Problem Background	4
	1.3 Problem Statements	6
	1.4 Research Objectives	8
	1.5 Research Scope	8
	1.6 Significance of Study	9
	1.7 Thesis Outline	10
2	**LITERATURE REVIEW**	**11**
	2.1 Introduction	11

2.2	Software Testing Overview	11
2.3	Software Testing Stages	13
	2.3.1 Modeling the Software Environment	13
	2.3.2 Selecting Test Scenarios	13
	2.3.3 Test Execution	14
	2.3.4 Quantifying Testing Process	14
2.4	Test Cases Generation	15
	2.4.1 Test Cases Generation Technique	15
2.5	Model-Based Testing (MBT) Overview	17
	2.5.1 Model-Based Testing Process	19
	2.5.2 MBT Technique for Test Cases Generation	20
	2.5.2.1 Finite State Machine (FSM)	21
	2.5.2.2 Theorem Proving	21
	2.5.2.3 Constraints Logic Programming and Symbolic Equation	21
	2.5.2.4 Model Checking	22
	2.5.2.5 Markov Chain	22
	2.5.2.6 Unified Modeling Language (UML)	23
	2.5.3 Related Work on MBT Technique for Test Case Generation	25
2.6	MBT Technique using UML Models for Test Case Generation	28
2.7	Test Case Generation for Concurrent System using UML Model	33
2.8	Automated Test Case Optimization Techniques	37
	2.8.1 Genetic Algorithm (GA)	38
2.9	Test Case Optimization Technique using UML Diagram	40
2.10	Chapter Summary	44

3	**RESEARCH METHODOLOGY**	**45**
	3.1 Introduction	45
	3.2 Research Operational Framework	46
	3.3 Research Design Framework	50
	3.4 Test Case Generation Evaluation Criteria	53
	3.4.1 Coverage Criteria	55
	3.4.2 Cyclomatic Complexity	56
	3.4.3 Number of Faults Detected	57
	3.4.3.1 Mutation Analysis	58
	3.4.4 Average Percentage of Fault Detected (APFD)	59
	3.5 Cases	59
	3.5.1 Case 1: Automatic Teller Machine Withdrawal Function (ATMW)	61
	3.5.2 Case 2: Automatic Teller Machine Pin Authentication (ATMPA)	62
	3.5.3 Case 3: Automatic Ticket Machine System (ATMS)	63
	3.5.4 Case 4: Conference Management System (CMS)	63
	3.5.5 Case 5: Robotic Wheel Chair (RWC)	64
	3.6 Chapter Summary	64
4	**TEST CASE GENERATION FEATURE ANALYSIS USING UML DIAGRAM**	**66**
	4.1 Introduction	66
	4.2 The Conducted Feature Analysis	67
	4.2.1 Test Cases Generation Technique by Boghdady *et al.*, (2011)	68
	4.2.1.1 Module 1: Generation of Activity Dependency Table (ADT)	69

4.2.1.2 Module 2: Generation of Activity Dependency Graph (ADG) 71
4.2.1.3 Module 3: Test Case Generation 72
4.2.2 Test Case Generation Technique by Sarma *et al.*, (2007) 75
4.2.2.1 Module 1: Generation of Operation Scenario 77
4.2.2.2 Module 2: Transformation Sequence Diagram into Sequence Diagram Graph (SDG) 78
4.2.2.3 Module 3: Test Case Generation 79
4.2.3 Test Cases Generation Technique by Swain *et al.*, (2012) 81
4.2.3.1 Module 1: Converting State Chart Diagram into State Transition Graph (STG) 82
4.2.3.2 Module 2: Test Case Generation 83
4.3 Feature Analysis Results 85
4.4 The Proposed Guidelines 88
4.5 Application of Guidelines on Cases 92
4.5.1 Guideline 1: Combination of Sequence Diagram with State Chart Diagram 93
4.5.1.1 Module 1: Conversion of State Chart Diagram (SCD) into State Chart Diagram Graph (SCDG) 93
4.5.1.2 Module 2: Conversion of Sequence Diagram (SD) into Sequence Diagram Graph (SDG) 94
4.5.1.3 Module 3: Integration SCDG and SDG into System Testing Graph (STG) 95

4.5.1.4 Module 4: Test Case Generation 96

4.5.2 Guideline 2: Extension of Sequence Diagram with Labeled Transition System (LTS) 100

4.5.2.1 Module 1: Transformation of Sequence Diagram into Labeled Transition System (LTS) 100

4.5.2.2 Module 2: Test Case Generation 102

4.5.3 Guideline 3: Sequence Diagram with Loop and Concurrent Fragment 105

4.5.3.1 Module 1: Converting Robotic Wheel Chair (RWC) Sequence Diagram (SD) into Sequence Graph (SG) 106

4.5.3.2 Module 2: Test Case Generation 108

4.6 Discussion 111

4.7 Chapter Summary 112

5 THE ENHANCEMENT OF TEST CASE GENERATION TECHNIQUE 113

5.1 Introduction 113

5.2 The Integration of the Selected Technique to Generate Test Cases 114

5.2.1 The Enhancement of Test Case Generation Technique 116

5.2.1.1 Step 1: Design Activity Diagram 119

5.2.1.1.1 Activity Diagram 119

5.2.1.2 Step 2: Extracted Activity from Activity Diagram Specification 123

5.2.1.2.1 XML Metadata Interchange 123

5.2.1.3 Step 3: Transforms Extracted Activities into Activity Graph 129

		5.2.1.4 Step 4: Generating Test Path from Activity Graph	130
		5.2.1.5 Step 5: Derived Test Cases	135
	5.3	Prioritization Test Cases using Genetic Algorithm	142
	5.4	Summary	148
6	**EVALUATION OF THE PROPOSED APPROACH**		**150**
	6.1	Introduction	150
	6.2	The Results of the Proposed and Original Technique	151
		6.2.1 Cyclomatic Complexity, *V(G)*	151
	6.3	Branch Coverage	155
	6.4	Mutation Analysis	156
	6.5	Average Percentage of Fault Detected (APFD)	166
		6.5.1 APFD Value for Fault Detected	166
		6.5.2 APFD Value for Prioritized Test Sequence and Non-Prioritized Test Sequence	169
	6.6	Results Discussion	174
	6.7	Chapter Summary	176
7	**CONCLUSION AND FUTURE WORKS**		**177**
	7.1	Research Summary	177
	7.2	Research Contributions	179
	7.3	Research Limitation	180
	7.4	Future Works	181
	REFERENCES		**182**
	Appendices A-C		189-203

LIST OF TABLES

TABLE NO	TITLE	PAGE
2.1	Test Case Generation Technique in Software Testing	16
2.2	Comparison of Different Testing Process in Software Testing	18
2.3	Summary MBT Technique for Test Cases Generation	24
2.4	MBT Techniques to Generate Test Cases	27
2.5	MBT Technique using UML Model for Test Case Generation	32
2.6	Comparison of Existing Test Case Generation Technique using UML Diagram	36
2.7	Comparison of Existing Test Case Optimization Technique using UML Model	43
3.1	Phases of Enhancement Process	54
3.2	Type of Coverage Criteria in Control-Flow	55
3.3	Selected Cases	60
4.1	Activity Dependency Table for Automatic Ticket Machine System (ATMS)	70
4.2	Test Cases for Automatic Ticket Machine System (ATMS)	73
4.3	Five operation scenario represented in quadruple form for ATMW	77
4.4	Test Cases for Automatic Teller Machine Withdrawal (ATMW)	79

4.5	Test Cases for Automatic Teller Machine Pin Authentication (ATMPA)	84
4.6	Comparison Number of Test Cases Generated	85
4.7	Type of Test Cases Produce using Different UML Diagram for ATMW	86
4.8	Type of Test Cases Produce using Different UML Diagram for ATMPA	86
4.9	Type of Test Cases Produce using Different UML Diagram for ATMS	87
4.10	The Proposed Guideline	90
4.11	Test Cases Generated for Automatic Ticket Machine System (ATMS)	98
4.12	Path Table for Automatic Teller Machine Pin Authentication (ATMPA)	103
4.13	Test Case 1	104
4.14	Test Case 2	104
4.15	Test Case 3	104
4.16	Test Case 4	104
4.17	Test Case 5	105
4.18	Test Case 6	105
4.19	Message Details of Sequence Diagram (MDSD)	108
4.20	Generated Test Cases for Robotic Wheel Chair (RWC)	110
5.1	Elements in Activity Diagram	120
5.2	Extracted Activities from the UML Activity Diagram for CMS Case Study	127
5.3	Test Cases from Generate Activity Test Paths	138
6.1	Basis Path Coverage Criteria	152
6.2	Generated Test Cases for Branch Coverage Criteria	157
6.3	Test Path with Fault Detected (Enhanced Technique)	159
6.4	Fault-Based Mutation Result (Enhanced Technique)	161

6.5	Test Path with Fault Detected (Original Technique)	163
6.6	Fault-Based Mutation Result (Original Technique)	164
6.7	Comparison Criteria for Improvement	165
6.8	Fault Matrix	171

LIST OF FIGURES

FIGURE NO	TITLE	PAGE
1.1	Software Development Life Cycle (SDLC) Phases	2
2.1	Generic Process of MBT (Karaman, 2014)	20
2.2	UML Model Category (Priya & Sheba, 2013)	29
2.3	Search-Based Method (Mohi-Aldeen *et al.*, 2014)	37
2.4	Basic Genetic Algorithm (GA) Steps	39
3.1	Research Operation Framework	49
3.2	Research Design Framework	51
4.1	Implementation Methodology	67
4.2	Activity Diagram for Automatic Ticket Machine System (ATMS)	69
4.3	Activity Dependency Graph for Automatic Ticket Machine System (ATMS)	71
4.4	Sequence Diagram for Automatic Teller Machine Withdrawal (ATMW)	76
4.5	Sequence Diagram Graph for Automatic Teller Machine Withdrawal (ATMW)	78
4.6	State Chart Diagram for Automatic Teller Machine Pin Authentication (ATMPA)	81
4.7	State Transition Graph for Automatic Teller Machine Pin Authentication (ATMPA)	82
4.8	State Chart Diagram Graph for Automatic Ticket Machine System (ATMS)	94

4.9	Sequence Diagram Graph for Automatic Ticket Machine System (ATMS)	95
4.10	System Testing Graph for Automatic Ticket Machine System (ATMS)	96
4.11	Labeled Transition System (LTS) elements	101
4.12	Labeled Transition System (LTS) for Automatic Teller Machine Pin Authentication (ATMPA)	102
4.13	Sequence Diagram for Robotic Wheel Chair (RWC) Case	106
4.14	Sequence Graph (SG) for Robotic Wheel Chair (RWC) Case	107
5.1	Framework of Integration Technique	115
5.2	Proposed Integration Technique	118
5.3	Main Elements in Activity Diagram	119
5.4	Activity Diagram for Registration Cancellation Use Case from Conference Management System (CMS) Case	122
5.5	Input file XMI	124
5.6	Extracted Activity Algorithm	125
5.7	Output from the Algorithm	126
5.8	Activity Graph for CMS (*Registration Cancellation*) Case	130
5.9	Generate Activity Test Path Algorithm	133
5.10	Test Path for Conference Management System (*Registration Cancellation*) Case	134
5.11	Details Activity of Test Path Generated for Conference Management System (Registration Cancellation) Case	135
5.12	Injected Concurrency Elements	136
5.13	Test Path Generated in Random Order	143

5.14	Assigned Weight to Activity Graph for Conference Management System (*Registration Cancellation*) Case	144
5.15	Weight for Different Path in Conference Management System (CMS) Registration Cancellation Case	145
5.16	Test Cases Prioritization Algorithm	146
5.17	Prioritized Test Path	147
6.1	Cyclomatic Activity Graph for CMS (*Registration Cancellation*) Case Study	151
6.2	Activity Graph for CMS (*Registration Cancellation*) Case Study (Kundu & Samanta, 2009)	153
6.3	Test Cases Generated from CMS (*Registration Cancellation*) Case Study (Kundu & Samanta, 2009)	154
6.4	Generated Test Path	159
6.5	Fault Mutation APFD Results	168
6.6	Comparison Based on APFD Values	172
6.7	Number of Fault Detected versus Sequence of Prioritized Test Cases	173
6.8	Number of Fault Detected versus Sequence of Non-Prioritized Test Cases	174

LIST OF ABBREVIATIONS

ABC	-	Ant Bee Colony
ACOToTSP	-	Ant Colony Optimization for Test Scenario Prioritization
AD	-	Activity Diagram
ADT	-	Activity Dependency Table
ADG	-	Activity Dependency Graph
AIG	-	Activity Interaction Graph
AG	-	Activity Graph
ASG	-	Activity Sequence Graph
ATMS	-	Automatic Ticket Machine System
ATMW	-	Automatic Teller Machine Withdrawal
ATMPA	-	Automatic Teller Machine Pin Authentication
BFS	-	Breadth First Search
CCFG	-	Concurrent Control Flow Graph
CFG	-	Control Flow Graph
CMS	-	Conference Management System
C-GA	-	Constraints-Based Genetic Algorithm
DFA	-	Deterministic Finite Automaton
DFS	-	Depth First Search
EAA	-	Extracted Activity Algorithm
EFSM	-	Extended Finite State Machine
FSM	-	Finite State Machine
GA	-	Genetic Algorithm
HC	-	Hill Climbing
MBT	-	Model-Based Testing
MDG	-	Message Dependency Graph

NDFA	-	Non-Deterministic Finite Automaton
OCL	-	Object Constraints Language
OP	-	Operational Profiles
PLC	-	Programmable Logic Controllers
PSO	-	Particle Swarm Optimization
RWC	-	Robotic Wheel Chair
SA	-	Simulated Annealing
SCG	-	State Chart Graph
SCSEDG	-	State Chart and Sequence Diagram Graph
SD	-	Sequence Diagram
SDG	-	Sequence Diagram Graph
SDLC	-	System Development Life Cycle
SFSNP	-	Semi-valid Fuzzing for the Stateful Network Protocol
SG	-	Sequence Graph
SMT	-	Satisfiability Modulo Theories
STG	-	State Transition Graph
SUT	-	System Under Test
TS	-	Tabu Search
UDG	-	Use Case Dependency Graph
UML	-	Unified Modeling Language
V&V	-	Verification and Validation
XML	-	Extensible Markup Language
XMI	-	XML Metadata Interchange

LIST OF APPENDICES

APPENDIX	TITLE	PAGE
A	Test cases generation technique by Boghdady *et al.*, (2011)	189
B	Test cases generation technique by Sarma *et al.*, (2007)	195
C	Test cases generation technique by Swain *et al.*, (2012)	200

CHAPTER 1

INTRODUCTION

1.1 Overview

When new technologies take off, modern software become larger and more complex. Computer applications have spread into every sphere of life for manipulation of several sophisticated applications and most of these applications are very large and complex (Biswal, 2010). Although growing difficulty and scope of software applications require more emphasis, a comprehensive testing is almost impractical. In a System Development Life Cycle (SDLC) as shown in Figure 1.1, testing is executed after the completion of development process of a system. Myers described software testing as an integral part of building a software (Myers *et al.*, 2011):

"It has been known for quite a while that, in an ordinary programming venture, pretty nearly 50% of the passed time and more than 50% of the aggregate expense are used in testing the system or framework being created. Given this learning, one would expect that program testing would have at this point been refined into an 'accurate science,' however this is a long way from the genuine case".

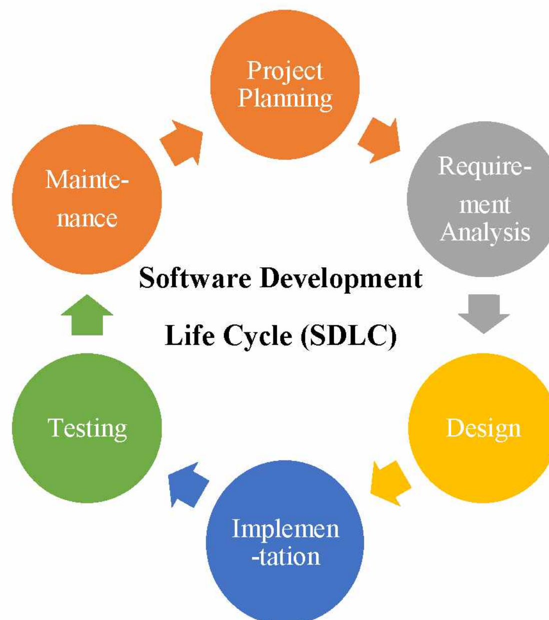

Figure 1.1 Software Development Life Cycle (SDLC) Phases

In term of software engineering, software testing is defined as a process of recognizing failures or defects in advance leading to producing a quality product. The main goal of software testing is to detect failures. Software failures are defined as observable differences between the behaviors of implementation and what are expected on the basis of system specification. Testing is a critical phase in an SDLC in guaranteeing software quality, which is typically performed to verify that the software is built according to customer's requirements. Software testing increases the efficiency and quality of a built software product and also ensures that the software is free from faults.

Software testing can be divided into two main approaches comprising white-box and black-box testing. White-box testing is a form of structural testing that considers the internal structure of a system or components (Williams, 2006). Such testing focuses on source code components. On the other hand, black-box testing, which is also known as functional testing, is a testing approach that overlooks the internal components of a developed software. Rather, black-box testing concentrates on the outputs generated in response to selected inputs and execution conditions. Scholars have utilized black-box testing approach in constructing test case, utilizing information retrieved from requirements and design specifications. In recent years,

there is revival of interest on one of black-box testing techniques—Model-Based Testing (MBT). MBT is progressively gaining scholars' attention from various fields, including industry and academia. MBT is a testing process that involves creating a test model utilizing different models such as Finite State Machine (FSM), Model Checking and Unified Modeling Language (UML), which are subsequently used to generate a collection of test case (Boghdady *et al.*, 2011). MBT is an evolving technique that depends on system behavior models for the generation of model traces, input, and expected output which are used to generate test case. Commonly, MBT employs models to generate test case that are able to verify the expected behaviors of the system.

In software engineering perspective, a test case consists of a set of conditions or variables that allows a tester to confirm whether a built software system functions as expected. Test case can be generated from requirements, source code, or design artifacts. In generating test case, there are numerous test generation techniques which have been proposed by scholars including, random testing, search-based testing, combinatorial testing, and model-based testing. Commonly, test cases are designed based on a program source code. For the most part, the developers of the software might not have enough time to generate adequate test case to test their software system after coding phase. This is because a large number of code are produced by the size and complexity of the system nowadays (Jena *et al.*, 2014). One of the solutions to mitigate this issue is to generate test case prior to coding phase. The generation of test case earlier prior to coding phase (i.e. design phase) makes test planning more effective and has the additional benefit of allowing test case to be available early in SDLC (Ikram *et al.*, 2015). For a complex system, the number of test cases generated could be large. One of the solutions to deal with a large collection of test case is to optimize their ordering for testing in order to detect faults earlier. There are several test optimization techniques available, such as test case prioritization, test case minimization, and test case selection. These techniques help testers to optimize testing resources such as time and cost. Optimization techniques are usually used in regression testing. Optimizing test case generated through UML models for regression testing could be useful in producing effective overall system testing.

1.2 Problem Background

Testing plays a vital role in guaranteeing the quality and reliability of a product (Jena *et al.*, 2014). As the difficulty and scope of systems expand attributed to changing customer's requirements, additional time and effort are needed to perform adequate testing. In testing process, there are three parts involved comprising test case generation, execution, and evaluation. Generation of test case is the most challenging and difficult step in testing as they determine the success of testing in producing quality products that meet customer's expectation.

Testing is a time consuming and labor-intensive task as it involves creating a huge collection of test case. This is attributed to growing software system size as customers request new system functionalities. The use of manual testing is impractical nowadays as it is both expensive and time consuming, particularly for modern software applications that consist of many intricate components. Generation of test case from source code can be arduous and ineffective. This is because hundreds if not thousands of lines of code are produced during software development. Manually tracing code line-by-line to discover faults requires a significant amount of time. Furthermore, test case generated from source code have been indicated to be inadequate in the case of component-based software development, where even the source code may not be accessible to the developers (Samuel *et al.*, 2008). Besides that, utilization of source code to test an object-oriented system is also a difficult and monotonous task.

Researchers have consistently produced new techniques to address the challenges of test case generation, aiming to reduce overall testing effort and time. It is ideal to generate test case at the design stage, which helps to produce reliable software (Jena *et al.*, 2014). Based on this notion, MBT technique has been introduced to support the production of quality software. MBT involves an automatic generation of test case using models from system requirements (Cartaxo *et al.*, 2007). Six MBT techniques have been introduced by scholars to generate test case including Finite State Machine (FSM), Theorem Proving, Constraints Logic Programming and Symbolic Execution, Model Checking, Markov Chain, and Unified Modeling Language (UML). Nowadays, model-based software development using UML notations has gained wide

attention among scholars, whereby numerous scholars have begun to extract useful information from UML diagrams to generate test case (Sawant & Shah, 2011). Evidently, UML models help developers to recognize software structure and discover test data attributed to high-level abstraction models. In addition, MBT strategy in generating test case allows testers to plan testing at an early phase in SDLC and permits parallel testing and coding (Kundu & Samanta, 2009). Furthermore, when test case is produced early, it allows developers to discover irregularities and ambiguities in requirement specification and design documents.

Even though MBT techniques can address the challenges in generating test case, they also suffer several challenges. MBT techniques typically generate a large set of test case and involves extra cost of modelling software under test. In addition, a full automation of MBT in generating test case is a challenging task as this demands complete and clear-cut system models as inputs (Felderer & Herrmann, 2015). Even though the generation of test case from UML diagrams is a rudimentary task for software testers, each type of UML diagram has its constraint in generating test case. Sequence diagram needs a loop combination fragment to describe the looping process, and to model the concurrency a combination fragment with the par operator are needed to show the parallel execution of the operation. On the other hand, a state machine diagram needs orthogonal components to express the concurrent process in the system and it becomes difficult if it involves complicated concurrent activity.

Due to this, it is difficult to select the most suitable diagram to generate test case for different features of a system. Moreover, it is especially difficult when a system contains concurrent processes. Testing concurrent object-oriented systems has to deal with the complexities arising from the physical distribution and parallel execution of the objects (Patnaik *et al.*, 2011). Furthermore, testing concurrent systems is a very critical task because the systems can show different responses depending on respective of concurrent situations (Khandai *et al.*, 2011). There may be test explosion, deadlock, synchronization and non-deterministic concurrent problems when parallel processes want to interact with each other. Based on Kundu and Samanta (2009) work, a non-deterministic fault occurs because they only test a relative sequence of the concurrent activity without considering the interleaving of activities between parallel

processes. Testing concurrent system requires not only to explore the space of possible inputs, but also the space of possible interleaving which make technique used by Kundu and Samanta (2009) lack in capturing the test path that involve the interleaving activities.

Besides that, as MBT techniques can possibly produce a very huge set of test case for any non-trivial models, many optimization methods are often applied to choose which test case to include in the generated set (Kanstrén & Chechik, 2014). There are different optimization techniques available such as test case selection, test case minimization, and test case prioritization. Test case prioritization determines the ordering of execution of test case in such a way that most beneficial test case are executed first (Mahali & Acharya, 2013). A number of different approaches have been studied to prioritize test case such as Tabu Search, Hill Climbing, Ant Bee Colony (ABC), Particle Swarm Optimization (PSO), and Genetic Algorithm (GA). These algorithms are concerned with identifying an 'ideal' ordering of test case that can reveal faults earlier. This way, critical problems can be found early in addition to allowing testers to measure how quickly faults are detected within a testing process.

1.3 Problem Statements

This study focuses on test case generation from different type of UML diagrams motivated by challenges faced in choosing suitable UML diagrams for generating test case. MBT techniques have begun to be used widely to generate test case replacing manual approaches. Researchers commonly utilize UML diagrams to derive test case. However, there are many types of UML diagrams that have been used to generate test case. The issue arises in choosing suitable UML diagrams as they do not address specific issues like iteration, looping, and concurrent activities in software systems. Specifically, issues that are faced by MBT using UML diagrams are as follows:

i. It is a difficult task to select a suitable diagram to generate test case for different features of software systems that consider looping, iteration and concurrent processes unsupported by some UML diagrams in their models.
ii. From existing works, MBT techniques have been widely used to generate test case for different types of software system. However, there is poor literature emphasis on dealing with concurrent software systems. Testing concurrent systems is a major challenge as testers need to verify legal sequence of interactions between multiple objects to access resources that are shared among processes.
iii. Generating test case in concurrent environment is a complex task due to interference of concurrent threads that may lead to concurrency problems. From Kundu and Samanta (2009) work, a non-deterministic arises when testing parallels activities in the Conference Management System (CMS) case.

Hence, this study emphasizes generating test case from UML diagrams focusing on concurrent software systems and solving the problem arising from concurrent activities. In order to solve the limitations of considered UML diagrams in generating test case and non-deterministic concurrent problems that arise from the current existing work, the main research question is stated as follows:

"How can an effective test case utilizing different UML diagrams for concurrent systems be produced?"

A set of research questions are constructed to address the aforementioned main research problem as follows:

i. RQ1: Which types of UML diagrams are capable of generating test case that consider looping, iteration, and concurrent processes in systems?
ii. RQ2: How can the current test case generation technique for concurrent system be enhanced using UML diagrams to solve non-deterministic concurrent problem?
iii. RQ3: How can GA be used to discover the critical test case generated using MBT technique?

1.4 Research Objectives

Based on the aforementioned problem statement and derived research questions, the objectives of this research are as follows:

i. To propose a guideline for selection of UML diagrams to generate test case based on looping, iteration and concurrent system.
ii. To enhance the current test case generation technique to solve the non-deterministic concurrent problem and achieve a maximum basis path, branch, and concurrent coverage criteria for defects to be detected.
iii. To implement optimization technique GA on the enhanced test case generation technique, by prioritizing test case so that critical test case can be run earlier.
iv. To evaluate the implementation of the enhanced test case generation technique on Conference Management System (CMS) and Automatic Teller Machine (ATM) cases using several metrics.

1.5 Research Scope

This section elaborates the scopes of this research. Some of the research scopes are addressed in detail in later chapters.

i. Focuses on one of the black-box testing techniques, which is MBT technique, where test case are derived from UML diagrams that are used to model user's requirements.
ii. Three UML behavioral models comprising activity diagram, sequence diagram, and state chart diagram have been implemented on looping, iteration and concurrent cases to compare the effectiveness on generating test case.
iii. Java programing language is used for implementation in case.
iv. Enterprise Architecture (EA) modelling tool is used, which supports UML 2.0 syntax/XMI 1.1 and 1.2.

1.6 Significance of Study

Since modern software are huge and complex in a real-life applications, exhaustive testing is impractical. In order to meet the convincing need of quick development, software industries are attempting to create quality software within a shorter period (Gantait, 2011). This research study investigates the existing UML behavioral models to generate test case such as activity diagram, sequence diagram, use case diagram, state chart diagram, and class diagrams. Based on the study, some types of UML diagrams have their limitations in generating test case for software systems that involve looping, iteration, and concurrent process.

This research also primarily intends to enhance the current test case generation technique proposed by Kundu and Samanta (2015). Their technique was capable of detecting faults such as looping fault and synchronization fault. However, it is not capable of detecting non-deterministic problem. Motivated by this, the enhance test case generation technique has been proposed for covering the non-deterministic faults. The enhanced technique leads to producing effective test case covering concurrent faults, and achieving a maximum path coverage criteria, branch coverage criteria, and concurrent coverage.

As a result, the knowledge gained from this research provide benefits other researchers that are looking into exploring software testing area, particularly on test case generation using UML diagrams. It helps software tester to select suitable UML diagram to be used in generating test case for different features of cases such as looping, iteration and concurrent. In addition, the suitable selection of diagram enables and are capable of covering concurrent fault, achieving path and branch criteria advantages in order to produce quality test cases and deliver the good software system to the user.

1.7 Thesis Outline

This thesis is organized as follows:

Chapter 1 discusses the overview of the research area, problem backgrounds, problem statements, aim, objective, scope and significance of the study. The main research questions are also stated in this chapter. Chapter 2 reviews software testing activities and model-based testing techniques that have been used to generate test case. This chapter also reviews related works on generation of test case based on UML models such as activity diagram, use case diagram, sequence diagram, and state chart diagram. Test case optimization techniques have also been discussed including a literature review of the current works in the field.

Chapter 3 describes the research operation framework that shows all related phases involved during the study. The proposed research design framework has also been discussed, which shows overall related research elements used for the research. Chapter 4 proposes a guideline of selecting UML model for generation of test case based on different features of cases.

Chapter 5 proposes an enhanced technique to generate test case from concurrent systems. This enhanced technique is an improvement of Kundu and Samanta (2009) test case generation technique for solving non-deterministic concurrent problem.

Chapter 6 presents the results of the enhanced test case generation technique. The results have been discussed, evaluated, and benchmarked with existing work. Chapter 7 summarizes and concludes the thesis. The thesis is concluded by restating the contributions with further discussions and exploring crucial issues concerning area for methodology improvements. The chapter also suggests directions for future study.

REFERENCE

Ali, Md Azaharuddin, *et al.* (2014).Test Case Generation using UML State Diagram and OCL Expression. *International Journal of Computer Applications.* 95(12): 7-11.

Alsmadi, Izzat. (2012). Using Test Case Mutation to Evaluate the Model of the User Interface. *Computer Science Journal of Moldova,* 20(1): 1-25.

Anand, Saswat, *et al.* (2013). An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software.* 86(8): 1978-2001.

Nur Fatimah As' Sahra (2015). *Test Case Prioritization Technique Using Sequence Diagram and Labeled Transition Systems in Regression Testing.* Universiti Teknologi Malaysia. Tesis Sarjana.

Biswal, Baikuntha Narayan (2010). *Test Case Generation and Optimization of Object-Oriented Software using UML Behavioral Models.* National Institute of Technology Rourkela. Tesis Sarjana.

Biswal, Baikuntha Narayan, *et al.* (2010). A Novel Approach for Optimized Test Case Generation Using Activity and Collaboration Diagram. *International Journal of Computer Applications,* 1(14): 67-71.

Boghdady, Pakinam N, *et al.* (2011). A proposed test case generation technique based on activity diagrams. *International Journal of Engineering & Technology IJET-IJENS,* 11(03): 1-18.

Cartaxo, Emanuela G, *et al.* (2007). Test case generation by means of UML sequence diagrams and labeled transition systems. *IEEE International Conference. Systems, Man and Cybernetics, ISIC.* 1292-1297.

Castro, Miguel, *et al.* (2008). Better bug reporting with better privacy. *ACM Sigplan Notices.* 319-328.

Dalal S. and Chhillar R. S. (2012). Case studies of most common and severe types of software system failure. *International Journal of Advanced Research in Computer Science and Software Engineering,* 2(8): 1-7.

Dalal, Siddhartha R, *et al.* (1999). Model-based testing in practice. *Proceedings of the 21st international conference on Software engineering.* 285-294.

Ebert C. and Cain J. (2016). Cyclomatic Complexity. *IEEE Software,* 33(6): 27-29.

Enoiu, Eduard P, *et al.* (2016). Mutation-Based Test Generation for PLC Embedded Software Using Model Checking. *IFIP International Conference on Testing Software and Systems.* 155-171.

Felderer, Michael, & Herrmann, Andrea. (2015). Manual Test Case Derivation from UML Activity Diagrams and State Machines: a Controlled Experiment. *Information and Software Technology*. 1-15.

Gantait, Amitranjan. (2011). Test Case Generation and Prioritization from UML Models. *Emerging Applications of Information Technology (EAIT), 2011 Second International Conference on.* 345-350.

Gebizli, Cereri Şahin, *et al.* (2015). Combining model-based and risk-based testing for effective test case generation. *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on.* 1-4.

Hamon, Grégoire, *et al.* (2004). Generating efficient test sets with a model checker. *Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on.* 261-270.

Helke, Steffen, *et al.* (1997). Automating test case generation from Z specifications with Isabelle, in *ZUM'97: The Z Formal Specification Notation* (pp. 52-71) Springer.

Horgan, Joseph R., *et al.* (1994). Achieving software quality with testing coverage measures. *Computer, 27*(9), 60-69.

Ikram, Muhammad Touseef, *et al.* (2015). Testing from UML Design using Activity Diagram: A Comparison of Techniques. *International Journal of Computer Applications,* 131(5).

Jena, Ajay Kumar, *et al.* (2014). A novel approach for test case generation from UML activity diagram. *Issues and Challenges in Intelligent Computing Techniques (ICICT), 2014 International Conference on.* 621-629.

Juristo, Natalia, *et al.* (2006). Software testing practices in industry. *IEEE software.* 23(4): 19-21.

Kanstrén, Teemu, & Chechik, Marsha. (2014). A comparison of three black-box optimization approaches for model-based testing. *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on.* 1591-1598.

Kaur, Arvinder, & Goyal, Shivangi. (2011). A bee colony optimization algorithm for fault coverage based regression test suite prioritization. *International Journal of Advanced Science and Technology*. 29: 17-30.

Keum, ChangSup, *et al.* (2006). Generating test cases for web services using extended finite state machine, in *Testing of Communicating Systems* (pp. 103-117). Springer

Khandai, Monalisha, *et al.* (2011). Test case generation for concurrent system using UML combinational diagram. *International Journal of Computer Science and Information Technologies, IJCSIT, 2*, 1-10.

Khurana N, and Chillar RS. (2015). Test Case Generation and Optimization using UML Models and Genetic Algorithm. *Procedia Computer Science, 57*, 996-1004.

Kim, Hyungchoul, *et al.* (2007). Test cases generation from UML activity diagrams. *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on.* 556-561.

Kim, Young Gon, *et al.* (1999). Test cases generation from UML state diagrams. *IEEE Proceedings-Software, 146*(4):187-192.

Kosindrdecha N. and Daengdej J. (2010). A test case generation process and technique. *Journal of Software Engineering, 4*(4): 265-287.

Kundu D. and Samanta D. (2009). A Novel Approach to Generate Test Cases from UML Activity Diagrams. *Journal of Object Technology, 8*(3): 65-83.

Kyaw A. A. and Min M. M. (2015). An Efficient Approach for Model Based Test Path Generation. *International Journal of Information and Education Technology, 5*(10): 763.

Lammermann, Frank, *et al.* (2008). Evaluating evolutionary testability for structure-oriented testing with software measurements. *Applied Soft Computing*. 8(2): 1018-1028.

Li H. and Lam C. P. (2004). Software Test Data Generation using Ant Colony Optimization. *International Conference on Computational Intelligence.* 1-4.

Linzhang, Wang, *et al.* (2004). Generating test cases from UML activity diagram based on gray-box method. *Software Engineering Conference, 2004. 11th Asia-Pacific.* 284-291.

Liu, Pan, *et al.* (2017). A study for extended regular expression-based testing. *IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS).* 14-23

Lobo, Fernando G, & Lima, Claudio F. (2007). Adaptive population sizing schemes in genetic algorithms. In *Parameter Setting in Evolutionary Algorithms* (pp. 185-204): Springer.

Lu G. and Miao H. (2014). An Approach to Generating Test Data for EFSM Paths Considering Condition Coverage. *Electronic Notes in Theoretical Computer Science, 309*. 13-29.

Ma, Rui, *et al.* (2017). Semi-valid fuzz testing case generation for stateful network protocol. *Tsinghua Science and Technology, 22*(5): 458-468.

Mahali P. and Acharya A. A. (2013). Model based test case prioritization using UML activity diagram and evolutionary algorithm. *International Journal of Computer Science and Informatics, 3*(2): 42-47.

Mahali, Prateeva, *et al.* (2015). Model Based Test Case Generation and Optimization Using Intelligent Optimization Agent, in *Information Systems Design and Intelligent Applications* (pp. 479-488.) Springer.

Mingsong, Chen, *et al.* (2006). Automatic test case generation for UML activity diagrams. *Proceedings of the 2006 international workshop on Automation of software test.* 2-8.

Mohammad Reza Keyvanpour, *et al.* (2012). Automatic Software Test Case Generation: An Analytical Classification Framework. *International Journal of Software Engineering and Its Applications, 6*(4): 1-16.

Mohi-Aldeen, Shayma Mustafa, *et al.* (2014). Systematic Mapping Study in Automatic Test Case Generation. In *SoMeT.* 703-720.

Myers, Glenford J, *et al.* (2011). *The art of software testing.* (3rd). Canada: John Wiley & Sons.

Nebut, Clementine, *et al.* (2006). Automatic test generation: A use case driven approach. *IEEE Transactions on Software Engineering, 32*(3): 140-155.

M. O. Odetayo. (1993). Optimal population size for genetic algorithms: an investigation. In Genetic Algorithms for Control Systems Engineering, *IEE Colloquium* on. 2/1-2/4. Offutt J. and Abdurazik A. (1999). Generating tests from UML specifications *«UML»'99—The Unified Modeling Language* (pp. 416-429). Springer.

Pahwa N and Solanki K. (2014). UML based test case generation methods: A review. *International Journal of Computer Applications*, 95(20): 1-6.

Panthi V. and Mohapatra DP. (2016). ACO based embedded system testing using UML Activity Diagram. *Region 10 Conference (TENCON), 2016 IEEE.* 237-242.

Patnaik, Debashree, *et al.* (2011). Generating Testcases for Concurrent Systems Using UML State Chart Diagram. *Information Technology and Mobile Communication* (pp. 100-105). Springer.

Priya S. S. and Sheba PD. (2013). Test Case Generation from Uml Models–A Survey. In *Proc. International Conference on Information Systems and Computing (ICISC-2013).* 1-11.

Rayadurgam S. and Heimdahl M. P. E. (2001). Coverage based test-case generation using model checkers. *Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings. Eighth Annual IEEE International Conference and Workshop on the.* 83-91.

Sabharwal, Sangeeta, *et al.* (2010). Prioritization of test case scenarios derived from activity diagram using genetic algorithm. *Computer and Communication Technology (ICCCT), 2010 International Conference.* 481-485.

Sahoo, Rajesh Ku, *et al.* (2017). Model Driven Test Case Optimization of UML Combinational Diagrams Using Hybrid Bee Colony Algorithm. *International Journal Intelligent Systems and Applications.* 43-45.

Samuel, Philip, *et al.* (2008). Automatic test case generation using unified modeling language (UML) state diagrams. *IET software, 2*(2): 79-93.

Samuel, Philip, *et al.* (2007). Automatic test case generation from UML communication diagrams. *Information and software technology, 49*(2): 158-171.

Sarma M. and Mall R. (2007). Automatic test case generation from UML sequence diagram. In *Advanced Computing and Communications, 2007. ADCOM 2007. International Conference on.* 60-67.

Sawant V. and Shah K. (2011). Construction of Test Cases from UML Models *Technology Systems and Management* (pp. 61-68). Springer

Sawant V. and Shah K. (2011). Automatic generation of test cases from UML models. In *International Conference on Technology Systems and Management (ICTSM).* 7-10

Shah, Syed Asad Ali, *et al.* (2016). Automated Test Case Generation Using UML Class & Sequence Diagram. *British Journal of Applied Science & Technology, 15*(3): 1-12.

Shanthi A. and MohanKumar G. (2012). A novel approach for automated test path generation using TABU search algorithm. *International Journal of Computer Applications, 48*(13): 975-888

Sokenou, Dehla. (2006). Test Sequences from UML Sequence Diagrams and State Diagrams. In *GI Jahrestagung (2).* 236-240.

Sumalatha V.M and Raju GSVP. (2013). Object Oriented Test Case Generation Technique using Genetic Algorithms. *International Journal of Computer Applications, 61*(20): 20-28

Sun, Chang-ai, *et al.* (2016). A transformation-based approach to testing concurrent programs using UML activity diagrams. *Software: Practice and Experience, 46*(4), 551-576.

Suresh Y. and Rath S. K. (2014). A genetic algorithm based approach for test data generation in basis path testing. *International Journal of Soft Computing and Software Engineering.* 1-7

Swain, Ranjita Kumari, *et al.* (2012). Generation and Optimization of Test cases for Object-Oriented Software Using State Chart Diagram. *International Journal of Soft Computing and Software Engineering.* 1-18

Swain, Ranjita Kumari, *et al.* (2014). Slicing-based test case generation using UML 2.0 sequence diagram. *International Journal of Computational Intelligence Studies 2, 3*(2-3): 221-250.

Swain, Santosh Kumar, *et al.* (2010). Test case generation based on use case and sequence diagram. *International Journal of Software Engineering, 3*(2): 21-52.

Swain, Santosh Kumar, *et al.* (2010). Test Case Generation Based on State and Activity Models. *Journal of Object Technology, 9*(5): 1-27.

Tachio Terauchi (2006). *Types for deterministic concurrency.* University of California, Berkeley. Tesis Doktor Falsafah.

Utting, Mark, *et al.* (2006). A taxonomy of model-based testing. Department of Computer Science, University of Waikato.

Wang Y. and Zheng M. (2012). Test case generation from uml models. In *45th Annual Midwest Instruction and Computing Symposium, Cedar Falls, Iowa.* 1-9

Weyuker, Elaine J. (1988). Evaluating software complexity measures. *IEEE transactions on Software Engineering, 14*(9), 1357-1365.

Whittaker J. A. and Thomason M. (1994). A Markov chain model for statistical software testing. *Software Engineering, IEEE Transactions on, 20*(10): 812-824.

Zhu, Hong, *et al.* (1997). Software unit test coverage and adequacy. *Acm computing surveys (csur), 29*(4): 366-427.

APPENDICES A

(Test case generation technique by Boghdady *et al.*, 2011)

1.1 Test Case Generation Technique by Boghdady *et al.*, (2011)

The original technique to generate test cases from ATM withdrawal case studies has been proposed by Boghdady *et al.*, (2011) They proposes an approach for generating test cases from activity diagram. Activity diagram has been chosen as a source of test case generation because it is one of the important UML models that used in representing the activity flow in the system. In this experiment, an activity diagram for Automatic Teller Machine Pin Authentication (ATMPA) are created as shown in Figure 1.1.

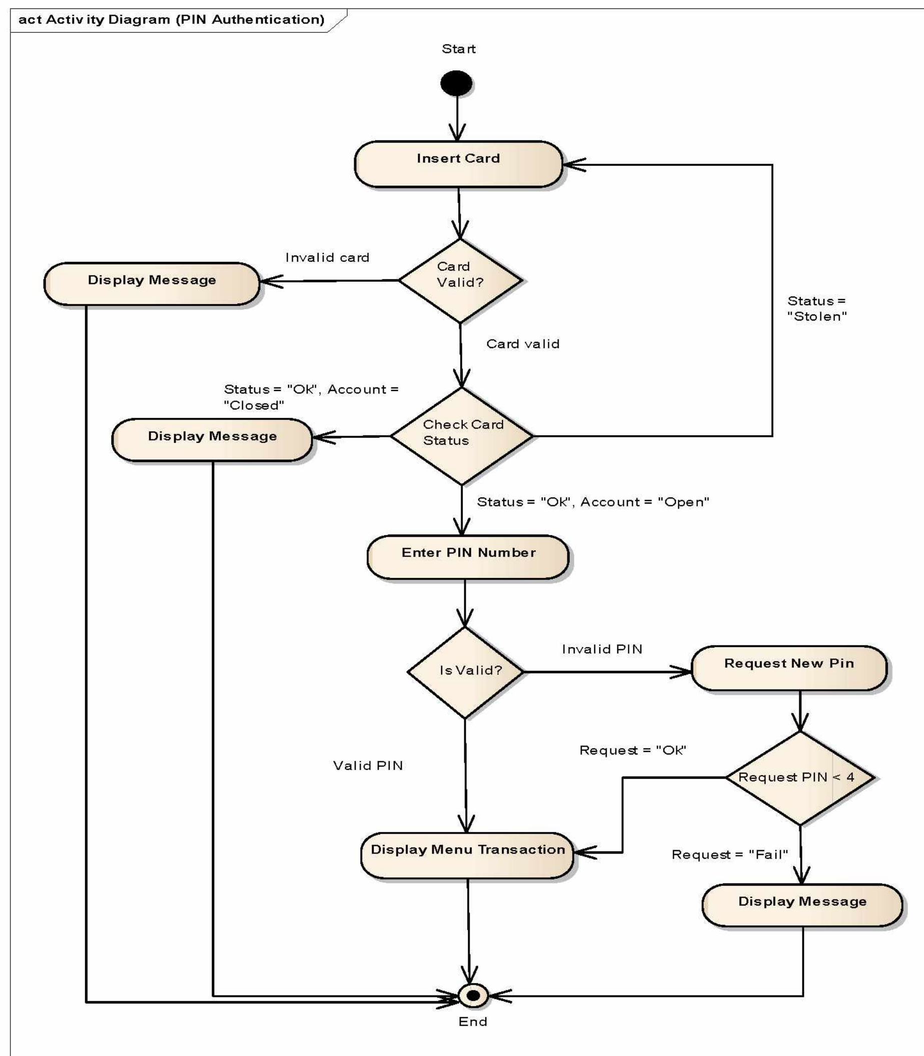

Figure 1.1 Activity Diagram for Automatic Teller Machine Pin Authentication (ATMPA)

1.1.1 Module 1: Generation of Activity Dependency Table (ADT)

In their work, they introduced an algorithm that automatically creates a table called Activity Dependency Table (ADT). This table will be used to creates a graph that called Activity Dependency Graph (ADG) and from this graph, the test cases is generated. In ADT, a symbol is given for each activities in activity diagram to showing the input and the expected output values for each activities. Table 1.1 shows the ADT for ATMPA.

Table 1.1: Activity Dependency Table for Automatic Teller Machine Pin Authentication (ATMPA)

Symbol	Activity Name	Controlling Entity: Entity Class	Dependency	Input	Expected Output
A	Insert Card	i: ATM Interface		Card	c:Card
B	Card Valid?	a: cardReader	A	Card	True (Valid card), False (Invalid Card)
C	Display Message	i: ATM Interface	B D I	False (Invalid card) False (Account closed) False (Invalid PIN)	"Please insert ATM card" "Account is closed" "PIN number is invalid"
D	Check Card Status	b: sessionMgr	B	True (Valid card)	True (Status = "Ok", Account = "Open"), False (Status = "Ok", Account = "Closed"), False (Status = "Stolen")

E	Enter PIN Number	b: sessionMgr	D	True (Status = “Ok”, Account = “Open”)	n: number PIN
F	Is Valid?	d: keyReader	E	n: number PIN	True (Valid PIN), False (Invalid PIN)
G	Display Menu Transacti on	i: ATM Interface	F	True (Valid PIN)	a: Bank
H	Request New PIN	d: keyReader	F	False (Invalid PIN)	r: request PIN
I	Request PIN < 4	d: keyReader	H	n: number PIN	True (Valid PIN < 4), False (Invalid PIN > 4, Cancel)
J	Return	i: ATM Interface	C G		r: request PIN

1.1.2 Module 2: Generation of Activity Dependency Graph (ADG)

After the creation of ADT, activity diagram is transformed into Activity Dependency Graph (ADG) using the symbol mentioned in Table 1. Each node in ADG represent the symbol that was mentioned in ADT. The transitions from one activity to another are represented by edges in the ADG. The presence of an edge from a node to another node is determined by checking the dependency column in the ADT for the current node’s symbol. Firstly, the nodes are created for all the symbols and draw a transition edge from each of the node refering to their dependency. The same procedures are applied continuously on every row in ADT to have the final ADG. Figure 1.2 illustrates the ADG for ATMPA.

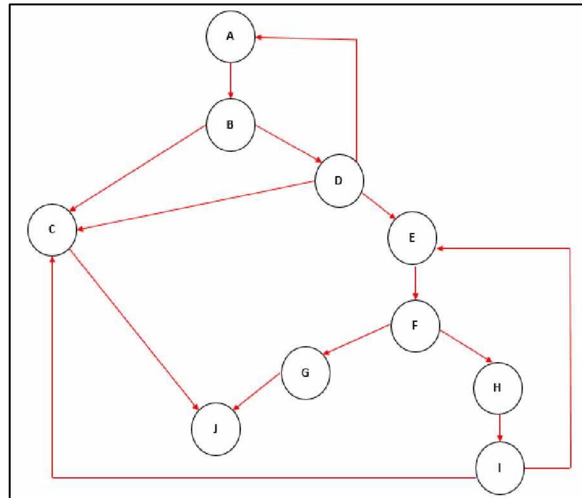

Figure 1.2 Activity Dependency Graph for Automatic Teller Machine Pin Authentication (ATMPA)

Figure 4.6 represent the ADG for ATMPA activity. Six possible paths are obtain from this ADG as listed below:

i. Path 1 : A-B-C-J
ii. Path 2 : A-B-D-C-J
iii. Path 3 : A-B-D-A
iv. Path 4 : A-B-D-E-F-G-J
v. Path 5 : A-B-D-E-F-H-I-C-J
vi. Path 6 : A-B-D-E-F-H-I-E-F-G-J

1.1.3 Module 3: Test Case Generation

To obtain the final test cases, the ADG is traversed and test cases are generated based on branch coverage criteria. The final test cases are generated using particular suggested algorithm called "GeneratingTestCasesSuite". This algorithm is apply on ADG to get all paths as listed above. All information are extracted from ADT and added to each test path to get the final test cases. Table 2.1 shows the generated test cases after the information is added to each test paths for ATMPA case study.

Table 2.1: Test Cases for Automatic Teller Machine Pin Authentication (ATMPA)

Test Case	Test Path	Node Input	Node Expected Output	Test Case Input	Test Case Expected Output
1	A B C	c: Card False (Invalid Card)	False (Invalid card) “Please insert ATM card”	Invalid Card	“Please insert ATM card”
2	A B D C J	c: Card True (Valid card) False (Status = “Ok”, Account = “Closed”	True (Valid card) False (Status = “Ok”, Account = “Closed” “Account is closed”	Valid Card	“Account is closed”
3	A B D A	c: Card True (Valid card) False (Status = “Stolen”)	True (Valid card) False (Status = “Ok”, Account = “Closed” Back to initial state	Valid Card	Back to initial state
4	A B D E F G J	c: Card True (Valid card) True (Status = “Ok”, Account = “Open” n: number PIN True (Valid PIN)	True (Valid card) True (Status = “Ok”, Account = “Open” n: number PIN True (Valid PIN)	Valid Card	Display Menu Transaction
5	A B D E H I C J	c: Card True (Valid card) True (Status = “Ok”, Account = “Open” n: number PIN False (Invalid PIN) r: request PIN False (Request PIN > 4)	True (Valid card) True (Status = “Ok”, Account = “Open” n: number PIN False (Invalid PIN) False (Request PIN > 4) “PIN number is invalid”	Valid Card	“PIN number is invalid” Reject card
6	A B D E H I E	c: Card True (Valid card) True (Status = “Ok”, Account = “Open” n: number PIN	True (Valid card) True (Status = “Ok”, Account = “Open” n: number PIN	Valid Card	Display Menu Transaction

	G J	False (Invalid PIN) r: request PIN True (Request PIN < 4)	False (Invalid PIN) True (Request PIN < 4)		

From Table 2.1, six final test cases was generated from ATMPA case study using activity diagram. The lists of test cases shows that test cases 5 and 6 cover iteration and looping process in this case study. The looping process perform when user repeat to enter their pin number if they insert an invalid pin number while iteration process perform when user repeatedly re-enter their pin number until they get a valid pin number. From the original technique, only test cases 1,2,3,4 and 6 was generated excluded test cases 5. In test cases 5, it cover faults in loop when user has entered a maximum number of trying the system will display "Pin number is invalid" and automatically eject the card.

APPENDICES B

(Test case generation technique by Sarma *et al.*, 2007)

2.1 Test Case Generation Technique by Sarma *et al*., (2007)

Technique that had been proposed by Sarma *et al.,* (2007) uses sequence diagram as a source of test case generation. In their approach, they created a UML sequence diagram and transform this diagram into a graphical representation called sequence diagram graph (SDG). Example of ATMPA case study is chosen to implement their technique. On the other hand, Figure 2.1 represents the implementation of their technique to another case study which is ATMS.

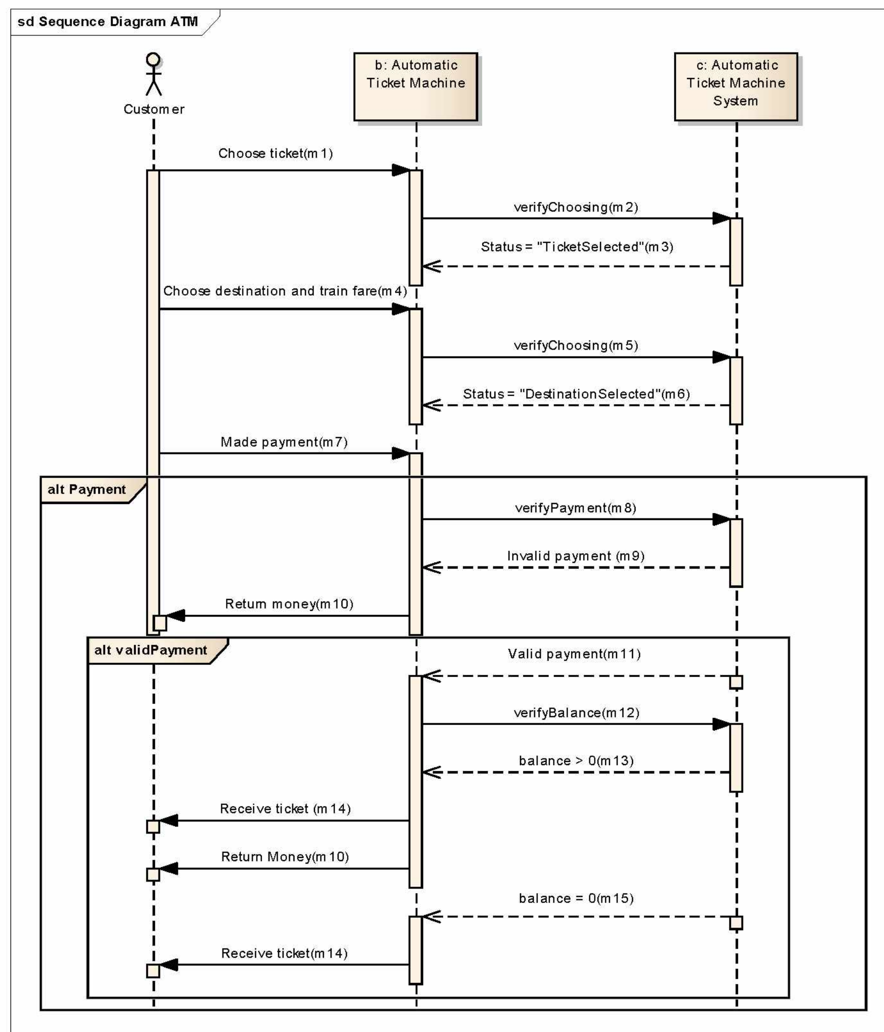

Figure 2.1 Sequence Diagram for Automatic Ticket Machine System (ATMS)

2.1.1 Module 1: Generation of Operation Scenario

To transform sequence diagram into sequence diagram graph (SDG), they proposed a methodology of operation scenario as a guide to form SDG. The operation scenario is define as a quadruple and definition of SDG is stated as below:

aOpnScn: <ScnId; StartState; MessageSet; NextState>

ScnId: A unique number that identifies each operation scenario.
StartState: Starting point of the *ScnId* that is, where a scenario starts.
MessageSet: Denotes the set of all events that occurs in an operation scenario.
NextState: The state that a system enters after the completion of a scenario.
An event that occurs in MessageSet denoted by a tuple:

<messageName; fromObject; toObject[/guard]>

An operation scenario is identified in order to create the SDG. From the sequence diagram in Figure 2.1, an operation scenarios are produced as shown in Table 2.1 for ATMS case study. State X is the start state for all the operation scenario, State Y is defined as a next state and State Z is a final state for all the operation.

Table 2.1: Three operation scenario represented in quadruple form for ATMS

<Scn1	**<Scn2**	**<Scn3**
State X	State X	State X
s1: (m1, a, b)	s1: (m1, a, b)	s1: (m1, a, b)
s2: (m2, b, c)	s2: (m2, b, c)	s2: (m2, b, c)
s3: (m3, c, b)	s3: (m3, c, b)	s3: (m3, c, b)
s4: (m4, a, b)	s4: (m4, a, b)	s4: (m4, a, b)
s5: (m5, b, c)	s5: (m5, b, c)	s5: (m5, b, c)
s6: (m6, c, b)	s6: (m6, c, b)	s6: (m6, c, b)
s7: (m7, a, b)	s7: (m7, a, b)	s7: (m7, a, b)
s8: (m8, b, c)	s8: (m8, b, c)	s8: (m8, b, c)

s9: (m9, c, b)	s11: (m11, c, b)	s11: (m11, c, b)
s10: (m10, b, a)	s12: (m12, b, c)	s12: (m12, b, c)
State Y>	s13: (m13, c, b)	s13: (m13, c, b)
	s14: (m14, b, a)	s15: (m14, c, b)
	s10: (m10, b, a)	s14: (m14, b, a)
	State Y>	State Z>

2.1.2 Module 2: Transformation Sequence Diagram into Sequence Diagram Graph (SDG)

After identifying all of the operation scenarios, the sequence diagram graph is created. There are transitions that occur from state X to state Y in each operation scenario. Each node connected in SDG is referring to these transitions. Figure 2.2 represent the SDG for ATMS.

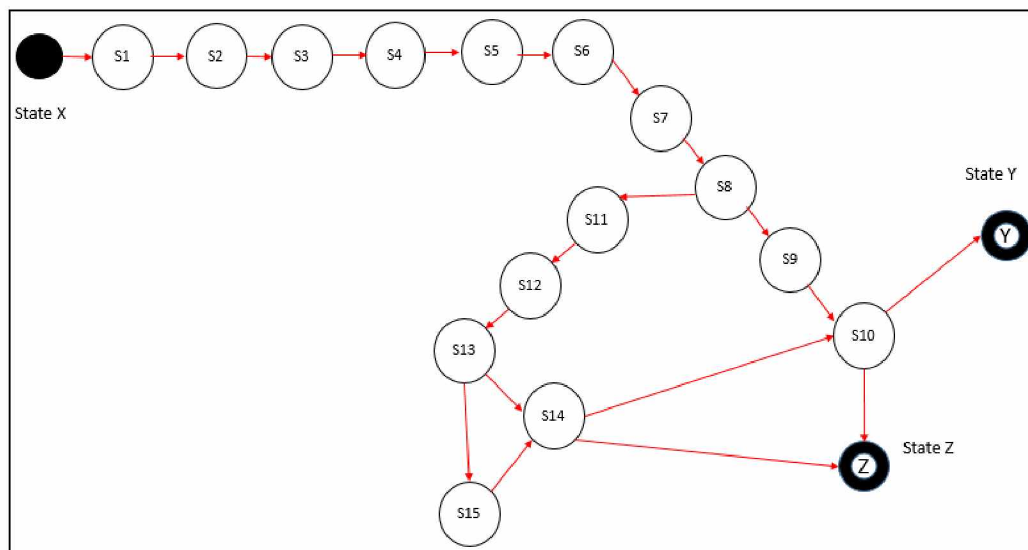

Figure 2.2 Sequence Diagram Graph for Automatic Ticket Machine System (ATMS)

SDG contains information that is needed for test cases generation. Each node in SDG is mapped to a communication from object A to object B in message set. Besides that, SDG

covers all paths from the start node to the final node to cover all interactions that occur in ATMS activity and to obtain the test paths. From the start node X to the final node Y and Z, three test paths has been obtained as listed below:

i. Path 1 : S1-S2-S3-S4-S5-S6-S7-S8-S9-S10 = Unsuccessful
ii. Path 2 : S1-S2-S3-S4-S5-S6-S7-S8-S11-S12-S13-S14-S10 = Successful
iii. Path 3 : S1-S2-S3-S4-S5-S6-S7-S8-S11-S12-S13-S15-S14 = Successful

To generate test cases, all the paths that are collected from SDG will be used as a reference to produce a set of test cases. Each test path will fetch the input and expected output for indexing the table of final set of test cases.

2.1.3 Module 3: Test Case Generation

To generate test case, a set of test sets defined to detect any faults when an object invokes a method of another object or whether message passing follow the right sequence to finish an operation. Coverage criteria is defined whereby given a test set A and sequence diagram B, T must cause each sequence of message path exercised at least once. The algorithm called "AlgorithmTestSetGeneration" was used to generate test set satisfying the coverage criterion. The SDG is traversed based on this coverage criteria and fault model to make sure each path in SDG would be visited to generate test cases. Three final generated test cases for ATMS are shown in Table 2.2.

Table 2.2: Test Cases for Automatic Ticket Machine System (ATMS)

Test Case Scenario	Input	Output	Post-Conditions
1	Ticket = "Select" Payment = "Insufficient"	Get Money	Display Menu
2	Ticket = "Select" Payment = "Sufficient" Balance = 0	Receive Ticket	Display Menu

3	Ticket = "Select" Payment = "Sufficient" Balance > 0	Receive Ticket and Get Money	Display Menu

Table 2.2 lists three final test cases for ATMS case study when sequence diagram were used for the generation. As can be seen in table above, three final test cases was derived compared to six final test cases when Boghdady *et al.* (2011) techniques was applied in this case study. From the table, test cases 1 until 3 cover a basis process to buy a train ticket such as choosing ticket, made payment and receive ticket. From the Figure 2.1, the sequence diagram for ATMS case study did not consider the looping activity in this case study. If the looping activity that occur when user make a selection of ticket type and destination is captured in the sequence diagram, another three test cases can be generated on this case study.

APPENDICES C

(Test case generation technique by Swain *et al.,* 2012)

3.1 Test Case Generation Technique by Swain *et al.,* (2012)

Using Swain *et al.,* (2012) technique, test cases are generated from state chart diagram and is implemented on ATMS case study. They choose state chart diagram as a source of test cases generation because it can give an abstract description of the behavior of a system and model dynamic nature of a system. In their work, they derived state transition graph (STG) from state chart diagram and all information to generate test cases are extracted from the STG. They also include the minimization of test cases by calculating node coverage for each test case to determine which test cases are covered by other test cases. In this experiment, their technique are implemented on another case study that is ATMW. Figure 3.1 represent the state chart diagram for ATMW case studies.

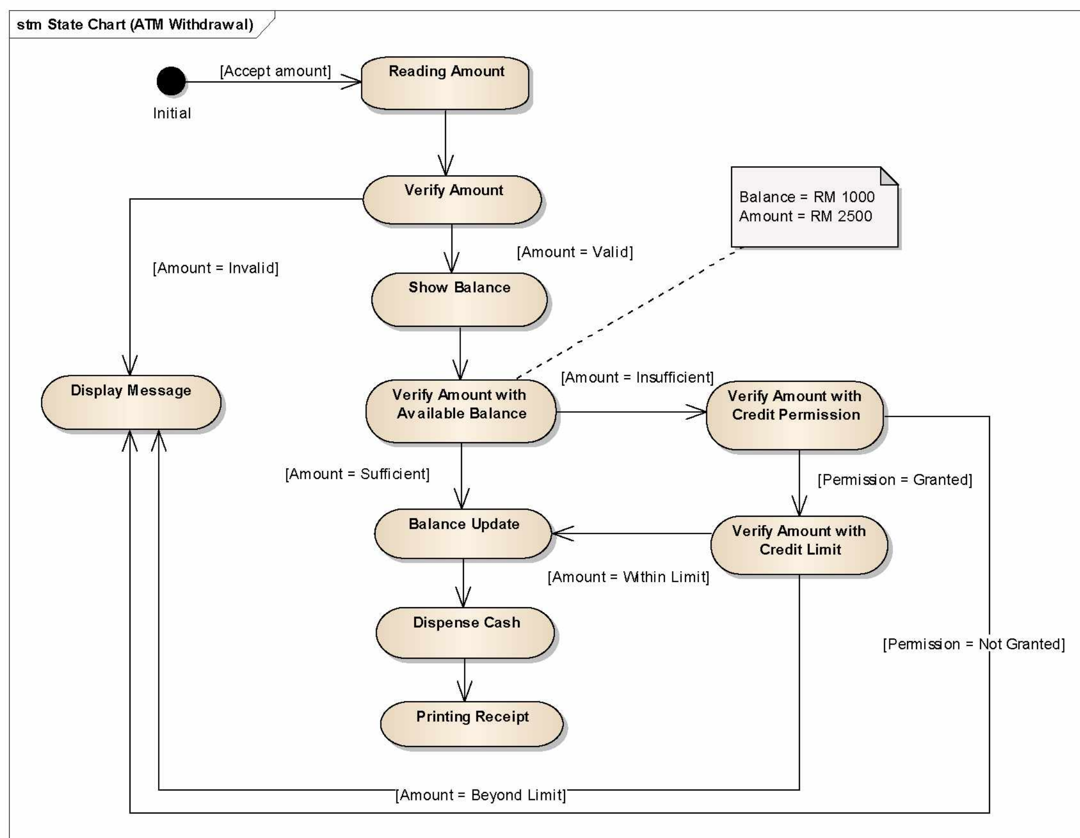

Figure 3.1 State Chart Diagram for Automatic Teller Machine Withdrawal (ATMW)

3.1.1 Module 1: Converting State Chart Diagram into State Transition Graph (STG)

For generating state transition graph, a definition of transition graph has been defined. A transition graph TG = (V_t, E_d) which (V_t) represents a set of vertices and (E_d) consisting a set of directed edges. In state chart diagram, a state will represent nodes and transitions between states will represent edges in STG. Besides that, each transition from one node to another will have a set of stages (ST) that consists of input data (ID), output data (OD) and also transition (TR). The STG for ATMW is presented in Figure 3.2

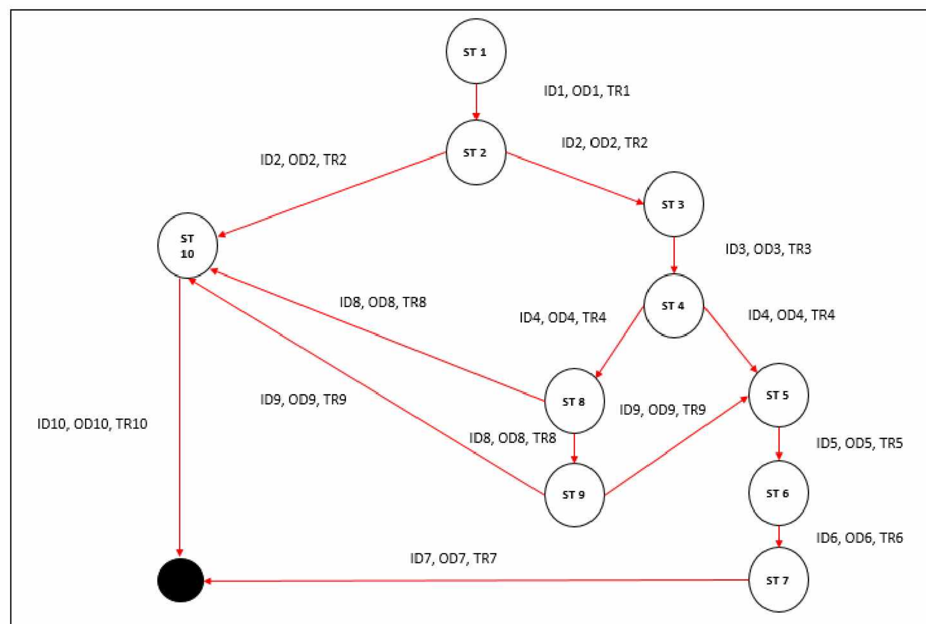

Figure 3.2 State Transition Graph for Automatic Teller Machine Withdrawal (ATMW)

From all the STG above, it can be seen that each transition from one node to next node has a set of stages that defines the input, output and transitions that occur for each activity in the system functions. To extract all of the information which are required to generate test sequence, this set of stages needs to show the connection between one nodes to the next node. This graph is then traversed using simple transition coverage by visiting all vertices in the graph.

3.1.2 Module 2: Test Case Generation

For generation of test cases, they applied the graphical travelling salesman problem which are determined the shortest possible path that visits each node exactly once and returns to the initial node. The graphical travelling salesman problem is defined as, if n is greater than 1, the transition graph has to be transformed $k - 1$ times. All possible paths of transition k will be obtain from the STG and added to the set of test cases. Before test cases are generated, all required information from the STG for ATMW are described as below:

ST= {St1, St2, St3, St4, St5, St6, St7, St8, St9, St10}

ID = {ID1, ID2, ID3, ID4, ID5, ID6, ID7, ID8, ID9, ID10}

OD = {OD1, OD2, OD3, OD4, OD5, OD6, OD7, OD8, OD9, OD10}

TR = {TR1, TR2, TR3, TR4, TR5, TR6, TR7, TR8, TR9, TR10}

TR1: {St1, St2}

TR2: {St2, St3}, {St2, St10)

TR3: {St3, St4}

TR4: {St4, St5}, {St4, St8},

TR5: {St5, St6}

TR6: {St6, St7}

TR7: {St7, Final}

TR8: {St8, St9}, {St8, St10}

TR9: {St9, St5}, {St9, St10}

TR 10: {St10, Final}

From the information above, it shows that transition between source and destination stage in TR2, TR4, TR8 and TR9 consists of two source stage. The test cases generated from this case study is shows in Table 3.1.

Table 3.1: Test Cases for Automatic Teller Machine Withdrawal (ATMW)

Test Cases	Information Extracted
1	{ST1, ST2, ST3, ST4, ST5, ST6, ID1, ID2, ID3, ID4, ID5, OD1, OD2, OD3, OD4, OD5, TR1, TR2, TR3, TR4, TR5}
2	{ST1, ST2, ST3, ST4, ST5, ST7, ST6, ID1, ID2, ID3, ID4, ID5, ID6, ID7, OD1, OD2, OD3, OD4, OD5, OD6, OD7, TR1, TR2, TR3, TR4, TR5, TR6, TR7}
3	{ST2, ST3, ST4, ST5, ST7, ST6, ID2, ID3, ID4, ID5, ID6, ID7, OD2, OD3, OD4, OD5, OD6, OD7, TR2, TR3, TR4, TR5, TR6, TR7}
4	{ST3, ST4, ST5, ST7, ST6, ID3, ID4, ID5, ID6, ID7, OD3, OD4, OD5, OD6, OD7,TR3, TR4, TR5, TR6, TR7}
5	{ST4, ST5, ST7, ST6, ID4, ID5, ID6, ID7,OD4, OD5, OD6, OD7, TR4, TR5, TR6, TR7}

Table 3.1 lists test cases that are generated from ATMW case study using state chart diagram. Based on the table, five test cases has been produce using this technique. We observed that the number of generated test cases from the three technique that are applied on ATMW case study is consistent using activity diagram, sequence diagram and state chart diagram. The reason for this because ATMW case study is free from looping and iteration activities and the three type of UML diagram can be used to generated test cases.