A TEST CASE GENERATION FRAMEWORK BASED ON UML STATECHART DIAGRAM

YASIR DAWOOD SALMAN

DOCTOR OF PHILOSOPHY
UNIVERSITI UTARA MALAYSIA
2018

Awang Had Salleh
Graduate School
of Arts And Sciences

Universiti Utara Malaysia

PERAKUAN KERJA TESIS / DISERTASI
(Certification of thesis / dissertation)

Kami, yang bertandatangan, memperakukan bahawa
(We, the undersigned, certify that)

YASIR DAWOOD SALMAN

calon untuk Ijazah **PhD**
(candidate for the degree of)

telah mengemukakan tesis / disertasi yang bertajuk:
(has presented his/her thesis / dissertation of the following title):

"A TEST CASE GENERATION FRAMEWORK BASED ON UML STATECHART DIAGRAM"

seperti yang tercatat di muka surat tajuk dan kulit tesis / disertasi.
(as it appears on the title page and front cover of the thesis / dissertation).

Bahawa tesis/disertasi tersebut boleh diterima dari segi bentuk serta kandungan dan meliputi bidang ilmu dengan memuaskan, sebagaimana yang ditunjukkan oleh calon dalam ujian lisan yang diadakan pada : ***10 October 2017.***
That the said thesis/dissertation is acceptable in form and content and displays a satisfactory knowledge of the field of study as demonstrated by the candidate through an oral examination held on: ***October 10, 2017.***

Pengerusi Viva: *(Chairman for VIVA)*	**Assoc. Prof. Dr. Yuhanis Yusof**	Tandatangan *(Signature)*
Pemeriksa Luar: *(External Examiner)*	**Prof. Dr. Abu Bakar Md Sultan**	Tandatangan *(Signature)*
Pemeriksa Dalam: *(Internal Examiner)*	**Dr. Rohaida Romli**	Tandatangan *(Signature)*
Nama Penyelia/Penyelia-penyelia: *(Name of Supervisor/Supervisors)*	**Dr. Nor Laily Hashim**	Tandatangan *(Signature)*

Tarikh:
(Date) ***October 10, 2017***

Permission to Use

In presenting this thesis in fulfilment of the requirements for a postgraduate degree from Universiti Utara Malaysia, I agree that the Universiti Library may make it freely available for inspection. I further agree that permission for the copying of this thesis in any manner, in whole or in part, for scholarly purpose may be granted by my supervisor(s) or, in their absence, by the Dean of Awang Had Salleh Graduate School of Arts and Sciences. It is understood that any copying, publication, or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to Universiti Utara Malaysia for any scholarly use which may be made of any material from my thesis.

Requests for permission to copy or to make other use of materials in this thesis, in completely or in part should be addressed to:

Dean of Awang Had Salleh Graduate School of Arts and Sciences
UUM College of Arts and Sciences
Universiti Utara Malaysia
06010 UUM Sintok

Abstrak

Pengesanan awal kesalahan perisian menawarkan lebih fleksibiliti untuk membetulkan kesalahan tersebut pada peringkat awal pembangunan sistem. Malangnya, kajian sedia ada masih belum cukup menyeluruh dalam menerangkan proses utama penjanaan kes ujian secara automatik. Malahan algoritma yang digunakan dalam penjanaan ujian kes tidak disediakan atau diterangkan dengan jelas. Kajian semasa juga hampir tidak menangani isu gelung dan laluan selari, malahan kriteria liputan yang dicapai adalah rendah. Oleh itu, kajian ini mencadangkan satu kerangka penjanaan kes ujian yang menjana kes ujian yang diminimumkan dan diprioritasikan daripada gambarajah UML keadaan dengan kriteria liputan yang lebih tinggi. Kajian literatur telah dilaksanakan untuk mengenal pasti isu dan jurang yang berkaitan penjanaan kes ujian, pengujian berasaskan model, dan kriteria liputan. Kerangka yang dicadangkan ini direka bentuk hasil daripada maklumat yang dikumpul dan telah mengenalpasti lapan komponen yang mewakili proses dalam penjanaan kes ujian. Komponen tersebut adalah jadual hubungan, graf hubungan, pemeriksaan konsistensi, meminimumkan laluan ujian, memprioritasikan laluan ujian, pemangkasan laluan, penjanaan laluan ujian dan penjanaan kes ujian. Sebagai tambahan, satu prototaip untuk melaksanakan kerangka turut dibangunkan. Penilaian kerangka yang dibangunkan melibatkan tiga fasa: prototaip, perbandingan dengan kajian terdahulu dan ulasan pakar. Dapatan kajian menunjukkan kriteria liputan yang paling sesuai bagi gambarajah UML keadaan adalah liputan semua keadaan, liputan semua peralihan, liputan semua pasangan peralihan, dan liputan semua laluan gelung bebas. Selain itu, kajian ini mencapai kriteria liputan yang lebih tinggi dalam semua kriteria liputan yang dinyatakan di atas, kecuali liputan semua keadaan apabila dibandingkan dengan kajian sebelumnya. Hasil ulasan pakar menunjukkan bahawa pakar domain bersetuju bahawa kerangka yang dicadangkan ini adalah praktikal, mudah untuk dilaksanakan kerana kesesuaiannya dalam menjanakan kes ujian. Algoritma yang dicadangkan menghasilkan keputusan yang betul, dan prototaip berupaya menjana kes ujian dengan berkesan. Secara umumnya, sistem yang dicadangkan diterima baik oleh pakar berdasarkan aspek kebergunaan, kebolehgunaan, dan ketepatannya. Kajian ini menyumbang secara teori dan praktikal dengan menyediakan kerangka penjanaan kes ujian alternatif awal yang mencapai liputan yang tinggi dan dapat dilaksanakan dengan efektif menggunakan gambarajah UML keadaan. Kajian ini turut menambahkan pengetahuan baru dalam bidang pengujian perisian khususnya kepada proses pengujian dalam teknik berasaskan model, aktiviti pengujian, dan alat sokongan pengujian.

Kata kunci: Kerangka penjanaan kes ujian, liputan gelung, laluan selari, kes ujian yang diminimumkan, kes ujian yang diprioritasikan.

Abstract

Early software fault detection offers more flexibility to correct errors in the early development stages. Unfortunately, existing studies in this domain are not sufficiently comprehensive in describing the major processes of the automated test case generation. Furthermore, the algorithms used for test case generation are not provided or well described. Current studies also hardly address loops and parallel paths issues, and achieved low coverage criteria. Therefore, this study proposes a test case generation framework that generates minimized and prioritized test cases from UML statechart diagram with higher coverage criteria. This study, conducted a review of the previous research to identify the issues and gaps related to test case generation, model-based testing, and coverage criteria. The proposed framework was designed from the gathered information based on the reviews and consists of eight components that represent a comprehensive test case generation processes. They are relation table, relation graph, consistency checking, test path minimization, test path prioritization, path pruning, test path generation, and test case generation. In addition, a prototype to implement the framework was developed. The evaluation of the framework was conducted in three phases: prototyping, comparison with previous studies, and expert review. The results reveal that the most suitable coverage criteria for UML statechart diagram are all-states coverage, all-transitions coverage, all-transition-pairs coverage, and all-loop-free-paths coverage. Furthermore, this study achieves higher coverage criteria in all coverage criteria, except for all-state coverage, when compared with the previous studies. The results of the experts' review show that the framework is practical, easy to implement due to it is suitability to generate the test cases. The proposed algorithms provide correct results, and the prototype is able to generate test case effectively. Generally, the proposed system is well accepted by experts owing to its usefulness, usability, and accuracy. This study contributes to both theory and practice by providing an early alternative test case generation framework that achieves high coverage and can effectively generate test cases from UML statechart diagrams. This research adds new knowledge to the software testing field, especially for testing processes in the model-based techniques, testing activity, and testing tool support.

Keywords: Test case generation framework, loop coverage, parallel path, minimized test cases, prioritized test cases

Acknowledgement

All praises and thanks to the Almighty, Allah (SWT), for giving me the strength, the patience, and the opportunity to complete this thesis. Besides, completing this thesis would not have been possible without a number of people who offered their unfailing support throughout the period of the study.

I would like to express my sincerest thanks and gratitude to my supervisor Dr. Nor Laily Binti Hashim for the continuous support of my PhD study and related research, for her patience, motivation, and immense knowledge. Her guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my PhD study.

To my family, thank you for encouraging me in all of my pursuits and inspiring me to follow my dreams. I am especially grateful to my parents, who have been a constant source of inspiration to me. It is their love, patience and encouragement during the PhD period that helped me through hardships. I always knew that you believed in me and wanted the best for me. My special gratitude to my brother and sister for supporting me spiritually throughout writing this thesis and my life in general.

I would also like to thank my colleagues at Information Technology, Universiti Utara Malaysia for their encouragement and support throughout this journey. I would also like also to extend my thanks and appreciation to all of my friends who have contributed in one way or another to help me complete this thesis successfully.

I could not have completed my thesis without the support of all these wonderful people!

Table of Contents

List of Tables

List of Figures

List of Appendices

List of Publications

Published

Hashim, N. L., & **Salman, Y. D.** (2011). An improved algorithm in test case generation from UML activity diagram using activity path. *Proceedings of the 3rd International Conference on Computing and Informatics, ICOCI*.

Salman, Y. D., & Hashim, N. L. (2014). An improved method of obtaining basic path testing for test case based on UML state chart. *Science International, 26*(4).

Salman, Y. D., & Hashim, N. L. (2016). Automatic Test Case Generation from UML State Chart Diagram: A Survey *Advanced Computer and Communication Engineering Technology* (pp. 123-134): Springer.

Salman, Y. D., Hashim, N. L., Rejab, M. M., Romli, R., & Mohd, H. (2017a). *Coverage criteria for test case generation using UML state chart diagram.* Paper presented at the AIP Conference Proceedings.

Salman, Y. D., Hashim, N. L., Rejab, M. M., Romli, R., & Mohd, H. (2017b). Coverage Criteria for UML State Chart Diagram in Model-based Testing. *Journal of Telecommunication, Electronic and Computer Engineering (JTEC), 9*(2-11), 85-89.

Salman, Y. D., & Hashim, N. L. (2017). Test Case Generation Model for UML Diagrams. *Journal of Telecommunication, Electronic and Computer Engineering (JTEC),* 9(2-2), 171-175.

Accepted

Salman, Y. D., Hashim, N. L., Rejab, M. M., Romli, R., & Mohd, H. (2017). Generating test cases for model-based testing and detecting deadlocks using Tarjan's algorithm.

Salman, Y. D. & Hashim, N. L. (2017). A Test Cases Minimization And Prioritized Method Using Firefly Algorithm Based On UML State Chart Diagram.

List of Abbreviations

BFS	Breath First Search
DFS	Depth First Search
EFSM	Extended Finite State Machine
FSM	Finite State Machine
GeMiTefSc	Generation and Minimization of Test Cases from State Chart
GTSC	Automated Generated Test Case Based on Statechart
IOCO	Input/Output Conformance
MBT	Model-based testing
OCL	Object Constraint Language
OMDAG	Object Method Acyclic Graph
OMG	Object Management Group
POS	Practical Swarm Optimization
PSO	Particle Swarm Optimization
RAD	Rapid Application Development
SAD	State Activity Diagram
SCCF	Statechart Coverage Criteria Family
SCOTEM	State COllaboration TEst Model
SRT	State Relationship Table
SRG	State Relationship Graph
SUT	System Under Test
TCG	Test Cases Generation
TCGP	Test Case Generation Paths
TeGeMiOOSc	Test Generation and Minimization for O-O Software with State Charts
TFG	Test Flow Graph
TGV	Test Generation with Verification
UML	Unified Modelling Language
VDT	Vertex Description Table
XML	Extensible Markup Language

CHAPTER ONE
INTRODUCTION

1.1 Introduction

This introductory chapter deliberates on the motivational aspects of software testing in general and automatic test case generation in practice, and focuses on using Unified Modelling Language (UML) diagrams as inputs to generate test cases.

This chapter presents the introduction to this study, beginning with the background of the study, which includes the background of software testing and automatic test case generation and the related literature. The next sections present the research problems, research questions, and research objectives. Subsequently, the scope of the research and the research framework will be discussed. Finally, the significance of the study and the terminologies will be presented. This chapter is concluded with an outline of the remaining chapters of this thesis.

1.2 Background of the Study

Computers and software are some of the major innovations in the history of mankind (Srivastav & Gupta, 2016). The use of computers plays a key role in the daily lives of people. The significant roles of computers in society and the increasing demand for complex computer applications makes software development difficult for software developers (Chavez, Shen, France, Mechling, & Li, 2016). Thus, the effort exerted and the cost of software development testing ultimately increases (Chen & Li, 2010).

Testing has been proven as an essential tool in enhancing the quality of code programming. Testing is also considered a critical part of today's software development (Rungi & Matulevičius, 2013). Therefore, in the practice of software development, testing remains the most vital part of quality assurance (Sood & Rattan, 2016).

Software testing aids in detecting software bugs and errors that cannot be detected by compilers (Patwa & Malviya, 2014). Furthermore, testing can guarantee software correctness to enhance the quality of the system (Tan, 2003). However, software testing is considered as a complicated task that requires the software tester to illustrate whether its purpose is achieved (Kim, Porter, & Rothermel, 2005). Practically or theoretically, testing is generally a difficult task.

Testing consumes a substantial amount of development time. Thus, developing an automatic test case generation algorithm for Model-Based Testing (MBT), which supports the commencement of the software testing process immediately after the design phase of the system lifecycle or as soon as the modelled requirements becomes available, is imperative (Oluwagbemi & Asmuni, 2014). Software testing is considered a critical part of the software development lifecycle (Gulia & Chugh, 2015) because software testing is performed during software development through a sequence of instructions of test inputs followed by expected outputs (Sahoo, Ojha, Mohapatra, & Patra, 2016b).

One of the software testing methods is the test case generation, where test cases can be declared as a classification of variables or conditions that fulfilled specific test coverage criteria (Wu & Fan, 2014). Coverage criteria are rules or requirements that

need to be satisfied by the test cases (Offutt & Abdurazik, 1999). Researchers stated that a significant amount of research has been targeted toward automated test case generation techniques (Xu, Kim, Kim, Rothermel, & Cohen, 2010; Yu, Martinez, Danglot, Durieux, & Monperrus, 2017). Generating test cases is considered an important activity in the MBT process (Li, Li, He, & Xiong, 2013a; Oluwagbemi & Asmuni, 2014). Software testers will discover whether a software system is executed according to the system requirements and the sequences of its executions by using test cases (Li et al., 2013a). However, test case generation is the most challenging and an extensively researched activity (Bertolino, 2003). Therefore, improving its effectiveness and reducing the cost of software testing by automating the test case generation have significant benefits (Rafi, Moses, Petersen, & Mäntylä, 2012).

Test cases could be generated from requirement specifications and design documents. For example, the UML statechart diagram is one of the diagrams used in the early life cycle of a system design (Lu & Tseng, 2010). Thus, this diagram can be used to generate test cases for software development to improve the efficiency and effectiveness of software testing (Kumaran, Kumar, & Kumar, 2011).

Test cases can be fully automated, in which the generation, evaluation, and execution of unit test cases can be automated and integrated into the programming task (Belli, Hollmann, & Kleinselbeck, 2009; Salman & Hashim, 2014). Then, writing test cases for bugs that are difficult to detect in automatic systems would be the only job of software testers. Test case generation becomes one of the most critical knowledge-demanding tasks because of its strong impact on the efficiency and effectiveness of the complete testing process (Bertolino, 2007; Zhu, Hall, & May, 1997).

A test case is a description of a test with mechanisms that describe inputs, events, or actions, and expected responses to define whether the feature of an application works properly (Shinde, 2013). Test cases are typically generated from manual or automatic inputs. Manual generation depends on the expertise of software testers who must perform the testing and detecting of errors (Sung & Paynter, 2006). Furthermore, the existing methods for the automatic generation of test cases needs to be enhanced and improved (Koong et al., 2012). The enhancement of the automatic generation of test cases can be implemented using minimization and prioritization (Singh & Shree, 2016). Therefore, there are advantages in developing such mechanism that increases the coverage and diversity of test cases, while minimize and prioritize the generated test cases (Singh & Shree, 2016).

There are a number of different approaches to aid the test case generation. The process in test case minimization is to identify and then eliminate the obsolete or redundant test case(s) from the generated test cases (Santosh & Singh, 2013). For the prioritization approach, the process is to identify the 'ideal' test cases that maximize desirable properties (Yoo & Harman, 2012). It was proved by empirical studies that the implementation of minimization and prioritization techniques in test case generation can be effective (Kim et al., 2005; Yoo & Harman, 2012).

A significant amount of research for the past decades has focused on automatic test case generation (Cartaxo, Neto, & Machado, 2007; Heumann, 2001; Javed, Strooper, & Watson, 2007; Kim, Kang, Baik, & Ko, 2007; Krishnachandra, 2016; Kundu & Samanta, 2009; Lilly & Uma, 2010; Linzhang et al., 2004; Mingsong et al., 2009; Mingsong, Xiaokang, & Xuandong, 2006; Oluwagbemi & Asmuni, 2015; Sahoo,

Mohapatra, & Patra, 2016a). Therefore, numerous techniques have been explored by researchers intensively and propelled to generate test cases. By contrast, software systems have become progressively complex (Salman & Hashim, 2016). For instance, they now use distinctive techniques within diverse programming languages and run on diverse platforms with components developed by diverse vendors (Anand et al., 2013).

The methods for automatic test case generation techniques that use UML-based testing can be categorised in several ways. The most common way to classify testing approaches is by diagram type (Shirole & Kumar, 2013). However, test case generation using these diagrams are ruled by intermediate representations in which these diagrams are converted. Therefore, this analysis can broadly classify testing approaches to metaheuristic and MBT techniques (Shirole & Kumar, 2013). These techniques focus on using UML models to generate test cases to enhance the testing of the system under test (SUT). The SUT can be usually described as a component within a containment tree (Gross, 2005), and are executed at the system, unit, and incorporation stages (Shirole & Kumar, 2013). The researcher work are based on these techniques, intermediate models, and coverage criteria that are clarified in Chapter 2 (section 2.7) to understand the key features of several test case generation research methods.

Metaheuristic refers to a process that pursues a solution to an optimization problem. However, finding a solution is not guaranteed (Eusuff, Lansey, & Pasha, 2006). Is uses a heuristic function, as a human would do, to guide the search. The heuristic search can either be a blind search or an informed search (Eusuff et al., 2006). Several metaheuristic techniques that generate test cases exist, such as ant colony optimization,

hill climbing, particle swarm optimization (PSO), genetic algorithm, simulated annealing, artificial immune systems, alternating variable method, and genetic programming (Shirole & Kumar, 2013). Ant colony optimization and genetic algorithm are the most widely used metaheuristic techniques in test case generation using UML (Sharma, 2014).

MBT is used to validate requirements, check the requirement's consistencies, and generate test cases that are focused on the behavioural aspects of the software (Society, 2014). Statechart diagrams, activity diagrams, and sequence diagrams, are the most commonly used UML structures to generate test cases (Shirole & Kumar, 2013).

During software development, UML is used to visualize, document, and specify the models of the software systems, including their designs and structures (Rumbaugh, Jacobson, & Booch, 2004). UML is one of the best modelling tools that can manage complex and large systems (Pandey & Jain, 2014). Furthermore, UML is the language that creates models, provides a life cycle, which is widely used to designate design and analysis the software specifications (Biswal, Nanda, & Mohapatra, 2008), and supports software development. However, the test case generation from the UML diagram is considered as a major challenge for researchers because of its implementation and covering most of the system under development (Gulia & Chillar, 2012; Schweighofer & Heričko, 2014; Tripathy & Mitra, 2012).

In previous studies, UML diagrams that are based on automatic test case generation has gained much attention by many studies (Hashim & Salman, 2011; Li, He, & Wu, 2012; Li et al., 2013a; Prasanna & Chandran, 2011; Schweighofer & Heričko, 2014; Shirole, Suthar, & Kumar, 2011; Swain, Panthi, Behera, & Mohapatra, 2012c).

Accurately generated test scenarios are vital to achieve test suitability, ensure software quality, and coverage criteria (Prasanna & Chandran, 2011). Moreover, UML diagrams would help software testers and developers understand the behaviours and dynamic properties of the system (Prasanna & Chandran, 2011).

The UML statechart diagram categorises the performance of a computer program or other processed works (Felicie, 2012). This diagram has many possible states. Entities or sub-entities are always in one of these states. In addition, the conditional transfer from one state to another is possible and well defined. Furthermore, this diagram can be applied as a model that generates test cases (Felicie, 2012).

The UML statechart diagram is a better option than other UML diagrams in test case generation because its lifecycle and the changes that it endures upon the delivery of an event are shown (Swain, Mohapatra, & Mall, 2010a). This diagram can also reveal unit-level faults (Abdurazik, Offutt, & Baldini, 2004). For example, a UML statechart diagram delivers further explanation of the action orders of the external system that are handled and recognized by the systems (Kumaran et al., 2011).

Coverage criteria is usually a rule or requirement that test cases need to satisfy (Paul & Jeff, 2008). According to Utting and Legeard (2010), many types of coverage criteria can be used with the UML statechart diagram, such as all-states coverage, all-configurations coverage, all-transitions coverage, all-transition-pairs coverage, all-loop-free-paths coverage, all-one-loop-paths coverage, all-round-trips coverage, and all-paths coverage.

The most frequently used theory to generate test cases from UML diagrams is the graph theory. In addition, in graph theory, depth-first search (DFS) is one of the basic algorithms used to generate test cases from the UML diagrams (Lammich & Neumann, 2015). DFS traverses the graph or model for as long as possible (i.e., until no non-visited vertex is left) to cover every branch before pursuing back and is the base of a gathering of automata and graph algorithms (Lammich & Neumann, 2015). In addition, many researchers have used DFS to identify all the possible paths of the test graph for the test cases to achieve the all-transition coverage (Swain et al., 2012c). However, the use of DFS in traversing loops will result in multiple appearances of some paths or path combinations in the test sequence (Mingsong et al., 2006). Therefore, DFS has subordinate coverage in other types, such as all-loop-free-paths coverage, all-one-loop-paths coverage, and all-round-trips coverage because the full combination of decision and loop states will result in path explosion (Mingsong et al., 2006).

This study investigates software testing and automatic test case generation. The statechart diagram from UML is used as a base for test case generation to develop the most suitable test framework before the programs are finalized in the design phase. Furthermore, a method is proposed to minimise the number of test cases and prioritize the test cases. Therefore, this study will generate test cases with the highest coverage criteria in the smallest possible number of test cases to decrease the manual process and the faults caused by human interaction.

1.3 Problem Statement

The error-finding cycle cost varies significantly from the software development life cycles (Stecklein et al., 2004). If software testing can be performed in the early stage, then the error can also be detected earlier. Therefore, the development period and expenses are reduced (Kim, Son, & Kim, 2011; Yadav, Patel, Arora, Uptu, & Jnu, 2016). In this study, the proposed approach identifies errors during the design phase using the UML statechart diagram by automatically generate test cases. There are significantly benefit from the automation and generation of this test cases (Binder, 2000). However, producing a large amount of test data will result in difficulties in testing. Therefore, the software tester that handles the test data will greatly benefit when the test data are minimized and prioritized (Rhmann & Saxena, 2016).

UML statechart diagram is required during design phase in the software development process (Felderer & Herrmann, 2015; Kumaran et al., 2011; Murthy, Anitha, Mahesh, & Subramanyan, 2006; Schweighofer & Heričko, 2014; Tsumaki & Morisawa, 2000). The UML statechart diagram is a better option than other UML diagrams in test case generation, because its lifecycle and the changes that it endures upon the delivery of an event are shown (Swain et al., 2010a). Also its specifies the transition of one object in the system (Tsumaki & Morisawa, 2000), during its life and the stimuli that cause the object to change its state (Shirole et al., 2011). State charts are used to represent the behaviour of an object (Shirole et al., 2011). Typically, it is used for describing the behaviour of classes. It shows how an object will react to an event (Swain et al., 2010a). The UML statechart diagram test cases can reveal unit-level faults better than other diagrams (Abdurazik et al., 2004).

Although various test case generation techniques are available (Hooda & Chhillar, 2014), the MBT approach has involved many scholars and continuous research is conducted to enhance the generation of minimized automatic test cases with the lowest cost and human effort (Ingle & Mahamune, 2015). Various test case generation methodologies that use the UML statechart diagram have been proposed by several researchers, software developers, and software testers by using many UML diagrams, algorithm types, and methods (Ali, Shaik, & Kumar, 2014; Chimisliu & Wotawa, 2013a, 2013b; Gulia & Chillar, 2012; Swain, Behera, & Mohapatra, 2012a, 2012b; Swain et al., 2012c). The discussion on these works is presented in section 2.6 of Chapter 2. Several researchers used test case generation with an extension of methods like, the state activity diagram (SAD), DFS, mutation analysis (Swain et al., 2010a), and test generation with verification (TGV) methods (Chimisliu & Wotawa, 2013b). Tools like input/output label transition systems (IOLTSs) and random test selection (Gnesi, Latella, & Massink, 2004). Also algorithm like Euler circuit (Li et al., 2012). Although all of these methods generate test cases, their works did not consider the minimization (Ali et al., 2014) or prioritization (Swain et al., 2012c) and contained limitation in coverage criteria (Chimisliu & Wotawa, 2013b), and every technique has its defect. However, a combination of different techniques in a framework is an effective solution to increase the reliability of the generated test cases (Farooq & Quadri, 2011; Pahwa & Solanki, 2014).

Several works have provided steps for the test case generation from a UML diagram, such as those by Boghdady, Badr, Hashim, and Tolba (2011b); Karambir and Kuldeep (2013), where they discussed the processes and components involved in test case generation. However, to the best of our knowledge no comprehensive framework that

can represent the entire process of test case generation is available until the time of this study.

In addition, few studies in this area, revealed their proposed algorithms or the testing processes used during the testing, among these studies are by Hartmann, Imoberdorf, and Meisinger (2000); Kansomkeat and Rivepiboon (2003); Kosindrdecha and Daengdej (2010); Santiago et al. (2006); Santiago, Vijaykumar, Guimarães, Amaral, and Ferreira (2008). This scenario results on these methods to may not be applicable in future work, or to be improved or applied enhancement on them. Furthermore, its implementation in test case tool or reproduction on test case generation in a fully automated manner is difficult.

Many of the test case tools were not integrated (Santiago et al., 2008). Tools that are used for test case generation demand significant effort from software testers because all testing processes require manual interference to make appropriate adjustments on the output of a tool to be used as input to another tool (Santiago et al., 2008). In addition, some of other tools used internally by an enterprise and not available to the public, whereas others are no longer actively developed (Anand et al., 2013).

Many studies in automatic test case generation from UML diagrams used the DFS algorithm to generate test paths (Kundu & Samanta, 2009; Nayak & Samanta, 2010; Patnaik, Acharya, & Mohapatra, 2011; Pilskalns, Andrews, Ghosh, & France, 2003; Shirole et al., 2011; Swain et al., 2012c; Swain et al., 2010a). The use of this algorithm results in loss of paths, especially loop paths, thereby decreasing loop coverage. Therefore, the generating an enhanced DFS algorithm or creating a new algorithm for

path generation is necessary to include the path coverage criterion and the loop path coverage (Mingsong et al., 2006).

In conducting test case generation, the quality or adequacy of test cases is often described using the coverage criteria (McQuillan & Power, 2005). Current test case generation techniques consume a large amount of time and cost with less testing coverage (Kosindrdecha & Daengdej, 2010). Many approaches, such as genetic algorithms, model checking, or graph search algorithms are used to perform such coverage criteria for UML diagrams (Weißleder, 2010). One of the gaps in the existing UML statechart diagram methods is the selection of a proper input graph that has enough complexity to generate an accurate coverage percentage and overcome the limitations of existing approaches, such as decision states and loops (Biswal, 2010). Swain et al. (2012a, 2012b); Swain et al. (2012c) used low-cyclomatic complexity UML statechart diagrams and did not prioritize generated test cases. In addition, Chimisliu and Wotawa (2012); Chimisliu and Wotawa (2013a, 2013b) applied only one coverage criteria, which is the transaction coverage, and generated a large number of test cases that were not minimized. Moreover, they did not prioritize their generated test cases. Therefore, a test generation method that generates minimized and prioritized test cases with more comprehensive test coverage criteria is highly required.

This study focuses on generating minimized and prioritized test cases that achieve the highest possible coverage criteria and handle complex inputs, such as decision and loop states. Therefore, more efficacious automatic test method is required. This study developed a framework that automatically generates test cases from UML statechart diagrams, in which detailed algorithms are provided. Additionally, a prototype has

been developed to implement this framework. As a result, a new tool has been created to automatically generate the minimum test case with high efficiency and additional comprehensive test coverage.

1.4 Research Questions

This study is an attempt to design and develop a framework and its combined algorithms that automatically generate minimized and prioritized test cases using UML statechart diagrams from the design documents. However, expenses and some issues need to be considered in automated testing. The issues that should be resolved, depending on which part of the process would be improved, are as follows:

i. What are the current test case generation methods and UML diagrams needed to propose a test case generation framework?
ii. What are the suitable coverage criteria covered by the proposed framework?
iii. How the test cases are generated using the proposed framework?
iv. How are the proposed test case generation framework and its algorithms evaluated?

1.5 Research Objectives

This study aims to propose a test case generation framework that generates minimized and prioritized test cases from UML statechart diagram with the highest coverage criteria and smallest in size and number of test cases. The framework with its methods present the entire process of test case generation. This study would also provide

evidence to prove that this framework meets the applicability requisite of test case generation. The specific objectives of the study are as follows:

i. To investigate the current practices of software test case generation methods that use the UML diagrams as an input, to design the proposed framework.
ii. To identify the suitable coverage criteria, which are covered by the proposed framework generated test cases.
iii. To develop an improved method that generates minimized and prioritized test cases using the proposed test case generation framework.
iv. To evaluate the proposed framework using prototyping, comparison with existing work, and expert review.

1.6 Research Scope

This study focuses on investigating software testing and the automatic generation of test cases using a UML statechart diagram. This research also includes the theoretical development and implementation for the framework and its algorithms. The graph theory was used as a base to convert the UML statechart diagram to the intermediate model. Firefly algorithm is used for the minimization and prioritization of this work. The programming style used to program the prototype focuses on open source and the use of an object-oriented programming approach as the basis of the development method. In addition, this study focuses on the UML statechart diagram created in the development cycle. However, nested states are not included in the scope of this study. In addition, this study focuses on MBT techniques.

The time consumed during testing has not been considered or measured because part of the research objectives is to validate the proposed framework using coverage criteria. In addition, the automating of testing result in reducing the cost, however the cost reduction is not been measured.

The prototype that was developed in this study is aimed at implementing the proposed framework and getting the comparison results of the used examples, however it is not ready to be commercialized.

1.7 Research Framework

A research framework defines the outcomes and set of research activities (Lithner, 2008). This framework presents the diagram components of this study that are connected to one another and built into this framework. Figure 1.1 shows the framework of this study.

Figure 1.1 shows the research framework, including the objectives, the methods used in achieving the research objectives, and the outcome from the objectives. The first phase aims to identify the technique for generating test cases, which includes the literature review. This phase focuses on finding and improving existing algorithms and proposed methods to accomplish the first objectives for test case generation. The second phase of this research framework aims to propose the required coverage criteria to test the generated test cases. The third phase includes the development of the framework algorithms to automatically generate minimized and prioritized test cases. This phase also includes the creation and development of a prototype that implements

the methods. The last phase involves the evaluation of the proposed framework and its algorithms and prototype. Chapter 3 discusses the details of all phases.

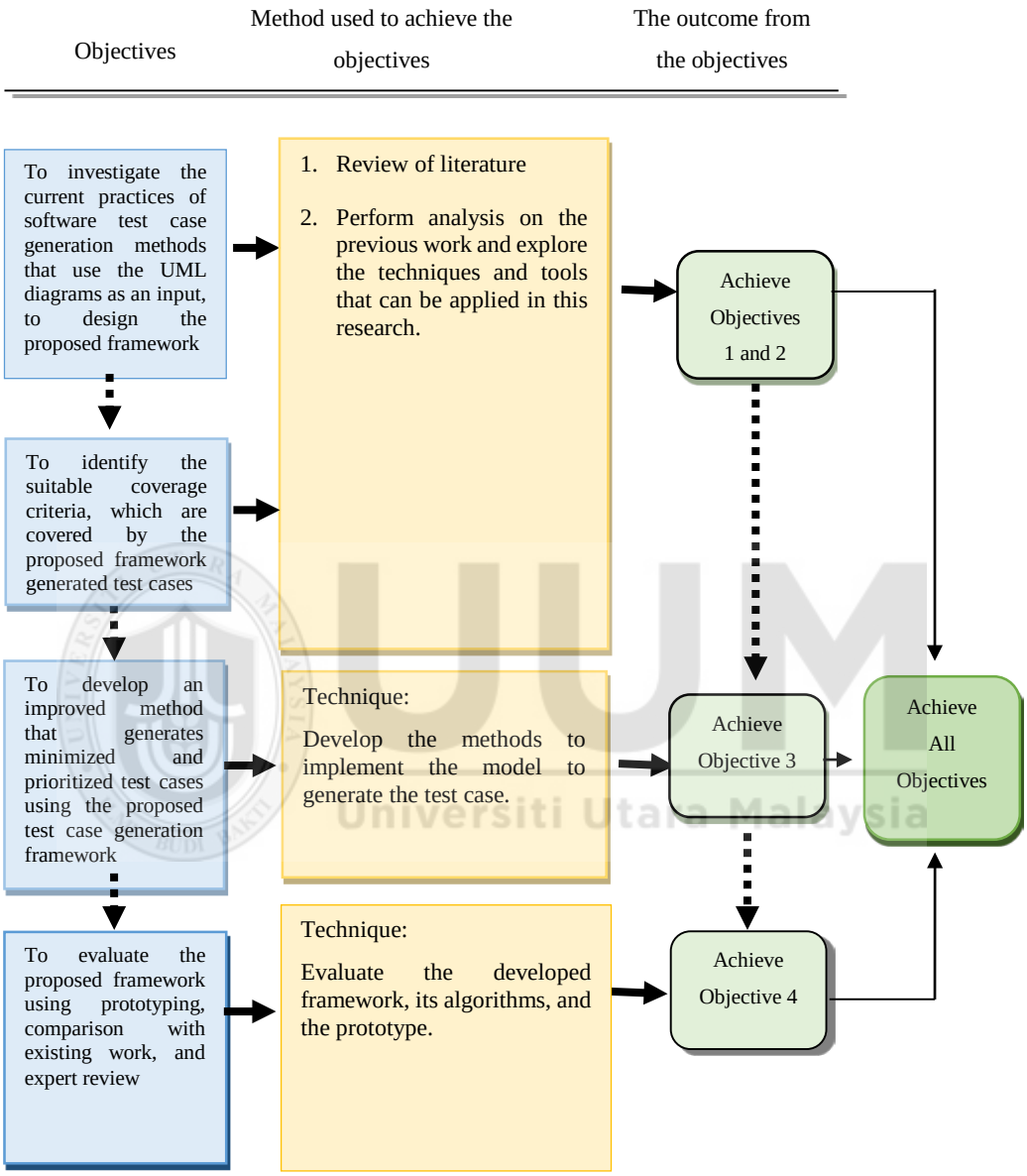

Figure 1.1. Research Framework

1.8 Research Contributions and Its Significance

At present, software-intensive systems increasingly influence people's lives. Thus, system features and functionalities require more qualifications. Subsequently, the need

for qualified and reliable systems has expanded. However, as the requirements increase, the complexity of software-intensive systems also increases with error-proneness, which is related to shortened development times.

- Body of knowledge

This study develops a framework to automatically generate minimized test cases from the UML statechart diagram, also prioritizing these test cases. This study contributes to software engineering, particularly on software testing, especially in generating test cases using model-based techniques, which generates test cases based on the design document, to validate and focus on the behavioural aspects of the software. This study contributes to traversal algorithm by proposing new algorithm, and to test paths or sequence by developing path pruning, also enhanced consistency checking to support the loops. In addition, contribute to minimization and prioritization by adapting firefly algorithm, the use of develop path weight equation, and the use of information flow. Moreover, this study contributes to test activities by including the expected results for each test case, which can aid test activities by providing improved methods and algorithms. Furthermore, the developed prototype can alleviate the burden of manual testing, thereby providing support to the testing tool for generating test cases from the UML statechart diagram. Figure 1.2 shows the contribution of software testing. Thus, this study is a worthwhile effort that is beneficial for software testers.

- Practical

This study primarily intends to benefit the software application industry by focusing on less costly and earlier alternative automatic test case generation techniques that will help software testers and developers by reusing UML statechart diagrams. The

proposed framework provides the fully required detailed for researchers and developer to create their own test case generation tool.

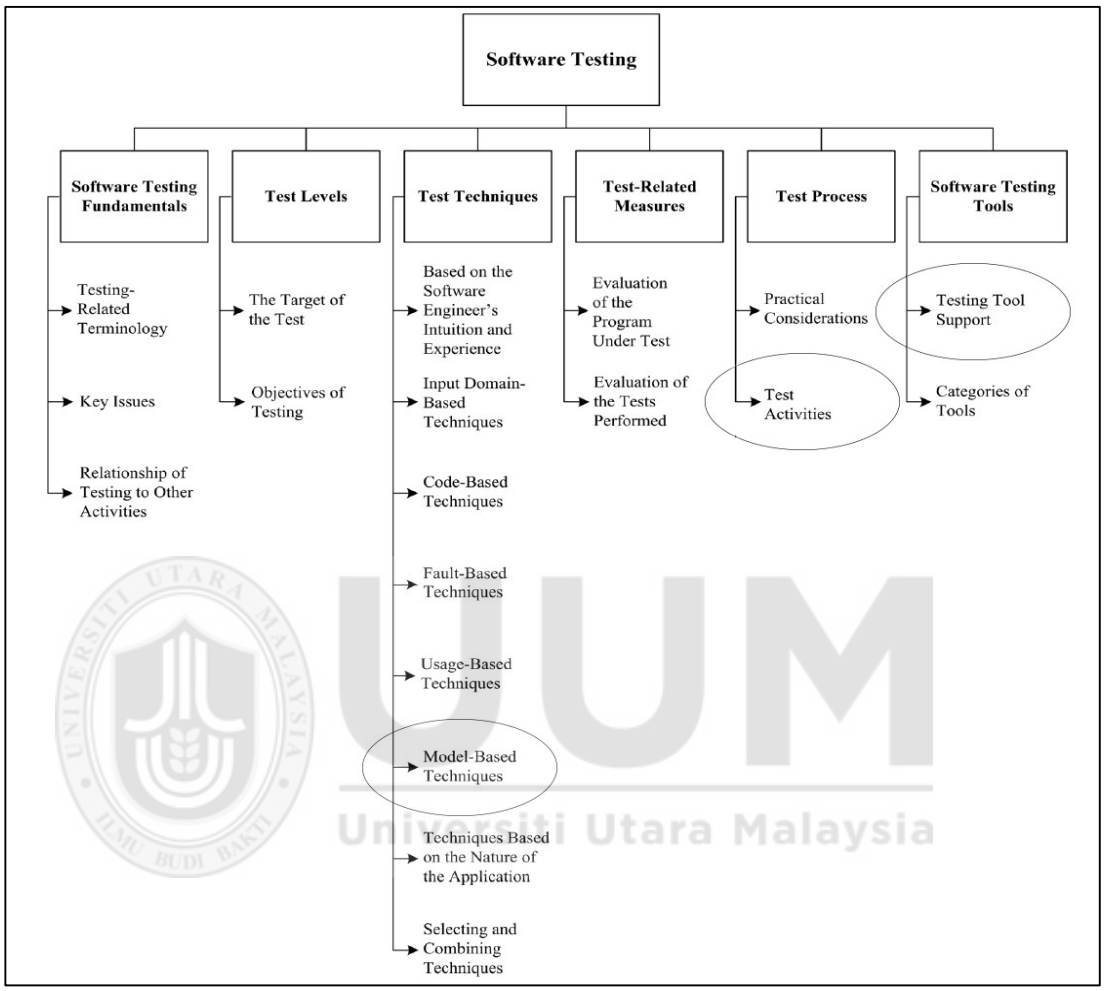

Figure 1.2. Software Testing Procedure
Adapted from Society (2014)

In addition, the practical advantages of this study are outlined as follows (Binder, 2000): Provide clear and between coverage criteria testing procedures for test case generation; suggest the development process to automatically generate an optimized test case from UML diagrams presented by the proposed framework; eliminate flaws in the manual input through the automation of the process in the developed prototype.

Moreover, automation is the only repeatable way to efficiently measure a large amount of input.

1.9 Terminologies for Software Testing

This study contains some software testing terms that require explanation. This section provides the definition of the terminologies that are used throughout this study.

Differentiation has been conducted among the many definitions and particularly refers to the following testing terms:

Definition 1 Path Testing: Path testing is a method that is frequently used to ensure that a set of paths or a particular path in the program are tested at least once (Shen & Abraham, 2000).

Definition 2 Testing: Testing is a software verification method that deduces execution results or traces that the SUT possesses certain good properties (Dssouli, Saleh, Aboulhamid, En-Nouaary, & Bourhfir, 1999).

Definition 3 Software Testing: This is a process that evaluates the software by executing and observing it (Ammann & Offutt, 2008).

Software testing includes, but is not limited to, the process of executing the program with the intent of finding fault, failure, and error that might exist in the software.

Definition 4 Test Case: A test case is composed of test case values, prefix values, expected results, and postfix values, which are necessary for the complete evaluation and execution of SUT (Ammann & Offutt, 2008).

In many different levels of abstraction, a test case can be existed. The most important difference is between concrete and abstract test cases. A test case is combination of three stages. The first stage is the initial state in which the test data are input into the system. The second stage involves inputting the test data into the system. The last stage is expecting the output from the system (Mall, 2009; Offutt & Abdurazik, 1999). This testing will provide the specification behaviour of the actual software to the output produced by the software in a particular test case.

Definition 5 Expected Results: When tests are executed, the result that will be produced is called expected result. This result will be recognized if and only if the program satisfies its intended behaviour (Ammann & Offutt, 2008).

The two most commonly applied problems related to software testing are identifying the details of the software behaviour and providing the right values to the software.

Definition 6 Test Requirement: A test requirement can be defined as a specific component of a software artefact that must be covered or satisfied by a test case (Ammann & Offutt, 2008).

Definition 7 Software Failure: Failure is external due to incorrect behaviours with respect to system requirements or other components from the expected behaviour (Ammann & Offutt, 2008).

1.10 Thesis Outline

This thesis consists of six chapters, including this chapter. The remaining chapters are structured as follows:

- **Chapter Two: Literature Review**

This chapter presents a discussion of the background information and related works of software testing, including an overview of UML diagrams and test case generation using these diagrams. This chapter also explains the MBT processes and issues concerning the automatic test case generation using UML diagrams. Then, the discussion focuses on the coverage criteria. In addition, test case optimization and prioritization, test case generation processes and components, and theoretical background are discussed. The chapter ends with a summary of its contents.

- **Chapter Three: Research Methodology**

This chapter is an introduction to the methodology used in the present study. The research methodology and its phases are presented. Furthermore, each phase is discussed in detail.

- **Chapter Four: Algorithm Development**

This chapter discusses the algorithms development that will be implemented in the proposed framework. Furthermore, coverage criteria calculation equations for the selected coverage criteria and prototype development are presented.

- **Chapter Five: Evaluation**

This chapter reports the evaluation of the proposed framework based on three stages, namely prototyping, comparison, and expert review.

- **Chapter Six: Conclusions**

This chapter begins by summarizing the study. Then, the contributions of this thesis are highlighted. The limitations and future work in related fields are addressed. Finally, a conclusion is provided.

CHAPTER TWO
LITERATURE REVIEW

2.1 Introduction

This chapter describes the state of the art in software testing, test case generation techniques and their automation, and the use of MBT. The discussion begins with the overview of testing, followed by test case generation. The theoretical background is elaborated, followed by MBT and UML diagrams. Analysing some related studies and their techniques on automatic test case generation using UML diagrams in general and the UML statechart diagram in particular. Then, the existing issues in optimization and prioritization of automatic test case generation are explained. Test case generation process and components are also discussed after that, as well as coverage criteria. The chapter ends with a summary.

2.2 Overview of Testing

Today, developing applications and high-quality systems with minimum errors and faults is necessary. Furthermore, cost and time should be as low as possible (Kull, 2009). Speeding up the delivery of services is the significant role of automated testing techniques for software development (Dustin, Garrett, & Gauf, 2009). Automatic testing is an essential task when familiarizing with technology and decreasing expenses are the goals. One of the processes in automatic testing is automatic test case generation. Test cases help a user test all possible combinations and compose an entire coverage of the application (Javed et al., 2012). Testing also provides areas in which the application works fine and the amount in which the testing has concluded (Karambir & Kuldeep, 2013).

Test cases can be generated based on the system requirement specification and design document (Hooda & Chhillar, 2014). This study focuses on test case generation from system design documents using the UML statechart diagram. The input values that a test case contains are reflected in the system. Thus, selected operations are tested using test cases. These input values can be parameters that launch the system or a series of input data (Ngah, 2012).

Test cases are created from the system design of the UML statechart diagram when a system will be tested. In ideal situations, the test case either passes or fails (Gotlieb, 2012). If all tests pass, then a symmetrical arrangement between the design document and the test is observed. If the tests fail, then the system encounters problems in generating expected results; thus, the system has errors. If a test is performed with defined situations but the system still fails, then errors are due to the system design. When a test is unsuccessful, the reason behind the cause of failure is identified (Hessel, 2006).

The effects on SUT by testing it can cause unpredictable behaviour. Hopper (1981) stated that the first unpredicted performance was caused by a moth trapped between the points of relay. Thus, the term “bug” is used to describe a failure, error, or fault in a computer program or system. However, this term does not appropriately define the different stages of fault, failure detection, and error propagation of error.

Failures can be detected directly by test cases because they are concerned with requirements. Figure 2.1 shows that one of the possible ways for a fault to cause error is by the activation and propagation to failure.

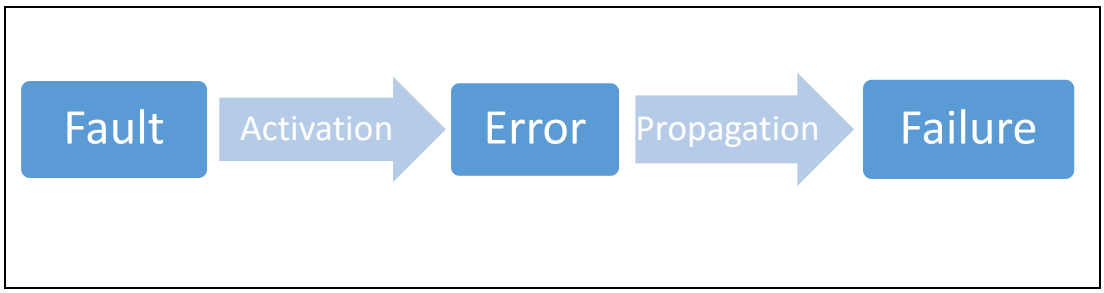

Figure 2.1. Relation of Fault, Error, and Failure

Software testing, mostly called fault detection technique, is considered as a failure detection technique because only failures can be identified by this test (Morell, 1984, 1990; Offutt, 1988). The next two subsections will be on software testing and its techniques, and automated software testing.

2.2.1 Software Testing and its Techniques

Testing is considered as an essential process to ensure the functionality of a system. The difficulty of testing is depending on the complexity of the SUT (Elallaoui, Nafil, Touahni, & Messoussi, 2016). Furthermore, development cost increases when the software complexity increases, thereby requiring much effort, time, and expertise (Elallaoui et al., 2016). Software testing is one of the most important and critical phases in software development process that cannot be ignored (Bentley, 2005; Kosindrdecha & Daengdej, 2010). Software testing is used to verify whether the system behaves in its intended way to reveal bugs in a system and to ensure that the system complies with its specifications (McQuillan & Power, 2005).

Similar to any other product, software requires testing. Nothing can be considered correct unless its functionality has been tested first. For physical products, testing can be as simple as using a product in all of its intended uses with the unpremeditated

products to be tested for errors, and formulating a conclusion that the product is satisfactory (Rapos, 2012). With software, the testing process is more complicated and often tends to be formal. Therefore, software testing has become an extremely important aspect of development (Rapos, 2012).

Although the defined development processes and helpful development tools increased, software development remains a largely manual process. Thus, errors, which are mostly caused by human fault, occur when a software is created. Error can be due to many possibilities, such as a misunderstanding in the user requirement, faults in the system, or even a programmer's mistake.

Practical-sized software usually offers a complicated set of possible ways of experimentation. Deciding the exact behaviour for software testing is one of the main difficulties. During experimentation, deciding whether the experimental behaviours are correct or not is also difficult. Therefore, new and enhanced testing and development techniques should be applied to face these challenges (Elallaoui et al., 2016).

Testing can be executed under several conditions. Observability and knowledge are two of the most effective features in SUT internal matters (Weißleder, 2010). Test cases can be generated using two main methods, namely, white box and black box (Sapna & Mohanty, 2008), as shown in Figure 2.2. The black-box technique is a functional or behavioural technique based on qualifications. This strategy disregards the internal structure of the tested object. Instead, it focuses on the required qualifications for object testing, thereby disregarding the method applied to the tested object (Aichernig, 2001).

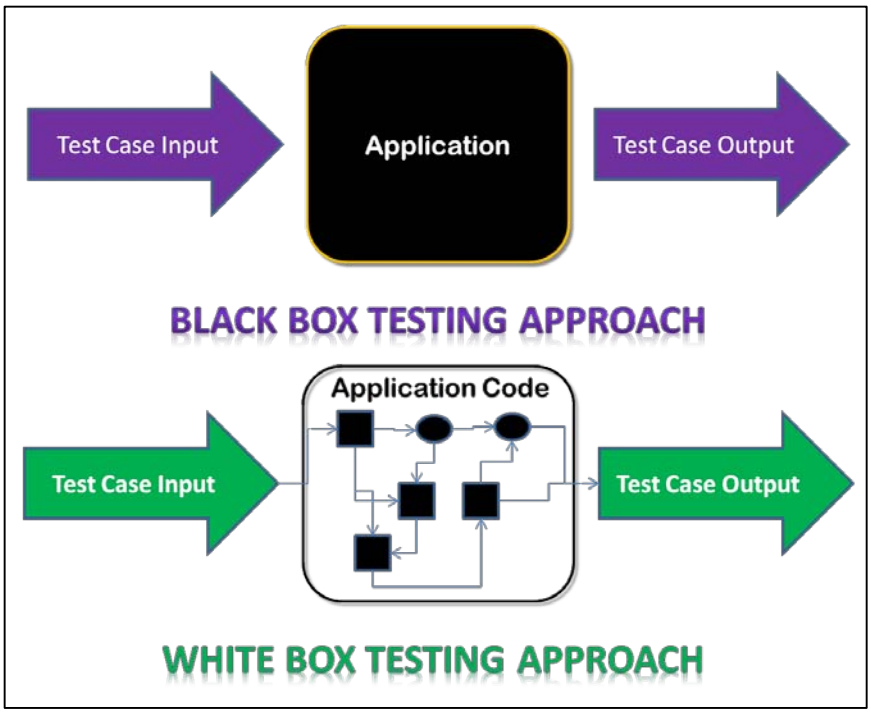

Figure 2.2. Comparison Between Black-box and White-box Testing
Source: Xu, Chen, Wang, and Rud (2016)

In the white-box testing model or structural testing, tests are generated based on the software structure or internal implementation, which tests the program at the structural level (McMinn, 2004). This model includes the choice of criterion, identification of a set of branches, paths, or vertices, and a test case generated (Ahamed, 2010). Some common examples of this strategy are data flow testing, which executes every statement as a minimum once; statement testing, which executes every branch as a minimum once; and branch testing, which tests the usage of all data objects (Nidhra & Dondeti, 2012).

White-box and black-box testing have both advantages and disadvantages. Therefore, a new approach called grey-box testing is formed to combine the advantages of these techniques (Linzhang et al., 2004). Grey-box testing techniques are used by white-box level to design tests that will be executed at the black-box level. This technique allows

the tester to have access to the internal information of SUT while tests are being designed (Lima & Faria, 2016). However, tests are performed under realistic circumstances, and therefore only failures will be discovered.

The most common testing tasks are test result evaluation and test case generation that are usually automated based on the SUT model. One of this testing is the MBT and according to Karambir and Kuldeep (2013), MBT is considered as a black-box testing technique.

2.2.2 Automated Software Testing

Automated testing is a well-established research area. Nevertheless, a gap in software testing application is recognized between academic and industrial research (Rafi et al., 2012). According to Rafi et al. (2012), automated testing can detect and provide solutions to many difficult and complex bugs.

In many areas, automation has been successful. Therefore, the use of an automated software testing programme to test another software programme is the next step of evolution that can be called automated software testing (Kelly, 1999).

The use of an automated software testing programme can significantly decrease the software development cost, increase testing result accuracy, complete test preparation in advance, and rapidly run tests (Srivastava & Kim, 2009).

The use of automated software testing is not a straightforward process. For many years, researchers have proposed various approaches and methods to develop test case generation (Bhat & Prashanth, 2014; Kaur & Harwinder, 2013; Mani & Prasanna,

2016; Mohi-Aldeen, Mohamad, & Deris, 2014; Oluwagbemi & Asmuni, 2015; Wu & Fan, 2014; Yemul, Vhatkar, & Bag, 2014; Zhang, Duan, Yu, Tian, & Ding, 2016). The development of these methods and techniques will result in significant cost savings and software testing automation support (Srivastava & Kim, 2009).

Figuring out the accuracy of a given part of a software is highly complicated. Software testing was traditionally performed manually and occasionally. However, a systematic, traceable, and systematic approach is required for the safety of the industrial environment. Automated tools are currently applied in the industry to perform and organize test cases (Prasanna & Chandran, 2011). Automation is essential for many reasons, where manually writing test cases can be tedious, and writing good test case can sometimes be more of an art than a science (Shamshiri et al., 2015). Manual production of test cases is tiresome and entails many errors (Kangas, 2008). The development process would be highly improved through automated test case generation because the most time-consuming parts of the process are preserved (Prasanna & Chandran, 2011).

Furthermore, automation would result in complete sets of test cases because of its systematic performance. However, some issues related to software automation need emphasis. Although generating a set of test cases can be automated, two important issues have to be considered. First, the generated test cases size should be considered because unnecessary test cases might be included, and the paths of the final test cases should be minimized (Ahmed, 2016; Belli & Hollmann, 2008). Second, selecting the best test case also needs to be attended by prioritizing the selected test cases (Sumalatha & Raju, 2014). The studies on software testing has suggested a variety of

solutions for the automation of test case generation. Some of these solutions were conducted and applied in software testing, such as for commercial products. However, many associated problems, such as the requirement for specialists with a higher skill level, the effect of new methods on people, and the need to increase the required tools for testing, should be eliminated before software testing can receive widespread support and acceptance.

2.3 Test Case Generation

The constitution of a test case will vary from one system to another, but in its simplest form, it will be a series of events that will result in a certain execution path, given certain conditions (Rapos, 2012). Values for attributes and parameters can be generated based on any constraint and can supply to the program for a test execution (Rapos, 2012).

Aside from software development, the testing phase is divided into three categories, namely, test case generation, test case evaluation, and test case execution (Karambir & Kaur, 2013). Compared with the other two categories, test case generation is the most challenging among the categories (Gulia & Chillar, 2012). Manually created test cases are usually time consuming and error prone; thus, the next logical phase is the automation of the test case (Schwarzl & Peischl, 2010b). Test case generation can save effort and time and reduce the number of faults and errors at the same time (Gulia & Chillar, 2012; Sahoo et al., 2016b). Likewise, the reliability of tests is increased and the costs of manual testing are reduced (Shamsoddin-Motlagh, 2012).

Figure 2.3 highlights the steps in the software testing life cycle. The software developer or software testers will be assisted in finding inconsistencies and uncertainties in the requirement specification and the design documents of the system for earlier test case generation (Shull, Rus, & Basili, 2000). When errors are removed early during the devolving life cycle, the time and development cost software systems decrease significantly.

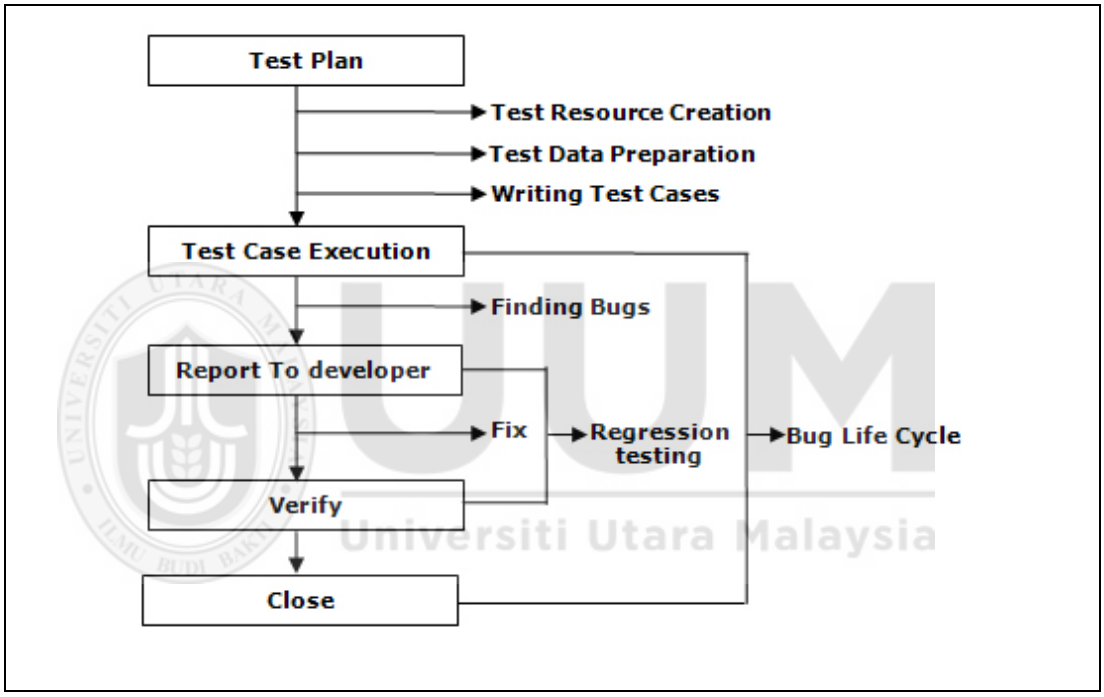

Figure 2.3. Software Testing Life Cycle
Source: Karambir and Kuldeep (2013)

2.3.1 Automatic Test Case Generation

The generation of a subjective test case requirement is a nontrivial problem. Several researchers have focused on the automation of test case generation in which various degrees of success are shown in the reported results. Different design artefacts and SUT methodologies are used in the automatic generation of test cases. The automatic generation of test cases will take and process the design artefacts as input, and then

generate test sequences based on certain pre-specified testing coverage criteria. Then, the exact test data for each test sequence are determined to form the test cases (Kaur & Gupta, 2013). This method builds the confidence of the developer and successively executes software testing in generating the test case for a set of data inputs (Jena, Swain, & Mohapatra, 2014).

Test case generation is an essential step in software testing. Test cases categorise the pre-test state and environment of SUT in addition to test conditions or inputs (Binder, 2000). A test case identified as a set of test inputs, states, and expected output is developed to verify an execution with a specific requirement or implementation of a specific program path (Lilly & Uma, 2010). Test cases aim to identify the communication conditions and problems that will be implemented in a test. Test case requirements will be necessary to verify the acceptability and success of any product implementation (Heumann, 2001).

Test case generation can be achieved from specifications and requirements, source code, or design document. Test cases are usually designed based on the software source code (Abdurazik & Offutt, 2000; Jena, Swain, & Mohapatra, 2015). This code will cause difficulties in test case generation, especially for mass-level testing (Jena et al., 2015). Generating test cases in the development cycle based on the requirement specification and design documents of the project will add as an advantage by enabling the early availability of tests in the software development life cycle (Kumaran et al., 2011) to create more effective test planning. Additionally, the advantage of design-based testing is to test the performance of the application based on the requirement specifications and design documents (Jena et al., 2014). However, manual test case

generation is time consuming and difficult (Jena et al., 2014). Thus, either a semi-automatic or an automatic test case generation based on the requirement specification and design document is usually anticipated (Krishnachandra, 2016).

The test case generation from UML diagrams includes many steps. The steps begins by storing the UML diagram information in a database-based table, which will subsequently transform the data into a graph model (Priya & Sheba, 2013). Next, the test paths are generated from the graph model in which these paths will help identify all possible routes that the software will follow and form these routes into a test case (Werner & Grabowski, 2012). These paths are the structural method of testing and test cases that will demonstrate every possible executable path for the program (Parnami, 2013). Two fixed vertices are included in the test paths. These vertices are established by the fact that every legal path must begin at the source vertex and end at the sink vertex (Schligloff & Roggenbach, 2002), which are called the start state and the end state. The number of vertex predecessors is its in-degree, and the number of successors of the vertex is its out-degree (Srikant & Shankar, 2007). A path from a vertex X_1 to a vertex X_k in a graph $G = (V, E)$ is a sequence of vertices $(X_1, X_2, X_3, \dots, X_k)$ such that $(X_i, X_{i+1}) \epsilon$ E for every i, $1 \leq i \leq k$ (Panthi & Mohapatra, 2012) as shown in Figure 2.6.

Test case generation has always been necessary and basic to the testing process (Bertolino, 2003). Incidentally, many researchers have conducted research on test case generation using UML diagrams (Jia & Liu, 2002; Jia, Liu, & Qin, 2003; Kaur & Singh, 2015; Mani & Prasanna, 2016; Oluwagbemi & Asmuni, 2015; Prasanna, Sivanandam, Venkatesan, & Sundarrajan, 2005). Other techniques include random and

goal-oriented techniques. A test case based on assumptions regarding fault distribution are controlled by random techniques. Intelligent techniques for automated generated test cases rely on complex calculations to identify test cases (Santiago et al., 2006).

Modelling languages can be used in the software requirement specification and design document. UML is the most extensively used modelling language. Thus, UML diagrams have been used by many researchers, such as sequence diagrams, statechart diagrams, and activity diagrams, to generate test cases using MBT case generation techniques (Chimisliu & Wotawa, 2013a; Nayak & Samanta, 2010; Oluwagbemi & Asmuni, 2015).

Examining a piece of software or model manually and formulating a number of tests that will use the program through a number of executions is possible. However, this process is infeasible and can often result in overlooking a particular case that causes software error or even an ideal case to ensure functionality (Lavagno, Markov, Martin, & Scheffer, 2016). Thus, automated test case generation has become an area of focus. Software development teams aim to provide a program or a system model and to have a complete set of tests that are automatically generated to test all desired executions. A number of possible methods for automatic test case generation have been developed because generating test cases by hand selection is error prone and time consuming (Rapos, 2012).

2.3.2 Automated Test Case Generation from Software Design

The automated generation of test cases has been proposed to reduce the challenges in test case generation (Korel, 1990). The quality of manual testing depends on the

experience and software design knowledge of the tester. Thus, automated test case generation can provide effective test cases with appropriate software design. This problem can also address those resulting from human errors and lack of testing experience.

The use of UML diagrams can produce test cases earlier in the development lifecycle and test the system before the coding cycle, given that the UML diagrams created in the system follow certain specifications. Such early generation of test cases will enable software developers to find uncertainties and inconsistencies in the system specification and design (Jain & Sheikh, 2014).

In the case of component-based software development, the use of program source code to generate a test case is proven to be insufficient because even the source code may not be available to software developers. Therefore, using design specifications to generate test cases is important (Samuel, Mall, & Bothra, 2008). In addition, creating new diagrams is not a necessity because the same diagrams created for the design phase are used for test case generation. Furthermore, test case generation based on design specifications has an additional advantage of providing test cases early in the software development cycle, thus making the test planning more effective (Samuel et al., 2008).

Software design and testing are both important in the software development lifecycle. Faultless software design helps software developers in developing a system, and excellent software design can support developers adjust to various software requirements during the software development process (Samuel et al., 2008).

The early detection of errors has become a serious issue. As shown in Figure 2.4, an error that is detected in the later cycles is extremely costly to repair (Tuple, 2010). Moreover, detecting a fault during the system testing is 10 times more costly than detecting the same fault during the system design. The same fault is up to 30 times more costly if detected during the system production. Unfortunately, requirements and design specifications are major sources of software bugs (see Figure 2.5). Studies have found that, in some cases, the proportion of such bugs to the overall detected bugs can be 50% or more (Perry, 2007).

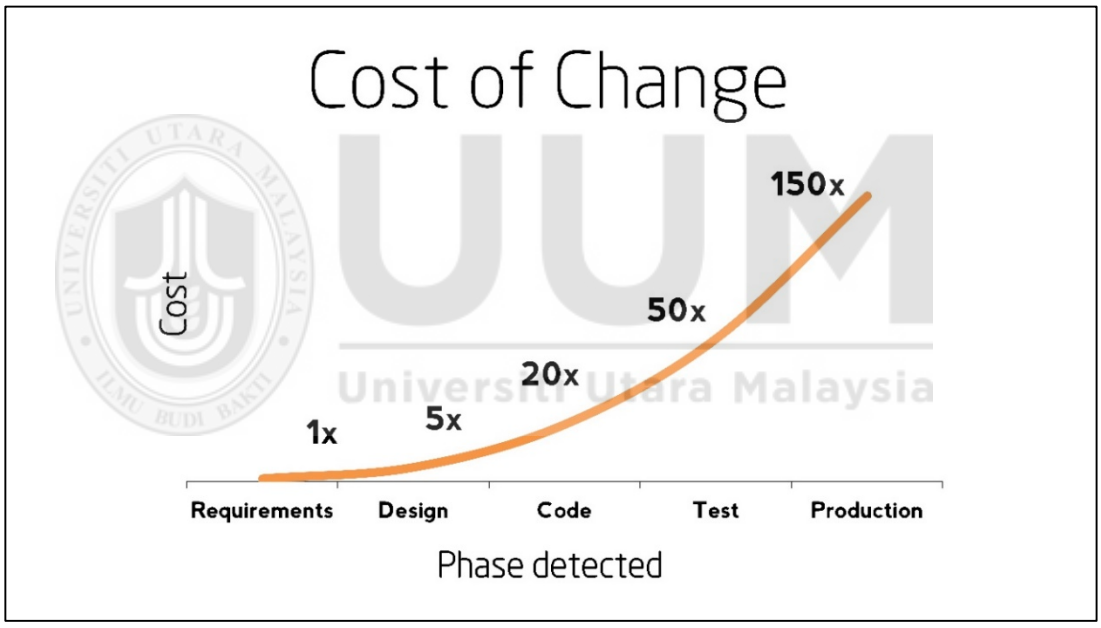

Figure 2.4. Comparative Graph for Cost of Software Repair by Development Lifecycle Phases
Adapted from Dawson, Burrell, Rahim, and Brewster (2010)

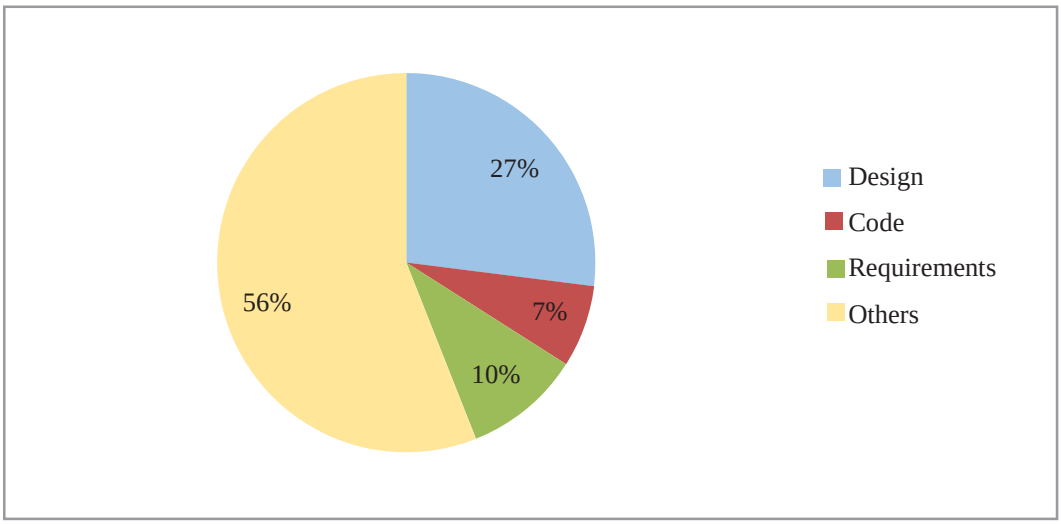

Figure 2.5. Fault Proportion According to Source Phase
Adapted from Rice (2010)

In addition, requirements represent the application from the perspective of the business as a whole or the user. Moreover, the design specification represents the application from the perspective of the software developer or the technical team. Therefore, test cases for the software system must be generated during the software design lifecycle. The next section cover several theories behind the generation of test cases applied in this study.

2.4 Theoretical Background

According to Kerlinger (1986), a theory is "a set of interrelated constructs (concepts), definitions, and propositions that present a systematic view of phenomena by specifying relations among variables, with the purpose of explaining and predicting the phenomena" (p. 9).

This study adapted two major theories to automatically generate the test cases: graph theory and automata theory. The following subsections will introduce these theories in the context of the current research.

2.4.1 Graph Theory

The paper published by Euler (1736) on the Seven Bridges of Königsberg is considered as the first paper in the history of graph theory. Graph theory is an important area of modern mathematics with many applications in social science, computer science, engineering, chemistry, genetics, business, and industry. This theory is a new science developed and invented to solve challenging problems of a "computerized" society, for which traditional areas of mathematics, such as calculus or algebra, are ineffective (Voloshin, 2009).

Graph theory is an area of mathematics that can assist researchers in utilizing the model information to test applications in many different ways (Robinson, 1999). Graph theory techniques have been an important part of MBT and several graph techniques (Shahzad, Raza, Azam, Bilal, & Shamail, 2009).

The adoption of graph theory techniques for MBT has been conducted by many researchers as the intermediate graph. In this study, the UML statechart is converted into a graph to generate the test paths.

The graph G is a set of vertices V together with a set of edges E and is presented as $G = (V, E)$.

An example of the graph is shown in Figure 2.6. This graph has $V = 6$ vertices and $E = 8$, where $V = (V_1, V_2, V_3, V_4, V_5, V_6)$ and $E = (E_1, E_2, E_3, E_4, E_5, E_6, E_7, E_8)$. $E_1 = (V_1 \rightarrow V_2)$, $E_2 = (V_2 \rightarrow V_3)$, $E_3 = (V_3 \rightarrow V_4)$, $E_4 = (V_1 \rightarrow V_3)$, $E_5 = (V_1 \rightarrow V_5)$, $E_6 = (V_5 \rightarrow V_1)$, $E_7 = (V_5 \rightarrow V_4)$, and $E_8 = (V_4 \rightarrow V_6)$ because each edge connects a pair of vertices; therefore, $E = \big((V_1 \rightarrow V_2), (V_2 \rightarrow V_3), (V_3 \rightarrow V_4), (V_1 \rightarrow V_3), (V_1 \rightarrow V_5), (V_5 \rightarrow V_1), (V_5 \rightarrow V_4), (V_4 \rightarrow V_6)\big)$ (Voloshin, 2009).

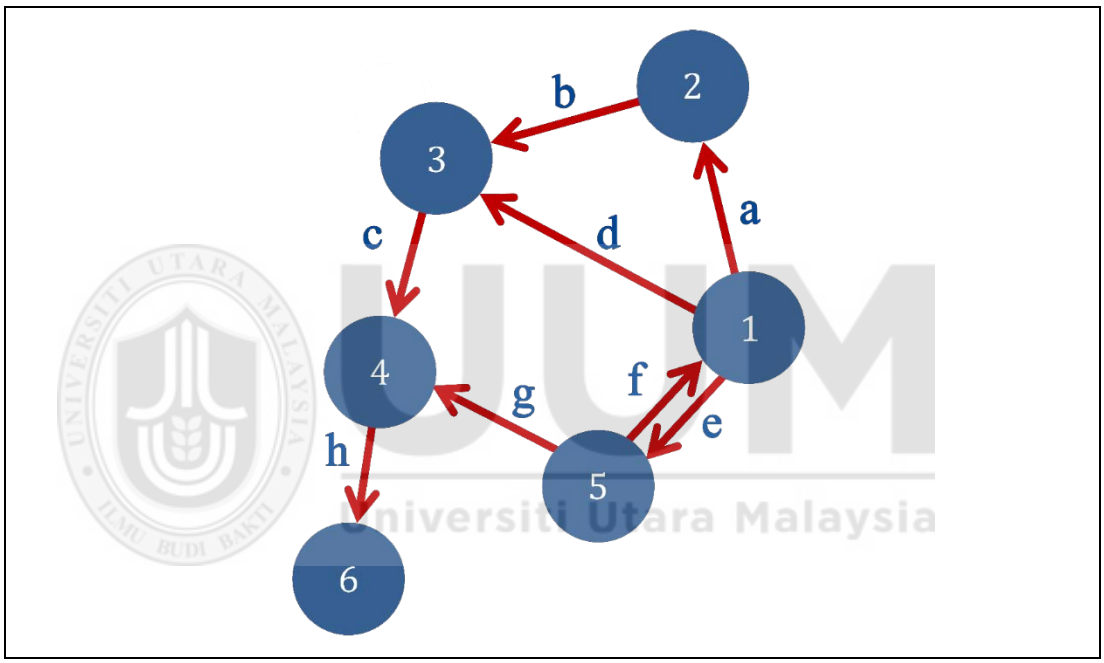

Figure 2.6. Graph Example

The graph representation is performed by using a square-name adjacency matrix, which has one row and one column for each vertex. If vertex V_i is connected by edge to vertex V_j, then (i, j) is 1; otherwise, it is 0 (Voloshin, 2009). For the graph in Figure 2.6, the adjacency matrix denoted by $A(g)$ is

$$A(g) = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

2.4.2 Automata Theory

Automata theory is a classical theoretical tool that is central to the development of computer science. This theory also has relevant applications in machine theory. The elegance and simplicity of this theory in "constructivist" applications is remarkable (Marijuán & Westley, 1992).

Automata theory is a theory in theoretical computer science and discrete mathematics. This theory is relevant to the study of abstract machines and automata, as well as other computational problems. This study automates the test case generation and uses deterministic automaton definition to present the UML statechart diagram as a quadruple ST = (E, Σ, H, T), where (Belli & Hollmann, 2008)

• E is a finite set of events,

• $\Sigma = (S, S_\Xi, S_\Gamma)$ is a triple of a set of states with S as a finite set of states,

$S_\Xi \subseteq S$ denoting the entries (initial states),

$S_\Xi \subseteq S$ the exits (final states),

• $H \subseteq S \times S$ is a hierarchy relation, and

• $T \subseteq S \times E \times S$ is a finite set T of transitions.

The set of states S comprises split sets of simple states S_{simple} and composite states S_{comp} consisting of AND- and XOR-states. The sets of initial and final states are

termed S_{Ξ} and S_{Γ}, respectively. Final states represent possible exits from the system. The set H defines a binary relation on the set S forming a tree. For an element, $(s, s') \in H$ holds that a state s is an immediate sub-state of state s'. Transitions must be deterministic and associated with an event (Belli & Hollmann, 2008).

The following section cover the characteristic of MBT, together with the common process in MBT test case generation.

2.5 Model-based Testing

MBT is considered a testing technique where test cases are derived from the model that identifies the predictable behaviour of a system (Bozkurt, Harman, & Hassoun, 2013). Formal models with exact semantics are very important because they are more appropriate for the automatic generation of test cases (Frantzen, Tretmans, & Willemse, 2006).

There are many techniques beside MBT to generate test case, like random test case generator, path oriented approach, and intelligent approach (Shah, Shahzad, Bukhari, Minhas, & Humayun, 2016). A random test case generator may create many test data; but might fail to find test case to satisfy requirements (Singh, 2014). A path oriented approach identifies path for which test case has to be generated, however the path might be infeasible, the test data generator might fail to find an input that will traverse the path (Wei & Xiaoxue, 2010). An intelligent approach generates test cases quickly but is quite complex (Prasanna et al., 2005). Comparing with these techniques, MBT is a valuable one, since it creates useful, flexible, and automated test cases from practically first day of development (Singh, 2014). Models are simple to modify,

generate innumerable test sequences, and allow the testers to get more testing accomplished in shorter time, also can be occupied from the software design documents (Prasanna et al., 2005). Even though varied test case generation approaches are available, MBT approach has attracted many researchers and still research is being carried out to optimize the generation of test cases with minimum human effort (Singh, 2014).

Tests that are produced through models are called MBT (Utting & Legeard, 2010). The idea for this type of testing was earlier known as specification-based testing (Utting, Pretschner, & Legeard, 2006). The benefit from this model is that MBT does not require a formal system specification; instead, it can represent several features of tested design phase (Pinheiro, Simão, & Ambrosio, 2014). In this section, a brief explanation of the processes in generic MBT is presented. Figure 2.7 illustrates five major parts of the MBT process, which are model, generator, concretize, execution, and analysis, which were introduced by Utting et al. (2006).

MBT is an important approach with many advantages that can lead to cost reduction and increased quality and effectiveness of a testing procedure (Schweighofer & Heričko, 2014). MBT, which uses UML diagram from design specifications for test case generation, overcomes the deficiencies that are extremely difficult to identify in the system state information, either from the code or from the requirement specifications. Therefore, MBT has been developed as a promising testing method (Pahwa & Solanki, 2014).

MBT is suitable when the requirements are formally specified through graphical notations, such as UML statechart diagrams, and when the test cases are generated

using the formal specifications. The use of this method for software testing is generally preferred for the following reasons (Saini & Srivastava, 2015):

1. MBT can be easily understood by both business and developer communities.
2. MBT separates the business rationale from the testing code.
3. MBT can quickly achieve automated testing.
4. MBT allows developers to switch testing instruments if the same model is required or utilized in various stages.
5. MBT focuses on requirement coverage.
6. MBT helps developers design more and code less.

The modelling phase is the initial stage. Normally, a system model that under testing is referred to as the abstract model because it is not as complicated as the actual system. The document specification or requirement creates this model, and it encodes the intended system behaviour. Furthermore, this phase includes a test plan that secures the needed requirements and considers the specifications of the design. Thus, this model would be applied in generating the test.

The next phase is generating test cases from the models. The tools that are applied in this stage are presented in the bold box line in Figure 2.7. Abstract test cases are produced from the abstract of the model. These abstract test cases select the main features that should be tested and removes the other details. The process of generating the test follows the coverage criteria to trace the test case requirements (Ammann & Offutt, 2008). Therefore, this study will focus on this stage to generate the test cases and adopt the coverage criteria.

In the third phase of this model, abstract test cases that were produced in the second stage are changed into practicable concrete test cases, and this process is referred to as test concretization or implementation. This phase also requires the support of tools and/or the aid of a programmer. This process is implemented through the implementation of many templates and mappings between the values of concrete and abstract test cases. In the fourth phase, the test is performed. However, the test needs to be adjusted to its environment before execution. This adjustment is done by using a test adapter tool. To perform the test against the real SUT, the test adapter adds the specific data of implementation to the tests. The test can be performed in two modes: online or offline. The tests and inputs generated in the online mode are used, which are also based on the response of SUT. The MBT tool manages and keeps track of the test results. However, in the offline mode, the saved forms of concrete test cases are created in the form of scripts that will be later performed through manual application or the use of certain tools.

In the last stage, the test results are analysed. This phase is similar to the conventional analysis of the test. In the case of errors in the test case, an analysis to identify the source of error is used. The error could be caused by a flaw or the test case in the system and in its setup.

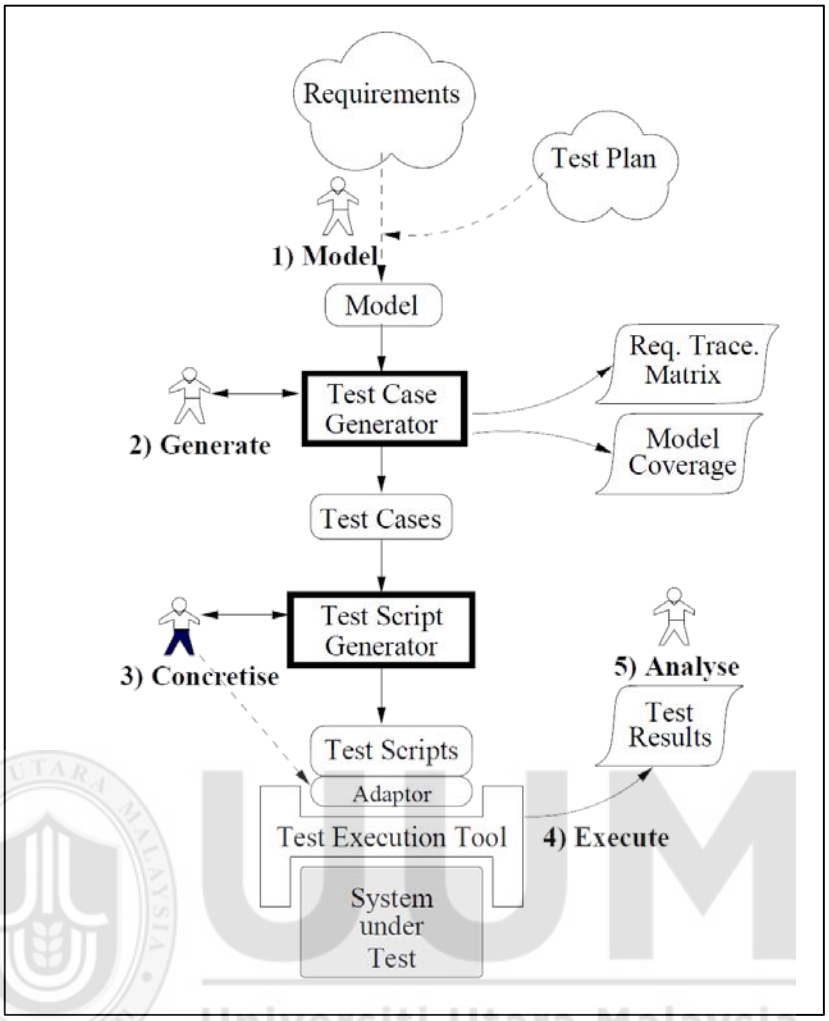

Figure 2.7. MBT Process
Source: Utting and Legeard (2010)

The next section elaborate on UML diagram in general and highlights the reasons for selecting UML statechart diagram as an input for generating the test cases.

2.6 UML Diagrams

UML is a modelling standard that is generally used in software engineering. UML incorporates a set of graphic notion techniques to produce visual models of object-oriented software-intensive systems (Sood & Rattan, 2016). In the 1990s at Rational Software, UML was developed by Grady Booch, Ivar Jacobson, and James Rumbaugh,

and was adapted and overseen by the Object Management Group (OMG) since 1997 (Sood & Rattan, 2016).

UML is a visual language for the documentation, construction, and specification of system artefacts (UML, 2004). Therefore, UML is a language that generally models and represents systems.

UML 2.0 is more extensive and complicated than the earlier version, also increased the amount of UML documentation. Figure 2.8 shows that the UML specification describes the two main kinds of UML diagrams, namely, structural and behavioural diagrams (Gupta, 2014).

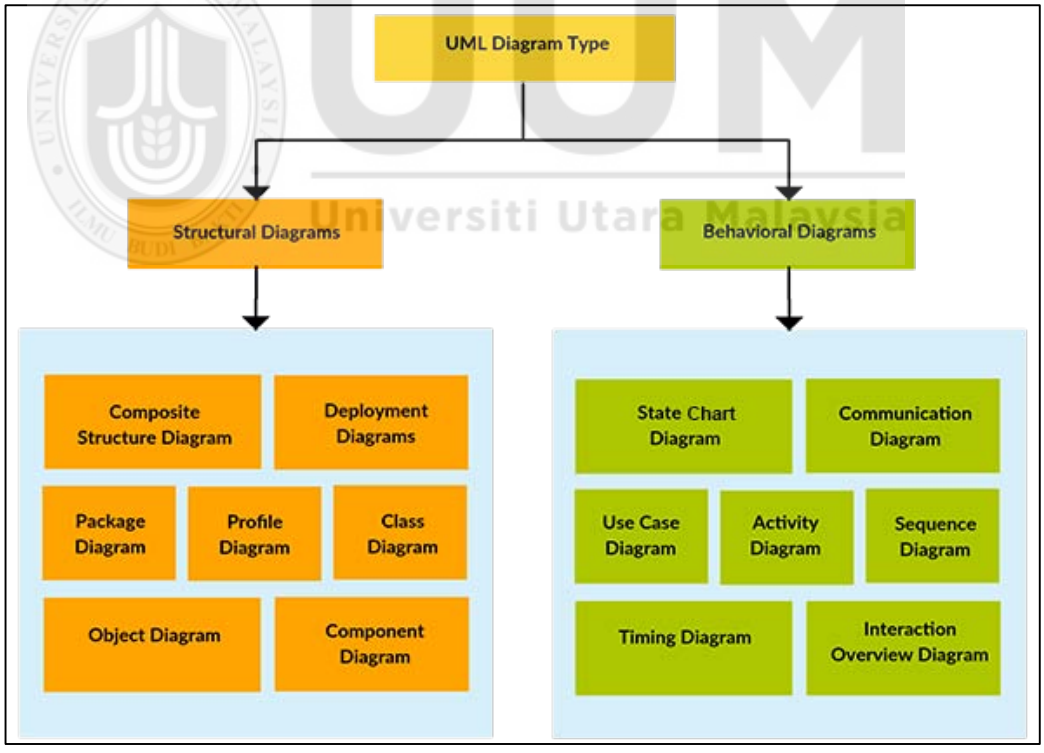

Figure 2.8. Overview of UML Diagrams
Source: Gupta (2014)

Structural diagrams show the system's static structure, various abstractions, and implementation levels in its fragments and the ways they are connected to one another. The meaningful ideas of a system are represented by the elements in a structure diagram, which are implemented in a real-world and abstract concept (Swain et al., 2012c).

Behaviour diagrams are used to represent the dynamic aspect and behaviour of the system (Knaak & Page, 2005). These diagrams are composed of a set of interconnected states and activities. Activities resemble the operations of object types, whereas states resemble their pre- and post-conditions. Structural diagrams represent the activity of the system's structure and its fragments on various stages of implementation and abstraction, which show the correlation among fragments (Na, Choi, & Lim, 2006).

UML is becoming an essential skill for anyone who is virtually incorporated in software projects (Wiegers & Beatty, 2013). The selection of modelling language in the system requirement and design are influenced by two reasons. First, the model offers a blueprint for developers to provide project managers the exact requirements needed to develop and precisely calculate the cost of a given project. Furthermore, UML is the bridge between non-technical users and technical developers (Ibrar, 2013).

Test cases are frequently generated from the code of the software after its implementation. Although UML diagrams are also used to generate test cases based on specification and design, in this testing the software developer is allowed to test the system before starting writing the code (Alhroob, 2012). Multiple diagram representations are provided by UML to describe the software design information from

different perspectives. UML was recently proposed as the new model to be used in the design and analysis of SUT (Weilkiens, 2011).

UML diagrams have become the source for test case generation because they are among the most preferred standard tools for the software development industry (Grant & Datta, 2016; Kim, Lively, & Simmons, 2006; Lange, Chaudron, & Muskens, 2006). Despite the use of UML diagrams in test case generation, several existing features do not consider the basic UML diagrams. These used diagrams are not consistent with the standard diagrams defined by OMG (2010). Using such modified diagrams that have extra techniques will require the developers and designers to obtain further knowledge of these tools.

The standard UML diagrams for test case generation are used in this study. The implementation of these standard diagrams will allow the developers and designers to focus only on the content of their design document without the additional burden of rephrasing their design documents in a different modified requested format.

Many studies on test case generation from UML diagrams are proposed because UML is considered as a standard in software development (Kernschmidt & Vogel-Heuser, 2013; Lange et al., 2006; Liu & Zhang, 2014). Furthermore, the availability, numerous support tools, and standardization of UML make it widely used in software development. In addition, transforming a suitable UML diagram to a source code can be easily done. Cavarra, Crichton, Davies, Hartman, and Mounier (2002) elaborated on the use of the UML diagram for automatic test case generation; they described each UML diagram component that could be used to generate test cases.

UML-based testing has been used by researchers for many years to produce test cases earlier in the development cycle (Rhmann & Saxena, 2016). Although prioritization techniques based on code are investigated by most researchers, the prioritization of test cases generated from UML diagrams has not received much research attention (Rhmann & Saxena, 2016).

Using UML diagrams to generate test cases is one of the most significant methods in software testing. One of the advantages of this method is covering the issues raised by object-oriented programs (Linzhang et al., 2004). However, using UML diagrams to generate test cases is a difficult and challenging task (Ali et al., 2014).

The UML statechart diagram is one of the most important UML diagrams because the appearance structure of the solution at the most detailed level is determined. In addition, the classes implemented and interaction with one another are shown (Bell, 2003).

Class diagram is another UML diagram that can be used to generate the test cases (Prasanna, Chandran, & Suberi, 2011). However, the behaviour of the system is not specified as compared to the UML statechart diagram. Furthermore, the UML class diagram does not contain any information on the target behaviour, and therefore cannot be used in test case generation (Doungsa-ard, 2012).

UML activity and statechart diagrams model the dynamic behaviour of the system, and the most frequently used UML diagrams for software design (Felderer & Herrmann, 2015; Schweighofer & Heričko, 2014). Essentially, the UML activity diagram is considered as a flowchart that shows the activity's flow of control. However, the UML

statechart diagram shows a state machine to ensure the state flow of control. A UML activity diagram is a special case of UML statechart diagram in which all or most of the states are activity states and the transitions are activated by completion of activities in the source state (Jürjens, 2005).

Using UML activity diagrams to generate test cases will not illustrate any state information of the system. Therefore, the state of the entire system during the execution of a use case, which is a collection of objects, remains unknown. However, the system behaviour may be varied to the same input depending on the state a system is in (Swain et al., 2010a). The UML statechart diagram shows the lifecycle of an object: the transitions that it undergoes upon receipt of an event. The UML statechart diagram test cases can reveal unit-level faults better than other diagrams (Abdurazik et al., 2004).

Test cases generated through UML statechart diagrams revealed 12% more integration level faults than those revealed by sequence diagrams. This result indicates that UML state diagrams are more efficient in revealing unit-level faults (Baig, 2009). Furthermore, Kansomkeat, Offutt, Abdurazik, and Baldini (2008) show in an experiment that testing using UML statechart diagram tests more faults than testing using a UML sequence diagram. In addition, twice as many tests from UML statechart than tests from sequence diagrams are observed. The UML statechart diagram has the same semantics as the other state-based specifications, thereby enabling the generalization of the proposed test case generation (Abdurazik & Offutt, 1999).

This current challenge exists in generating test cases from software design lifecycle rather than in the coding cycle of the system development (Makker & Singh, 2011).

This condition will result in the use of a proper model in the software system, which is the first step of the solution because UML diagrams are used to represent different software design issues (Berardi, Calvanese, & De Giacomo, 2005). The UML statechart diagram can be used to represent parallel activities and hierarchical relationships that usually are presented in modern complex software (Santiago et al., 2008). This illustration helps in testing the design cycle in the software system using the generated test cases. The next section describes the UML statechart diagram and its benefit in test case generation.

2.6.1 UML Statechart Diagram

This section explains the principles of the UML statechart diagram. The statechart structure is based on statechart formalism. David Harel firstly introduced this diagram in 1987. This diagram is a visual modelling language that represents finite state automata with added parallelism, hierarchy, broadcast communication, and history. Harel created the formalism to describe large and reactive systems because he believed that such a method was not available at that time (Harel, 1987).

In UML statechart diagrams, as shown in Figure 2.9, the basic elements are the rounded rectangles that represent the states and labelled arrows that indicate transitions. The composition for transition is “Event [guard condition] / action,” in which the event is considered as a message that is sent (Kansomkeat & Rivepiboon, 2003). In the UML statechart diagram, states, events, and transitions are the fundamental components and the main building blocks (Specification, 2007). Conceptually, an object remains in a state until an event causes it to transit to another state (Samuel et al., 2008).

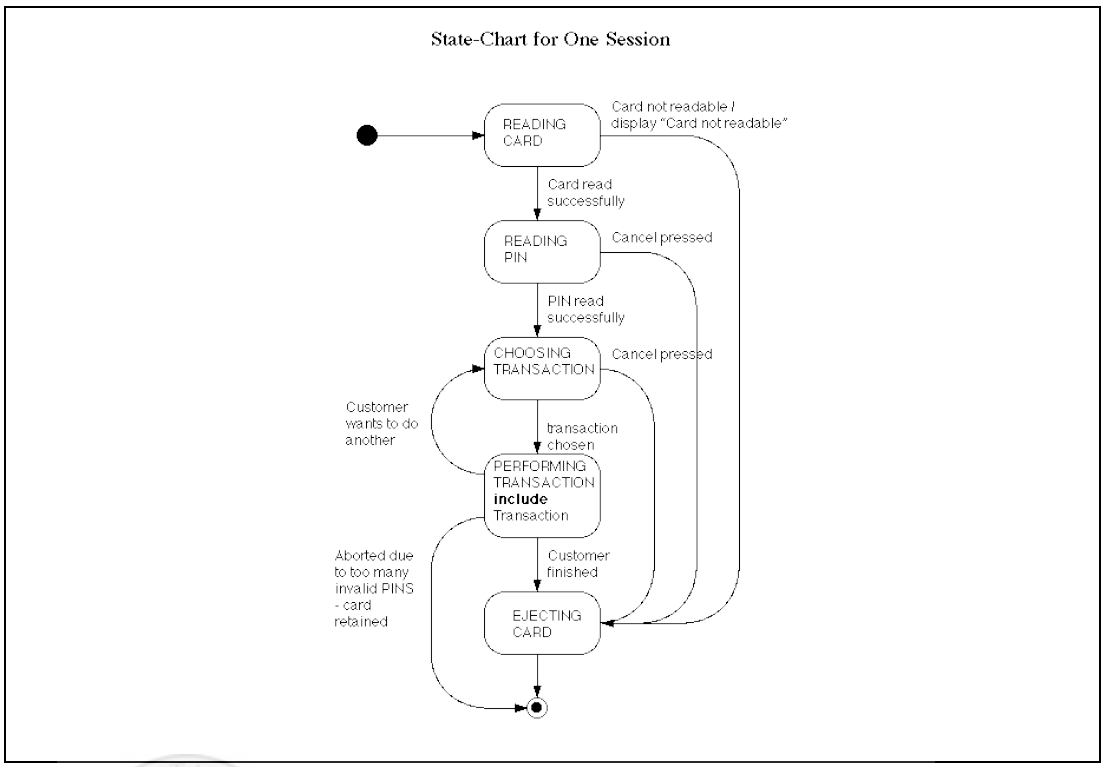

Figure 2.9. Simple UML Statechart Diagram for ATM Machine Transactions

A state may include other summarized sub-states, and the state is called a composite state. One of the special states in generating test paths is the starting state, which indicates the first condition of every test path. Another special state is the end state, which indicates the last condition and the end of all the paths (Gross, 2005).

The UML statechart diagram has two types, namely, composite or simple type. A composite state entails one or more regions. A simple state does not have any sub-states. A composite state can be either sequential or concurrent. A composite state cannot be in more than one sub-state at any time but can be in any one of its sub-states (Specification, 2007). OMG suggests that in a concurrent type, an object and logic of its sub-states determine the state. The object is regarded to be in all the concurrent states simultaneously.

UML statechart diagrams use states and state transitions to describe software behaviours or a single object (Yemul et al., 2014). The diagrams define the dynamic software behaviour in terms of how it responds to external input (Kansomkeat et al., 2008). These diagrams are used to aid the software developer in depicting the dynamic behaviour of the entire system or in better understanding any complex functionality or a single object in a system or a sub-system (Yemul et al., 2014). Therefore, this diagram can naturally be considered as a good source for unit testing (Kansomkeat et al., 2008).

Finite state machines (FSM) had been used to describe the reactive components of models with reactive components for more than half a century before UML statechart diagrams were introduced (Drusinsky, 2011). However, when FSMs were applied to larger problems, the models were unreadable and cluttered because FSMs were sequential and flat. Therefore, relatively simpler systems, such as protocols, are modelled using FSMs. More complex systems, such as the engine controller of an aircraft, are modelled using the UML statechart diagram (Mathur, 2008). In addition, UML statechart diagrams have built-in capabilities in their environment to describe their interaction with multiple objects (Drusinsky, 2011). Furthermore, UML statechart diagrams, being an extension of FSM, with added functions, are the most popular language for modelling reactive components (Drusinsky, 2011).

The UML statechart diagram is a rich extension of the FSM. The UML statechart diagram needs to be handled differently when used as an input to generate test cases (Kaner & Fiedler, 2013). The UML statechart diagram is considered as a source for test case generation and is not the item under test (Doungsa-ard, 2012). Instead, the

UML statechart diagram implementation is under test. Such an integration is also known as implementation under test (Kaner & Fiedler, 2013). For example, a UML statechart diagram may represent the model of a user login while an implementation under test is its integration.

This study aims to generate a test case from the system design cycle that achieves most of the system coverage criteria. UML statechart diagrams are selected as the input of this study. The UML statechart diagram describes the changes in the system, which can be represented by an attribute value or state of the system. The UML activity diagram or the UML case diagram are not practical to be used for test case generation as an inputs because they describe the system more from the business view, which is the functional requirement, and not that of the software developer (Al-kahlout, B. salha, & El-haddad, 2017; Doungsa-ard, 2012). The UML statechart diagram describes the system based on the programmer view to map the software more easily.

2.7 Test Case Generation in Model-based Testing

This section surveys the current research in test case generation using MBT from UML diagrams. The information and data obtained from this review will aid in devolving the framework and selecting the coverage criteria.

Researchers such as Gnesi et al. (2004); Kansomkeat and Rivepiboon (2003); Kim, Hong, Bae, and Cha (1999) have paid considerable attention to automatic test case generation from UML diagrams. Numerous efforts were paid to use UML diagrams to generate test cases (Linzhang et al., 2004; Mingsong et al., 2006). In addition, studies have been conducted on the test case generation from UML-diagram-based activity,

which uses a grey box method to generate test cases (Linzhang et al., 2004). At the same time, more researchers have worked on generating test cases from UML statechart diagrams (Ali et al., 2007; Kosindrdecha & Daengdej, 2010; Swain et al., 2012c).

Although the focus of this study is on the test case generation from UML statechart diagram, test case generations from other types of diagrams are also investigated to identify the commonalities or trends in the test case generation algorithm being applied. It has been spotted after passing through different approaches that UML diagrams like activity, sequence, and statechart have been used to generating test cases, and their techniques share some similarities in some components. The following section presents test case generation approaches using UML activity, sequence diagram, and statechart.

2.7.1 Test Generation Approaches Using UML Activity Diagram

As shown in Figure 2.10, UML activity diagrams clarify the sequential control flows of activities. The UML activity diagram uses a kind of directed graph as its graphical illustration. The action node in a UML activity diagram is represented by a rectangle with rounded corners. This node represents the execution of an operation on input data, and new data are generated to deliver an outgoing edge.

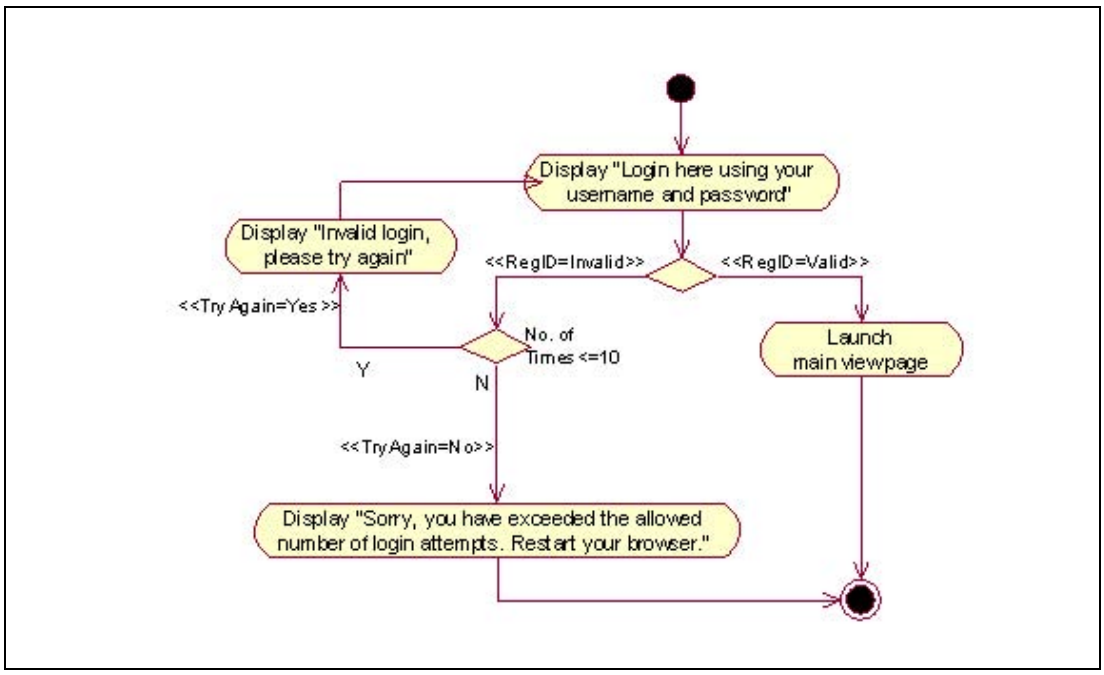

Figure 2.10. Simple UML Activity Diagram for Login Screen

The control flows of activities in the UML activity diagram are used to model the dynamic aspects of a control flow of an operation or a group of objects, which shape a kind of design specifications for the software (Chen et al., 2009).

The test case is directly extracted from the UML activity diagrams, which model complex processes that have loops, parallelism, and event-driven behaviour. The UML activity diagrams can also be used to specify the workflow and business process or to model the behaviour of some use cases (Eshuis, 2006).

The predicted behaviour of an operation, which deals with coverage criteria, are described in the following sections. Successively, each test scenario provides complete information on the test case generation. Finally, the application of the category-partition method makes potential input/output constraints (Ostrand & Balcer, 1988) that can achieve path coverage from the generation of the test cases. However, this

technique at any time of execution overlooks information related to the state of the objects inside the system.

Mingsong et al. (2006) offered to acquire a reduced test set implementation using UML activity diagrams. They focused on generating the test cases for Java programs randomly and achieved the execution traces of the program by executing the programs while applying the test cases. They also acquired reduced test cases by comparing simple paths with program execution traces. The help of the plain path coverage criterion avoids the path explosion because the loops and concurrency are available. Their adequacy coverage criteria for the UML activity diagrams are based on the matching between UML activity diagrams and the paths execution traces of the implementation codes of the program. They generally dealt with activity coverage, transition coverage, and simple path coverage. However, their approach was limited to UML activity diagrams that do not contain loops or concurrency.

Chen, Mishra, and Kalita (2008) proposed an approach for automatic test case generation using UML activity diagrams. To generate properties, the researchers used specification coverage and design models and model checking to enable directed test generation. Their technique achieved activity coverage, key path coverage, and transition coverage. To generate directed tests, the researchers defined and used the fault model of the specification model. In their study, the UML activity diagram was converted to intermediate model as the formal model. Then, the properties were generated from the coverage criteria. Finally, to generate required tests, the properties were applied on the formal model using model checking.

Fan, Shu, Liu, and Liang (2009) proposed a technique for test case generation from sub-UML activity diagram to introduce composite activity diagram hierarchically. Their technique generated test cases based on (intermediate model) composition trees generated from UML activity diagram by taking the functional decomposition, round-robin strategy, and bottom-up integration testing strategy into consideration. Furthermore, their coverage criteria were based on transition coverage and activity coverage.

Kansomkeat, Thiket, and Offutt (2010) proposed a method for generating test cases from UML activity diagrams, which is called the condition classification tree. Then, intermediate model condition classification trees were generated by analysing the UML activity diagrams, which were then used to create test cases and test case tables. In addition, to introduce faults, they used mutation analysis to evaluate test sets based on the number of mutants that failed or were killed. However, the mutants were generated manually.

Kundu and Samanta (2009) proposed an approach to generate test cases using UML activity diagrams. In their approach, they translated the UML activity diagrams into an activity graph. From the result of the activity graph, they used DFS and breadth first search (BFS) algorithm to generate test cases. These generated test cases are based on an activity path coverage criterion and are used to cover loop faults and organization. To achieve UML activity diagram coverage, they considered a coverage criterion called activity-path coverage criterion.

Hashim and Salman (2011) proposed a test case generation algorithm from a UML activity diagram, where they generated the test case by converting the UML activity

diagram into an activity graph that store all the activity information. The graph was used to automatically generate an activity path, which contains all the possible test case paths. Then, from all the stored information and the paths, the test case was generated automatically. Furthermore, a prototype was created to implement and test the algorithm.

Boghdady et al. (2011b) proposed a newly enhanced methodology to generate test cases automatically from UML activity diagrams using the extensible markup language (XML) form. The XML for each UML activity diagram in any system was transferred to an activity dependency table, which covers a reduced form of all the functionalities in the UML activity diagram. A directed graph called activity dependency graph was automatically generated using the activity dependency table, which was used in combination with the table to generate all the possible test case paths. In their study, to achieve minimization, they reduced the test case paths before generating the final efficient set of test cases. To accomplish their validation, they implemented the cyclomatic complexity technique to the generated test case paths to calculate the lower bound for the generated test case paths. Thus, the general performance of the testing process was optimized with respect to saving time and effort.

Table 2.1

Test Case Generation Methods Using UML Activity Diagram

Author(s)	**Input model**	**Method**	**Intermediate model**	**Coverage criteria**
Mingsong et al. (2006)	Activity diagram	Random testing	-	Activity, transition, simple path

Table 2.1 Continue

Author(s)	Input model	Method	Intermediate model	Coverage criteria
Chen et al. (2008)	Activity diagram	Coverage driven	Formal model	Activity, key path, transition
Fan et al. (2009)	Activity diagram	Bottom-up testing strategy activity diagram	Composition tree	Transition, activity
Kundu and Samanta (2009)	Activity diagram	DFS and BFS traversal	Activity graph	Activity path
Kansomkeat et al. (2010)	Activity diagram	Condition classification tree method, mutation analysis	Condition classification trees	-
Hashim and Salman (2011)	Activity diagram	Activity path	Activity graph	Activity path
Boghdady et al. (2011b)	Activity diagram	XML form	Activity dependency graph	Hybrid

Table 2.1 represents the test case generation based on a UML activity diagram. It shows the input models that have been used for test case generation, which is the activity diagram. Furthermore, the intermediate models that generated an intermediary between the input model and the generated paths, and the coverage criteria, are also clarified in this table.

As observed in Table 2.1, amongst the seven papers that report credible evidence, most papers used activity graph as the intermediate model. There are also only one paper that did not use intermediate model which is by Mingsong et al. (2006). Moreover, the

table shows that the intermediate model is quite important in assisting the generation of the test cases. In addition, the methods used in these papers to generate the paths from the intermediate model can be used also on different UML diagrams as can be seem in Table 2.1. The method proposed by Kundu and Samanta (2009) to construct the UML activity diagram can be adapted in this study to be used on UML statechart diagram construction to be used later in path pruning (refer to Section 4.3.1). In addition, Boghdady et al. (2011b) in their work described the converting from the intermediate table to intermediate graph, was also adapted in this study (refer to Section 4.3.3). An activity diagram has transition coverage, activity coverage, concurrent-path coverage, and simple-path coverage criteria (Shirole & Kumar, 2013). However, the transition coverage is the most commonly used method in the activity diagrams, as presented in Table 2.1.

2.7.2 Test Generation Approaches Using UML Sequence Diagram

In the UML sequence diagram shown in Figure 2.11, the control structures and the sequence of messages between the objects, which contains group of objects and messages, are described. The lines in the objects represent messages and the lifelines represent objects. The messages show an association among the objects to complete the system functionality, and they are exchanged from top to bottom in a natural order sequentially.

A UML sequence diagram is an illustration of the successful and unsuccessful event collaboration between the objects. Consequently, this diagram is useful in integration testing (Specification, 2007). For each use case, to understand the dynamic behaviour of the system, a UML sequence diagram is drawn.

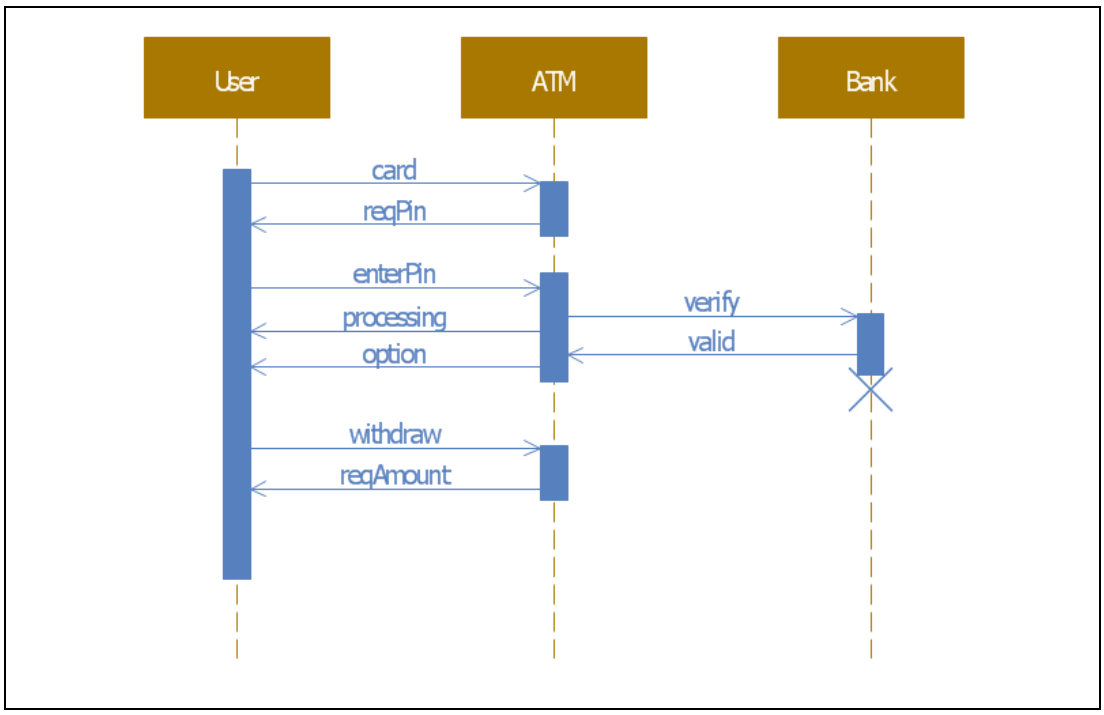

Figure 2.11. Simple UML Sequence Diagram for ATM Machine

Pilskalns et al. (2003) presented a method to generate test cases using UML sequence diagrams. They also presented a new graph that summarizes many paths that exist between objects, using their own method, and called it object method acyclic graph (OMDAG). To generate the test cases, they generated test paths by crossing OMDAG using DFS or BFS algorithms. Thereafter, they selected the suitable parameter and attribute values for object instantiations that cause the execution of the required test sequence. Finally, all tests were defined in the object method execution table. To accomplish the coverage criteria of this study, all message paths, and full predicate coverage were measured.

Li, Li, Qing, and Chen (2007) presented an approach to automatically generate a test case from a UML sequence diagram. To amend the UML limitations, they used the Object Constraint Language (OCL). In the beginning, they constructed a tree representation of sequence diagrams. Then, the traversal of the constructed tree for

selecting conditional predicates from the sequence diagram was carried out. Then, to generate test data, they selected conditional predicates and OCL pre- and post-condition expressions for each path. A function minimization technique was also used to generate the test data. Furthermore, the coverage criteria focused on message, path, and condition coverage.

Dinh-Trong, Ghosh, and France (2006) presented a systematic approach to generate test inputs from UML class and sequence diagrams. Test case generation information was collected in a directed graph called variable assignment graph from sequence and class diagrams. This approach adopts a symbolic execution method to derive test input constraints from the presented graph and solves these constraints with the alloy constraint solver. In addition, their coverage criteria focused on all message paths and condition coverage.

Shirole and Kumar (2010) proposed a hybrid approach to generate test cases for MBT that uses the information from UML sequence diagram and genetic algorithm. Test cases have evolved through generations to correct sequence flows of execution. Therefore, they used a method named call dependencies to show the sequence diagram, which is useful for integration testing. The generation of test cases using genetic algorithm improved the exception coverage as well as method coverage. The coverage criteria focused on message sequence coverage.

Nayak and Samanta (2010) proposed an approach to generate test cases from the information embedded in UML sequence diagrams, class diagrams, and OCL constraints. In their study, a structured composite graph was generated from class diagrams and OCL constraints to improve a sequence diagram with constraint

information and attribute. They generated the test specifications from the structured composite graph. In addition, their coverage criteria focused only on all path coverage.

As shown in Table 2.2, the UML sequence diagrams played a role in generating test cases because these diagrams were also part of the requirements and development diagrams. The table illustrates the input model, the method to generate the test case, the use of the intermediate model as a medium, and the coverage criteria. In addition, the use of DFS is quite prominent in generating the paths. Moreover, for the coverage criteria, all message and path conditions are commonly used.

Table 2.2

Test Case Generation Methods Using UML Sequence Diagram

Author(s)	**Input model**	**Method**	**Intermediate model**	**Coverage criteria**
Pilskalns et al. (2003)	Sequence diagram	DFS or BFS traversal	Object method directed, acyclic graph	All message paths, full predicate
Li et al. (2007)	Sequence diagram	Traversal, function minimization	Scenario tree	Message, path, condition
Dinh-Trong et al. (2006)	Sequence diagram	Symbolic execution, constraint solver	Variable assignment graph	All message paths, condition
Shirole and Kumar (2010)	Sequence diagram	Genetic algorithm	Call dependency graph	Message sequence
Nayak and Samanta (2010)	Sequence diagram	DFS traversal, symbolic execution	Structured control graph	All paths

2.7.3 Test Generation Approaches Using UML Statechart Diagram

Statechart diagrams in UML can be used to construct the dynamic aspects of a system. This diagram consists of transitions, states, actions, and events (Rumbaugh et al., 2004) and emphasizes the flow of control from state to state by illustrating a state machine. A UML statechart is a comprehensive FSM with concurrency, hierarchy, and communication, and these extensions allow small diagrams to express complex behaviour in a modular method (Utting & Legeard, 2010).

The purpose of generating a test case using a UML statechart diagram is to verify the relationship among the behaviour, state transition, state, action, and event (Kim et al., 2011). This technique is used to determine if the system specifications are fulfilled through the state-based motion of the system. In the state-based system, three reasons caused the fault. First, the state diagram cannot accurately transfer the system function specification. Second, the UML statechart diagram configuration is erroneous or unreliable. Finally, the statechart diagram is converted to programmable code (Kim et al., 2011).

Kim et al. (1999) proposed a method to produce test case generation for class testing by using UML statechart diagrams. By deriving test cases, their method transformed a UML statechart diagram into an extended finite-state machine (EFSM). In the resulting EFSMs, broadcast communications were removed and the concurrent and hierarchical structures of states were compressed. By transforming EFSMs, data flow was defined into flow graphs. In the flow graphs, the conventional data flow was applied to analyse the techniques. However, their work only discussed a method for the generation of test cases and thus an automated environment would be needed in order to support the total

process of class testing, also they focused on the unit testing of classes, but did not consider inter-relationships between classes.

Hartmann et al. (2000) extended the UML diagrams with particular representations to generate a design-based testing situation. First, a UML statechart diagram was issued to define the active behaviour of each system part. Then, the connections between the parts were identified. By interpreting the state diagrams, a global FSM was obtained. This global FSM relates to the integrated system behaviour, which was used to generate the test cases. The authors aimed at the stub generation problem, but how their method reduces the number of manually crafted stubs remains unclear. The interaction between the components was conducted via message exchange, which did not contain parameters and values. In this thesis, no constraint on the message was used. Furthermore, components interacted via message exchange containing parameters and values.

Kansomkeat and Rivepiboon (2003) developed a transformation method from UML statechart diagrams into intermediate diagrams that were used to generate test sequences. The test cases were generated automatically from UML statechart diagrams created by the Rational Rose tool. The testing coverage criterion was used to guide the generation of test cases and to cover the intermediate model testing flow graph (TFG) from the all-state coverage and all-transition coverage. Based on their fault detection abilities, the test cases measured the effectiveness. From the generated test cases, the results of simple test experiments had high effectiveness. However, usually, more than one object often participated in the execution of a use case. Therefore, testing using

this approach with the chance of such behaviour occurring will be difficult. In addition, the approach did not generate multiple test data because of the lack of coverage.

Offutt, Liu, Abdurazik, and Ammann (2003) developed a method to automatically generate test cases from UML statechart diagrams by changing events for Boolean class attributes. The developments of many useful coverage criteria that were centred on the UML statechart diagrams were found to be effective. Class-level testing was the aim of their approach. This method attains transition pair coverage, transition coverage, and full predicate coverage. Offutt and Abdurazik (1999) also provided suitable visions on counting test prefaces that contain necessary inputs for the test values to place the software into the suitable state. In their study, all transitions were assumed to be triggered by change events. In addition, their approach did not handle guards. In comparison, their work was not limited to any particular type of event or transition. The developed algorithm from their work will handle change events, time events, and transitions with guards.

Gnesi et al. (2004) offered a formal test case generation by providing a mathematical basis for conformance testing and automatic test case generation for UML statechart diagram that was established on an operational semantic. With transitions labelled by input/output pairs, they proposed a formal conformance testing relation for input-enabled transition systems. To succeed in the specified requirements, testing the software was identified as conformance testing. Considering the formal specification, a conformance relation defines the accuracy criterion of the implementation. However, proper test selection strategies are needed to apply the test generation algorithm in practice.

Briand, Labiche, and Cui (2005) focused on creating a methodology using UML statechart diagram to define the system state required for each event or transition, which are part of the paths to be tested, input values for the parameter for all actions, and events associated with these transitions. Their work generated a test case specification involving a possible sequence of transitions. A requested sequence tree was also constructed to develop the test restraints for the transition sequences and to acquire the interactions among state-dependent objects in their work.

Li and Lam (2005) presented an approach to generate test sequences from UML statechart diagrams using ant colony optimization. A UML statechart diagram was transformed into an intermediate model called a directed graph. By exploring the directed graph by a group of ants cooperatively, test sequences were generated. From this generation, all-state coverage was achieved.

Santiago et al. (2006) presented a method to automate test case generation from UML statechart diagrams using a software specification model. This method converts the UML statechart diagram model into an XML-based language table. Moreover, by using the performance chart tool, they generated an intermediate model as FSM based on control flow. Their intention was to determine that by using a higher-level technique, such as UML statechart diagrams, a complex software with clarity and rich details can be presented. UML statechart diagrams are able to model a complex system more realistically and provide hierarchy and parallelism for it. Although these conditions are not enough to guarantee that a test case generation approach is successful, an improvement was still observed especially when the conditions have been compared with the use of Condado as an unconnected tool with FSM

specification. In addition, the Condado implements the switch cover method for the control part. A switch is a transition-to-transition pair, and their method generated test cases to cover all pairs of transitions in the model in the coverage criteria.

Murthy et al. (2006) suggested a new foundation to generate test cases using the UML statechart diagram as the basis model of behaviour. They also defined a test-ready UML statechart diagram, which indicates that the model is ready with data for a test generator to generate test scripts automatically from it. To generate the paths, the researchers started from the starting vertex with a state transition and reconnoitred the next vertex subsequent of its state transitions. A satisfied state transition provided guard condition. The researchers solved the problem of generating the test case from a UML statechart diagram by defining all the sentential forms derivable from an equivalent extended context-free grammar model. Additionally, in the convergence criteria, they achieved the path coverage and the basic path coverage.

Ali et al. (2007) projected a method for state-based integration testing. Their work produced an intermediate test model called state collaboration test model (SCOTEM) from the corresponding UML statechart diagrams and UML collaboration diagrams. SCOTEM copies all possible paths for object state changes where message sequences may be produced. Then, the model produces test paths centred on several coverage criteria. For them, revealing the state-dependent interaction errors is the goal behind the generated test cases. Their work reflects the analysis of all possible states of cooperating levels in an interface.

Santiago et al. (2008) presented an environment called automated generated test case based on statechart (GTSC) that allows a test designer to generate test cases based on

statechart test criteria and FSM methods. This interesting characteristic allows test sequence generation from both statechart and FSM techniques, which are based on the same FSM. However, other comparisons need to be made, namely, all-paths-k-C0-configuration of the statechart coverage criteria family (SCCF) as well as the round-trip route testing offered by Binder (2000) and all-paths-k-configurations. Similarly, more comparisons between the latest FSM-based methods are available, such as state counting, and some SCCF criteria. Such an analysis will be enabled with the help of mutation testing by GTSC in applying these test criteria methods.

Kosindrdecha and Daengdej (2010) proposed a new method to generate and prepare both test data and test case based on UML statechart diagram, called TGfMMD method. This method has been developed to verify the UML statechart diagram before the generation of both test cases and test data from the extended UML statechart diagram. However, this method has not been tested in a complex UML statechart diagram.

Swain et al. (2010a) proposed a novel technique to generate test cases automatically from UML statechart diagram and activity diagram. They constructed an intermediate representation based on the model, which they named SAD. They generated the test case from the use of SAD generation, DFS, and mutation analysis. In addition, to detect harmonization of the UML statechart diagram as well as activity diagram faults within a use case of the system exercise, an activity synchronization in the context of multiple state combinations was used. They also achieved transition coverage and state/activity path coverage. For the testing, they have implemented a prototype tool based on their

approach. However, in their study, the tester should select the test data for each test case manually.

Shirole et al. (2011) also worked on the automatic generation of a test case using a UML statechart diagram. The researchers used the genetic algorithm as a medium for their tool by combining the information from the UML statechart diagram. They proposed a search-based approach to handle infeasible paths and test data generation. They also used the following steps to generate the test cases. First, the UML specifications were transformed into EFSM. Second, the EFSM was transformed into an extended control flow graph. Third, test sequences were generated using genetic algorithm and DFS. Finally, the test cases were selected using data-flow techniques. In the coverage criteria, they focused on state cover, transition cover, all-definition cover, and all du-paths. However, the UML statechart diagrams that they considered were very simple, which led to reduced coverage when dealing with scenarios that are more complex. Full path coverage is not obtained because of the use of DFS and fitness function.

Li et al. (2012) presented a test case generation approach that takes UML statechart diagrams as inputs. The researchers first constructed the UML statechart diagram to conform to system requirements. Then, the .mdl file of the UML statechart diagram was analysed, and the main information of the UML statechart diagram was extracted and converted into a directed graph. Finally, an algorithm was designed to construct the Euler circuit based on a directed graph and test cases were generated automatically by Euler circuit algorithm. Their specified test coverage criteria were the state coverage and transition coverage of UML statechart diagram to minimize the number

of test cases. Although generated test paths were minimized, they still contained redundant transitions.

In an earlier study, Swain et al. (2012c) proposed an approach to automatically generate test cases from a UML statechart diagram. First, the researchers constructed the UML statechart diagram for a given object. Then, the UML statechart diagram was traversed, conditional predicates were selected, and these conditional predicates were transformed into source code. Then, the test cases were generated and stored by using function minimization technique. From the UML statechart diagram, they performed a DFS to select the associated predicates. After selecting the predicates, they predicted an initial dataset. They generated test predicate conditions from a UML statechart diagram, which were used to generate test cases. Their technique accomplished limited coverage in test cases such as transition pair coverage, state coverage, action coverage, and transition coverage. The technique also achieved full-predicate coverage by generating test data for each conditional clause. Moreover, the technique can handle transitions with guards and achieve a transition path coverage. In the present study, the quantity of test cases is minimized. By contrast, Swain et al. reached transition path coverage in testing the limitations decided by simple predicates, but the test case needs to be optimized.

Additionally, Swain et al. (2012b) proposed an approach for test case generation, namely, test generation and minimization for O-O software with statechart (TeGeMiOOSc). The researchers started by analysing the system, which was tested and accepted by users, and then by building the UML statechart diagram. After they converted the given UML statechart diagram into an intermediate model, they named

it as a state transition graph. DFS was used to form test sequences and generate all the possible paths. Then, all the valid sequences of the application were obtained until the final edge was reached. Finally, they minimized a set of test cases by calculating the state coverage for each test sequence. In the same year, Swain et al. (2012a) performed a similar experiment to generate a test case from the UML statechart diagram, which was called generation and minimization of test cases from statechart (GeMiTefSc). First, the researchers built a UML statechart diagram model for SUT. Next, they conjugated a state transition graph from a UML statechart diagram. Then, by using the graph, all the required information were extracted. Next, they generated the test cases by applying Wang's algorithm (Linzhang et al., 2004). Finally, they minimized the set of test cases by calculating the state coverage for each test case, which helped them determine the test cases that were covered by other test cases. However, after creating the intermediate graph, the researchers relied on DFS to generate the paths, which resulted in reduced coverage when the UML statechart diagrams have loops and feedbacks in it. Moreover, by using minimization, they minimized a set of test cases, which caused overlapping or neglecting some of the important data, thereby leading to less coverage.

Chimisliu and Wotawa (2012) in their earlier work proposed a method for generating test cases aiming to automatically achieve transition coverage and state coverage of the model. Their proposed approach presents an automatic transformation of the system composed of communicating a UML statechart diagram into a language of temporal ordering specification. They also showed how to generate test cases in a semi-automatic way by using an input from the user as explanations on the UML diagram. In their work, the generated test case coverage criteria did not contain any

rejected transitions. Thus, the generation process was not as efficient as in the case when the user provides explanations that can be used as rejected transitions in the test purpose.

In their more recent work, Chimisliu and Wotawa (2013a) and Chimisliu and Wotawa (2013b) proposed an improved tool for test case generation from UML statechart diagram by using control, data, and communication dependencies. They generated the test cases by using the TGV technology (Claude & Thierry, 2002), which is a test case generator from the analysis and the construction of distributed processes toolbox. For the coverage criteria, their generation technique was intended to achieve transition coverage only. Therefore, the lack of coverage indicates the need to enhance this method or obtain a novel one.

Li, Li, Tan, and Xiong (2013b) presented an approach using extended context-free grammar to generate test cases from a UML statechart diagram. They used the context-free grammars and UML statechart diagram as inputs, to perform an automated consistency simulation for UML specification. First, they refined the source file of the UML statechart and transformed it automatically into an intermediate model called directed diagram. Then, they introduced the concept of PLAY-Tree; the consistency checking of the UML statechart diagram is defined, where the existence of a corresponding PLAY-Tree in all successful branches was checked. Their work only satisfied the transition coverage criterion and state coverage criterion from many paths.

Ali et al. (2014) proposed a test-case-based technique using the UML state diagram. They transferred the UML statechart diagram into an intermediate graph, which is the FSM. Each node in this graph stores the necessary information for the test case, which

will be generated later. They also used additional parameters for test case generation, including pre- and post- conditions and object constraint language. By using FSM as input to BFS, Ali et al. generated and transformed all basic paths to obtain a suitable test case using the test-set generation algorithm. The generated test cases satisfied the transition, transition pair, and state coverage criteria. However, apart from ignoring loops, these cases required additional inputs to satisfy the coverage criteria.

Table 2.3

Test Case Generation Methods using UML Statechart Diagram

Author(s)	**Input model**	**Method**	**Intermediate model**	**Coverage criteria**	**Evaluation**
Kim et al. (1999)	Statechart	Data flow	EFSM	-	Comparison
Hartmann et al. (2000)	Statechart	Test Development Environment, test specification language (TSL)	Directed graph	Transitions	Comparison
Kansomkeat and Rivepiboon (2003)	Statechart	Parsing TFG, mutation analysis	TFG	State, transition	Mutation Analysis
Offutt et al. (2003)	Statechart	Spectest, software cost reduction	Specification graph	Transition, full predicate, transition pair, complete sequence	Comparison
Gnesi et al. (2004)	Statechart	IOLTSs, random test selection	-	-	Comparison

Table 2.3 Continue

Author(s)	Input model	Method	Intermediate model	Coverage criteria	Evaluation
Briand et al. (2005)	Statechart	Normalization and analysis of operation contracts and transition guards	Invocation sequence tree	Transitions, transition pairs, full predicate, round-trip paths	Case Study
Li and Lam (2005)	Statechart	Ant colony optimization	Directed graph	States	-
Santiago et al. (2006)	Statechart	PerformCharts and Condado	FSM	Transition pair	Case Study, Simulation
Murthy et al. (2006)	Statechart	Extended UML statechart model	Context-free grammar model	Path, basic path	-
Ali et al. (2007)	Collaboration and statechart	SCOTEM constructor, test path generator, test executor	SCOTEM	Basic path, transition, N-path, and path	Mutation Testing
Santiago et al. (2008)	Finite state machines and statechart	Switch cover, distinguishing sequence and unique input/output methods	FSM	Transitions	Comparisons
Kosindrdecha and Daengdej (2010)	Statechart	TGfMMD method	Sketch diagram-based technique	States	Comparison
Swain et al. (2010a)	Statechart and	SAD generation, DFS, mutation analysis	SAD	State, transition, path	Mutation Analysis

Table 2.3 Continue

Author(s)	Input model	Method	Intermediate model	Coverage criteria	Evaluation
	activity chart				
Shirole et al. (2011)	Statechart	Genetic algorithm	Extended control flow graph	State, transition, definition, and du-path	Empirical Study
Li et al. (2012)	Statechart	Euler circuit algorithm	Directed graph	State, transition	Comparison
Swain et al. (2012c)	Statechart	DFS, Model JUnit	Statechart graph	State, transition, transition pair	Comparison
Swain et al. (2012b)	Statechart	TeGeMiOOSc	State graph	State, action, transition, transition path, condition	Comparison
Swain et al. (2012a)	Statechart	GeMiTefSc	State graph	State, action, transition, path, condition	Comparison
Chimisliu and Wotawa (2012)	Statechart	TGV test case generation tool	Test purpose	Transition, state	A case study comparison
Chimisliu and Wotawa (2013a)	Statechart	TGV and the Input/Output Conformance (IOCO) theory	Test purpose	Transition	A case study comparison

Table 2.3 Continue

Author(s)	Input model	Method	Intermediate model	Coverage criteria	Evaluation
Chimisliu and Wotawa (2013b)	Statechart	TGV and IOCO	Test purpose	Transition	A case study Comparison
Li et al. (2013b)	Statechart	Extended context-free grammar	Directed diagram	Transition, state	-
Ali et al. (2014)	Statechart and use case	BFS	FSM	Transition, transition pair, state	Comparison

Table 2.3 reviews the studies of some researchers in the past decade and the input models that they used, such as the UML statechart diagram or its combination with others, and the method they used to generate the test cases. Additionally, the intermediate model and coverage criteria are illustrated in the table.

These studies illustrated that the integration of UML statechart diagram in generating the test case for the software development process and the MBT is important. The conclusions from these studies describe that most of them need to translate the UML statechart diagram into other descriptions, such as a graph or a table (intermediate model), which are derived from the test cases. In the present study, an intermediate table was adapted. In addition, several studies focused on the use of DFS as a basis to generate the test paths. However, the present work provides an algorithm to generate the test paths. Furthermore, this review emphasizes the importance of achieving all-state coverage and all-transaction coverage in conducting coverage criteria.

This section have reviewed 24 studies in generating test cases from UML statechart diagram. In most of the techniques, it has been observed that their number of processes ranges from four to six for generating test cases. It has also been perceived that few of the techniques used one supporting diagrams with UML statechart diagram in order to generate test cases. It has been spotted that there are similarities in the processes to generate the test cases using UML sequence, activity, and statechart diagram, like the intermediate table and test path generation. The inconsistency in the process used in generating test cases for the existing works has raised the need to have a framework that will have a complete set of process to generate test cases. This proposed framework is based on the review of all these diagrams.

From the literature survey, it has been analysed that only quarter of the studies have consider using minimization on the generated test cases or sequences, and most of this studies used metaheuristic algorithms to achieve this, however prioritization was not adapted with the use of UML statechart diagram at the time of this study. So far, there is no technique that claims to generate test cases in an optimal way and still, there is a rich space available for researchers to work in this area.

Similarly, out of total number of studies, comparison was most commonly used with 60% of the total reviewed studies used it to evaluate the generated test cases. Mutation testing was also used by 12% of studies; however, this method is a structural testing technique, which uses the structure of the code to guide the testing process. Nevertheless, 12% of studies did not reveal their evaluation methods.

Likewise, from surveying these studies, in Table 2.3 to pin down the most frequently used coverage criteria in test case generation from UML statechart diagram; it shows

that the most commonly used coverage criteria are all-transition coverage (36%), all-state coverage (24%), and all-transition-pair coverage (12%), as illustrated in Figure 2.12.

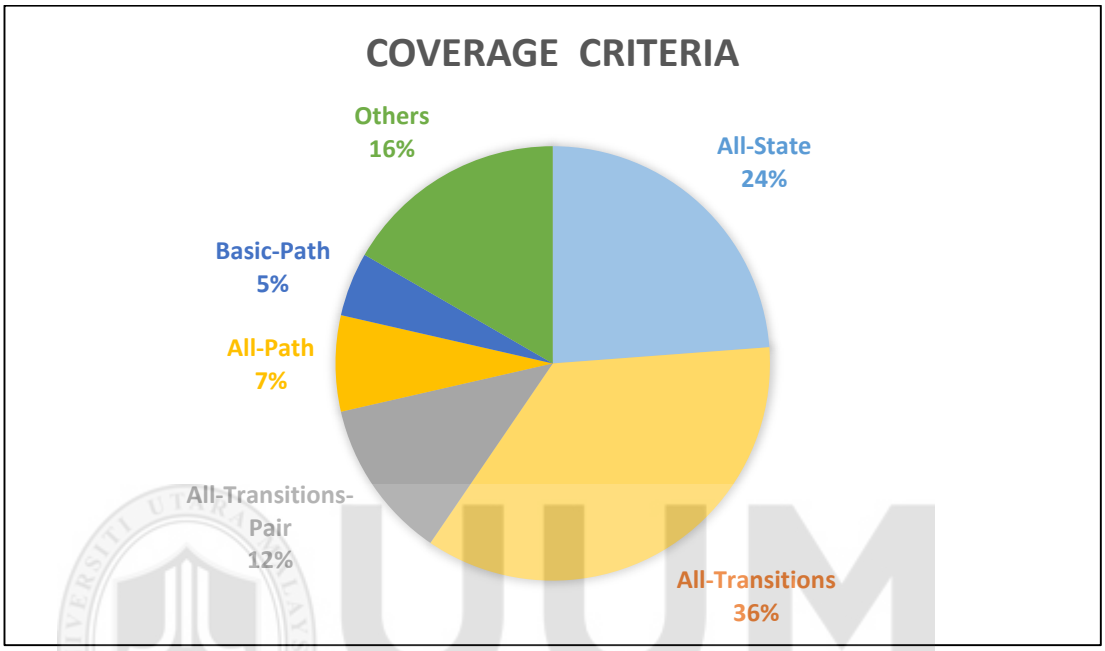

Figure 2.12. Coverage Criteria from Previous Work

To compare the present study with the previous works, five studies have been selected, including those by Ali et al. (2007), Swain et al. (2010a), Swain et al. (2012c), Chimisliu and Wotawa (2013b), and Ali et al. (2014), which are presented in Table 2.3. These studies have been selected according to the inputted diagram, outputted test cases, revealed coverage criteria percentage, and method structure. The full comparison can be viewed in Section 5.2.2 of Chapter 5.

2.8 Test Case Minimization and Prioritization

To reduce redundancy in generated test cases for SUT, two approaches have been explored. The first approach is test case minimization technique that is used to

eliminate the redundant test cases (Omotunde et al., 2016). This technique is used to automatically select a minimum set of test cases used before for testing a complex software product (Xiong, 2011). The second technique is test case prioritization, where the test cases are prioritized chronologically based on their importance and coverage (Omotunde et al., 2016). This method prioritizes and schedules test cases in an appropriate order. Test cases that are having higher priority must be run before than the lower priority test case in order to minimize time, cost and effort during software testing phase (Ghai & Kaur, 2017). In both approaches, the generated test cases should necessarily detect faults in the system while maintaining a good coverage (Omotunde et al., 2016).

The quality of the system is evaluated by executing the test cases. To measure the quality of the generated test cases that contain both important and unimportant test cases, which need to be reduced by using some systematic procedure. Test case generations need to be effective in terms of both time and resources (Sumalatha & Raju, 2014). In the generated test cases, the possibility of redundant test cases needs to be reduced and eliminated, which leads to the process of test case minimization. Test case minimization is also called test case reduction (Hooda & Chhillar, 2014). The purpose of test case minimization is to reduce the number of the test cases using method and technique, while maintaining the coverage criteria (Sumalatha & Raju, 2014).

Minimization procedure is applied to maximize coverage, decrease computational complexity, increase fault detection rate, and minimize running time (Sumalatha & Raju, 2014). Studies were conducted to generate a minimized number of test cases

with the same coverage criteria as the original generated test cases (Ahmed, 2016; Belli & Hollmann, 2008; Srivastava, Baby, & Raghurama, 2009; Srivatsava, Mallikarjun, & Yang, 2013). These studies addressed the test case minimization problem with the advantage of producing test cases that are optimal while considering the coverage criteria.

Generating optimal test sequences and prioritizing the test sequences are still challenging tasks (Panthi & Mohapatra, 2015). According to Tomar and Singh (2016), no complete method is able to find optimal test cases up to the present. However, many researchers used a number of methods to reach optimal possible test cases. The most commonly used methods used by researchers to minimize the number of test cases include ant colony optimization, bee colony optimization, PSO, genetic algorithm, and firefly algorithm (Dubey, Singh, & Singh, 2016; Gulia & Chillar, 2012; Kulkarni, Naveen, Singh, & Srivastava, 2011; Mala, Kamalapriya, Shobana, & Mohan, 2009; Rhmann & Saxena, 2016; Sahoo et al., 2016a). These methods try to generate test data in an automated manner to facilitate the task of software testing (Srivatsava et al., 2013). Therefore, numerous studies have been conducted to minimize the test sequences or test cases (Srividhya & Alagarsamy, 2014).

As shown in Table 2.4, the genetic algorithm is commonly used to minimize the number of test cases. However, the genetic algorithm includes no memorization, delayed convergence, risk of suboptimal solution, and nonlinear optimization (Baudry, Fleurey, Jézéquel, & Le Traon, 2005; Mala, Ruby, & Mohan, 2012). Therefore, a global optimal solution using genetic algorithm has no guarantee of success even when it is reached (Mala & Mohan, 2009). In addition, generating optimized test cases

requires more time compared to other methods (McCaffrey, 2009). Bee colony optimization for test case minimization seemed to work effectively for programs with small sizes. However, as the size of software increases, finding paths and test data becomes more difficult (Lam, Raju, Ch, & Srivastav, 2012) because the bee colony optimization method may be trapped in local search space and the number of iterations is quite high (Srivatsava et al., 2013).

The firefly algorithm, is a new nature-inspired algorithm, it is widely used to solve minimization problems, also results in efficient prioritization of the generated test cases (Choudhary, Gigras, & Rani, 2016; Kwiecień & Filipowicz, 2012; Panthi & Mohapatra, 2015). According to Hashmi, Goel, Goel, and Gupta (2013) the firefly algorithm performed really well in optimizing the results. The firefly algorithm has various advantages like being robust, accurate, and easy to be implemented (Choudhary et al., 2016). In study conducted by Sahoo et al. (2016a), they found that the test cases processed by firefly algorithm in compared with PSO, bat, harmony search, and cuckoo search, reveals optimal result with efficiently in very less time and with more accuracy. Furthermore, compared to the genetic algorithm and PSO techniques, the firefly algorithm reduces the overall computational effort by 86% and 74%, respectively (Panthi & Mohapatra, 2015; Yang & He, 2013). In addition, according to a survey by Kavita, Shilpa, Yogita, Payal, and Akshath (2015), the Meta heuristic approach firefly algorithm has proven to be successful minimization test case generation method. Their results covers each and every vertex of the graph of problem under test. Therefore, this study uses a firefly algorithm to minimize and prioritize test cases.

Table 2.4

Test Case Minimization Methods

Author(s)	Method	Objective
McCaffrey (2009)	Genetic algorithm	Generation of minimal all-pair test cases
Mala and Mohan (2009)	Bee colony optimization	Non-pheromone-based test case optimization
Dahiya, Chhabra, and Kumar (2010)	Bee colony optimization	Automatic generation of structural software tests
Mala and Mohan (2010)	Hybrid genetic algorithm	Test case optimization during the solution generation process by improving the quality of test cases
Suri, Mangal, and Srivastava (2011)	Genetic algorithms and bee colony optimization	Regression test case reduction
Srivastava et al. (2009)	Ant colony optimization	Optimal test path identification
Panthi and Mohapatra (2015)	Firefly algorithm	Prioritization of test sequence generation
Rhmann and Saxena (2016)	Firefly algorithm	Prioritization of generated test paths
Dubey et al. (2016)	Ant colony optimization	Test case optimization for automated testing
SahSahoo et al. (2016a)	Firefly algorithm	Test sequence generating and optimize the generate test sequence

2.8.1 Firefly Algorithm

The firefly algorithm is a bio-inspired metaheuristic algorithm (Panthi & Mohapatra, 2015) that was proposed at Cambridge University by Xin-She Yang; the concept is inspired by the behaviour of fireflies. Approximately 2,000 species compose the firefly

species, and most of them produce rhythmic and short flashes of light. Their generated flashing light may serve as warning signals or an element of courtship rituals (Kwiecień & Filipowicz, 2012; Yang, 2010). The firefly algorithm is inspired by the flash pattern and characteristics of fireflies. This technique is used for solving optimization problems (Rhmann & Saxena, 2016).

Test case generation is demanding and costly. Thus, an effective technique that will minimize redundant generated test cases is needed. Furthermore, for effective testing, the concept of test prioritization is often applied to run the test cases, which may reveal faults earlier in the testing process (Rhmann & Saxena, 2016).

The objective function of the firefly algorithm is based on differences in light intensity of a given optimization problem. Brightness helps fireflies to move toward brighter and more attractive locations and to obtain optimal solutions (Kwiecień & Filipowicz, 2012). The firefly algorithm uses the following idealized rules (Yang, 2010):

1. All fireflies are unisex, so regardless of their sex, one firefly will be attracted to the brightness of other fireflies.
2. Attractiveness is related to the brightness of fireflies. Therefore, for any two fireflies, the less bright one will move toward the brighter one. Attractiveness is relative to distance; brightness decreases as the distance between the fireflies increases. If the two fireflies have the same level of brightness, then one of them will move randomly.
3. The brightness of a firefly is determined or affected by the landscape of the objective function.

Based on these rules, the basic steps of a firefly algorithm can be summarized in the pseudocode shown in Figure 2.13. The two essential components of the firefly algorithm are the formulation of attractiveness of the firefly and the variation of light intensity. For simplicity, this study assumes that the attractiveness of a firefly is determined by its brightness.

```
Begin
   1) Objective function: f(x),  x = (x_1, x_2, ..., x_d);
   2) Generate an initial population of fireflies x_i  (i = 1, 2, ..., n);.
   3) Formulate light intensity I so that it is associated with f(x)
      (for example, for maximization problems, I ∝ f(x) or simply I = f(x);)
   4) Define absorption coefficient γ

   While (t < MaxGeneration)
      for i = 1 : n (all n fireflies)
         for j = 1 : n (n fireflies)
            if (I_j > I_i),
               Vary attractiveness with distance r via exp(−γ r);
               move firefly i towards j;
               Evaluate new solutions and update light intensity;
            end if
         end for j
      end for i
      Rank fireflies and find the current best;
   end while

   Post-processing the results and visualization;

end
```

Figure 2.13. Pseudocode for Firefly Algorithm
Source: Yang and He (2013)

The intensity of light is inversely proportional to the square of the distance, say d, from the source. Thus, the intensity at $I(d)$ varies according to the inverse square law,$I(d) = I_s/d^2$, where I_s is the intensity at the source point. In the simplest form, the brightness on intensity I of a firefly at a particular location x can be chosen as $I(d) \propto$ f(x).

When light passes through a medium with light absorption coefficient of λ, the light intensity I varies with distance, say d, which is given as follows:

$$I(d) = I_0 e^{-\lambda d^2} \tag{2.1}$$

where I_0 is the intensity at the source. The approximation of the Gaussian form in Equation 2.1 is obtained by using the combined effect of inverse square law and absorption, which is given as follows:

$$I(d) = \frac{I_0}{1 + \lambda d^2} \tag{2.2}$$

Similarly, the attractiveness of a firefly can be defined as follows:

$$A(d) = \frac{A_0}{1 + \lambda d^2} \tag{2.3}$$

where $A_0$is the attractiveness at $d = 0$ and $A(d)$ is the attractiveness of the vertex at distance d.

2.8.2 Minimization and Prioritization Methods in Test Case Generation

In this section, a review of the techniques used for an automatic test case generation with test case minimization and/or prioritization is presented.

Srivastava et al. (2009) proposed a technique that used ant colony optimization for path prioritization; the researchers used the directed graph to show the system and presented different paths of the model during the execution. Their method automatically selects the best path sequence that covers the maximum coverage by calculating the strength of each path.

Panthi and Mohapatra (2015) proposed a firefly-optimization-based approach for test sequence generation and prioritization using a composite state in the UML state machine diagram. Using the proposed algorithm, a group of fireflies can effectively explore the UML state machine diagram and automatically generate test sequences to achieve the test adequacy requirement. Redundant exploration of the state diagrams and the iteration over the state loops are avoided through the construction of the feasible control flow graph. The use of the firefly algorithm resulted in the efficient prioritization of the generated test sequences. However, they did not generate the test cases or consider about coverage criterion.

Rhmann and Saxena (2016) proposed a UML-model-based test paths generated from UML activity diagram using the firefly algorithm. Their approach is based on the complexity of different constructs of the UML activity diagram. They used cyclomatic complexity and information flow metric to prioritize generated test paths. Cyclomatic complexity and information flow metric can be calculated from the adjacency metric of the flow graph of the UML activity graph.

Dubey et al. (2016) proposed an optimized test case system for the automated testing using ant colony optimization. To improve the performance of the testing process, they used data mining techniques to reduce the size of the test cases. In their study, a technique called parallel early-binding recursive ant colony optimization system was presented with automated testing to provide an efficient way of software testing.

Sahoo et al. (2016a) proposed the firefly algorithm to generate test sequence using test data and then optimize the generated test sequence. Test data values are selected based on the fitness function. Their work described how the test sequence are generated using

the firefly algorithm and how they are useful in finding the optimal solution to maximize the problem. In their study, they found that the firefly algorithm is more accurate than other methods and the algorithm is able to generate automated test cases with test data efficiently.

The previous studies (Dubey et al., 2016; Panthi & Mohapatra, 2015; Rhmann & Saxena, 2016; Sahoo et al., 2016a; Srivastava et al., 2009) focused on minimization and prioritization for the test sequence, where they only generate the paths and didn't generate the test cases. They used many types of methods and techniques to achieve their objectives. However, these studies provide preliminary data on the test cases as test sequence; also, the coverage criteria of the generated sequences was not taken into consideration. The conclusion from these studies describe that the use of firefly algorithm is the optimal selection for minimization and prioritization of the present study generated test cases.

2.9 Test Case Generation Process and Components

Software developers often use a framework to design their systems. A framework is a concept where the software provides general functionality that can be changed by user code, thus providing application specific software. A framework is therefore a universal and reusable software platform for the efficient development of applications (Waller, Dresselhaus, & Yang, 2013). A framework is a software environment that is designed to simplify application development and system management for a specialized application domain. It is a layered structure indicating what kind of programs can or should be built and how they would connected (Bernstein, 1996). A framework may be for a set of functions within a system and how they interrelate; the

layers of an operating system; the layers of an application subsystem; how communication should be standardized at some level of a network; and so forth. This section presents the studies that explained the processes and important components used in generating test cases to design the test case generation framework.

The components of a test case differ from system to system. However, in its simplest form, the components are a series of events that lead to a certain execution path with certain conditions. The values for attributes and parameters can be generated on the basis of any constraint and then supplied to the program for test execution (Rapos, 2012).

Test case generation has a strong influence on the effectiveness and efficiency of the complete testing process and is one of the most critical knowledge-demanding tasks (Bertolino, 2007; Zhu et al., 1997). Test cases are typically generated from manual or automatic inputs. Manual generation depends on the expertise of the software testers. However, existing methods for the automatic generation of test cases still need to be enhanced and improved (Koong et al., 2012).

The test case generator contains three main phases, which are essential in generating a test case from UML diagrams (Vernotte et al., 2014). These phases are shown in Figure 2.14. The first phase analyses the developed components of the system and delivers the data to the second phase. The second phase investigates the data to determine the appropriate paths; these paths may represent the high coverage criteria. The third phase tests these paths as arguments. The third phase may provide feedback to the second phase regarding any impracticable paths (Edvardsson, 1999).

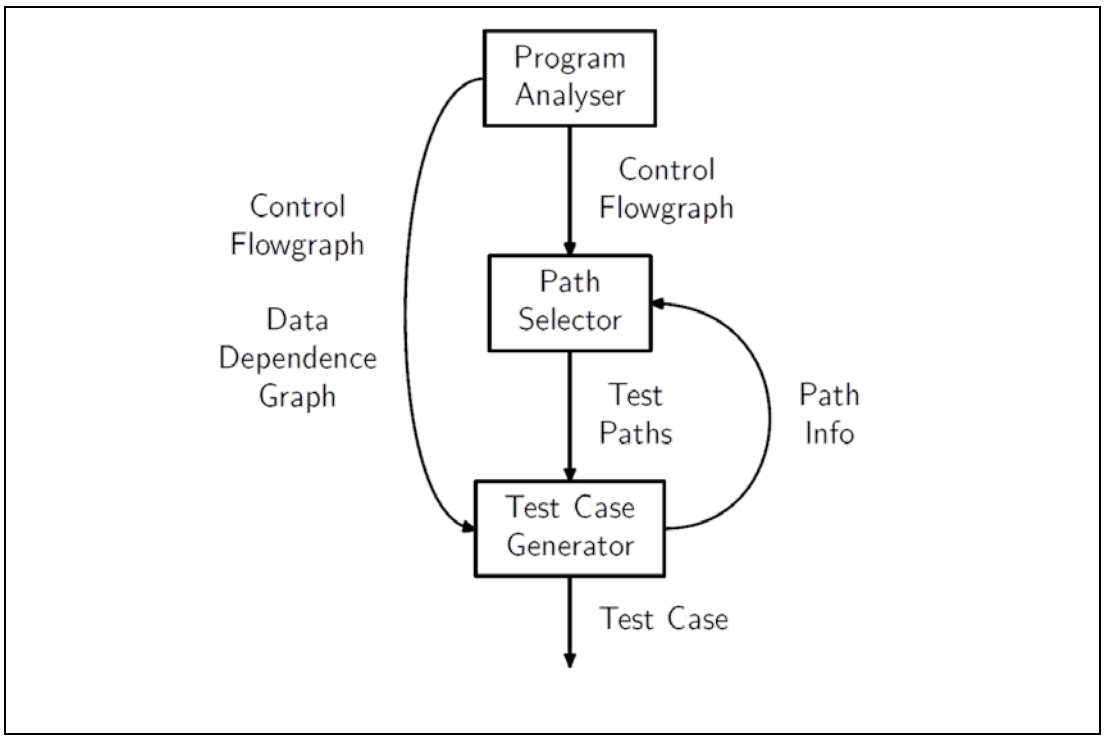

Figure 2.14. Architecture of a Test Case Generator System
Adapted from Edvardsson (1999)

Test cases will be generated with the help of stored strings in the database, which are in the form of tables. In the database, the table of the UML statechart diagram documents each particular message of the UML statechart diagram to generate test cases by extracting the correlating information (Karambir & Kuldeep, 2013). When a class name has been found, it will be entered to the database with the related class attributes along with its operations, attributes, cardinality, dependency, inheritance classes, and every stored string.

According to Boghdady, Badr, Hashem, and Tolba (2011a); Shanthi and Kumar (2012); Verma and Dutta (2014), a reduced form of the stored database (i.e., a dependency table) is needed. This table is generated from the database, created for each UML diagram in any system, and is called a state relationship table (SRT), which covers all the functionalities in the UML diagram. The SRT is then used to

automatically generate a directed graph called state relationship graph (SRG), which is used in conjunction with the SRT as an intermediate model (Boghdady et al., 2011b). The SRG will be used later to generate all possible test paths. Furthermore, the reliability of the intermediate model will increase by conducting a consistency check and entering automatic information.

Converting the UML statechart diagram specifications into an SRT does not generate a blind SRG product of the basic states within each parallel component. The generated machine is the possible combination of configurations based on the simulated events (Santiago et al., 2006). Moreover, one or more arcs can be pruned to avoid generating a large graph (Salman & Hashim, 2017) because it removes unnecessary vertices on the entire graph (Kang, Lee, Lee, Yoon, & Shin, 2015). Pruning reduces the graph needed to be tested and thus minimizes the number of vertices that are processed (Chan & Lim, 2007). Notably, graph pruning has a serious drawback of not testing the entire machine (Santiago et al., 2006). By contrast, a modified version named path pruning ignores only the states with minimum effect on the system, thereby making it applicable to generate test cases for complex systems.

The SRG describes the logic structure of a software module as follows: the vertices represent computational statements or expressions, the edges represent the transfer of control between vertices, and each possible execution path of the module has a corresponding path from the entry to the exit vertex of the graph (Shirole et al., 2011).

Therefore, after formulating all the necessary information, an algorithm is necessary to generate all the possible paths (Hashim & Salman, 2011; Kundu & Samanta, 2009) based on several possible coverage criteria. From the generated paths, a test case

generation algorithm will generate the test case (Swain et al., 2012b; Swain, Mohapatra, & Mall, 2010b).

According to Hooda and Chhillar (2014), the next phases of the life cycle of a test case (as shown in Figure 2.15) are test case generation, test case selection, test case minimization, and test case prioritization. However, Srivatsava et al. (2013) suggested minimizing and prioritizing the test paths before generating test cases because the test paths are more modifiable as the test case generation depends on the data of test paths.

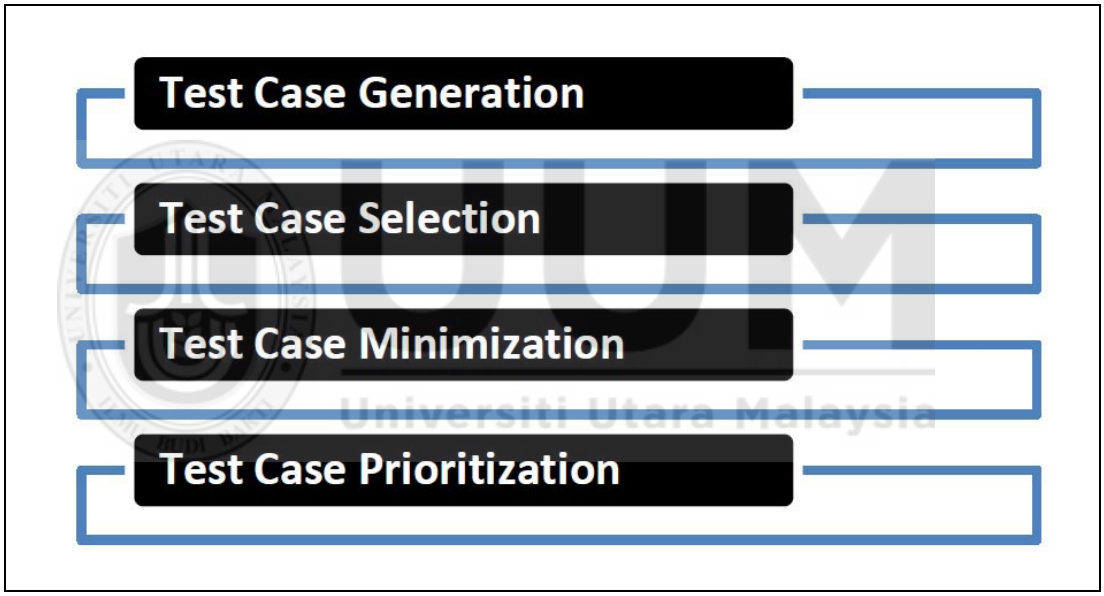

Figure 2.15. Test Case life cycle
Adapted from Hooda and Chhillar (2014)

Test case selection is a method of selecting a subset of test cases from a test suite to reduce the time, cost, and effort in the software testing process. This method is highly similar to the test case minimization technique (Hooda & Chhillar, 2014). Therefore, this study will adapt the test case minimization technique in its framework and select the first state from the prioritization as the best test case.

From the preceding discussion, this study will propose a test case generation framework that combines the preceding processes and components. The proposed framework for test case generation will be composed of eight modules, which include SRT, SRG, consistency checking, test path minimization, test path prioritization, path pruning, test path generation, and test case generation. The framework is shown in Fig. 3.2 in Chapter 3.

2.10 Test Coverage Criteria Selection

This study aims to measure the quality of generated test cases. To measure the quality of a set of test cases, a criterion is necessary (Miller, Padgham, & Thangarajah, 2010). A sequence of conditions that satisfy certain coverage criteria are called test cases (Rhmann & Saxena, 2016). Moreover, coverage criteria are used to evaluate how well a system is exercised by a set of test cases (Fraser & Wotawa, 2007). Therefore, this section considers coverage criteria measurement for the generated test cases.

Coverage criteria (or adequacy criterion) on software systems can be defined as the set of conditions and rules imposing a set of test requirements on a software test (Saifan & Mustafa, 2015). A number of coverage criteria are available for testing, and most of them are based on the information of control and data flows (Hong & Ural, 2004). Test coverage criteria enhance the generation of comprehensive test cases based on the number of elements to cover or visit within a diagram.

Coverage criteria are a popular heuristic means to measure the fault detection capability of test cases (Weißleder, 2010). A test coverage criterion is a crucial factor

in validating and analysing the test adequacy of test cases (Shirole & Kumar, 2013). It can also be used to direct and stop the test case generation processes.

Test coverage specifies the degree of the testing standard such as basis path testing or path testing being achieved. The whole performance from the beginning to the end is represented in a path (Kusumoto, Matukawa, Inoue, Hanabusa, & Maegawa, 2005). Path testing is a testing technique where a set of paths is selected from the domain of all possible paths through the program (Goodubaigari, 2013).

A series of statements, instructions, or high-level design is called a software path. This path begins with a decision, junction, or entry and comes to end at the same or different decision, exit, or junction. Moreover, the path may experience many decisions, processes, and junctions once, twice, or more (Mall, 2009).

The use of the test case generation to drive path testing is thus suitable. The next problem is the testing criteria. The program input domain can be divided into a path by using a suitable test criterion. In addition, the strongest criterion in the path testing family is the path coverage (Goodubaigari, 2013).

The performing path from the beginning to end, which performs any loop only once, is called basis path testing. This path can be identified from other forms of basis path through one-state activity nod or one-edge activity nod (Salman, Hashim, Rejab, Romli, & Mohd, 2017). Basis path testing also refers to the testing of all basis paths. This path testing fulfils the requisites of branch testing, and the independent paths that can be applied to make an arbitrary path will be tested (Jorgensen, 2013). In other words, basis path testing is a combination of branch testing and path testing.

When applying model-based coverage criteria to a model, they can be compared by subsuming them. This subsuming coverage criterion is considered stronger than the individually subsumed coverage criterion. For example, in satisfying the coverage, all transition coverage is considered the minimum coverage criterion. Most of the commercial test generator tools are only able to satisfy slightly weak coverage criteria (Budnik, Subramanyan, & Vieira, 2008). For example, the Smartesting LTD tool is only able to cover all-transition coverage criteria (Weißleder & Sokenou, 2010). Therefore, the only choice for the users of such tools, when they want stronger coverage criteria, is to buy a new test generator tool or create their own. In this study, the test case generation framework was developed in such a way that it cover more coverage by fulfils the impartment coverage criterion. This current methods for supporting the test case generation is now only able to satisfy a limited set of coverage criteria (Weißleder, 2010).

For every test case generation method, certain targeted features need to be specifically tested in the system. The tested targeted features can be specified using test coverage criteria. The full-coverage criteria based on a model will be achieved when the test reaches all model parts at least once (Pahwa & Solanki, 2014).

This section introduces the eight most common transition-based coverage criteria used in MBT test case generation, namely, all-state coverage, all-configuration coverage, all-transition coverage, all-transition-pair coverage, all-loop-free-path coverage, all-one-loop-path coverage, all-round-trip coverage, and all-path coverage (Utting & Legeard, 2010). These criteria are shown in Figure 2.16.

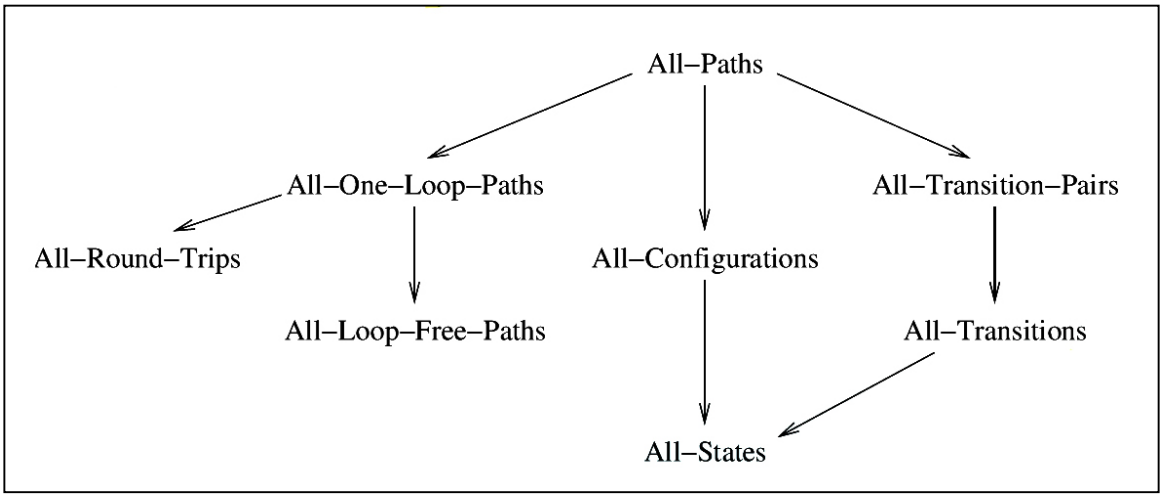

Figure 2.16. Hierarchy of Transition-based Coverage Criteria
Source: Utting and Legeard (2010)

Notably, the all-loop-free-paths, all-one-loop-paths, and all-round-trip coverage criteria can be relatively inadequate because they do not guarantee that all states (let alone all transactions) are covered (Utting & Legeard, 2010). Also from surveying the studies related to automatic test case generation using UML state diagram in Table 2.3; that the most commonly used coverage criteria are all-state coverage, all-transition coverage, and all-transition-pair coverage, as can be seen in Figure 2.12.

Using an extreme example, a UML statechart diagram primarily loops around a self-transition a few times until a counter reaches a particular value, which then enables the transition leading to the rest of the UML statechart diagram (Utting & Legeard, 2007). For this example, the all-loop-free-path criterion can be satisfied with an empty test case, the all-round-trip criterion can be satisfied with only a single test (one loop around the self-transition), and Binder's algorithm for generating an all-round-trip test case can generate tests containing unsatisfiable guards, thereby disabling execution (Utting & Legeard, 2010).

This finding shows that these coverage criteria should be combined with other criteria, such as all-state or all-transition criteria, to ensure that the entire UML statechart is covered (Utting & Legeard, 2007). Utting and Legeard (2010) recommended that all test cases generated from transition-based models satisfy all-transition coverage as a minimum measure of quality. The following are the proposed coverage criteria for the UML statechart diagram:

- **All-state Coverage**

Visiting every model state at least once through a test case is required (Li & Lam, 2005; Utting & Legeard, 2010). This criterion covers all states in every statechart diagram for basic test generation. State coverage is a test adequacy criterion requiring tests to check the output variables of a program. All variables defined when executing a test scope (even those that are invisible, such as private fields of objects) are considered by state coverage (Swain et al., 2012c).

However, the all-state coverage criterion is considered the weakest structural coverage criterion (Devroey et al., 2014); still, few studies adapted this coverage criterion (Chimisliu & Wotawa, 2012; Kansomkeat & Rivepiboon, 2003; Kosindrdecha & Daengdej, 2010; Li & Lam, 2005; Li et al., 2012; Shirole et al., 2011; Swain et al., 2012a, 2012b; Swain et al., 2012c). Therefore, for its importance and wide usage, this coverage criterion is considered in this study.

- **All-transition Coverage**

The transition coverage specifies that each transition must be fired at least once in some test cases (Devroey et al., 2014; Utting & Legeard, 2010). To test a transition, the test case requires that the object under test be in the accepting state of the

transition. The technique does not place any constraint on how to reach the accepting state (Al Dallal & Sorenson, 2006). This coverage criterion is proposed by several authors in generating test cases from statechart diagrams (Ali et al., 2007; Chimisliu & Wotawa, 2012; Chimisliu & Wotawa, 2013a, 2013b; Hartmann et al., 2000; Kansomkeat & Rivepiboon, 2003; Li et al., 2012; Offutt et al., 2003; Santiago et al., 2006; Santiago et al., 2008; Shirole et al., 2011; Swain et al., 2012a, 2012b; Swain et al., 2012c; Swain et al., 2010a). Therefore, this coverage criterion is one of the most commonly used, and this study considers this coverage criterion.

- **All-transition-pair Coverage**

The all-transition-pair coverage considers adjacent transitions successively entering and leaving a given state. This coverage specifies that, for each state, each couple of exiting transition has to be fired at least once (Devroey et al., 2014). Thus, the all-transition-pair coverage includes the all-transition coverage. The all-transition-pair coverage criterion generates more test cases than the all-transition coverage criterion (Blanco, Fanjul, & Tuya, 2010). Given that the all-transition-pair coverage is not widely used by researchers, Briand et al. (2005); Offutt et al. (2003); Santiago et al. (2006) used the all-transition-pair coverage in their studies. For the transition coverage, pairs that are executable by at least one product are considered in the ratio that covers the parallel path (Devroey et al., 2014). Therefore, this study considers this coverage for its importance to the parallel path.

- **All-configuration Coverage**

Visiting every configuration of the UML statechart diagram at least once is required. This coverage criterion and the all-state coverage for systems with no

parallelism are the same (Utting & Legeard, 2010). Thus, for this study, this coverage criterion is not considered.

- **All-one-loop-path Coverage**

All-one-loop-path coverage returns all paths containing one cycle at most; therefore, each generated path contains one and only one repeated state at most (Muniz, Netto, & Maia, 2015). In other words, this condition requires visiting all the loop-free paths through the model, including all paths that loop once (Utting & Legeard, 2007). Muniz et al. (2015) covered all-one-loop-path coverage for MBT but not for UML statechart diagram in their study. The present work considers this coverage because this study focused on loops.

- **All-loop-free-path Coverage**

In the loop-free coverage, every loop path must be traversed at least once. A path that does not contain any type of repetition is called loop-free path (Utting & Legeard, 2010). Notably, this coverage does not frequently cover all transitions. Similarly, this coverage does not constantly cover all states. However, all-one-loop-path test cases include all paths of the all-loop-free-path coverage criterion. Therefore, using all-one-loop-path coverage is sufficient, and the loop-free-path coverage is not considered in the present study.

- **All-round-trip Coverage**

This coverage criterion is similar to the all-one-loop-path criterion because it requires a test for each loop in the model. Furthermore, the test only has to perform one iteration around the loop. Nevertheless, this coverage is weaker than the all-

one-loop-path coverage because all the paths preceding or following a loop do not require testing (Utting & Legeard, 2010). Therefore, all round trips will be overlooked and the all-one-loop-path coverage will be chosen instead. However, Briand et al. (2005) used all-round-trip coverage in their study.

- **All-Path Coverage**

The all-path coverage specifies that each executable path should be followed at least once when executing the abstract test case on it (Devroey et al., 2014). The all-path criterion corresponds to the exhaustive testing of the statechart diagram model (Utting & Legeard, 2010). Few studies consider this coverage in their coverage criteria (Ali et al., 2007; Murthy et al., 2006; Shirole et al., 2011; Swain et al., 2012a) because it is generally impractical, given that such models typically contain an infinite number of paths due to loops (Utting & Legeard, 2010). The present study does not consider this coverage because it focuses on parallel paths and loops.

Based on the preceding coverage criteria, all-state coverage is the weakest coverage, but it still awaits acknowledgement for its importance and comprehensive use. All-transition coverage and all-transition-pair coverage are important in parallel paths because they cover all decision and guard states. These coverage criteria are used by most of the reviewed papers. In all-loop-free-path, all-one-loop-path, and all-round-trip coverage, the use of the all-loop-free-path coverage is efficient by itself, given that the test from it covers all-one-loop-path and all-round-trip coverage. Conversely, all-path coverage is impractical because, in loop cases, this coverage requires an infinite number of paths.

2.11 Summary

This chapter has highlighted the concepts of software testing, MBT, test case generation and its automation and specified UML and statechart diagrams, as well as the advantage of using these diagrams in generating test cases. In addition, coverage criteria are highlighted, as well as the theories that will be used in this study. This chapter also reviewed related literature regarding the test case generation techniques, as well as path sequence minimization and prioritization based on generation from UML diagrams in general and UML statechart diagram in particular. This study shows the possibility of automatically generating test cases using UML statechart diagrams with enhanced coverage criteria.

This literature reviewed the research that has been conducted to automatically and semi-automatically generate the test case using the UML statechart diagram, activity diagram, and sequence diagram.

CHAPTER THREE
RESEARCH METHODOLOGY

3.1 Introduction

The purpose of this chapter is to discuss the research methodology to be used in this study to present the approaches used in conducting this research. First, the research phases are emphasized. Then, each phase is separately discussed in different sections. In Section 3.3.1, the information gathering phase is discussed. Then, the design phase, where the main components of the algorithms and its implementation, as well as a description of the prototype development that are used to implement the framework and how it is integrated, are presented in Section 3.3.2. The evaluation phase is explained in Section 3.3.3, which contains the three stages of evaluation of this study, while Section 3.3.4 provides the conclusion phase.

3.2 Design Research

This study used the design science approach to achieve all objectives as outlined in Chapter 1. The selection of this approach is based on the philosophical foundation of this study, the process involved, and the research outcomes. March and Smith (1995) described design science research as a process that aims to “produce and apply scientific knowledge of tasks or situations to create effective artefacts” to enhance the practice. Furthermore, design research is viewed as an “improvement research” due to its nature in problem solving and performance improvement.

Similarly, March and Smith (1995) emphasized that the design science approach includes two essential activities: building and evaluating. In this approach, building is

"the process of constructing an artefact for a specific purpose" and evaluation is "the process of determining the performance of the artefact." Nevertheless, outcomes such as algorithms, working prototypes, processes, techniques, user interfaces, methodologies, and frameworks can also be considered as valid artefacts under the design research (Norshuhada & Shahizan, 2010).

According to Zelkowitz and Wallace (1998), the research methodologies can be classified into three main categories: observational, historical, and controlled. The observational methods consist of gathering the relevant information during the development of the project. The historical methods gather existing information regarding established projects. On the one hand, the controlled methods are classical methods for the design and experiment used in other technical methods for the statistical validity of the results.

On the other hand, Offermann, Levina, Schönherr, and Bub (2009) highlighted three main phases in designing a research process, which are as follows: problem identification, solution design, and evaluation. Furthermore, Moret and Shapiro (2001) highlighted that the algorithms and methodology of experiments contain theoretical, experimental, and simulation research. However, depending on the research needs, the study may contain one or more characteristics from this component.

These mentioned research methodologies could be generally implemented in design science research. However, a specific methodology may be required in this study. Methodological difficulties of software engineering research have not been resolved yet. Thus, researchers have to create a research approach that is suitable for their problem at hand (Easterbrook, Singer, Storey, & Damian, 2008).

Software testing may combine several different issues, such as humans and tools, and may refer to computer science (Pimenta, 2006); therefore, different research fields are necessary. By combining the mentioned research approaches, the new methodology was generated. Thus, the uncertainty of software development and its technical basis is clearly addressed.

3.3 Phases of Research Methodology

As mentioned, the processes that took place in this study are reflected in the recently proposed research process based on March and Smith (1995); Moret and Shapiro (2001); Offermann et al. (2009); Zelkowitz and Wallace (1998). The outline of the research phases comprises four phases, which include information gathering, development and design, evaluation, and conclusion as shown in Table 3.1. The technique for each phase is further discussed in the subsections.

The research process consists of a sequence of steps; however, they are not always sequentially executed. In this study, the steps often require iterations of processes. The implementation of this process results in the design research artefacts, as presented under the outcome column.

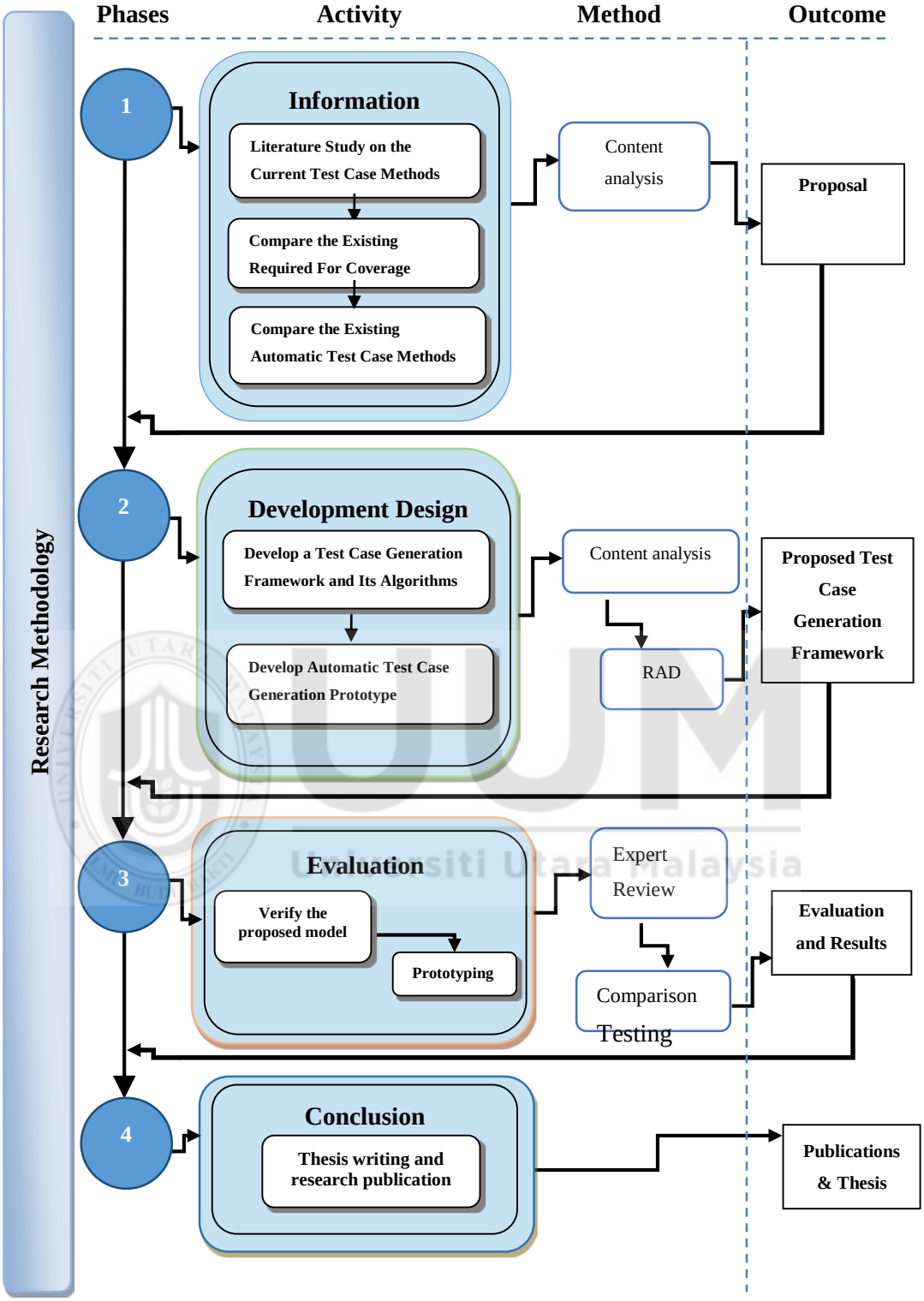

Figure 3.1. Steps of Research Methodology

3.3.1 Phase One: Information Gathering

The initial phase of the research methodology is information gathering. The main activities involved at this stage are synthesizing and collecting information and studying the main topics for defined and considered relevant problems. The outcome of this phase are as follows: understanding the scope and the objectives of this study and identifying the particular problem and the problem statement. At this stage, the test case generation approaches and methods are reviewed, as well as the problems that were faced by the previous researchers. The current test case generation algorithms and the background reviews are presented in the literature review in Chapter 2 of this thesis. Once the proposal is finished, the plan for the entire project is clarified.

This phase is achieved through the following steps: (1) literature study on the current test case methods, (2) comparison of the existing requirements for coverage criteria, and (3) comparison of the existing automatic test case generation algorithms. Each step involved in this phase is described in the following subsections.

a) Literature Study on Current Test Case Methods

The literature study is performed by reviewing the previous works to identify the issues and gaps related to the domain of the study.

Testing is always related to software development process. Thus, the related literature on software testing models, tools, software development methodologies, and techniques were obtained by reading the printed and online references. Among the

references are journals, proceeding papers, standard documentation, books, and unpublished theses.

The existing test case generation methods and models used in different environments were analysed and reviewed in detail to identify their strong and weak points to suggest a test case generation framework. Ideal feature lists were compared to the test case generation methods, models have been studied and their possible up-to-date literature were collected. This list will provide a formal and solid framework for comparing the existing relevant methods.

b) Compare the Existing Coverage Criteria

This phase is designated to compare and analyse the existing current coverage criteria for test case generation algorithms using content analysis.

Content analysis can be described as the process of obtaining efficient knowledge regarding the proposed study, where the information can be attained from many sources of information, including text, audio, video, and other forms of sources (Sharp, Rogers, & Preece, 2007).

The content analysis is conducted by making marginal notes on the sources of the automatic test case generation and marking it when interesting or relevant information is found. Then, the notes in the margins are reviewed, and the different types of information found regarding the different methods and graphs are listed. By reading through the list and categories in such a way where each item offers a description, listing them as major methods or graphs, comparing and contrasting the various major and minor methods or graphs, will lead to identifying whether or not the categories

can be linked in any way. At that time, the methods and graphs have then been collected and examined in detail. In addition, the fitness of the methods and graphs have been considered and their relevance was listed in a table. A review on all of the categories was conducted to establish whether some of the categories could be merged or sub-categorized. Then, the research returns to the original transcripts and ensures that all the information has been categorized.

In this work, the objective of the content analysis was to develop an enhanced algorithm to automatically generate the test case, including the coverage criteria and other related issues. The literature and content analysis has been presented in Chapter 2.

c) Compare the Existing Algorithms

By completing the content analyses from the previous section, this phase compares and analyses the current component of test case generation algorithms through (a) input model, (b) method, (c) intermediate model, and (d) coverage criteria (a), as shown in Tables 2.1, 2.2, and 2.3 (see Chapter 2). The objective of these comparative studies was to explore and compare the existing development algorithms and methods proposed by several researchers and developers in terms of coverage criteria and steps to be accomplished. The analysis was based on achievement and limitations of the methods. These studies also resulted in determining the main components of the automatic test case generating algorithm, as well as the proposed framework to develop the algorithms. The results of these comparative studies has been discussed in Chapter 2.

3.3.2 Phase Two: Development and Design

The first step in building the working system will be at the development stage, and development is a systematic method of research. Therefore, this phase will be designated to (a) develop a test case generation framework and its algorithms, (b) develop its prototype, and (c) calculate the coverage criteria. This stage involves using the output from the gathered information phase to plan a strategy for developing the instruction.

a) Develop a Test Case Generation Framework and Its Algorithms

In this phase, the proposed framework and its algorithms was developed to generate the test cases from UML statechart diagrams. Based on the proposed development method in Figure 3.2, the development targets was achieved by using the following processes and components:

1. Use the UML statechart diagram to define and represent the software development specifications.
2. Automatically construct the SRT by (a) fulfilling the hierarchical relationships based on the influences entered from the UML statechart diagram, (b) automatic checking and storing for existing symmetric ancestor descendent or parent–child hierarchical relations for every pair of states, (c) avoiding the inconsistency problem by the automatic detection for classes relationships based on a set of rules, and (d) automatic deduction of new hierarchical relations (if available).
3. Automatically create the SRG from the SRT using the relation that has been stored in the table.

4. Generate all the possible paths using test case paths generation algorithm from the SRG.
5. Check the generated test paths using consistency checking.
6. Minimize the generated test paths to select the best test paths.
7. Prioritize the minimized test paths.
8. Remove the duplication and unnecessary state from the generated test paths using path pruning to avoid illegitimate test cases.
9. Automatically generate test cases from the pruned generated test paths.

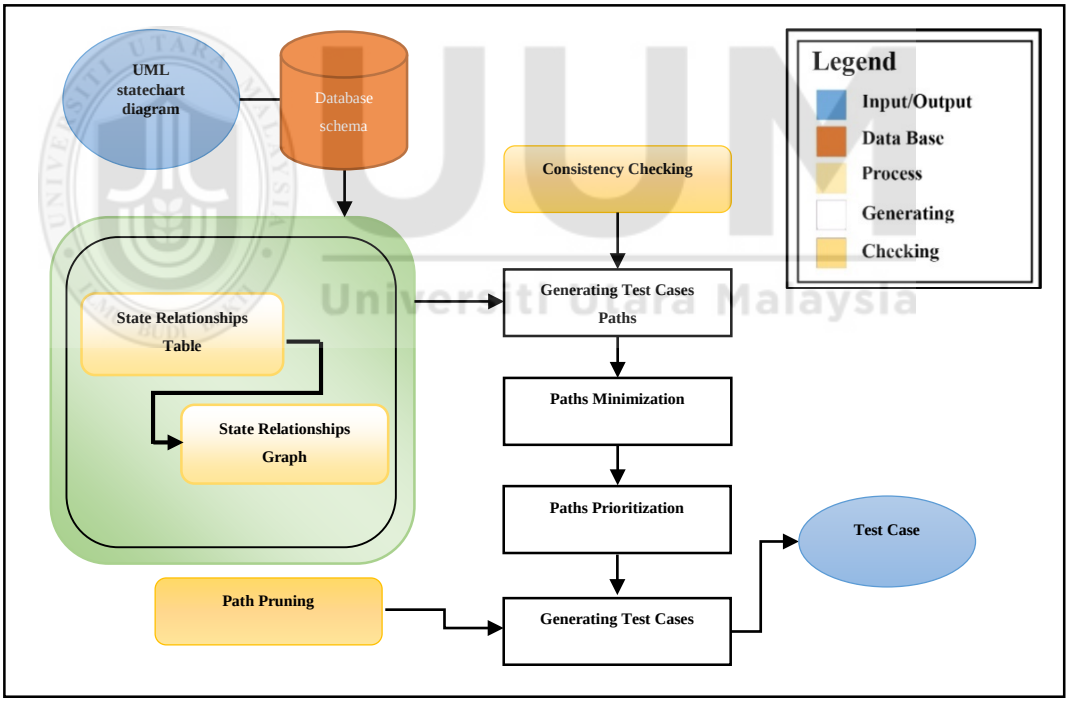

Figure 3.2. The Proposed Development Framework Phases

The framework will have the UML statechart diagram as inputs and the test cases as outputs as the goal of this study is to generate test case using the UML statechart diagram.

b) Coverage Criteria Calculation

In this phase, the selected coverage criteria were calculated using elements coverage equation to be implemented on the generated test cases and will be compared later with previous related studies.

c) Develop the Prototype

In this phase, a prototype was developed to automatically generate the test cases and created based on the proposed framework. After designing the test case generation framework, this research proceeds with the development of the prototype as shown in Figure 3.3. The completed design was transformed into an executable form.

Prototyping is the process of translating systems specification into a physical outcome to gain users' feedback (Dix, 2009). In the prototyping approach, user involvement is at its core. By adopting the approach, the activities involved may improve the understanding of users of the system, along with its information needs and its capabilities.

According to Dix (2009), prototyping has three main approaches: throwaway process (Carmel & Becker, 1995), incremental process (Sprague Jr & Carlson, 1982), evolutionary process (Keen, 1980), aims at building the decision support technologies in a simple step with a feedback from users. The throwaway means that the knowledge gained from the prototype is used in the final design. Incremental prototype refers to the release of the final product as a series of components that have been separately prototyped. By contrast, the evolutionary prototype serves as a basis for the next

iteration of the design. This repetitive process aims to ensure that the development process is properly progressing.

In this study, the prototype was used through evolutionary cycles. By using this type of prototypes, the users will be able to comment on usability, look-and-feel, as well as the flow of the prototypes.

The prototype development was utilized in the rapid application development (RAD) model, which will be designed based on the work of Martin (1991). RAD is characterized as having a substantial advantage from the other models of methodology. One of the advantages of RAD is the possibility of starting early to develop and design the system. The RAD methodology is characterized as flexible, allowing modification of the design even after evaluation. The RAD methodology comprises four phases: requirement planning, user design, construction, and cutover, as illustrated in Figure 3.3.

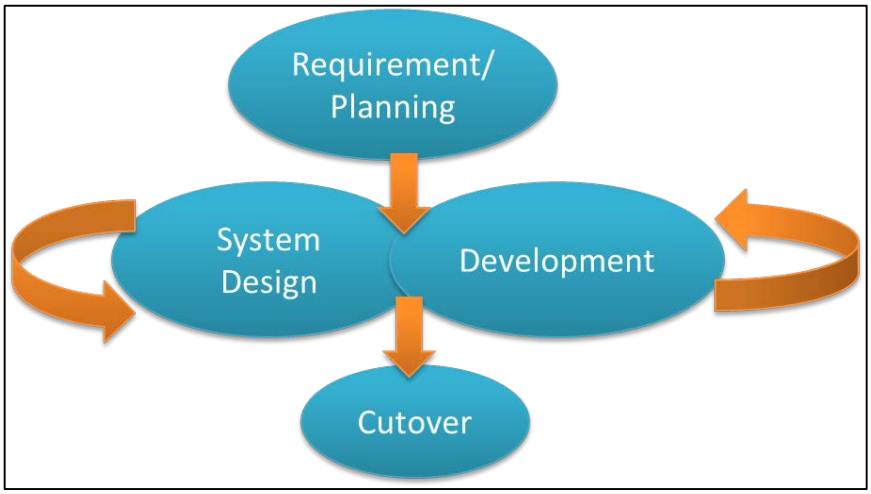

Figure 3.3. Rapid Application Development Model

Through the requirement planning phase, the prototype life cycle of the system combines elements of planning and analysis. In addition, in the user design phase, the users interact with analysts of the systems and developers of the model. Based on the data gathered from the previous studies, the outcome of this phase is identifying the sequence action of the prototype. In the construction phase, the focus is on application development and programming to ensure that tasks are similar to the prototype. The prototype was developed by using PHP language, JavaScript, CSS, and HTML. The design interface of the prototype was conducted and the classes was designed. The design of the database for the prototype was executed using MySQL language. The prototype was evaluated by an expert in software engineering.

3.3.3 Phase Three: Evaluation

Selecting the evaluation technique is a crucial step in all performance evaluation projects (Jain, 1990). Evaluation is a set of research methods and associated methodologies with a distinctive purpose. The first stage of this research evaluation start with prototyping.

Prototyping is considered because it is extensively acknowledged by software developers for early development testing. Apart from the prototyping method to validate the developed algorithm, a comparison with five previous studies has also been conducted. Furthermore, four different UML statechart diagrams are used to show examples of loop and parallel problems. At this stage, an expert review approach was also adapted.

The combination of these evaluation methods will ensure that the final implementation of the automatic test case generation framework represents an approach of software development that has proven benefits in terms of coverage criteria.

a) Prototyping

According to Schwarzl and Peischl (2010a), the implementation of the algorithm in a prototype will reveal more bugs and errors that are absent during simulation and manual testing. Also, Costagliola, Ferrucci, and Francese (2002) revealed that prototyping will help the developer to determine the requirements to develop the expected product. A prototype was developed to validate the proposed algorithm, clearly explain the work, and assist in understanding the techniques for the test case generation (Zhang & Liu, 2013). The prototype was also evaluated using an expert review in the expert review stage. Furthermore, four different UML statechart diagrams were be used. These diagrams were adapted from Inamdar (2015); Lauder and Kent (2001); Popp et al. (2009).

b) Comparison with Previous Test Case Generation Methods

The results of this study were compared with data from previous major and most recent studies of automatic test case generation from UML statechart diagram in terms of coverage criteria, including all states coverage, transitions coverage, transition-pairs coverage, and loop-free paths coverage. Five studies were selected to be compared with Ali et al. (2007), Swain et al. (2010a), Swain et al. (2012c), Chimisliu and Wotawa (2013b), and Ali et al. (2014). These studies were selected according to their similarity in objectives and methods to the objectives of this study. Revealing the

coverage criteria percentage was also critical because the coverage criteria were based on the comparison.

c) Expert Review

Expert review is an essential step in the implementation and development of projects and systems. This activity provides an evaluation of documents, design concepts, and artefacts to meet quality objectives (Garousi, 2010). Based on the demonstrated value of expert reviews in software engineering, numerous industry experts have listed this review at the top of the list of desirable software development practices (Boehm & Basili, 2005). Expert reviews are usually performed by individuals who are not associated with the original design team (O'Neil, 2001).

Expert review in software engineering particularly refers to a type of review in which a creation is examined by one or more experts to evaluate its quality and practical content (Wiegers, 2002a). The purpose behind an expert review in verification is to illustrate a disciplined engineering practice to detect defects and correct them, thereby preventing their occurrence in the functional use of the product or the system (Chrissis, Konrad, & Shrum, 2011). Data collected during the expert review process is used not only to correct defects but also to improve and evaluate the development process itself (Garousi, 2010).

In system development, expert review is recognized as a significant way to improve the quality of the developed software and serves as a complement for testing of other products (Wiegers, 2002b). Therefore, the framework, algorithms, and prototype evaluation in this study is conducted through expert review.

As Shneiderman and Plaisant (2005) stated that different experts tend to find different problems, having between three to five expert reviewers is suggested to be highly productive and sufficient. Also according to Olson (2010) the number of expert reviewers tends to be small, ranging from two or three experts (Holbrook, Krosnick, Moore, & Tourangeau, 2007; Presser & Blair, 1994; Theis, Frood, Nishri, & Marrett, 2002). The evaluation review was conducted in two phases: the first was with an academic expert and the second will be domain experts, who are software testers and developers. This section was conducted with four academicians who have experiences in software testing or/and software engineering domains. Meanwhile, the second expert review process will be conducted with three software developers or/and software testers.

Expert review has also been found to be one of the most effective ways to promote productivity and quality of design processes not only in software engineering but also in other engineering disciplines (Garousi, 2010).

An inspection is the most rigorous and systematic type of peer review. Inspection follows a distinct multistage process with specific roles assigned to individual participants (Wiegers, 2002b). All experts examined the same questions using forms developed by the author.

The procedures for the expert review are arranged in the following manner: (a) setting up the review form based on the selected evaluation attributes, (b) conducting the review, (c) analysing the results, and (d) amending the model and algorithm (Zaibon & Shiratuddin, 2010). Verifying the proposed framework involves the following three activities:

i. Identifying the potential experts

The first task is identifying the characteristics of the selected experts as suggested by Hallowell and Gambatese (2009). The characteristics of these experts include (1) being currently attached to the field of study under examination, (2) being employed in practice in an academic or professional business, (3) having an advanced degree in the field, and (4) having at least five years of professional experience.

ii. The second task is determining the technique or method for conducting the expert review approach.

The framework was verified by ensuring practicality, clarity, and completeness, as well as the correctness of the algorithms. In addition, the effectiveness of the prototype was evaluated, as well as the overall accuracy, usefulness, and usability of the proposed system. Finally, understandability of the documentations was evaluated.

iii. Email and interview approaches were used to contact the experts.

Invitations to become experts for the study were sent through e-mail. The related documents were then sent to the experts who agreed to verify the framework and its processes. They provided feedbacks through in-depth interviews.

The seven experts are sufficient for the purpose of the expert review (Shneiderman & Plaisant, 2005). The following are activities involved during the expert review process (Mohamed, 2015):

1. The researcher conducts a presentation to provide an overview of the study and explain its components, also provide detailed documents for the framework, its algorithms, and the comparison results (from Sections 4.3, 4.4, and 5.2.2).
2. The expert run the prototype, go through the steps, saw the results, and try all its functions.
3. The experts review the framework, algorithms (SRT, TCGP, minimization, prioritization, and TCG), coverage criteria results, and prototype.
4. The experts fill in the verification form and provide their comments.
5. The researcher updates the software processes based on the comments of the experts.

The feedback from the identified experts was collected and analysed to modify and improve the proposed framework. Details are presented in Section 5.2.2.1. The following section describes the instrument design that was used during the expert review.

- **Instrument Design**

Interview questionnaire can be defined as a set of questions that are answered by the respondents whose responses are documented (Sekaran and Bougie, 2010). In evaluating the proposed system, the contents of the instrument from previous works were obtained from different fields, such as general software development, multimedia applications, and project management, which include the works by Al-Tarawneh (2014); Bahrin (2011); Mohamed (2015). Additionally, outcomes from the theoretical study, including Avancena and Nishihara (2015); Joo, Lin, and Lu (2011); Naik and Tripathy (2011); Salah, Paige, and Cairns (2014); Shiratuddin et al. (2013); Vaziri and

Mohsenzadeh (2012) were also applied, as shown in Table 3.1. The choices of the evaluation attribute selection are based on the most appropriate to define the dimensions that are under evaluation. Table 3.1 describes the selected evaluation dimensions.

Two measurement scales, “agree/disagree” with comments/suggestions, are used as semi-structured instrument evaluation tools as employed by Mohamed (2015). Therefore, the feedbacks on the evaluation of the proposed framework are discussed in Section 5.2.2.2. Details on the evaluation measures are presented in Appendix A.

Table 3.1

Construct Descriptions

DIMENSIONS	**DESCRIPTIONS**	**SOURCE**
Practicality	The proposed framework of automatic test case generation from UML diagrams can practically be implemented in the real world.	(Mohamed, 2015)
Clarity	As a whole, the framework is workable and the steps in the framework are easily followed.	(Bahrin, 2011; Mohamed, 2015; Shiratuddin et al., 2013)
Completeness	The essential items of the proposed framework are complete, satisfactory and suitable to generate test cases.	(Naik & Tripathy, 2011; Vaziri & Mohsenzadeh, 2012)
Correctness	The algorithms: State Relationships Table (SRT), Test Cases Paths Generation (TCGP), minimization, prioritization, and Test Cases Generation (TCG), provide correct results and achieve its objectives.	(Naik & Tripathy, 2011)
Effectiveness	The prototype automatically generates the test cases from the UML statechart diagram, for which it is intended.	(Avancena & Nishihara, 2015; Joo et al., 2011)
Accuracy	The system provides correct test case result to the inputted UML statechart diagram.	(Naik & Tripathy, 2011; Salah et al., 2014)

Table 3.1 Continue

DIMENSIONS	DESCRIPTIONS	SOURCE
Perceived Usefulness	The proposed system is useful for the software tester in improving the coverage criteria quality of test case generation.	(Calisir & Calisir, 2004; Mohamed, 2015; Salah et al., 2014)
Usability	Using the proposed system would make generating the test cases easy for the software tester.	(Calisir & Calisir, 2004; Mohamed, 2015; Salah et al., 2014)
Understandability	All documentations are clearly and simply written such that procedures, rules, and algorithms are readable and can be easily understood.	(Naik & Tripathy, 2011; Salah et al., 2014; Vaziri & Mohsenzadeh, 2012) (Mohamed, 2015)

3.3.4 Phase Four: Conclusion

The resulting generated test cases that have been evaluated in the evaluation phase, confirmed the proposed algorithm. Consequently, the goal knowledge of the research was obtained. The conclusion chapter describes and discuss the finding and result of this study, as well as the limitation and possible future expansion of the proposed algorithm.

3.4 Summary

Concisely, this chapter is dedicated to elaborate the processes involved in this study to achieve all objectives. The phases in the methodology include four major phases: information gathering, development and design, evaluation, and conclusion. Each phase is further described in detail in terms of the activities that are involved in the study.

CHAPTER FOUR
ALGORITHMS DEVELOPMENT

4.1 Introduction

In this chapter, the requirements and goals of the proposed work are deliberated. The generating of the test case from different methods is labelled. Furthermore, the objectives for the new algorithm are provided. Therefore, this chapter describes the automatic test case generation algorithms and their implementation, which were developed as part of this thesis.

This chapter is organized as follows. Section 4.2 presents the design goal. The proposed and improved algorithms to generate the test cases are discussed in Section 4.3. Next, the coverage criteria calculation is presented in Section 4.4. The prototype development is presented in Section 4.5. Finally, the chapter is summarized in Section 4.6.

4.2 Design Goal

Software testing is one of the most expensive and time-consuming activities in software development. A well-tested software system will be validated by the customer before being accepted. Practitioners and researchers have attempted to automate the system to increase reliability and reduce the cost of manual testing (Prasanna et al., 2005; Shamsoddin-Motlagh, 2012).

Test cases can be mapped and directly derived from system design. Additionally, when the test cases are generated early, software testers can usually find ambiguities and inconsistencies in the design documents. The cost of developing software systems will

definitely be reduced as errors are eliminated early during the development lifecycle (Prasanna et al., 2005).

Automatic test case generation was proposed to achieve a balance between the quality and amount of test cases because random test case generation does not always ensure the quality of the test case. Moreover, random text case generation mostly does not perform well in terms of coverage criteria (Han & Kwon, 2008). Therefore, the ultimate goal of this study is to increase coverage and reliability of software testing by automating and generating it in the design phase through improving and creating algorithms that automatically generate test cases with the highest coverage criteria.

4.2.1 Parallel Path Problem and Loop Problem

Two problems have prevented researchers from generating test data from UML statechart diagram with 100% coverage: parallel path problems and loop problems (Doungsa-ard, Dahal, Hossain, & Suwannasart, 2008). For example, the path from the initial state to the final state can be easily generated when no loop exists inside the UML statechart diagram. When the loops occur, the number of parallel paths are increased as the number of loops in the path can vary. This condition is called state explosion problem (Schroeder, Kim, Arshem, & Bolaki, 2003).

Loop and parallel path problems are examples of cases which demand large computation time as the test case generation techniques cannot find the test case to explore these parts (Edvardsson, 1999).

In the parallel path problem, one sequence of paths is insufficient to cover every transition in the SUT. Therefore, more than one path is required to cover every transition and state from the initial to the final state. A parallel path refers to every path that starts from the initial state and ends by the final state (Yan, Jiang, & Eynard, 2008). An example of a parallel path in the ATM system is shown in Figure 4.3. The initial to the final state has seven paths, as shown in Figure 4.10.

Loop problem occurs in either loop entry condition, loop terminating condition, increment operation, or decrement operation. When generating test cases using the UML statechart diagram, the main issue encountered by the software tester is the loop problem. For example, when no loop states exist in the UML statechart diagram, the paths from the initial to the final state can be easily generated. The situation in which the transitions are formed as a loop in the UML statechart diagram is known as the loop problem.

With the proposed framework in this thesis, this study has an obligation to cover the loops in the used UML statechart diagram. Currently, the proposed searching cycles or loops in graph techniques do not satisfy the requirement in this study because they could not extract the exact loop but could check whether a graph has loops (Doungsa-ard, 2012).

4.3 Proposed Framework to Generate Test Cases

Test cases are used to detect the software system faults. According to Kundu and Samanta (2009), automatic test case generation is gaining acceptance from software specialists. Advantages of automatic test generation include but are not limited to

reduction of software development time and early detection of faults. This section discusses the overview of the proposed approach to generate a test case from the UML statechart diagram.

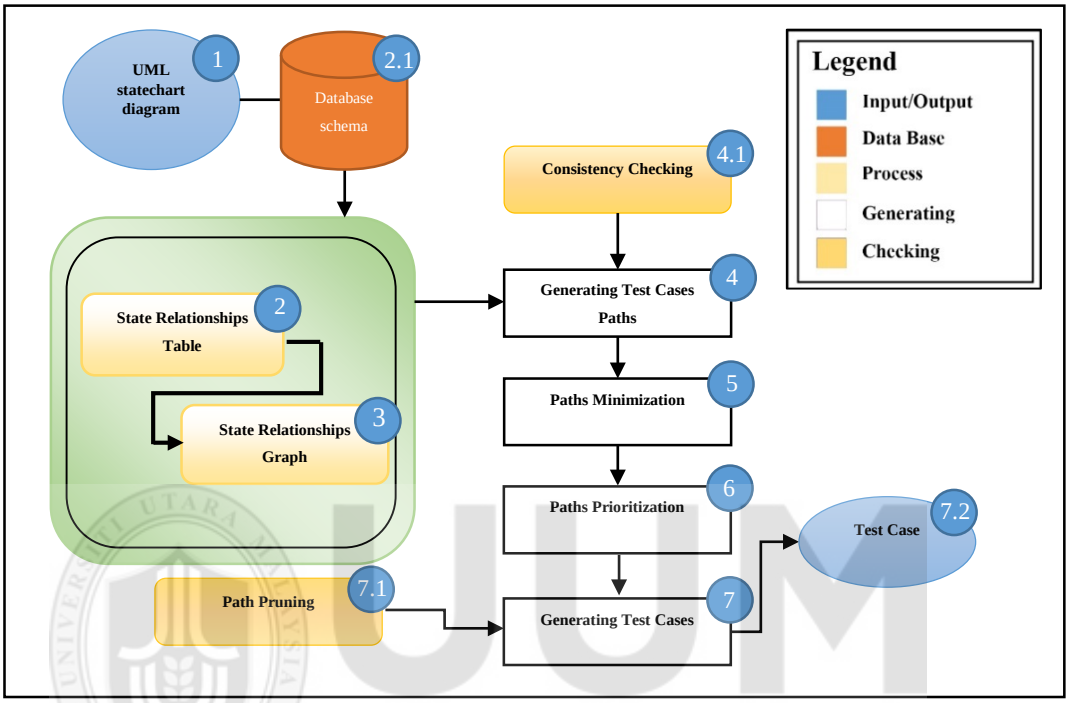

Figure 4.1. Proposed Framework for Automatic Test Case Generation

The proposed framework for test case generation will comprise seven modules: construction of UML statechart diagram, state relationship table, state relationship graph, test case path generation, test case path minimization, test case path prioritization, and test case generation, as shown in Figure 4.1.

The seven steps will be described in detail in the following subsections. In addition, each step will be illustrated with a running example of the UML statechart diagram of the ATM system, as shown in Figure 4.3.

4.3.1 Construction of UML Statechart Diagram

In this research, the UML statechart diagram has been selected to automatically generate test cases because this diagram provides a way to model the behaviour of the system by analysing it in response to input data on how the state of the system changes (D'Souza, Rao, Sharma, & Singh, 2012). However, the process goes through a few steps before generating test cases. This section presents one of these processes to formulate the UML statechart diagram. In accordance with this, the assumption is that the vertices represent states and edges represent the transitions among the states (Aggarwal & Sabharwal, 2012). Edges, which are usually drawn with arrows to indicate their direction, connect to different kinds of vertices in a direction. An outgoing edge from a vertex represents a transaction with an event, wherein an event has a Boolean guard condition associated with it. An action is allied with the edge. The vertices in this study will represent the state, initial state, and final state.

According to Booch (2005), UML statechart diagrams address the dynamic view of a system. These diagrams are especially important in modelling the behaviour of a class, an interface, or a collaboration. These diagrams also highlight the event-ordered behaviour of an object, which is especially useful in modelling reactive systems. A UML statechart diagram consists of five parts: state, transition, event, action, initial state, and final state (Booch, 2005; UML, 2004). These constructs are shown in Figure 4.2.

The UML statechart diagram will be transferred later to a graph from where a graph G will be presented as follows (Voloshin, 2009):

$$G = (V, E) \tag{4.1}$$

where

$V =$ vertex,

$E =$ edge,

$G =$ graph.

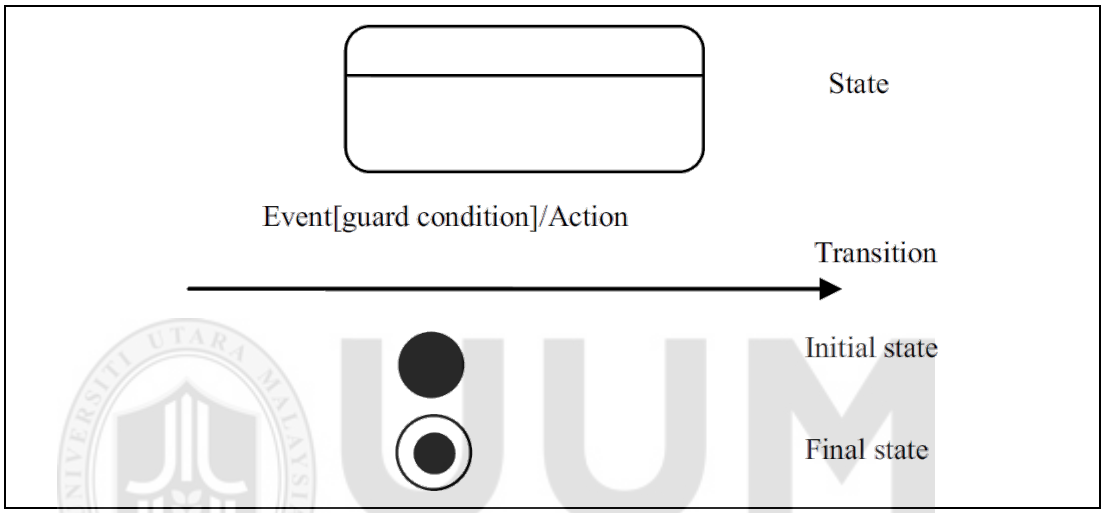

Figure 4.2. Main Constructs Used in UML Statechart Diagram
Source: Aggarwal and Sabharwal (2012)

This graph will comprise a nonempty set of V and set of E (Diestel, 2012). Each edge is a pairing of two vertices. For example, the sets $V = (V_1, V_2, V_3, \dots)$ and $E = \{(V_1 \rightarrow V_2), (V_2 \rightarrow V_3), \dots\}$. Graphs have natural visual representations in which each vertex is represented by a point and each edge by a line connecting two points, as shown in Figure 4.7.

Cyclomatic complexity metric is used to measure the complexity of each diagram used in this study. This complexity metric was selected because it quantitatively measures the logical capability of a program (Oladejo & Ogunbiyi, 2014). The number of basic paths is equal to the cyclomatic complexity of G. A path through a flow graph is a

sequence of edges. This path indicates the flow of control in the corresponding program. From a program with control graph G, basic path (BP) can be calculated by the following formula (Kaner & Fiedler, 2013):

$$BP(G) = E - V + 2 \quad (4.2)$$

where

$BP(G)$ = basic path cyclomatic complexity,

E = number of edges of the graph,

V = number of vertices of the graph.

The existence of loop vertices can significantly increase the number of paths. A few assumptions on the distribution of the inputs can be used to estimate the number of paths in a program in the presence of loops, thereby deriving a few tests to check loops. Furthermore, the graph is considered simple when it does not have loops or parallel edges (Bozeman et al., 2015). This can be attributed to the addition of a loop, which will increase the value of cyclomatic complexity by one and each traversal of the loop body adds a condition to the program, thereby increasing the number of paths by at least one (Jain & Sheikh, 2014). Occasionally, the executed number of loop depends on the input data and cannot be determined before program execution. This finding becomes another cause of difficulty in determining the number of paths in a program. In this study, the loop, as well as the number of decision vertices in it, will be executed once. An example of the UML statechart diagram of the ATM system was adapted from Ali et al. (2014) as shown in Figure 4.3, with five vertices and eight edges except for the loop. The $BP(G) = 8 - 5 + 2 = 5$ is the cyclomatic complexity in Equation 4.2.

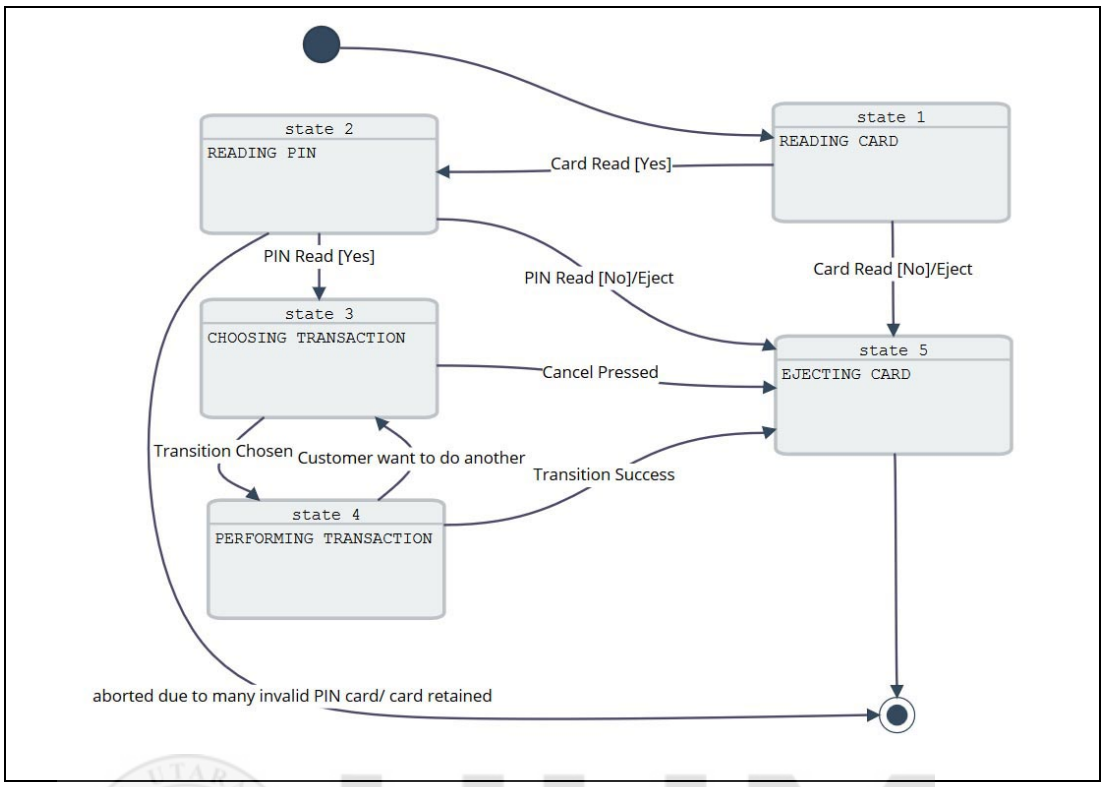

Figure 4.3. UML Statechart Diagram of ATM System

As shown in Figure 4.3, State 1 represents the ATM card reading. If the card read guard condition is Yes, then it will read the PIN code. However, if the card read guard condition is No, then it will eject the card. A similar result is expected in reading the PIN. If the PIN guard condition is Yes, then it will be processed to the selection of a transaction; the card will be ejected if the PIN guard condition is No. However, the card will be retained and aborted if an invalid PIN is entered. The user can select the transaction. Then, the transaction will be performed or cancelled. Finally, the card will be ejected. In performing a transaction, the customer can select between conducting another transaction that results in a loop. Then, the customer finishes the transaction and ejects the card.

- **Decomposing Statechart Diagram into Edges and Vertices**

A transition in the UML statechart diagram is translated into edges in an automaton, with intermediate locations when necessary (Håkansson & Mokrushin, 2004). The UML statechart diagram constraints operation implementations and determines system behaviour and structure (Gogolla, Hamann, Hilken, Sedlmeier, & Nguyen, 2014). Generation of edge structure is based on the edge vector as shown in Equation (4.3). The edge vector is responsible for merging a proper object with its message by assuming that m_i is a message between vertices, E_i as the existing edge, and v_i, v_i' are, respectively, the state vertices immediately before and after message m_i is executed. The source and destination of message m_i are signified by the source and destination vertices. Thus, the UML statechart diagram edges will be presented as follows:

$$E_i(v_i \rightarrow m_i \rightarrow v_i') \tag{4.3}$$

Identified edges are insufficient to establish a coherent testing scenario. These edges are connected by several types of relationships in the UML statechart diagram. These relations must be identified to build an integrated testing scenario graph (Alhroob, 2014).

A set of rules was necessary to generate test cases from the UML statechart diagram. Therefore, mapping constructs for the UML statechart diagram into different types of vertices were proposed. From the most commonly used elements when modelling UML statechart diagram and the newly proposed element, the following constructs are considered as contributors to the structure of UML statechart diagrams (Cruz-Lemus, Maes, Genero, Poels, & Piattini, 2010): *State* (State), *InitialState* (Start State),

FinalState (End State), *Decision* (Decision State), *Loop* (Loop State), *Output* (Decision State associated with End State), *DecisionLoop* (Decision State associated with Loop State), *GuardLoop* (Guard State associated with Loop State), *Guard* (Guard State), and *SimpleState* (simple state). These constructs were enhanced, and a new vertex description table (VDT) was produced as illustrated in Table 4.1.

Table 4.1

Vertex Types Description

No.	Constructs of UML statechart diagram	Vertex type of SRG
1	*State*	Vertex of type *state;* its associated string is state name
2	*InitialState*	Vertex of type *state* without any incoming edge
3	*FinalState*	Vertex of type *state* with no outgoing edge
4	*Decision*	Vertex of type *state* with condition string, which has two or more outgoing edges
5	*Loop*	Vertex of type *state* with one edge performing a loop
6	*DecisionLoop*	Vertex of type *Decision* with one of its edge performing a loop
7	*Guard*	Vertex of type *decision* with a Boolean expression
8	*GuardLoop*	Vertex of type *Guard* with one edge performing a loop
9	*SimpleState*	Vertex of type *state* that has only one outgoing edge and is connected to *finalState*
10	*Output*	Vertex of type *decision* with one edge connected to *finalState*

4.3.2 State Relationships Table

After decomposing and setting rules for the UML statechart diagram, the actual sorting and saving into the database will begin. The works of Ali et al. (2014); Boghdady et al. (2011a); Jena et al. (2014); Shanthi and Kumar (2012); Verma and Dutta (2014) used a relationship table. In this study, an automatically generated relationship table is proposed to set the relationships of states in a systematic way. The relationship table is enhanced to be an SRT to automatically set the relationships between the vertices. The central objective of this module is to automatically generate SRT from the UML statechart diagram using the proposed rules and algorithm. This table aims to show all the necessary information that can be useful for the system and generates the test cases in the final stage. Figure 4.6 shows the pseudocode to capture the relationship of every state of the UML statechart and convert it into SRT, as shown in Table 4.2.

The state table is a method to simplify the large systems in a comprehensive manner. This tabular form is made for convenience to specify states, inputs, transitions, and outputs (Tewari & Misra, 2015). The table follows Equation (4.1) because the SRT is converted to a graph *G*. In *G*, vertices represent states, and edges represent transitions between states (Diestel, 2012).

Other elements, such as d, represent the maximum number of vertices in one graph as this work uses *E* and *V*. Given a vertex set $(V_1, V_2, \dots, V_i)$ connected by $E(i \rightarrow \text{j})$ as $(1 > i, j \geq d)$, their relationships should be determined according to a set of rules, which has been collected and developed for this study. The list of rules that explain the relationship conditions between vertices to enhance the extraction process is shown in Figure 4.4.

Rule 1: The first vertex will be indicated as V_1, and the final vertex will be V_d. The Start vertex has no ancestor, and the End vertex has no descendant (Mathur, 2008).

Rule 2: When a vertex with only one descendant vertex is connected as $E\{(V_i \rightarrow V_j)\}$, V_i and V_j are presented with one edge (Mathur, 2008).

Rule 3: In case of the current vertices (V_i) with NULL destination edge, V_i becomes connected with an edge to V_d as $E\{(V_i \rightarrow V_d)\}$.

Rule 4: If an edge $(V_i, V_j) \in$ E exists, then V_i becomes the predecessor of V_j when $i < j$ and V_i becomes a successor of V_j when $i \geq j$ in their relationship (Mathur, 2008). Also, a new indication for loop is flagged in $type_i$.

Rule 5: Each vertex can be connected to a maximum of two edges $E\{(V_i \rightarrow V_j), (V_i \rightarrow V_{j'})\}$. Therefore, in the case of $E\{(v_i \rightarrow V_j), (V_i \rightarrow V_{j'}), (V_i \rightarrow V_{j''})\}$, or more edges, a new vertex is created for each additional edge. The state information $V_{i'}$ is duplicated from V_i without creating a duplicated path and is then for V_i as $E\{(V_i \rightarrow V_{i'}), (V_i \rightarrow V_{j''})\}$ and for $V_{i'}$ as $E\{(V_{i'} \rightarrow V_j), (V_{i'} \rightarrow V_{j'})\}$.

Figure 4.4. Edges and Vertices Relationship Conditions

In the example of the fifth rule as shown in Figure 4.5 (a), the graph has three vertices (V_1, V_2, V_3). Figure 4.5 (b) shows the conversion of the vertices to $(V_1, V_2, V_{2'}, V_3)$ and the new edges relation to $E\{(V_2 \rightarrow V_{2'}), (V_2 \rightarrow V_1)\}$ and

$\mathrm{E}\{(\mathrm{V}_{2'} \rightarrow \mathrm{V}_3), (\mathrm{V}_{2'} \rightarrow \mathrm{V}_\mathrm{d})\}$ because V_2 has 3 edges, namely, $\mathrm{E}\{(\mathrm{V}_2 \rightarrow \mathrm{V}_1), (\mathrm{V}_2 \rightarrow \mathrm{V}_3), (\mathrm{V}_2 \rightarrow \mathrm{V}_\mathrm{d})\}$.

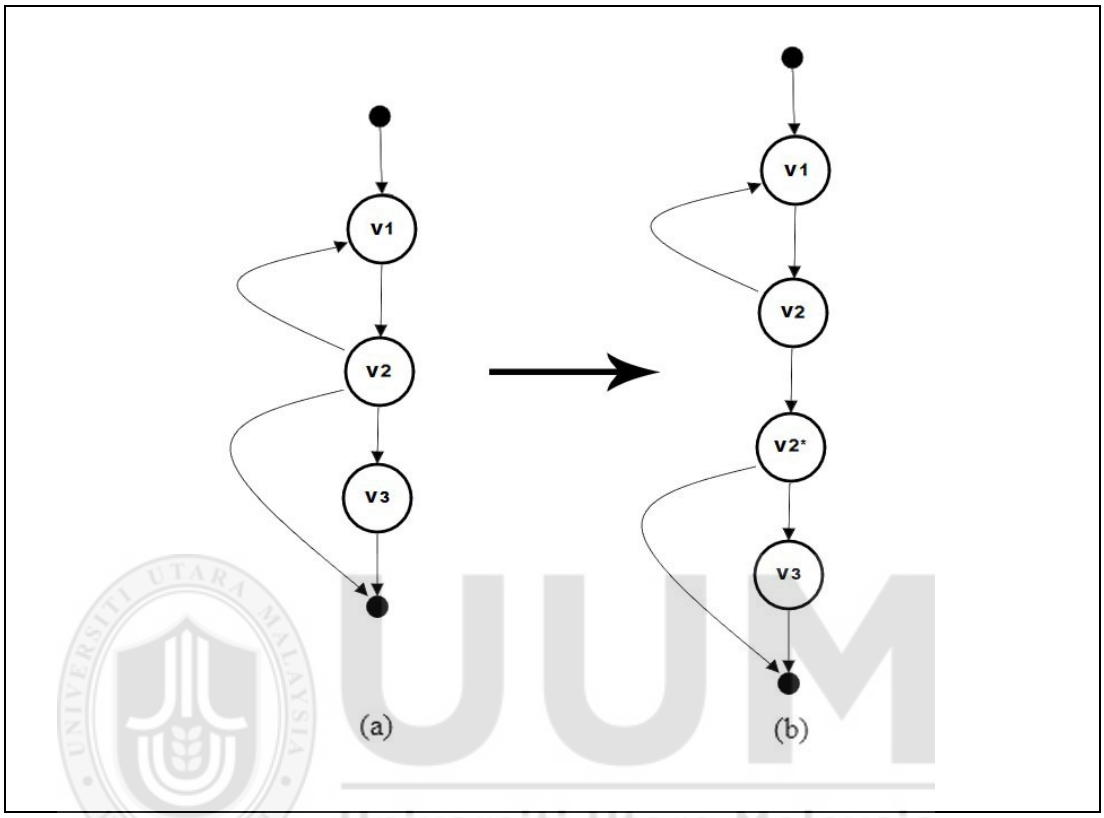

Figure 4.5. Rule 5, Clarification Example

As adapted from Kot (2003), a UML statechart diagram can be a quadruple as shown in Equation (4.4):

$$S_c = (S_s, T, V_\mathrm{a}, S_0) \tag{4.4}$$

where

$S_s =$ a set of simple vertices,

$T =$ a set of edges,

$V_\mathrm{a} =$ a set of variables used in the statechart,

$S_0 =$ an initial state of the statechart.

Each S_s will be an input to the SRT algorithm to automatically generate the SRT and store it in the database.

```
SRT Algorithm
Input: S_c = (S_s, T, Va, S_0)
Output: SRT

1       endVertices ← getLengthOf (Ss);
2       for counter ←1 to endVertices do
3               V[counter] ← vertices value;
4               Va[counter] ← variables value;
5               counterplus ←1;
6               while Transitions[counter] have real number do
7                       if (counterplus=1) then
8                          Vfirst(counter) ←Transitions[counter][counterplus];
9                       and if (counterplus=2) then
10                           Vsecond(counter) ←Transitions[counter][counterplus];
11                      else then
12                              creatNewVertices V'(counter);
13                              Vs(counter) ←V'(counter);
14                              V'first ←Transitions[counter][counterplus-1];
15                              V'second ←Transitions[counter][counterplus];
16                      end if
17                      increase counterplus by 1;
18              end do
19              if (counter=1) then
20                      Type[counter] ← "initialState";
21              end if
22              if (Vfirst[counter] = NULL AND counter ≠ endVertices) then
23                      Vfirst[k] ← endVertices;
24              end if
25              if (counter < Vsecond[counter] and Vsecond[counter] ≠ NULL) then
26                      decisionCounter++;
27                      Vtype[counter] ← "decision";
28              end then
29              if (counter > Vsecond[counter]) then
30                      j ← Vsecond[counter];
31                      set Vtype[counter] to "loop";
32                      for j to endVertices do
33                              if (Vsecond[j] ≠ NULL and Vsecond[j] > j) then
34                                      loopcounter++;
35                              end then
36                              set Vnumber[counter] to loopcounter;
37                  end do
38              end then
39              increase counter by 1;
40      end do
41      Type[counter] ← "finalState";
42      set Vnumber[1] to counter+1;
```

R2 (lines 7–10), R4 (lines 11–15), R1 (lines 19–20), R3 (lines 22–23), R5 (lines 25–38), R1 (lines 40–41)

Figure 4.6. SRT Algorithm

A process starts with an *InitialState*, which goes through a number of transitional states with various edges and ends with the *FinalState*. Initial vertex can be easily detected from SRT as S_0 is the *InitialState* and $\mathrm{E}\{(S_0 \rightarrow \mathrm{V}_1)\}$ is the initial edge. Each state is examined, and its value, type, and connecting edges are determined by applying the rules of SRT relationship conditions.

Table 4.2

State Relationships Table

V_i	V_j	$V_{j'}$	Va_i	m_i	$m_{i'}$	$type_i$	*Vnumber*
S_0	1					*Initial State*	5
1	2	5	Reading card	Card Read [Yes]	Card Read [No]/ Eject	*decision*	
2	2`	5	Reading PIN		Cancel pressed	*decision*	
2`	3	d		PIN Read [Yes]	Aborted due to many invalid PIN cards/Card Retained	*decision*	
3	4	5	Selecting transaction	Transition selected	Cancel pressed	*decision*	
4	5	3	Performing transaction	Transaction Success	Customer wants to do another transaction	*loop*	1
5 d	d		Rejecting card			*Simple State* *Final State*	

Table 4.2 shows the SRT, which contains eight columns: the vertices symbol for each state (V_i), the next two vertices (V_j) and ($V_{j'}$) that the state vertices transition to, the state data (Va_i), the next two events (m_i) and ($m_{i'}$) performed by each state, the vertex type ($type_i$) which differentiates the decision state, the normal state, the loop state, the final state, and the initial state. Finally, the value for the current vertices (*Vnumber*) is

calculated. The modified SRT is created by using the UML statechart diagram as an input, as shown in Figure 4.3. As an example, the first state will have 1 as its value in (V_i), the next two vertices will be 2 in (V_j) and 5 in ($V_{j'}$), the state data "reading card" stored in (Va_i) will transition to 2 "Card Read [Yes]" in (m_i) and 2 "Card Read [No]/ Eject" in ($m_{i'}$). Its type is a "decision" vertex in ($type_i$), and the final (Vnumber) is left blank as this vertex has no necessary value.

4.3.3 State Relationships Graph

This section presents a discussion regarding the proposed intermediate diagram known as an SRG, which has been subjected by most of the previous studies as shown in Table 2.4. SRG captures the information presented in UML statechart diagrams that are stored in SRT and works as the intermediate model to generate the test paths. The proposed SRG is made from vertices and edges as SRG = (V, E). In SRG, vertices represent states and edges represent transitions between states (Diestel, 2012). Without any loss of simplification, the assumption is that a unique vertex that represents the start exists. In addition, one vertex represents the ends. The process of generating SRG uses the SRT as input and goes through the following steps:

Step 1: Place the *InitialState* at the top of the tree as the start vertices.

Step 2: Position the *FinalState* as the root of the tree.

Step 3: If a vertex without any outgoing edge and its type are not a *FinalState*, then connect this vertex to a *FinalState* with an edge.

Step 4: Detect the initial edge after *InitialState* and check if it was previously visited.

Step 5: Obtain the next vertex connected to the current edge.

Step 6: If the current vertex was previously visited but has more than one edge, flag this vertex and revisit again.

Step 7: If the current vertex was previously visited and does not have an unvisited edge, proceed to the next vertex.

Step 8: Repeat steps 6 and 7 until the *FinalState* vertex is reached.

Step 9: If the current vertex type is *FinalState*, then consider the graph finished.

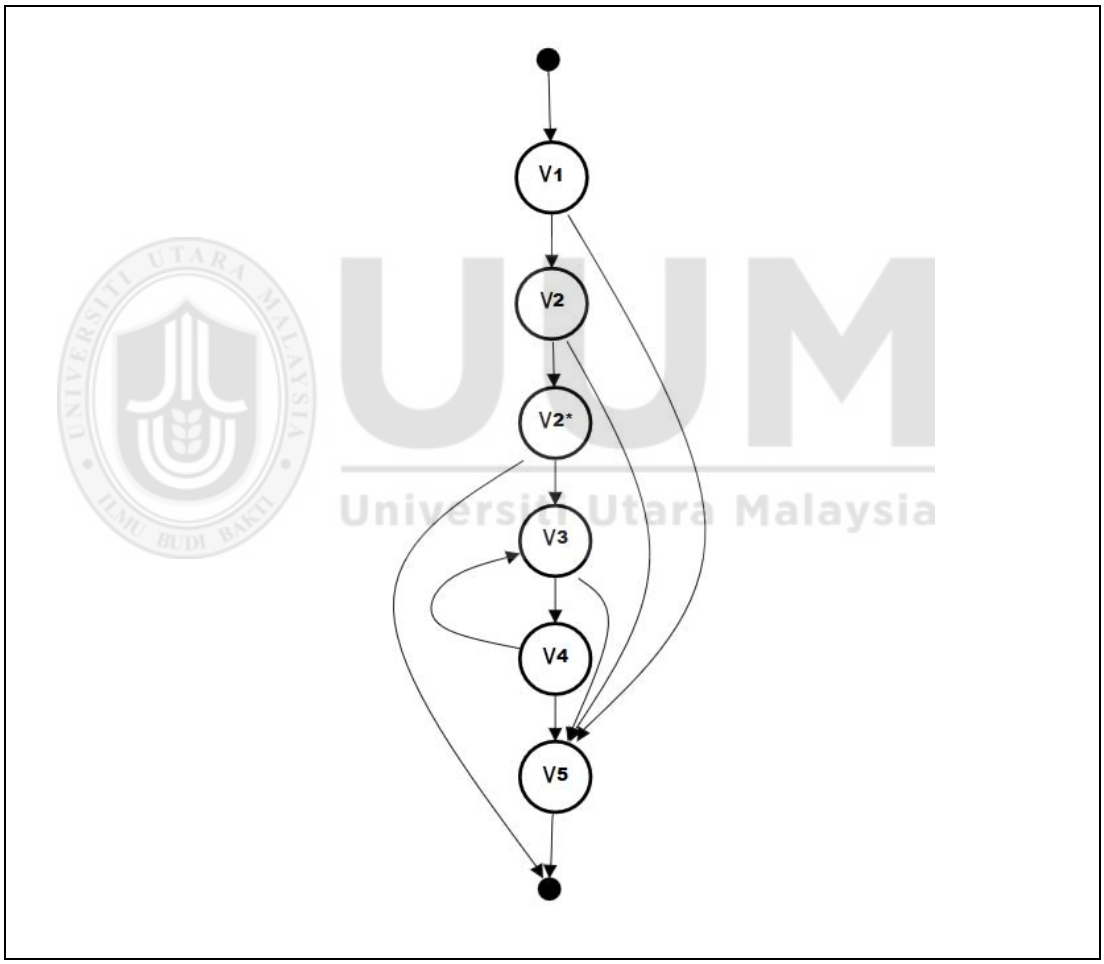

Figure 4.7. State Relationship Graph

The SRT is accomplished to automatically generate the SRG. The value provided for each state in the SRT column (V_i) is used to name the vertices in the SRG, where each vertex represents a state in the UML statechart diagram. A loop on the SRT will be

checked from vertex to vertex to determine the connection between the vertices and edges. Therefore, checking the two preceding vertices (V_j) and ($V_{j'}$) columns in the SRT for the symbol of the current vertex will determine the direction and location of an edge from one vertex to another. Specifically, if it contains the symbol of the previous vertex, then an edge from the previous vertex to the current one is drawn in the SRG. Otherwise, a backtrack search in the SRG is conducted until the vertex whose symbol is mentioned in the column of the current vertex is found. In addition, an edge is created from the SRG to the current vertex until all the rows in the SRT are finished. Synchronization, decisions, and loops are demonstrated using edges as well. An example of SRG is shown in Figure 4.7.

4.3.4 Generating Test Case Paths

In this section, the description of the algorithm to test case generation paths (TCGP) are presented. The first path in a set could be any complete path through G that starts at vertex *Start*, ends at vertex *End*, and does not iterate any loop more than once. The subsequent paths can be derived by changing the outcome of one of the conditions in any of the paths derived so far such that the new path is not identical to any path already derived (Kaner & Fiedler, 2013).

The existing studies used the DFS algorithm as a base graph optimization technique to generate the paths, as shown in Table 2.4. DFS was used to traverse the graph whenever possible. In DFS, edges are explored from the most recently discovered vertex v, which still has unexplored edges leaving it. This process continues until all the reachable vertices from the original source vertex are discovered (Tripathy & Mitra, 2012). However, when this algorithm is applied to the example graph in Figure

4.7, it generates five test paths as shown in Figure 4.8 as the DFS does not fully handle loops, thereby leading to loss of paths (Kim et al., 2007; Mingsong et al., 2006). Therefore, the common DFS algorithm will cause the loss of paths; thus, creating an enhanced DFS algorithm or other algorithms to generate the paths is unnecessary (Mingsong et al., 2006). The modification is performed to generate paths, and the number of times a vertex can be visited depends on the number of decisions and loop vertices present. If two-decision vertices are present, then a vertex is visited twice; if two-loop vertices are present, then the vertex gets a chance of being visited four times.

TP 1: [S → 1 → 2 → 3 → 4 → 5 → E]

TP 2: [S → 1 → 2 → 3 → 5 → E]

TP 3: [S → 1 → 2 → 5 → E]

TP 4: [S → 1 → 2 → E]

TP 5: [S → 1 → 5 → E]

Figure 4.8. All Possible Test Paths Using DFS Algorithm

In this work, all possible test paths are generated using the proposed TCGP algorithm in Figure 4.9, which guarantees visitation of all the UML statechart graph vertices and achieving total path. The TCGP algorithm is applied on the SRG to obtain all the possible test paths.

The test paths that have been generated using the TCGP algorithm cover all the conditions, branches, and loop states. A specific flag has been created to test that every loop has been visited as least once. Therefore, the coverage criteria of the basic paths have been addressed. A test path comprises successive vertices forming a complete path from the start vertex in an SRG to the end vertex. Figure 4.3 represents 12 edges

$E_i(i = E_1, E_2, \dots, E_{12})$ epitomizing a UML statechart diagram to a message $m_j(j = m_1, m_2, \dots, m_{11})$ between two vertices. The algorithm uses TCGP as shown in Figure 4.9 to produce seven test paths as shown in Figure 4.10. Then, the proposed algorithm automatically generates seven test cases.

The proposed improved algorithm of the test paths generation is shown in Figure 4.9. This algorithm will generate all the possible paths from the UML statechart diagram. From this generation, the number of paths will be minimized and prioritized. Then, the test cases will be generated from the minimized and prioritized test paths. The TCGP algorithm will be following these steps to generate the paths:

Step 1: Calculate the basic paths of the graph $BP(G)$ and generate $BP(G)$ empty paths slot starting with *initialState* to prevent the algorithm from going into looping and into an infinite number of paths.

Step 2: Trace the last vertex in the path and follow it to the next vertex V_j. Generate a new vertex until a decision, loop, or *finalState* vertex type is reached. In the case of loop or *finalState*, the path generation will stop and move to the next path. However, in the case of decision, the generated path up to this point will be copied to other empty paths and continue with V_j, of the decision vertex.

Step 3: Repeat Step 2 until all the paths end with loop or *finalState* vertex.

Step 4: Calculate $BP(G)$ from the next vertex V_j of the loop vertex and generate $BP(G)$ empty paths slot starting with *initialState* to generate the remaining loop paths.

Step 5: Repeat steps 2–4 until the entire paths end with *finalState* vertex.

TCGP Algorithm

Input: SRG&SRT
Output: Set of test paths

```
01      Begin:
02      //calculate the basic paths of the state relationship graph
03      Basic paths ← decision vertex + 1;
04      Generate the empty basic paths and set first vertex as S
05      While not reaching the final vertex do
06              //Start the navigating from the first vertex
07              If the current vertex = 0 then
08                      Set V ← 1;
09              End IF
10              //Navigate the vertices and generate new path for every decision type vertex
11              If V_j' is empty then
12                      Set the path next vertex to V_j;
13                      Set current V ← V_j;
14              End IF
15              Else
16                      Duplicate the current path and set it as new path;
17                      Set the path next vertex to V_j;
18                      Set the new path next vertex to V_j';
19                      Set current V ← V_j;
20              End Else
21              If current vertex of type 'loop' OR 'finalState' then exit the loop;
22      End While
23      //calculate the basic paths after the loop vertex
24      Basic paths ← decision vertex after loop+ 1;
25      Generate the empty basic paths and set first vertex as S
26      While there are more row in the path array do
27              Select the last vertex in the current path and set its vertex value to V
28              //Navigate only the loop paths
29              If the current V not of type 'finalState' then
30                      While not reaching the final vertex do
31                              //Navigate the remain vertices after the loop vertex
32                              If V_j' is empty or V_j'<V then
33                                      Set the path next vertex to V_j;
34                                      Set current V ← V_j;
35                              End IF
36                              Else
37                                      Duplicate the current path and set it as new path;
38                                      Set the path next vertex to V_j;
39                                      Set the new path next vertex to V_j';
40                                      Set current V ← V_j;
41                              End Else
42                      End While
43              End If
44      End While
45      End Begin
```

Figure 4.9. TCGP Algorithm

TP 1: [S→1→2→3→4→5→E]

TP 2: [S→1→5→E]

TP 3: [S→1→2→5→E]

TP 4: [S→1→2→E]

TP 5: [S→1→2→3→5→E]

TP 6: [S→1→2→3→4→3→4→5→E]

TP 7: [S→1→2→3→4→3→5→E]

Figure 4.10. All Possible Test Paths Using TCGP Algorithm

- **Consistency Checking**

Cain et al. (2003) introduced the consistency problem and proposed a technique to detect inconsistency relations. All previous attempts to recover this problem was observed to be done by manual adjustment. Although Cain et al. (2003) proposed a technique to detect the inconsistency relations, this technique does not avoid or treat inconsistencies.

Each UML statechart diagram is subjected to basically compute the minimum number of test paths that must be covered to check its capabilities in covering the functionalities of the UML statechart diagram. This complexity technique is used to perform the previous computation (Boghdady, Badr, Hashem, & Tolba, 2012). The number of paths can be measured by the number of decision vertices + 1 (Kumar & Mathew, 2014). The number of paths will depend on the loop location, as well as the decision vertices between the loop vertex and the final state as this study deals with loops.

Therefore, a new equation has been proposed, the total paths (TP) equation, which will calculate the number of paths as shown in Equation (4.5). Then, the TP will be compared with the number of the path generated and its generation will be confirmed.

$$TP(G) = \mathrm{T} + \sum_{n=1}^{L}\left((TL_n + 1) + \sum_{n=1}^{LL}(TL_n + 1)\right) + 1 \quad (4.5)$$

The following are the variables used:

- TP = total paths (value of the minimum number of test paths that must be generated),
- T = total number of vertices that are of *Decision* type,
- L = total number of vertices that are of *Loop* type,
- TL = total number of vertices that are of *Decision* type between the *Loop* type vertex and *Final State* type vertex,
- LL = total number of vertices that are of *Loop* type inside the loop.

The second modified equation will be as follows because this work does not consider coverage for all paths as mentioned in Chapter 2:

$$TP(G) = \mathrm{T} + \sum_{n=1}^{L}(TL_n + 1) + 1 \quad (4.6)$$

Equation (4.6) is applied on the SRG shown in Figure 4.7 and Table 4.2, where $T = 4$ *vertices*, $L = 1$ *vertex, and* $TL = 1$ *vertex*, then

$$TP(SRG) = 4 + \sum_{n=1}^{1}(1 + 1) + 1 = 7$$

The cyclomatic complexity is considered an upper bound for the branch coverage criterion and the predicate/condition coverage criterion (Boghdady et al., 2011b). The

cyclomatic complexity also shows the expected path sequence for SRG. In addition, the low certainty of test paths is seven, which is the same number as that achieved by the TCGP algorithm. Then, this part ensures that the generated test paths pass the consistency checking.

4.3.5 Test Case Path Minimization

In this section, after generating the test case paths, test case minimization was conducted to reduce the generation of the test cases paths numbers while maximizing test coverage and generate an effective size of generated test cases.

Test case minimization starts by assuming each visited or amount of visited edge E_i in a specific path as 1 and 0 for unvisited edge. The generated path was converted to path weight as shown in Table 4.3. The weight of a path is the summation of the weights of the path traversed (Ruohonen, 2013). Therefore, this study proposed Equation 4.7 to calculate weight values W_v to determine each path weight of transactions in the system, as shown in Table 4.3.

$$W_v = \frac{\sum_{i=0}^{R} E_i}{\sum_{i=0}^{n} f_i} \tag{4.7}$$

where R represents the total number of edges and in this example is equal to 12. $f_i = 1$, where n is the number of states. Table 4.3 shows the value of W_v for each single path. As an example, the first path E_i summation is equal to 6 because it visits six edges, and f_i summation equals to 7 because it contains seven different vertices. Therefore, $W_v = \frac{6}{7} = 0.85$.

Table 4.3

Path Weight for Each Path

TP	S→1	1→2	1→5	2→3	2→5	2→E	3→4	3→5	4→5	4→3	5→E	W_v
1	1	1	0	1	0	0	1	0	1	0	1	0.85
2	1	0	1	0	0	0	0	0	0	0	1	075
3	1	1	0	0	1	0	0	0	0	0	1	0.8
4	1	1	0	0	0	1	0	0	0	0	0	075
5	1	1	0	1	0	0	0	1	0	0	1	0.83
6	1	1	0	1	0	0	1	0	1	1	1	0.77
7	1	1	0	1	0	0	1	1	0	1	1	0.87

After generating the path weight, the next step starts by calculating the path coverage for each single path, as shown in Table 4.4. Let the test cases TP be a set of test paths, $TP = (T_1, T_2, T_3, \dots, T_n)$. If one of the TP achieves full coverage, then this test case will be selected. If more than one test path achieves full coverage, the path with lower W_v will be selected. When no test case achieves full coverage, selecting an effective set of test cases that will achieve full coverage by its combination is necessary. Now, this step is presented through an algorithm.

In most cases, one testing path cannot achieve full coverage, as there may be many paths from several decision vertices, as shown in Table 4.4, where the sixth path achieves all-state and all-one-loop-path coverage, but not achieving all-transition and all-transition-pair coverage. An approach has been proposed in this study to select more than one testing path to increase the testing coverage using the firefly algorithm. Then, the selection continues until it reaches full coverage. The selection method for the next best testing path depends on the firefly algorithm in the edges contained in the best testing path. In other words, the next best testing path should contain various edges as possible compared with the best testing path (Alhroob, 2012) with the lowest weight possible. The testing paths, which are eliminated, have the largest similarity degree.

Table 4.4

Coverage Criteria for Each Path

TP No.	All state	All transition	All-transition pairs	All-one-loop paths
1	100%	54%	44%	0%
2	57%	63%	11%	0%
3	71%	27%	22%	0%
4	57%	63%	22%	0%
5	85%	45%	33%	0%
6	100%	63%	55%	100%
7	100%	63%	55%	100%

The path weight (as shown in Table 4.3) and coverage criteria for each path are generated first (as shown in Table 4.4, refer to Section 4.4 for the calculation method). The proposed intermediate graph is converted to an adjacency matrix and then used to generate a guidance matrix for the graph. Adjacency matrix is a two-dimensional matrix that indicates the relationship between vertices and edges (Srivatsava et al., 2013).

Next, the value of each element of the adjacency matrix is specified. If connectivity between nodes i and j is detected, then the elements $a_{ij} = 1$ and $a_{ij} = 0$ otherwise, (Das, 2014), as shown in Table 4.5. The following are the steps in creating an adjacency matrix (Das, 2014):

Step 1: Construct an $n \times n$ null matrix (let it be $Adj(i, j)$).

Step 2: Check whether an edge exists for all vertices.

Step 3: If $E(V_i, V_j)$ = = 1.

$Adj(i, j)$ = 1;

Step 4: Repeat step 3 for all values of *i*.

The adjacency matrix in Table 4.5 was created best on the graph in Figure 4.7 as an example. However, vertices 2 and 2` were combined.

Table 4.5

Adjacency Matrix

States	0	1	2	3	4	5	6
0	0	1	0	0	0	0	0
1	0	0	1	0	0	1	0
2	0	0	0	1	0	1	1
3	0	0	0	0	1	1	0
4	0	0	0	1	0	1	0
5	0	0	0	0	0	0	1
6	0	0	0	0	0	0	0

Then, the created adjacency matrix is used to generate a guidance matrix. A guidance matrix holds guidance factors to probe the fireflies in making decisions at predicate vertices in choosing the path (Srivatsava et al., 2013). The out degree of a vertex is the total number of edges that move out from a vertex, and a vertex with an out degree greater than 1 is defined as a predicate vertex (Srivatsava et al., 2013). It is used for the decision matrix for a given graph. For a firefly at a predicate vertex, the decision to choose a path or not is carried out by referring to the guidance factor in the guidance matrix. It blocks the global view of the domain or graph. The guidance factor GF can be defined as follows (Srivatsava et al., 2013):

$$GF = 10\left(CC_i\big((V - i) - 0.1\big)\right) \tag{4.8}$$

The guidance value for the final state is usually set to 1,000 or any high value. The cyclomatic complexity (CC_i) of the given vertex i can be calculated by the following formula (Kaner & Fiedler, 2013):

$$CC = E - V + 2 \tag{4.9}$$

where E is the number of edges of the graph and V is the number of vertices of the graph.

Fireflies at a predicate vertex use the guidance factor as discussed above to traverse the vertex. Therefore, the brightness can be defined as follows:

$$\text{Brightness function= (1/guidance factor)} \tag{4.10}$$

Thus, a firefly at a predicate vertex follows the guidance factor with a lower value.

In the example in Figure 4.3, the number of vertices is 7, and the number of edges is 11; therefore, the Cyclomatic Complexity equal to 6. However, the Cyclomatic Complexity for each vertex should be obtained (using Equation 4.9) to calculate the guidance value. For example, for the third state, $CC_3 = 4 - 3 + 2 = 3$, and for the same state, $GF_3 = 10\left(3\left((7-4) - 0.1\right)\right) = 117$, as shown in Table 4.6.

Table 4.6

Guidance Value

States	Cyclomatic Complexity (CC)	Guidance value (GF)
0	6	414
1	6	354
2	5	245
3	3	117
4	2	58
5	1	19
6	1,000 [END vertex infinity]	1,000 [finial state]

The guidance matrix (Table 4.7) is only a look-up/decision table of the adjacency matrix with each guidance factor corresponding to every edge. Table 4.7 was created

based on Table 4.5 by multiplying each state value by the guidance value from the same state in Table 4.6.

Table 4.7

Guidance Matrix

States	**0**	**1**	**2**	**3**	**4**	**5**	**6**
0	0	354	0	0	0	0	0
1	0	0	245	0	0	19	0
2	0	0	0	117	0	19	1000
3	0	0	0	0	58	19	0
4	0	0	0	117	0	19	0
5	0	0	0	0	0	0	1000
6	0	0	0	0	0	0	0

Then, the algorithm will generate the first path = [0, 1, 5, 6] by starting from state 0 and searching the lowest value in the row, and in this case, it is 354 which represents state 1. Therefore, the first sequence (0, 1) is created. Then, from state 1, proceed to the next state with the lowest value. In this case, it is 19. Thereafter, create (1, 5). State 5 will end to state 6 to create (5, 6). Then, all the visited states in Table 4.7 [(0, 1), (1, 5), (5, 6)] will be replaced with zero as in Table 4.8. The next execution will generate the rest of the paths until all the states are equal to zero.

Path 2 = [1, 2, 5]

Path 3 = [2, 3, 5]

Path 4 = [2, 6]

Path 5 = [3, 4, 5]

Path 6 = [4, 3]

The fifth path starts with 3, and the sixth path ends with 3. Therefore, they will be combined as [4, 3, 4, 5].

Table 4.8

Guidance Matrix after First Path

States	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	245	0	0	0	0
2	0	0	0	117	0	19	1000
3	0	0	0	0	58	19	0
4	0	0	0	117	0	19	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0

The algorithm will match each optimal path sequence with the test paths in Figure 4.10 to optimize the test cases, and the matched path is chosen. When more than one matched path is present, choose the lowest path weight W_v from Table 4.9 between the selected match paths.

TP 2: [S→1→5→E]

TP 3: [S→1→2→5→E]

TP 5: [S→1→2→3→5→E]

TP 4: [S→1→2→E]

TP 6: [S→1→2→3→4→3→4→5→E]

Figure 4.11. Optimized Test Paths

The highest coverage percentage of a testing path that can cover a system is the best path. However, the highest percentage does not mean the largest number of vertices. Each path has its own coverage, as illustrated in Table 4.4.

This method minimized the number of test paths to five (see Figure 4.11) from the seven test paths, as shown in Figure 4.10, where the first and seventh paths have been

deleted. However, the experiment shows that the minimization method depends on the complexity of the inputted graph, especially on the numbers of the loop in it.

The combination of these five paths leads to achieving all-state coverage, all-transition coverage, all-transition-pair coverage, and all-one-loop coverage, as shown in Table 4.9.

Table 4.9

Coverage Criteria Percentage for Minimized Paths

TP No	All state	All transition	All-transition pairs	All-one-loop paths
2, 3, 5, 4, 6	100%	100%	100%	100%

Figure 4.12 shows the test case minimization from 10 UML statechart diagram examples where the total minimization from the total number is 31%.

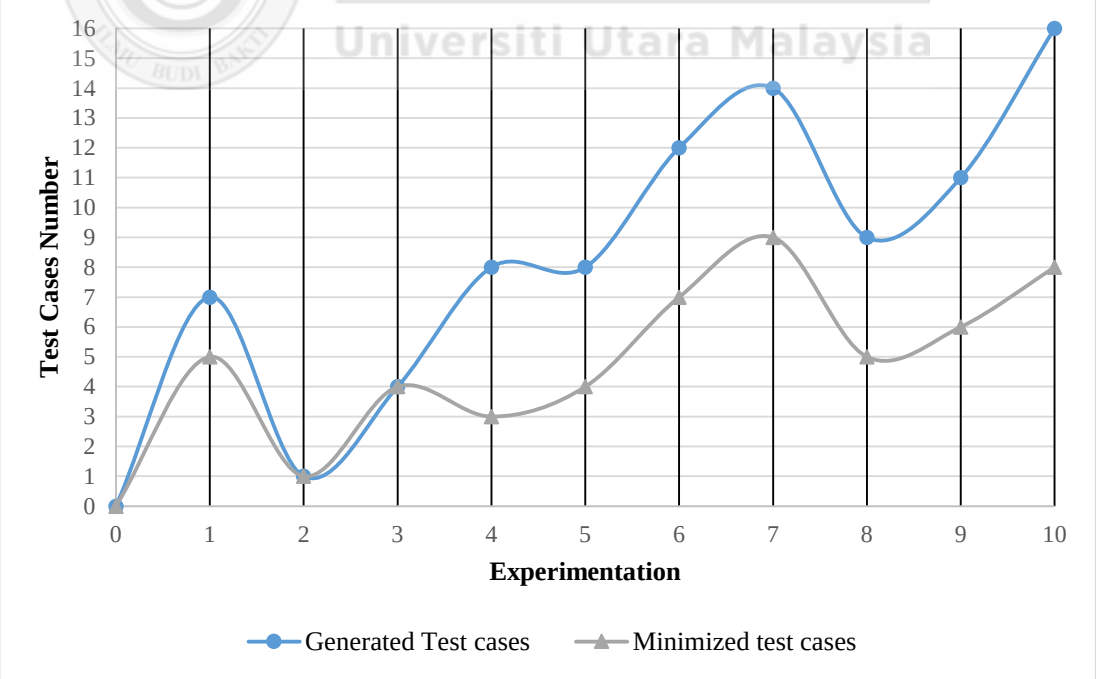

Figure 4.12. Test Case Minimization

4.3.6 Test Case Path Prioritization

Testing depends on fixed resources; thus, path prioritization is needed to schedule the order of test execution (Ahmad & Baharom, 2017). Test path prioritization involves scheduling the test cases systematically to improve the performance of regression testing (Rothermel, Untch, Chu, & Harrold, 2001). Path prioritization means finding the critical paths that a tester might want to test and/or prioritize.

Ten fireflies are generated at each vertex of the state relationship graph of the UML statechart diagram for prioritization of the generated test paths. The brightness of each firefly is determined by the following formula (Yang, 2010):

$$A_i = \frac{A_0}{(1 + \gamma d)} \tag{4.11}$$

where A_0 is the brightness of the firefly at the first vertex and the scaling factor is 100 to maintain the brightness values above zero, to avoid purely random search, and γ is the light absorption coefficient obtained using the following equation (Rhmann & Saxena, 2016):

$$\gamma = CC_i + IF_i \tag{4.12}$$

where CC_i is the cyclomatic complexity at node vertex I and IF_i is the information flow metric (Gries & Schneider, 2005) applied to system design component. The IF value for each vertex is calculated using the following equation (Gries & Schneider, 2005):

$$IF_i = (\mathrm{FANIN}_i \times \mathrm{FANOUT}_i)^2 \tag{4.13}$$

where FANIN_i is the number of edges in vertex *i* and FANOUT_i is the number of edges out from the vertex *i*.

$d_i$is the maximum random distance from the end vertex to that of vertex *i* in which the fireflies are deployed, and vertices at the same level have the same distances (Srivatsava et al., 2013). The maximum random distance value will be from $\big((V - i) - r\big)$, where V is the number of vertices of the graph, i is the vertex, and r is a random number between 0.1 and 1.0. The 10 generated fireflies for each state are shown in Table 4.10, where d_i represents the random distance and $A_i$is the brightness of the firefly.

Table 4.10

Calculation of Brightness Values of 10 Fireflies

V	1		2		3		4		5		6		7		8		9		10	
	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i
0	6.9	2.36	6.8	2.39	6.7	2.43	6.6	2.46	6.5	2.5	6.4	2.54	6.3	2.58	6.2	2.62	6.1	2.66	6	2.7
1	5.9	1.67	5.8	1.69	5.7	1.72	5.6	1.75	5.5	1.79	5.4	1.82	5.3	1.85	5.2	1.89	5.1	1.92	5	1.96
2	4.9	1.44	4.8	1.47	4.7	1.51	4.6	1.53	4.5	1.56	4.4	1.64	4.3	1.63	4.2	1.67	4.1	1.71	4	1.75
3	3.9	1.33	3.8	1.37	3.7	1.41	3.6	1.44	3.5	1.48	3.4	1.52	3.3	1.57	3.2	1.62	3.1	1.67	3	1.72
4	2.9	5.43	2.8	5.62	2.7	5.81	2.6	6.02	2.5	6.25	2.4	6.49	2.3	6.76	2.2	7.04	2.1	7.35	2	7.69
5	1.9	3	1.8	3.16	1.7	3.34	1.6	3.55	1.5	3.77	1.4	4.03	1.3	4.33	1.2	4.67	1.1	5.08	1	5.56

Table 4.11 shows the separate calculation for cyclomatic complexity and information flow for each vertex and the firefly brightness for that specific vertex after including the random factor.

Table 4.11

Objective Function

Vertex	Cyclomatic Complexity CC	Information Flow IF_i	Firefly brightness A_i
0	6	0	2.36
1	6	4	1.82
2	5	9	1.5
3	3	16	1.53
4	2	4	6.49
5	1	16	3.16

According to Srivatsava et al. (2013), the mean firefly brightness from the first to the last vertex in a specific path is the sum of firefly brightness accumulated at the end vertex by the number of fireflies. That is,

$$\text{Arithmetic mean of brightness (AMB)} = \frac{\sum_{i-0}^{n} A_i}{\sum_{i=0}^{n} f_i} \tag{4.14}$$

However, W_v was added to Equation (4.14) to guide the fireflies into selecting the best test path.

The mean of brightness at every path is calculated using the following equation:

$$\text{AMB} = \frac{\sum_{i-0}^{n} A_i}{\sum_{i=0}^{n} f_i} + W_v,$$

$$\text{AMB} = \frac{\sum_{i-0}^{n} A_i}{\sum_{i=0}^{n} f_i} + \frac{\sum_{i=0}^{R} E_i}{\sum_{i=0}^{n} f_i},$$

$$\text{AMB} = \frac{\sum_{i-0}^{n} A_i + \sum_{i=0}^{R} E_i}{\sum_{i=0}^{n} f_i} \tag{4.15}$$

where $f_i = 1$, i is the state/vertex, A_i is the firefly brightness, and n represents the number of vertexes.

Table 4.12

Test Path Prioritization

Test ID	Test path	AMB
TP 6	S→1→2→3→4→3→4→5→E	3.9842
TP 3	S→1→2→5→E	3.2095
TP 2	S→1→5→E	3.1137
TP 5	S→1→2→3→5→E	3.0725
TP 4	S→1→2→E	2.8912

The mean of brightness in every path is calculated using Equation (4.15). Table 4.12 shows prioritization of test paths arranged according to its mean of brightness value for each generated optimized test paths. The test path that will have the highest mean brightness value will have the highest priority and will be tested first (Rhmann & Saxena, 2016). Similarly, other paths will be tested based on their mean of brightness value. From the table, the optimized test path 6 has the highest brightness value, thereby having a high-priority status followed by the third, second, fifth, and fourth paths.

4.3.7 Generating Test Cases

In this stage, details of each symbol in each path are extracted from the SRT and added to its corresponding vertex in the test path to obtain all the final test cases. The test case will be generated automatically using the proposed algorithm.

A test case TC has a triple value (I, S, and O), where I is the data input that acts as a function input to initiate the process, S is the state that indicates the process of retrieving the test data, and O is the expected output of the system (Jena et al., 2014; Mani & Prasanna, 2016; Sharma & PrakashSonwani, 2015). The set of input values for the test case $I\ (m_1, m_2, \dots, m_i)$ is assumed as a set of messages and the state steps

when the method is executed $S\ (v_1, v_2, \dots, v_{i-1})$ for vertices except the last vertex because it will be the expected output resultant values in object $O(v_i)$. Therefore, the equation will be represented in a test case as follows:

$$TC = [I(m_1, m_2, \dots, m_i), S(v_1, v_2, \dots, v_{i-1}), O(v_i)] \quad (4.16)$$

As a part of the test case generation, obtaining the necessary values of the three components of a test case will be from the TCGP itself. For this case, SRT and VDT will be used in constructing test cases.

a) Path Pruning

The first part of the proposed algorithm for generating test cases is called pruning. It is used to prune the generated test paths before generating the final test cases. By using the pruning, some redundant test cases can be reduced as a test case set which meets the test adequacy criteria (Mingsong et al., 2006). Chen, Poon, and Tse (1999) proposed an algorithm to improve the value of SRG. They observed that the algorithm for removing duplicates has many limitations. To overcome these limitations, a method has been proposed for pruning the vertices and improving the value of the test cases.

This method reduces the size of the test case itself by selecting only the vital information from the SRT using the proposed vertex types as shown in Table 4.1. The generated test case is smaller in content but at the same time functional to perform the testing. Path pruning illustrates the detection and deletion of unwanted vertices with the help of steps as shown in Figure 4.13. However, this process will lead to lowering of the state coverage because this method aims to reduce the number of states.

- When the type of the current $\boldsymbol{v_i}$ of TCGP is either *Start State* or *End State*, it will be ignored.
- When the type of the current $\boldsymbol{v_i}$ of TCGP is *State* and the previous vertex is not of type *Decision, Start State, Guard, Loop, GuardLoop,* or *DecisionLoop,* it will be ignored.
- When the type of the current $\boldsymbol{v_i}$ of TCGP is *Guard* or *GuardLoop*, then $\boldsymbol{v_i}$ information is ignored and only its edge information $\boldsymbol{m_i}$ is considered.
- When the type of the current $\boldsymbol{v_i}$ of TCGP is *Output,* then add $\boldsymbol{v_i}$to *State* and insert $\boldsymbol{v_i} \rightarrow \boldsymbol{m_i} \rightarrow \boldsymbol{v_d}$ into *Expected Output*. End the test case.
- When the type of the current $\boldsymbol{v_i}$of TCGP is *Simple State,* add $\boldsymbol{v_i}$ to *Expected Output*. End the test case.

Figure 4.13. Path Pruning Steps

b) Produced Test Cases

The final part is generating the test cases using the test cases generation (TCG) algorithm as shown in Figure 4.14, which will take the TCGP output and use it with pruning as an input to generate all possible test cases that achieve the proposed coverage criteria. The algorithm starts by taking all paths, and then traces each path vertex from start to end, and finally uses each vertex saved information in SRT to apply the information position in the requested test case.

TCG Algorithm

Input: UML start charts path from TCGP algorithm
Output: Set of *test cases*

```
01      Begin:
02      Navigate each path from the start to final vertex;
03      Set current path number into TestCase[Test Case No];
04      While there are more row in the path array do
05              If current vertex not of type 'finalState,' then
06                      If the next vertex is the path = v1, then
07                              Set m_i into TestCase[Input];
08                      End If
09                      Else If the current vertex not of type 'output,' then
10                              Set m_i, into TestCase[Input];
11                      End If
12              End If
13      End While
14      While there are more row in the path array do
15      If the current vertex not of type 'finalState' & 'simpleState' & 'guard' & 'output,' then
16                      Set Va_i into TestCase[state];
17              End If
18              If the current vertex of type 'output' and the next vertex of type 'finalState,' then
19                      Set Va_i into TestCase[state];
20              End If
21      End While
22      While there are more row in the path array do
23              If the current vertex of 'simpleState,' then
24                      Set Va_i into TestCase[expected output];
25              End If
26              If the current vertex of type 'output' and the next vertex of type 'finalState,' then
27                      Set m_i, into TestCase[expected output];
28              End If
29      End While
30      End Begin
```

Figure 4.14. TCG Algorithm

Table 4.13 shows the final expected test case output. The table contains four columns, namely, current test case number, its input, state, and output. These columns reflect Equation (4.16).

Table 4.13

Generated Test Cases

TC No.	Input	State	Expected output
1	Card read [yes], PIN read [yes], Transition chosen, customer want to do another, transition chosen, transition success	Choosing transaction, Performing transaction, Choosing transaction, Performing transaction	Ejecting card
2	Card read [yes], PIN read [no]		Ejecting card
3	Card read [no]		Ejecting card
4	Card read [yes]	Reading PIN	Aborted due to many invalid PIN card /card Retained
5	Card read [yes], PIN read [yes], Cancel Pressed	Choosing transaction	Ejecting card

4.4 Coverage Criteria Calculation

A coverage criterion can be measured on any program during software development, for example, design models, requirements, or source coded. The coverage criterion is satisfied when a test case fulfils a set of test requirements in terms of structural elements. Coverage is usually counted as the percentage of test requirement satisfaction. The coverage criteria assess the quality and completeness of the test cases. Coverage criteria are resulting from popular heuristics to measure the fault detection capability of test cases (Shirole & Kumar, 2013). Clearly specifying the coverage criteria is important because they are frequently used to measure the effectiveness of test case generation (Ali, Briand, Hemmati, & Panesar-Walawege, 2010). The percentage of criteria coverage is used to evaluate the accuracy or quality of test case

generation approaches. The calculation formula for the percentage of coverage criteria is depicted in Equation 4.17. The following formula indicates the number of elements contained in the UML diagram, which are exercised in the generated test cases (Oluwagbemi & Asmuni, 2015):

$$E_c = \left(\frac{E_{tcs}}{E_{tcUML}} \times 100\right) \tag{4.17}$$

where E_c indicates the elements' coverage, E_{tcs} denotes the number of elements exercised in the test cases, and E_{tcUML} refers to the number of elements in the UML diagram.

The following subsections discuss the calculation of the used coverage criteria.

4.4.1 All-State Coverage

Full coverage can be achieved when every state of the UML statechart diagram is visited at least once and by applying all-state coverage to the test model. Through the sets $V_i = (V_1, V_2, V_3, \dots)$ and because the total number of $V_t = 5$ without the *'Start State'* and *'End State'* in the example, every V_i in the graph should be covered at least once, as shown in Figure 4.3. The all-state coverage percentage of the all-state coverage C_{AS} can be achieved by devising the visited vertex V_v on the total V_t the total coverage:

$$C_{AS} = \left(\frac{V_v}{V_t} \times 100\right) \tag{4.18}$$

The proposed test cases achieve $V_v = 5$ vertices; therefore , $C_{AS} = \left(\frac{5}{5} \times 100\right) = 100\%$.

4.4.2 All-transition Coverage

Full coverage is achieved when the test cases visit every transition of the UML statechart diagram at least once and by applying all-transition coverage to the test model. Each transition has a pre-vertex and a post-vertex (Paul & Jeff, 2008). Then, assume all-transitions = AT so that $AT \in E$ and all-transitions coverage C_{AT}. In Figure 4.3, given that E=11 in the example, the following E should be covered at least once:

$E_1(V_0 \rightarrow V_1)$
$E_2(V_1 \rightarrow V_2)$
$E_3(V_1 \rightarrow V_5)$
$E_4(V_2 \rightarrow V_3)$
$E_5(V_2 \rightarrow V_5)$
$E_6(V_2 \rightarrow V_d)$
$E_7(V_3 \rightarrow V_4)$
$E_8(V_3 \rightarrow V_5)$
$E_9(V_4 \rightarrow V_5)$
$E_{10}(V_4 \rightarrow V_3)$
$E_{11}(V_5 \rightarrow V_d)$

Each E has Boolean flags (0) and (1), and its total is E_d. The total coverage is calculated as follows:

$$C_{AT} = \left(\frac{E_d}{AT} \times 100\right) \quad (4.19)$$

4.4.3 All-transition-pair Coverage

The test cases should visit each pair of existing transitions of the UML statechart diagram at least once to obtain a full all-transition-pair coverage for the test model.

Then, assume all-transition-pair coverage = C_{AP} so that $C_{\mathrm{AP}} \in E$. In Figure 4.3, given that $V_{decision} = 4$ in the example, the following $V_{decision}$ should be covered at least once:

$V_{\mathrm{d1}}[(V_1 \rightarrow V_2), (V_1 \rightarrow V_5)]$
$V_{\mathrm{d2}}[(V_2 \rightarrow V_3), (V_2 \rightarrow V_5), (V_2 \rightarrow V_{\mathrm{d}})]$
$V_{\mathrm{d3}}[(V_3 \rightarrow V_4), (V_3 \rightarrow V_5)]$
$V_{\mathrm{d4}}[(V_4 \rightarrow V_3), (V_4 \rightarrow V_5)]$.

Each $V_{decision}$ has Boolean flags (0) and (1), and its total is V_{dt}. The total coverage is obtained using the following equation:

$$C_{\mathrm{AP}} = \left(\frac{V_{dt}}{V_{decision}} \times 100\right) \tag{4.20}$$

The proposed test cases achieve all the 11 transitions pairs; therefore , $C_{\mathrm{AP}} = \left(\frac{11}{11} \times 100\right) = 100\%$.

4.4.4 All-one-loop-path Coverage

Full coverage can be achieved when the generated test paths from the UML statechart diagram visited every loop plus all the paths that loop once by applying all-one-loop-path coverage to the test model.

$$E_{AOLP} = \left(\frac{LT}{TP} \times 100\right) \tag{4.21}$$

where E_{AOLP} refers to the all-one-loop-path coverage and LT is the total number of generated loop test cases. All the paths are required to precede or follow a loop to be tested; thus, $TP = loop \times (included\ desion + 1) = 1(1 + 1) = 2$.

For example, in Figure 4.3, an all-one-loop-path test case would include the two path tests of the all-loop-free-path coverage criterion (TP6 and TP7); therefore, it achieves full coverage because $E_{AOLP} = \left(\frac{2}{2} \times 100\right) = 100\%$.

4.5 Prototype Development

The design and development phases of the prototype applied in the framework are proposed in this study. Each phase in the framework has components embedded in the prototype development.

Nowadays, many software developers are adopting iterative development methodologies highlighted by RAD cycles. In iterative life cycles, testing is conducted at many stages of development unlike waterfall development life cycles, where testing is done at the end of the project. Identifying the flaws early is an advantage to reduce the cost and time of system development. RAD has been proven a valuable software strategy (Konstantinou, 2013).

As stated by Martin (1991), "RAD is a development life cycle designed to give much faster development and higher-quality results than those achieved with the traditional life cycle." In general, software is allowed by the RAD development life cycle to be written much faster, and the requirements are in turn allowed to be changed much easier (Beynon-Davies, Carne, Mackay, & Tudhope, 1999; Martin, 1991; Ooi, Shahrizal, Noordin, Nurulain, & Norhan, 2014).

Using the RAD methodology, the design and development of the prototype were analysed to implement the suggested test case generation module and algorithm. Four stages are included into the RAD methodology, namely, requirement planning, user design, construction, and cutover stages, as shown in Figure 3.3.

In the requirement planning stage, the combined elements of the prototype that are related to the system development life cycle must be obviously understood. The test

case generator takes the UML statechart diagrams as an input and produces a set of test cases based on analysis and algorithms. At the same time, it identifies and generates test paths based on evaluated results. The primary objective of the prototype is to automatically generate a test case. Each phase in the framework has components embedded in the prototype development. In general, the prototype is generated in four stages: statechart, intermediate graph, generated paths, and generated test cases.

In the user design stage, users interact with developed models and system analysts. The target users were defined as software tester, software developer, and programmer who can use the prototype in their testing. The prototype can be accessed through a Web-based application to be more accessible. In addition, the user has the capability to draw the UML statechart diagram and automatically generate the test cases while displaying the steps.

In the construction stage, application and program development are the focus. The prototype system is developed in JavaScript, PHP, and MYSQL. Initially, in the implementation phase, previous stages are considered.

In the cutover stage, the final phase of RAD is implemented, including data conversion and testing. Evaluating user satisfaction is considered the essential part of the software development process.

The prototype was designed, and the results were evaluated based on the test case generator framework. In addition, the prototype has a graphical interface that allows users to construct, edit, and analyse UML statechart diagrams interactively.

The following screen shots present the user interface and screens of the prototype. The designed prototype begins on the first page of the application, and the homepage allows the registered users access to provided UML statechart examples, as shown in Figure 4.15.

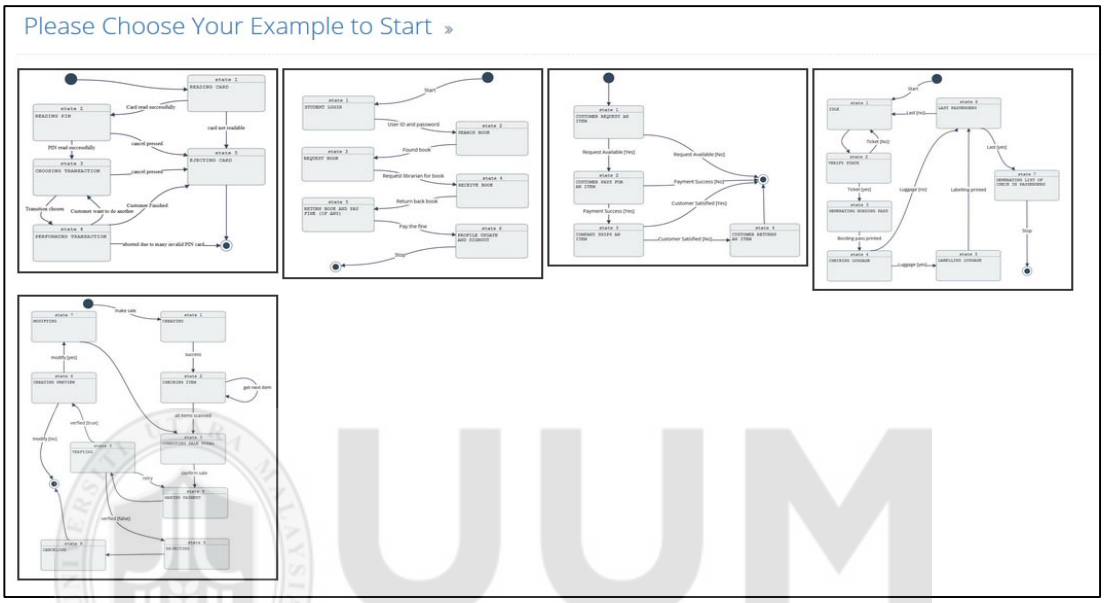

Figure 4.15. Test Case Generation Prototype

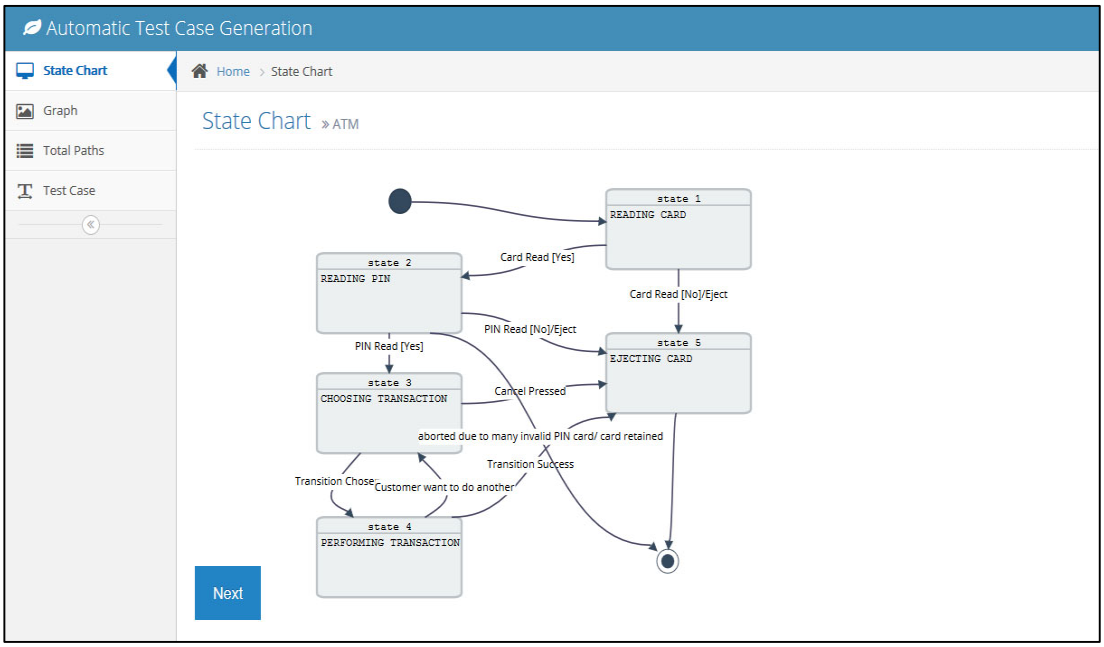

Figure 4.16. Test Case Generation Prototype in the Statechart Page

The UML statechart diagram of ATM system is shown by choosing the first example in the prototype, as presented in Figure 4.16.

The prototype will show the complete process to the user; therefore, by clicking on the next button, the intermediate graph of the selected UML statechart diagram page is generated, as shown in Figure 4.17.

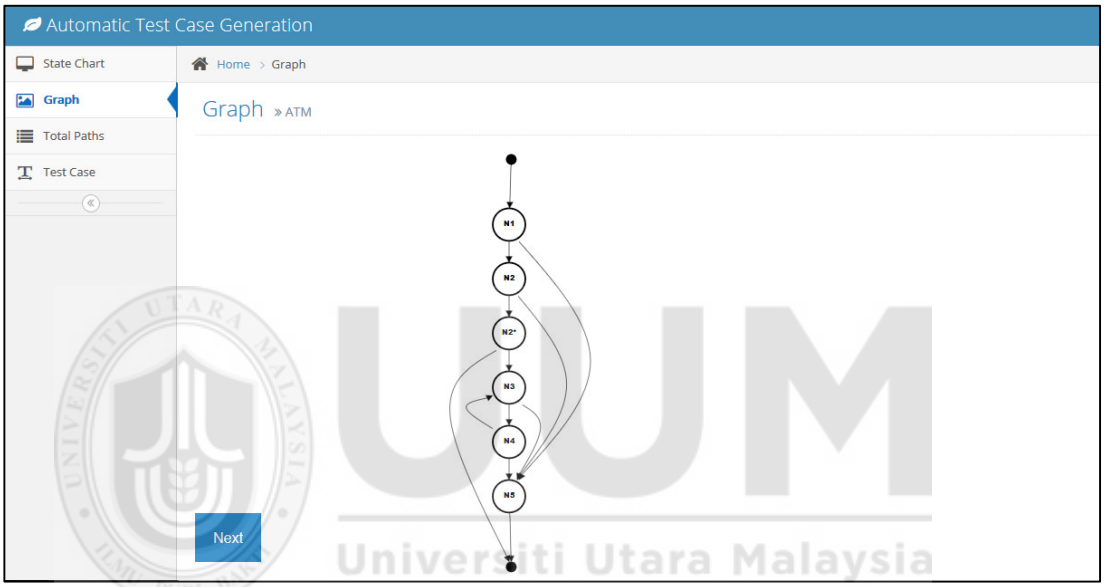

Figure 4.17. Test Case Generation Prototype in the Graph Page

Figure 4.18 summarizes the test case paths of a certain test scenario. Figure 4.19 shows a sample of the detailed description of the test cases with the ability to approve or reject these test cases. This detailed description provides structured information on how the tester interacts with the system. It details which input the tester has to provide, what output is expected, and what actions the tester should take.

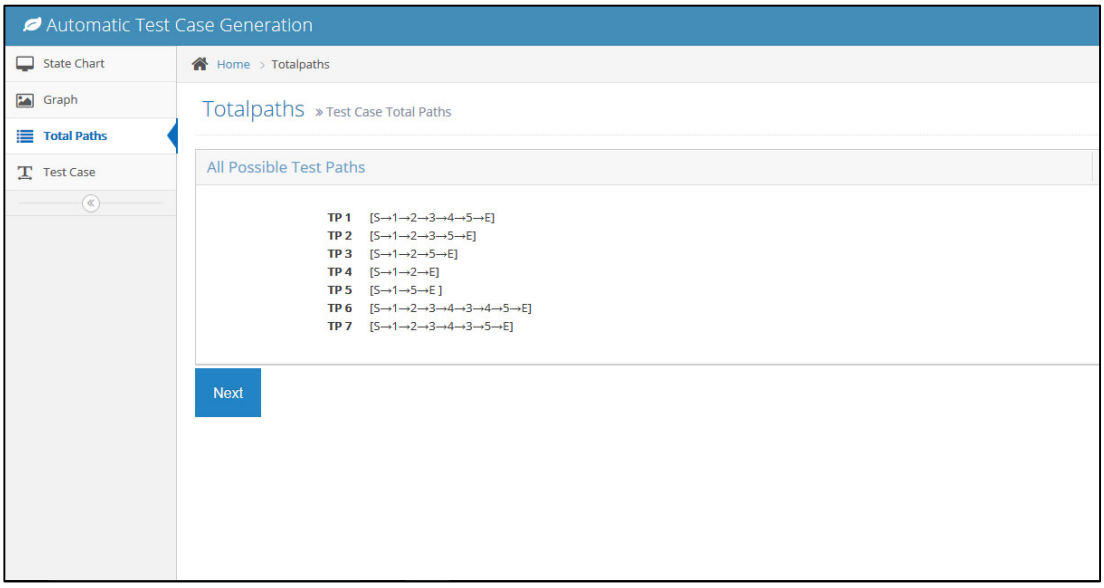

Figure 4.18. Test Case Generation Prototype in the Total Path Page

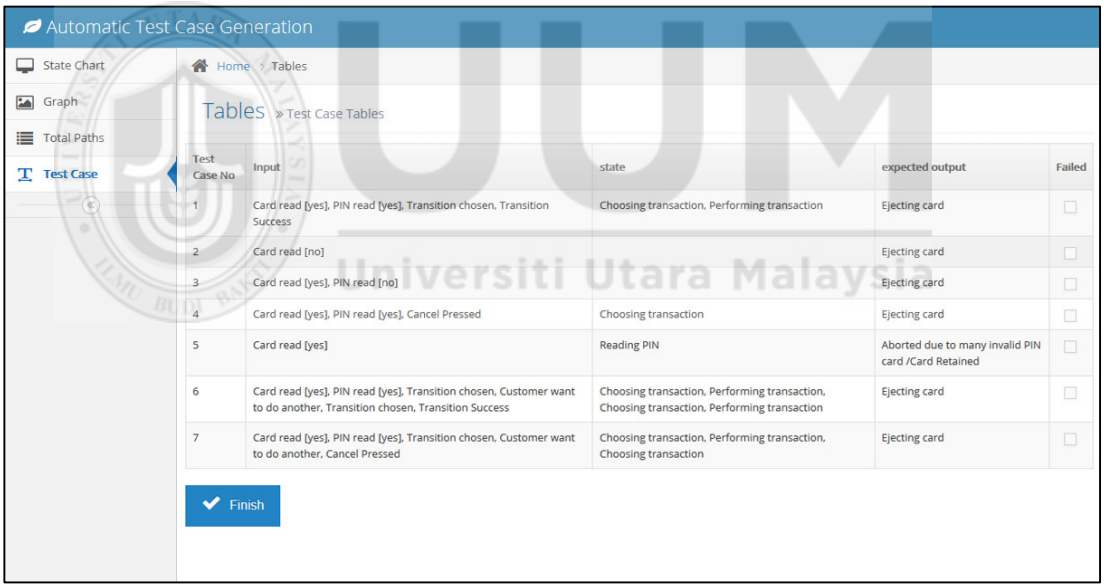

Figure 4.19. Test Case Generation Prototype Test Case Page

4.6 Summary

This chapter describes the components of the proposed framework, including rules, tables, and algorithms. The test case generation framework is proposed specifically to provide the software tester with approaches for designing and developing automatic test case generation application. Seven phases (i.e., construction of the UML statechart diagram, SRT, SRG, test case path generation, minimization, prioritization, and test case generation) are described in detail. Then, the framework will achieve four coverage criteria: all-state coverage, all-transition coverage, all-transition-pair coverage, and all-one-loop-path coverage in which the coverage criterion calculation has been presented. Afterwards, the framework and its methods was implemented in the prototype development. Objectively, this framework is intended for the software tester to follow, in developing an automatic test case generation system.

CHAPTER FIVE
EVALUATION

5.1 Introduction

In this chapter, evaluation of the proposed framework, methods, and algorithms was conducted. The evaluation start by generate the test cases using the prototype then comparing other similar test case generation methods. Furthermore, the domain experts, who are knowledge experts and software practitioners, carried out the final stage in the evaluation process. These stages are discussed further in this chapter.

5.2 Research Framework Evaluation

The main goal of the evaluation phase is to test the proposed framework and its algorithms to automatically generate the test cases from the UML statechart diagram. In addition, they are constructed specifically to ensure that it performs according to expectation (Sommerville, 2011).

The proposed system was evaluated by comparing test case generation methods with the two evaluation methods suggested by Sherwood and Rout (1998), the expert review, and development of a prototype of the automatic test case generation program. The combination of the three evaluation methods ensures that the final implementation of the framework can generate test cases using the UML statechart diagram that are proven beneficial in terms of coverage criteria.

The proposed system was evaluated in three stages, namely, prototyping, comparison, and expert review. They are further discussed in the next sections. The next subsections discuss the implementation of the framework and the use of the examples

to generate the coverage criteria. Then, the result was used for comparison with previous studies.

5.2.1 Prototyping and Examples

Prototype development was conducted as a part of evaluation. Prototyping does not count as coding because the prototype is developed only to explore how parts of the product work (Kaner, Falk, & Nguyen, 1999). Prototyping is widely acknowledged by software developers for early development testing (Bahrin, 2011). Therefore, implementing a prototype that can process the UML statechart diagram as an input, apply all transformation steps on that diagram, and assemble the expected test cases is needed to prove the proposed method.

In addition to the ATM, the other four UML statechart diagram examples that will be implemented in the prototype to evaluate the proposed framework and calculate the total average coverage criteria are the following: university library, online shop, airline check-in, and retail point of sale, as shown in Figure 4.3 (Inamdar, 2015; Lauder & Kent, 2001; Popp et al., 2009). For each example, the result of the test coverage of all-state coverage, all-transition coverage, all-transition-pair coverage, and all-one-loop coverage will be presented.

a) UML Statechart Diagram of a University Library

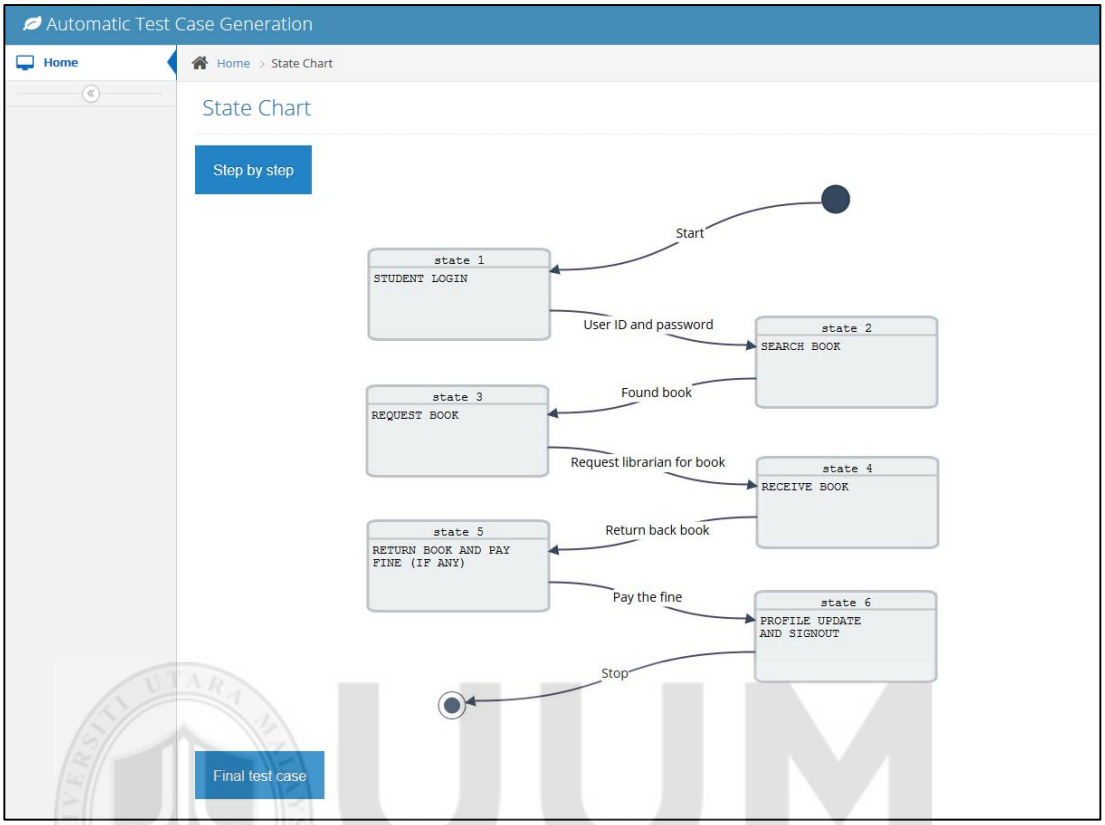

Figure 5.1. UML Statechart Diagram of a University Library

A UML statechart diagram of a university library has been imported in the prototype, as shown in Figure 5.1. The UML statechart diagram starts when a student needs to login using his/her username and password. After obtaining the access, he/she can search for a book, and when the book is found, he/she will be able to request for the book. Once the librarian has made the request, he/she will receive the book. Then, he/she will return the book and pay any fine if necessary. In the end, the system will update the user profile and terminate the session.

Using the SRT algorithm as explained in Chapter 4 (see Figure 4.6), the prototype will generate the SRT of the selected example and store the information in the database, as shown in Table 5.1.

Table 5.1

SRT of a University Library UML Statechart Diagram

V_i	V_j	$V_{j'}$	Va_i	m_i	$m_{i'}$	$type_i$	*V number*
S_0	1					*Initial State*	1
1	2		Student login	User ID and password		*State*	
2	3		Search book	Found book		*State*	
3	4		Request book	Request librarian for book		*State*	
4	5		Receive book	Return back book		*State*	
5	6		Return book and pay fine (if any)	Pay the fine		*State*	
6	d		Profile update and sign out	Stop		*Simple State*	
d						*finalState*	

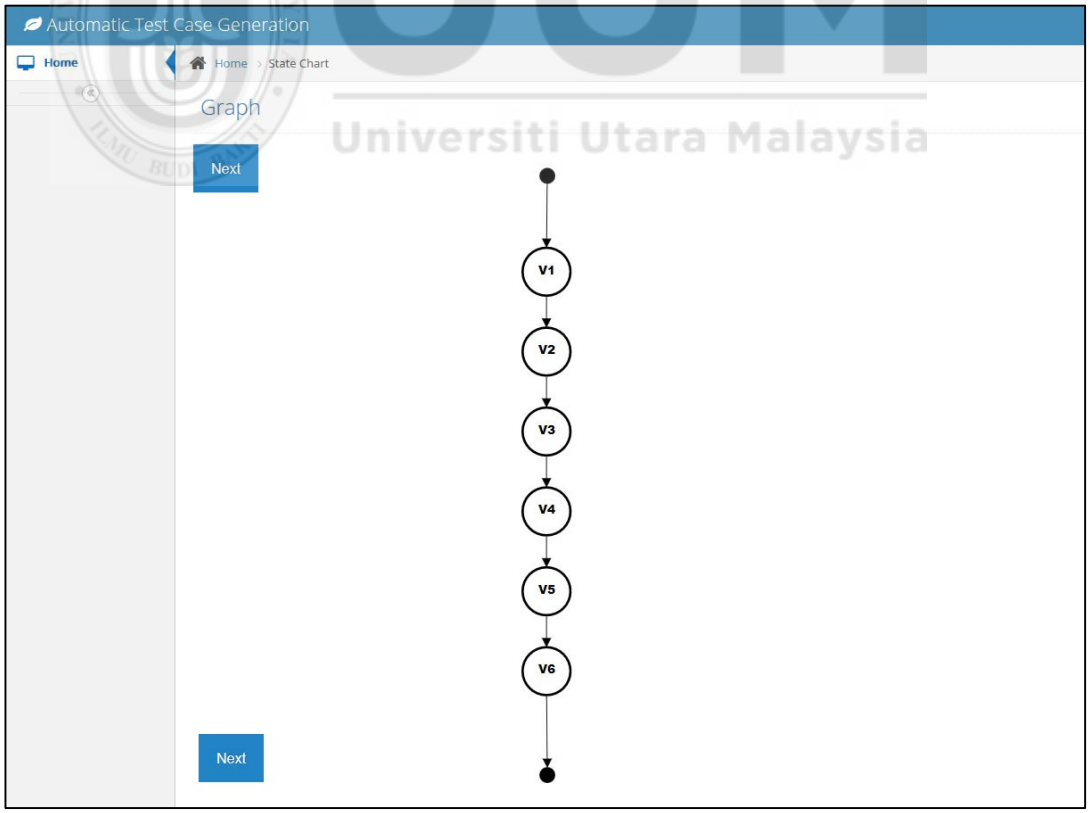

Figure 5.2. Chart Relationship Graph for a University Library UML Statechart Diagram

The next step is generating test cases in the SRG format, which will be shown by the prototype when the user clicks on the “Step by step” button, as shown in Figure 5.1. The created SRG is shown in Figure 5.2.

After clicking the next button in the SRG page, the prototype will generate all possible paths using the TCGP algorithm (see Figure 4.9) and demonstrate them as shown in Figure 5.3.

TP 1: [S→1→2→3→4→5→6→E]

Figure 5.3. All Possible Test Paths for a University Library UML Statechart Diagram

This example has one test path; thus, it does not need to be minimized or prioritized. Therefore, the final generated result will be the test cases as shown in Table 5.2.

Table 5.2

Test Cases for UML Statechart Diagram of a University Library

TC No.	**Input**	**State**	**Expected output**
1	User ID and password, pay the fine	Return book and pay fine (if any)	Profile update and sign out

The coverage criterion for the above example is calculated after implementing the example in the prototype, using coverage criteria calculation in Section 4.4, for later comparison, as shown in Table 5.3.

Table 5.3

Coverage Criteria Percentage for UML Statechart Diagram of a University Library

States (C_{AS})	**Transition (C_{AP})**	**Transition pairs (C_{AP})**	**One loop path(C_{AL})**
50%	100%	null	null

Table 5.3 shows the total coverage criteria percentage for the example in Figure 5.1, which contains four columns, namely, all-state coverage, all-transition coverage, all-transition-pair coverage, and all-one-loop coverage. The all-transition-pair coverage and all-one-loop coverage are *null* because the example does not contain a decision or loop vertex.

b) UML Statechart Diagram of an Online Shop

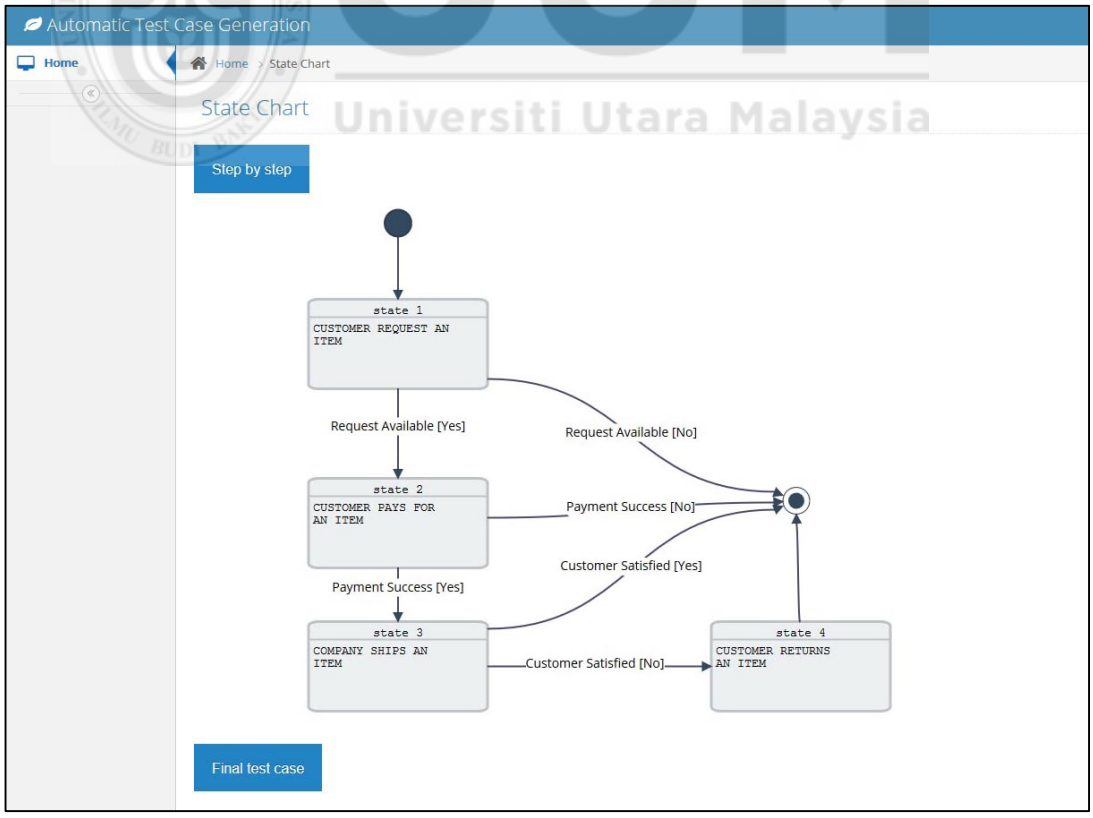

Figure 5.4. UML Statechart Diagram of an Online Shop

As shown in Figure 5.4, state 1 is a customer request for an item. If the requested item is available, then he/she can proceed to state 2 where the customer pays for the item, and if the item is not available, then the transaction is terminated. After the payment is successful, the company will ship the item, or end the transaction if payment is unsuccessful. Thereafter, if the customer is satisfied with the item, the process will end; otherwise, the customer will return the item.

Using the SRT algorithm as explained in Chapter four (see Figure 4.6), the prototype will generate the SRT of the selected example and store the information in the database, as shown in Table 5.4.

Table 5.4

For UML Statechart Diagram of an Online Shop

V_i	V_j	$V_{j'}$	Va_i	m_i	$m_{i'}$	$type_i$	**Vnumber**
S_0	1					*Initial State*	4
1	2	d	Customer requests an item	Request available [Yes]	Request available [No]	*decision*	
2	3	d	Customer pays for an item	Payment Success [Yes]	Payment Success [No]	*decision*	
3	4	d	Company ships an item	Customer satisfied [No]	Customer satisfied [Yes]	*decision*	
4	d		Customer returns an item			*Simple State*	
D						*finalState*	

The next step is the generation of test cases in the SRG format, which will be shown by the prototype when the user clicks on the “Step by step” button, as shown in Figure 5.4. The created SRG is shown in Figure 5.5.

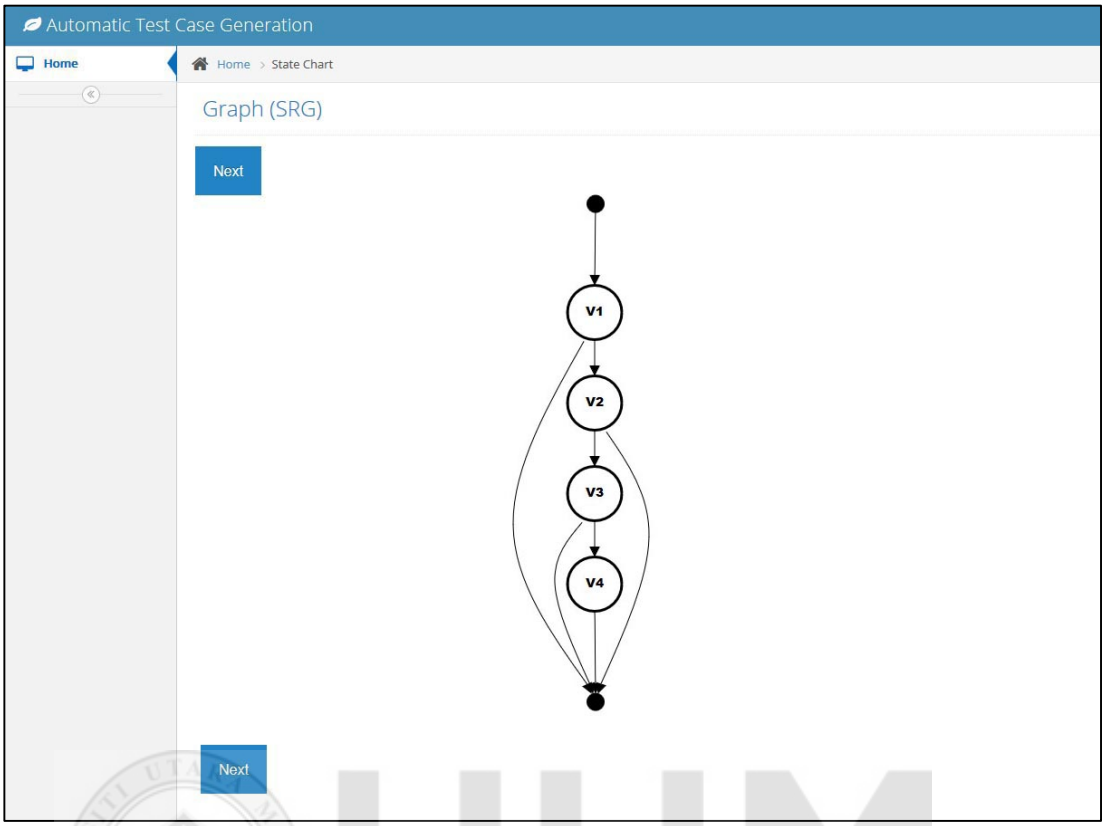

Figure 5.5. Chart Relationship Graph for the UML Statechart Diagram of an Online Shop

After clicking the next button in the SRG page, the prototype will generate all the possible paths using the TCGP algorithm (see Figure 4.9) and demonstrate them as shown in Figure 5.5.

TP 1: [S→1→2→3→4→E]

TP 2: [S→1→E]

TP 3: [S→1→2→E]

TP 4: [S→1→2→3→E]

Figure 5.6. All Possible Test Paths for the UML Statechart Diagram of an Online Shop

After generating all the possible paths, the minimization stage is conducted. However, it generates the same amount of test paths as shown in Figure 5.7.

TP 1: [S → 1 → E]

TP 2: [S → 1 → 2 → E]

TP 3: [S → 1 → 2 → 3 → E]

TP 4: [S → 1 → 2 → 3 → 4 → E]

Figure 5.7. Optimized test paths for the UML statechart diagram of an online shop

After optimizing the test paths, the mean of the brightness value is calculated for each path to prioritize the paths, as shown in Table 5.5. The table shows the order of the test paths according to importance to test the most important test case first. Details on the minimization and prioritization of this example are presented in Appendix B.

Table 5.5

Test Path Prioritization for the UML Statechart Diagram of an Online Shop

Test ID	**Test path**	**Brightness value**
TP 4	0 → 1 → 2 → 3 → 4 → 5	6.1010304355335
TP 3	0→ 1 → 2 → 3 → 5	4.6998174561816
TP 1	0 → 1 →5	4.5997256564649
TP 2	0 → 1 → 2 → 5	4.4963083323801

The final generated result will be the test cases, as shown in Table 5.6.

Table 5.6

Test Cases for a UML Statechart Diagram of an Online Shop

TC No.	Input	State	Expected output
1	Request available [Yes], Payment success [Yes], Customer satisfied [No]	Company ships an item	Customer returns an item
2	Request available [Yes], Payment success [Yes]	Company ships an item	Customer satisfied [Yes]
3		Customer request an item	Request available [No]
4	Request available [Yes]	Customer pays for an item	Payment success [No]

After implementing the example in the prototype, the coverage criteria for the above example is calculated using the coverage criteria calculation in Chapter four to be used later for comparison, as shown in Table 5.7.

Table 5.7

Coverage Criteria Percentage for a UML Statechart Diagram of an Online Shop

States (C_{AS})	Transition (C_{AP})	Transition pairs (C_{AP})	One loop path(C_{AL})
100%	100%	100%	null

Table 5.7 shows the total coverage criteria percentage for the example in Figure 5.4, which contains four columns, namely, all-state coverage, all-transition coverage, all-transition-pair coverage, and all-one-loop coverage. The all-one-loop coverage is *null* since the example does not contain a loop vertex.

c) UML Statechart Diagram of an Airline Check-in

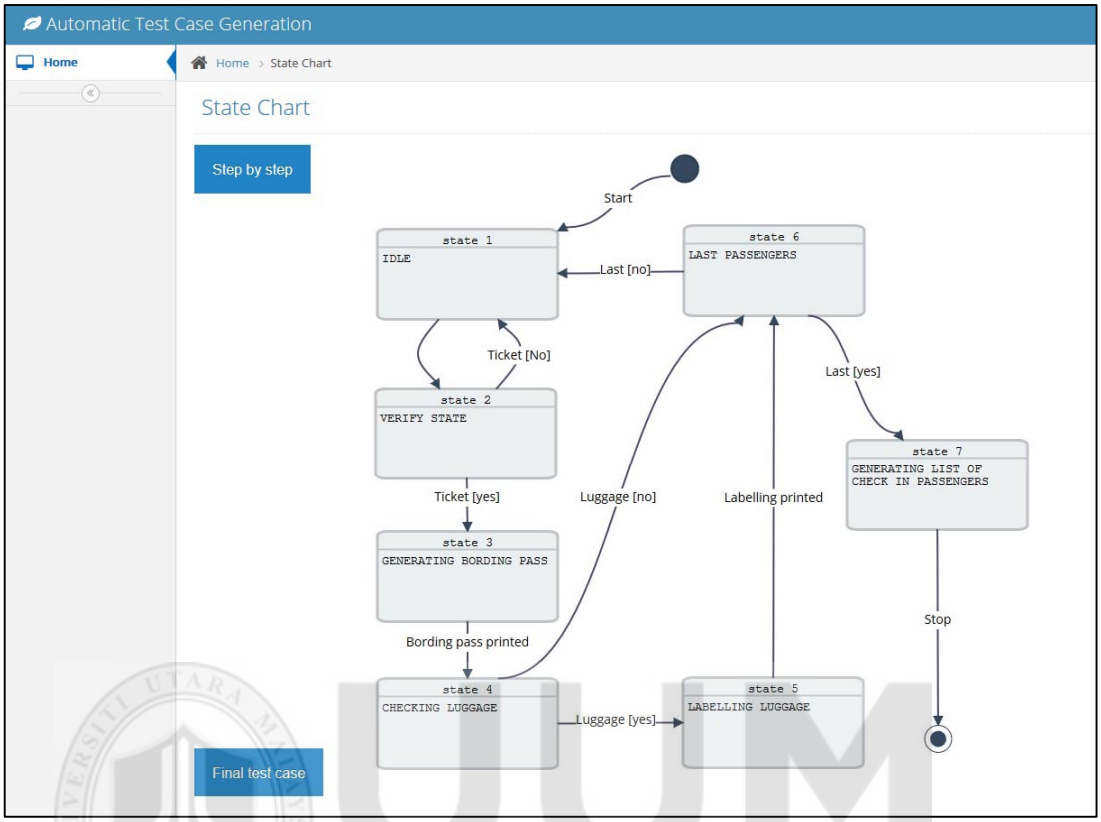

Figure 5.8. UML Statechart Diagram of an Airline Check-in

As presented in Figure 5.8, the statechart for an airline check-in starts with an idle state. Then, it proceeds to verify the state with a guard condition of whether to generate the boarding pass when the guard condition value is Yes or Reject, but returns to idle state if the guard condition value is No. In generating boarding pass state, the boarding pass is printed, followed by check luggage state. This state also has a guard condition whether to proceed to labelling the luggage (if any) or directly to the last passenger state. Thereafter, the labelling the luggage state proceeds to the last passenger state. The last passenger state checks whether the last passenger in the airplane manifest has been reached to generate a list of check-in passengers or back to the idle state if it is

not achieved. After generating a list of check-in passenger state, the statechart reaches the final state.

Using the SRT algorithm as explained in Chapter Four (see Figure 4.6), the prototype generates the SRT of the selected example and store the information in the database, as shown in Table 5.8.

Table 5.8

SRT of a UML Statechart Diagram of an Airline Check-in

V_i	V_j	$V_{j'}$	Va_i	m_i	$m_{i'}$	$type_i$	*Vnumber*
S_0	1			Start		*Initial State*	6
1	2		Idle			*State*	
2	3	1	Verify state	Ticket [yes]	Ticket [no]	*loop*	
3	4		Generating boarding pass	Boarding pass printed		*State*	
4	5	6	Checking luggage	Luggage [yes]	Luggage [no]	*decision*	
5	6		Labelling luggage	Labelling printed		*State*	
6	7	1	Last passengers	Last [yes]	Last [no]	*decision*	
7	d		Generating list of check-in passengers	Stop		*Simple State*	
d						*Final State*	

The next step is the generation of test cases in the SRG format, which will be shown by the prototype when the user clicks on the “Step by step” button, as shown in Figure 5.8. The created SRG is shown in Figure 5.9.

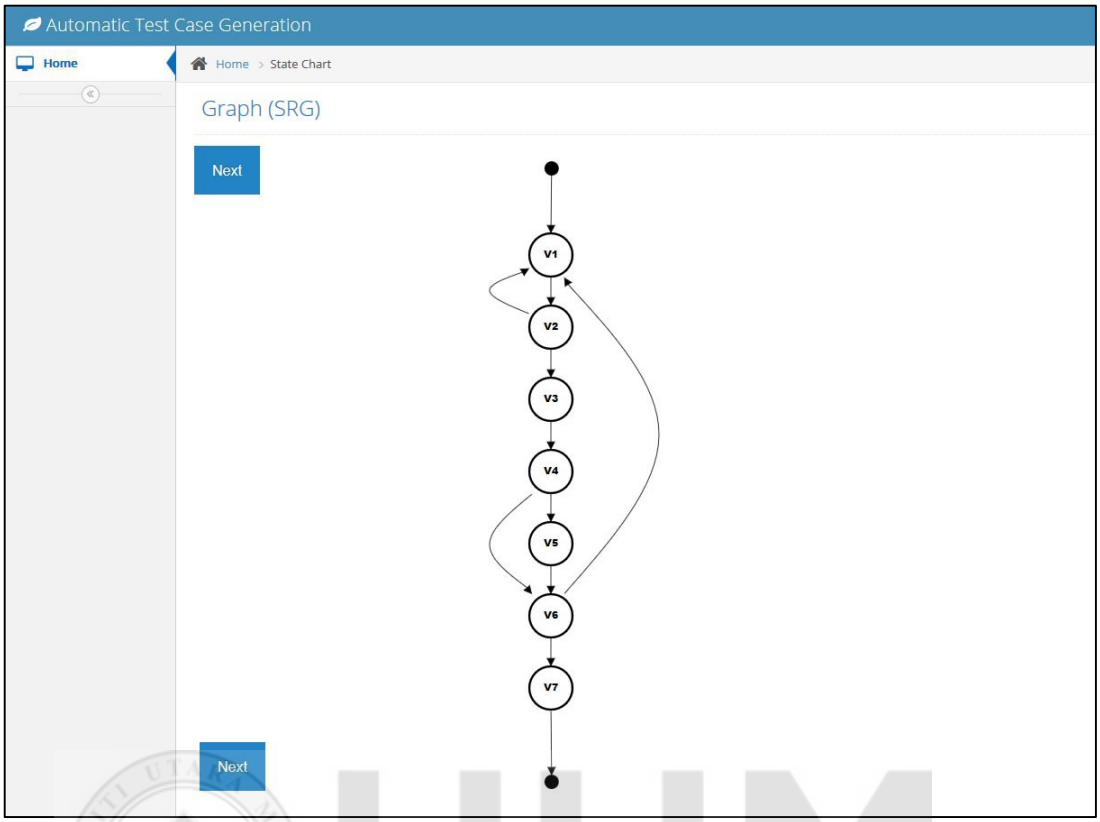

Figure 5.9. Chart Relationship Graph of a UML Statechart Diagram of an Airline Check-in

After clicking the next button in the SRG page, the prototype will generate all possible paths using the TCGP algorithm (see Figure 4.9) and demonstrate them as shown in Figure 5.10.

TP 1: [S→1→2→3→4→5→6→7→E]

TP 2: [S→1→2→3→4→6→7→E]

TP 3: [S→1→2→3→4→5→6→1→2→3→4→5→6→7→E]

TP 4: [S→1→2→3→4→5→6→1→2→3→4→6→7→E]

TP 5: [S→1→2→3→4→6→1→2→3→4→5→6→7→E]

TP 6: [S→1→2→3→4→6→1→2→3→4→6→7→E]

TP 7: [S→1→2→1→2→3→4→5→6→7→E]

TP 8: [S→1→2→1→2→3→4→6→7→E]

Figure 5.10. All Possible Test Paths of a UML Statechart Diagram of an Airline Check-in

After generating all the possible paths, the minimization stage will start by selecting the best paths as shown in Figure 5.11.

TP 2: [S→1→2→3→4→6→7→E]
TP 4: [S→1→2→3→4→5→6→1→2→3→4→6→7→E]
TP 8: [S→1→2→1→2→3→4→6→7→E]

Figure 5.11. Optimized test paths of UML statechart diagram of an airline check-in

After optimizing the test paths, the mean of the brightness value is calculated for each path as shown in Table 5.9 to prioritize the paths. Table 5.9 shows the order of test paths according to their importance to test the most important test case first. Details on the minimization and prioritization of this example are presented in Appendix B.

Table 5.9

Test Path Prioritization of a UML Statechart Diagram of an Airline Check-in

Test ID	**Test path**	**Brightness value**
TP 2	0→1→2→3→4→6→7→0	8.3417877259468
TP 8	0→1→2→1→2→3→4→6→7→8	6.8306759195254
TP 4	0→1→2→3→4→5→6→1→2→3→4→5→7→8	6.7065188982599

The final generated paths will be used to generate the test cases as shown in Table 5.10.

Table 5.10

Test Cases of UML Statechart Diagram of an Airline Check-in

TC No.	Input	State	Expected output
1	Ticket [yes], Boarding pass printed, Luggage [no], Last [yes], Stop	Idle, Generating boarding pass	Generating list of check-in passengers
2	Ticket [yes], Boarding pass printed, Luggage [yes], Labelling printed, Last [no], Ticket [yes], Boarding pass printed, Luggage [no], Last [yes], Stop	Idle, Generating boarding pass, Labelling luggage, Generating boarding pass	Generating list of check-in passengers
3	Ticket [no], Ticket [yes], Boarding pass printed, Luggage [yes], Labelling printed, Last [yes], Stop	Idle, Generating boarding pass, Labelling luggage	Generating list of check-in passengers

After implementing the example in the prototype, the coverage criterion for the above example is calculated using the coverage criteria calculation in Chapter four to be used later for comparison, as shown in Table 5.11.

Table 5.11

Coverage Criteria Percentage of a UML Statechart Diagram of an Airline Check-in

States (C_{AS})	Transition (C_{AP})	Transition pairs (C_{AP})	One loop path (C_{AL})
100%	100%	100%	100%

Table 5.11 shows the total coverage criteria percentage for the example in Figure 5.8, which contains four columns, namely, all-state coverage, all-transition coverage, all-transition-pair coverage, and all-one-loop coverage.

d) UML Statechart Diagram for a Retail Point of Sale

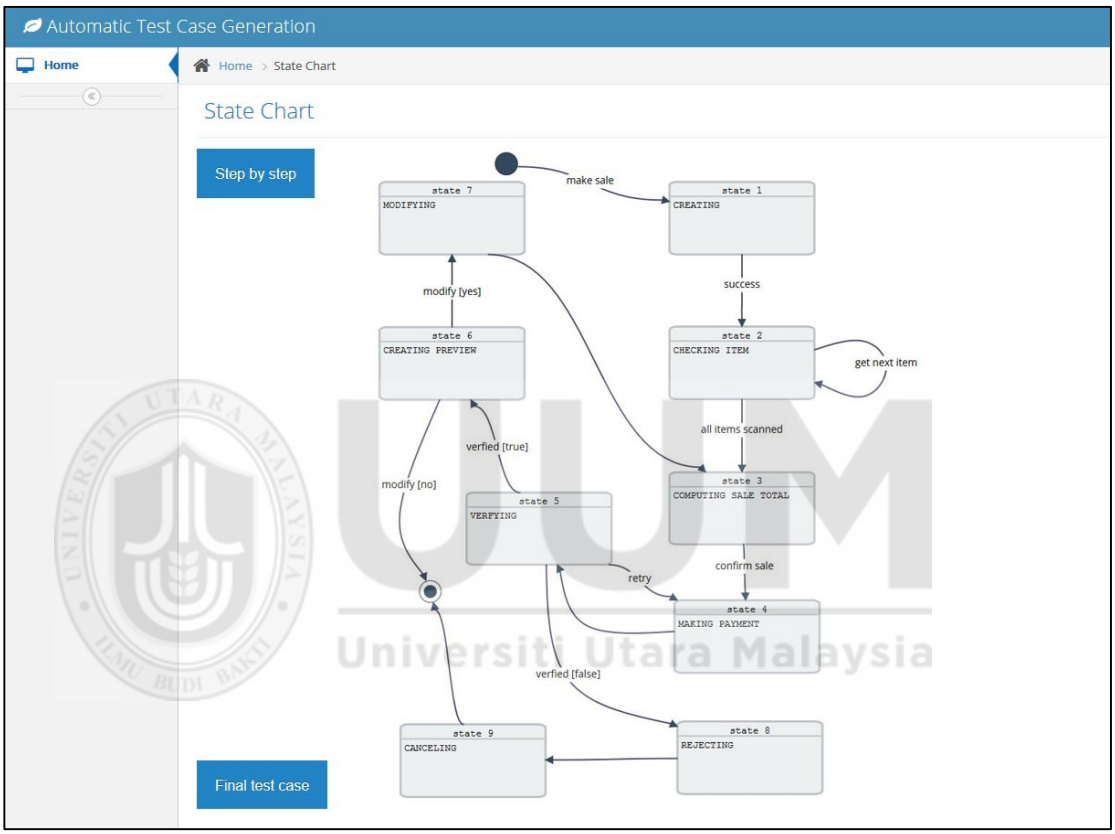

Figure 5.12. UML Statechart Diagram for a Retail Point of Sale

In Figure 5.12, the system starts by making a sale. Then, a shopping cart is created. Thereafter, the cart is ready for adding of items and proceeds to computing the sale total or adds new items. After computing the sale total, the sale is confirmed to proceed for the payment. In make payment state, the payment is verified and proceeds to creating a preview of the entire sale when the payment is approved or direct to rejection when the payment is not approved, then to cancelling of the transaction, and finally

ending the case. In creating a preview, if the user does not need any modification, then the transaction will end; otherwise, the user is directed to modification and proceeds to computing the total sale and repeats the rest of the procedure.

Using the SRT algorithm as explained in Chapter four (see Figure 4.6), the prototype will generate the SRT of the selected example and store the information in the database, as shown in Table 5.12.

Table 5.12

SRT for A UML Statechart Diagram for a Retail Point of Sale

V_i	V_j	$V_{j'}$	Va_i	m_i	$m_{i'}$	$type_i$	*Vnumber*
S_0	1			Make sale		*Initial State*	4
1	2		Creating	Success		*State*	
2	3	2	Checking item	All items scanned	Get next item	*loop*	
3	4		Computing sale Total	Confirm sale		*State*	
4	5		Making payment			*State*	
5	5`	4	Verifying		Retry	*loop*	
5`	6	8		Verified [true]	Verified [false]	*decision*	
6	7	d	Creating preview	Modify [yes]	Modify [no]	*decision*	
7	3		Modifying			*loop*	
8	9		Rejecting			*State*	
9	d		Cancelling			*Simple State*	
d						*Final State*	

The next step is generation of test cases in the SRG format, which will be shown by the prototype when the user clicks on the “Step by step” button, as shown in Figure 5.12. The created SRG is shown in Figure 5.13.

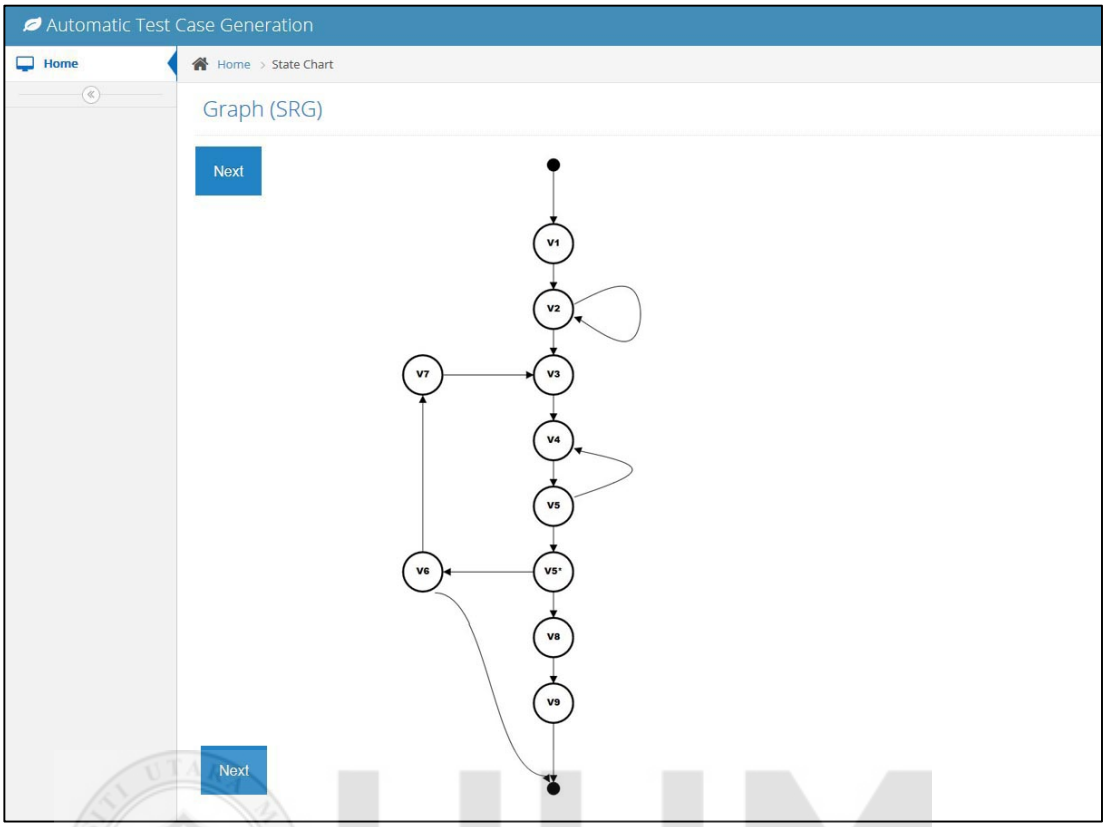

Figure 5.13. Chart Relationship Graph for UML Statechart Diagram for a Retail Point of Sale

After clicking the next button on the SRG page, the prototype will generate all possible paths using the TCGP algorithm (see Figure 4.9) and demonstrate them as shown in Figure 5.14.

TP 1: [S→1→2→3→4→5→8→9→E]
TP 2: [S→1→2→3→4→5→6→E]
TP 3: [S→1→2→3→4→5→6→7→3→4→5→8→9→E]
TP 4: [S→1→2→3→4→5→6→7→3→4→5→6→E]
TP 5: [S→1→2→3→4→5→4→5→8→9→E]
TP 6: [S→1→2→3→4→5→4→5→6→E]
TP 7: [S→1→2→2→3→4→5→8→9→E]
TP 8: [S→1→2→2→3→4→5→6→E]

Figure 5.14. All Possible Test Paths for UML Statechart Diagram for a Retail Point of Sale

After generating all the possible paths, the minimization stage will start by selecting the best paths as shown in Figure 5.15.

TP 1: [S→1→2→3→4→5→8→9→E]
TP 8: [S→1→2→2→3→4→5→6→E]
TP 4: [S→1→2→3→4→5→6→7→3→4→5→6→E]
TP 6: [S→1→2→3→4→5→4→5→6→E]

Figure 5.15. Optimized Test Paths for UML Statechart Diagram for a Retail Point of Sale

After optimizing the test paths, the mean of the brightness value is calculated for each path as shown in Table 5.13 to prioritize the paths. Table 5.13 shows the order of test paths according to their importance to test the most important test case first. Details on the minimization and prioritization of this example are presented in Appendix B.

Table 5.13

Test Path Prioritization for a UML Statechart Diagram for a Retail Point of Sale

Test ID	**Test path**	**Brightness value**
TP 1	S→1→2→3→4→5→8→9→E	4.9599705586331
TP 4	S→1→2→3→4→5→6→7→3→4→5→6→E	2.6504048295212
TP 8	S→1→2→2→3→4→5→6→E	2.3697922355156
TP 6	S→1→2→3→4→5→4→5→6→E	2.3482365027468

The final generated result will be the test cases as shown in Table 5.14.

Table 5.14

Test Cases for a UML Statechart Diagram for a Retail Point of Sale

TC No.	Input	State	Expected output
1	Make sale, success, all items scanned, confirm sale, verified [false]	Creating, checking item, Computing sale total, Verifying, rejecting	Cancelling
2	Make sale, success, all items scanned, confirm sale, verified [true], modify [yes], make sale, success, all items scanned, confirm sale, verified [true], modify [no]	Creating, checking item, Computing sale total, Verifying, creating preview, Computing sale total, Verifying	Creating preview
3	Make sale, success, all items scanned, get next item, confirm sale, verified [true], modify [no]	Creating, checking item, Checking item, computing sale total, verifying	Creating preview
4	Make sale, success, all items scanned, confirm sale, retry, verified [true], modify [no]	Creating, checking item, Computing sale total, Making payment, verifying	Creating preview

After implementing the example in the prototype, the coverage criteria for the above example is calculated using coverage criteria calculation in Chapter four to be used later for comparison, as shown in Table 5.15.

Table 5.15

Coverage Criteria Percentage for a UML Statechart Diagram for a Retail Point of Sale

States (C_{AS})	**Transition (C_{AP})**	**Transition pairs (C_{AP})**	**One loop path(C_{AL})**
88.8%	100%	100%	100%

Table 5.15 shows the total coverage criteria percentage for the example in Figure 5.13, which contains four columns, namely, all-state coverage, all-transition coverage, all-transition-pair coverage, and all-one-loop coverage.

5.2.2 Comparison with Previous Studies

The decreasing cost and time of testing will be accomplished by automating the testing in addition to eliminating human error (Khandai, Acharya, & Mohapatra, 2011). The automated test cases satisfy a coverage criterion if, for every entity defined by the coverage criterion, there is a test sequence in the test cases exercising the entity (Hong & Ural, 2004). Therefore, this section focuses on determining the coverage achievement using the proposed framework and algorithms. The evaluation of the algorithm was conducted to ensure that the framework meets its intended requirements in terms of coverage criteria. In other words, the proposed framework is intended to increase the accuracy of coverage criteria by covering the loops and parallel paths.

Figure 5.16 shows that the detailed coverage criteria percentage of proposed generated test cases framework, which are achieved by implementing five UML statechart diagrams case studies (1) ATM system, (2) university library, (3) online shop, (4)

airline check-in, and (5) retail point of sale. The coverage criteria percentage being calculated using the proposed equations in 4.4.

Table 5.16

Result of Achieved Coverage Criteria

Case study	Coverage Criteria Percentage			
	State	**Transition**	**Transition pairs**	**One-loop paths**
1	100%	100%	100%	100%
2	50%	100%	null	null
3	100%	100%	100%	null
4	100%	100%	100%	100%
5	88.8%	100%	100%	100%
Total	87.76%	100%	100%	100%

The quality of the test case generated by the proposed framework is measured by the coverage criteria. This section presents a graph that compares the proposed method with other five existing test generation techniques based on the following measurements as shown in Table 5.17: (a) all-state coverage, (b) all-transition coverage, (c) all-transition-pairs coverage, and (d) all-one-loop-path coverage. These techniques were developed by (i) Ali et al. (2007), (ii) Swain et al. (2010a), (iii) Swain et al. (2012c), (iv) Chimisliu and Wotawa (2013b), and (v) Ali et al. (2014).

First, Ali et al. (2007) proposed a method to generate test case, which combines the information from UML collaboration diagrams and statechart diagram. They transformed these diagrams to an intermediate graph. Then, they traverse the

SCOTEM graph using their proposed algorithm SCOTEM constructor to generate the test paths. Then, they execute each test path to generate the test cases, which are generated manually. In addition, Ali et al. (2007) generated a large set of test cases. However, even if this method generates a greater size of tests (no minimization), these tests do not maximize the test coverage. Moreover, they did not implement the prioritization.

Second, Swain et al. (2010a) proposed a method to generate test cases automatically from the UML statechart and activity diagram. They construct their graph based on SAD and traverse this graph using DFS. The present study achieves all-transition and all-one-loop-path coverage, while the method of Swain et al. (2010a) does not. Thus, in comparison with Swain et al. (2010a), the present study has a substantial benefit in terms of higher coverage in all-state and all-transition-pair coverage. Furthermore, Swain et al. (2010a) generated many redundant test cases. In compare in this thesis, the number of test cases is minimized, and the generated test cases are prioritized.

Third, in another work, Swain et al. Swain et al. (2012c) used the UML statechart diagram for test case generation directly. In their method, they converted the UML statechart diagram to a state graph, which traversed using DFS. They applied their minimization function and generated the test case thereafter. The test criteria they used do not cover loop-path coverage, and Swain et al. (2012c) achieved less coverage criteria in the transition pair coverage compared with the proposed method. Also in their minimization method, they calculated the vertex coverage for each test case and determine which test cases are covered by other test cases, this will result in selecting

more test cases to achieve the required coverage in compared to the proposed firefly method. In addition, their study did not prioritize the generated test cases.

Fourth, Chimisliu and Wotawa (2013b) proposed an improved tool for test case generation from a UML statechart diagram using control, data, and communication dependencies. For the coverage criteria, their generation technique aimed at achieving transition coverage only, and they do not minimize the generated test cases or prioritize them.

Finally, Ali et al. (2014) reported that the UML statechart diagram is used in their method by extracting the information from pre-condition, post-condition, and use case to build a test case with the aid of OCL. They used deterministic finite state machine as an intermediate model to be traversed by BFS to generate a test sequence. A significant advantage of the present study in comparison with the work of Ali et al. (2014) is that the present study has minimized the number of test cases and reduced their sizes. Furthermore, the generated test cases are prioritized in testing while maintaining the coverage criteria.

Thus, in comparison with the work of Ali et al. (2014); Ali et al. (2007); Chimisliu and Wotawa (2013b); Swain et al. (2012c); Swain et al. (2010a), the proposed method has substantial benefit in terms of the coverage achieved, where it achieved higher coverage with smaller number and size of the test cases. These existing studies do not ensure all-one-loop-path coverage testing, but the proposed method ensures it. The coverage criteria comparison is shown in Table 5.16. However, for the proposed method, the all-state coverage decreased from 100% to 87.7% after generating the test cases from the test case paths because the present study applies path pruning before

generating the test cases in which one of its objectives is to reduce the number of states. Therefore, even it achieves less coverage in all-state coverage, the generated test cases are more efficient. In addition, to comparison with these studies, the proposed work has prioritized the generated test cases.

Table 5.17

Comparison Result of Coverage Criteria

Study	**Coverage Criteria Percentage**			
	State	**Transition**	**Transition pairs**	**One-loop paths**
Ali et al. (2007)	×	91%	×	×
Swain et al. (2010a)	71%	×	65%	×
Swain et al. (2012c)	100%	100%	58.17%	×
Chimisliu and Wotawa (2013)	×	100%	×	×
Ali et al. (2014)	100%	100%	100%	×
Proposed work	87.76%	100%	100%	100%

Figure 5.16 shows that the proposed method generates the highest coverage criteria in all-one-loop coverage and those of Ali et al. (2014); Swain et al. (2012c) generate the highest coverage criteria in all-state coverage for the test cases. The other three techniques by Ali et al. (2014); Chimisliu and Wotawa (2013b); Swain et al. (2012c) cover the all-transition coverage of the UML statechart diagram. Chimisliu and Wotawa (2013b) approach has the least types of coverage criteria compared with other techniques. Ali et al. (2014); Swain et al. (2012c) methods achieve all-transition-pair coverage.

In the conclusion, the proposed method is the most recommended method to generate

minimized and prioritized test cases with 100% coverage criteria aimed at all-state coverage, all-transition coverage, all-transition-pair coverage, and all-one-loop path for the UML statechart diagram.

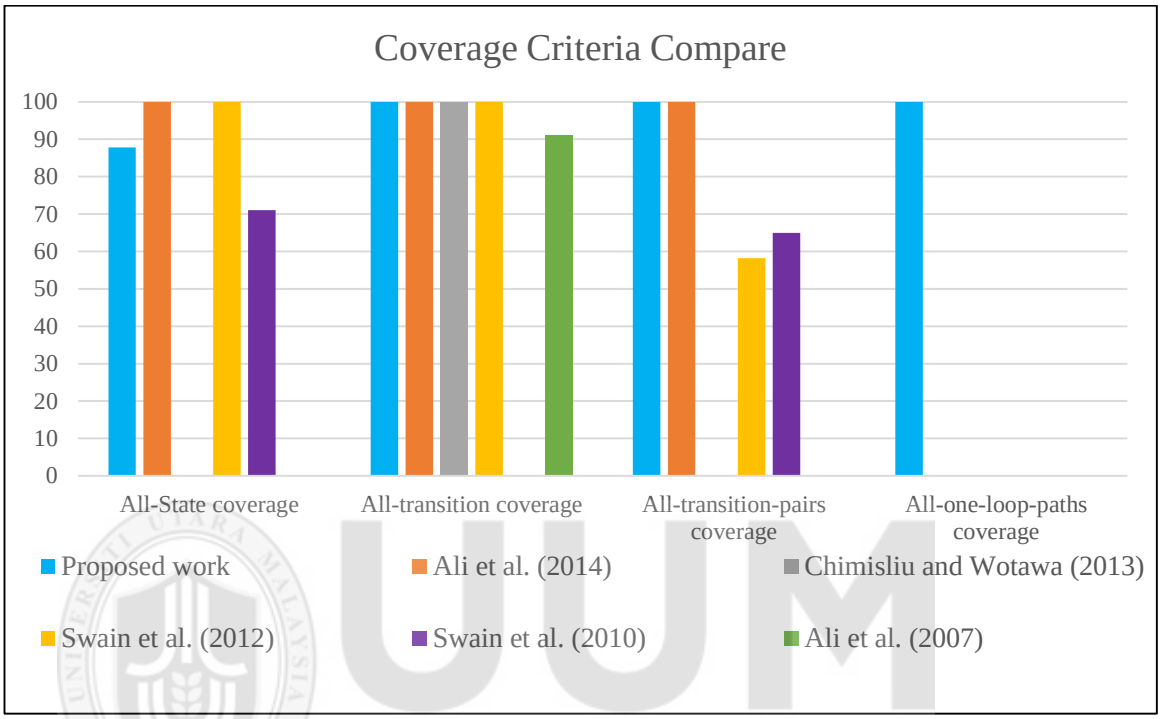

Figure 5.16. Test Coverage Criteria Chart of Comparison Result

5.2.3 Expert Reviews

In this section, expert review was conducted because it can be easily implemented in addition to being fast and affordable. Moreover, expert reviews have been acknowledged as a significant way to detect and correct faults (Komuro & Komoda, 2008; Wiegers, 2002b). Furthermore, having a human to evaluate the correctness of a given output is effective (Ammann & Offutt, 2008).

Consequently, the present study adapted this technique for the verification process. This approach also has been used in the field of software engineering to evaluate or obtain practitioner experience (Daneva & Ahituv, 2011).

Four professionals from software engineering and algorithm-related fields were identified as the potential experts. In addition, three domain experts from the software industry were added because they are potential users of the framework and prototype and they can provide feedback based on their practices in real-world projects.

Prior to conducting the interview with the experts, the interview guide was developed. The principles of preparing interview guides were adapted, whereby the discussion was planned to be started by general topic, which is the introduction of the study. Then, the next agenda was to obtain the weight values, continued with the evaluation of the proposed framework. These key sequential activities were determined based on their relative importance to the study, as suggested by the second principle of preparing interview guide (Stewart & Shamdasani, 2014). Additionally, the materials used during the interview session were prepared, namely the presentation slides, documents for the participants and the questioner for the evaluation process.

According to the activities involved during the expert review process, the researcher conducts a presentation to provide an overview of the study and explain its components, also what to be expected to evaluate in this work by showing the evaluation form. Then a presentation will be conducted to explain the process of the framework and its algorithms, also the results and comparison with the previous studies. At the same time, the researcher provides detailed documents that include the framework, algorithms (TCGP, TCG, minimization and prioritization), coverage criteria results, and the prototype also the results and comparison with the previous studies to be reviewed by the experts. After that, the expert was able to run the prototype and try its functions. In the end, the experts fill in the evaluation form

(Appendix A) and provide their comments and feedback. In addition, as result from the interview and the form, the researcher updates the software processes based on the comments and suggestion from the experts.

Table 5.18 summarizes the knowledge experts' background. The background of domain experts and activities related to the review are discussed in Section 5.3.

Table 5.18

Experts' Background

	ID	Position	Expertise	Years of Experience	Institutions
Domain	Expert A	Software analyst	Software development, Web development, and database analyst	7 years	Uniutama Solution Sdn. Bhd
	Expert B	Research development project manager	Project management, software development, software development, software engineering, and software testing	7 years	PT Jingdong Indonesia Pertama
	Expert C	System analyst	Software development, Web development, and database analyst	7 years	Uniutama Solution Sdn. Bhd
Academic	Expert D	Senior lecturer	Software engineer, combinatorial testing generation, and search-based optimization algorithms	10 years	University Malaysia Pahang
	Expert E	Associate professor	Optimization algorithms, swarm algorithms, and grid computing	15 years	Universiti Utara Malaysia
	Expert F	Senior lecturer	Software engineering, software testing, algorithm design	8 years	Universiti Malaysia Perlis
	Expert G	Senior lecturer	Multimedia, Web design, Web development, and game-based learning	5 years	Universiti Utara Malaysia

- **Results for the Review**

Concisely, in Table 5.19, all of the experts agreed firstly that the framework achieved practicality, clarity, and completeness. Secondly, the proposed algorithms accomplished correctness. Thirdly, the prototype was effective. Fourthly, the system achieved accuracy, perceived usefulness, and usability overall. Finally, the documentation was comprehensible.

Meanwhile, Expert A is expert in the industrial domain, expert A agreed on the usefulness of the system in software testing practices for industry, and found that they system can reduce the time and cost. On the other hand, Expert D concluded that the present study improved the test case generation process by generating high coverage test cases, and the use of optimization algorithm benefit in increases the consistency of the generates test cases. In addition, Expert D suggested adding some terms in the processes to highlight the contribution of the proposed work.

Expert E is an expert in swarm algorithms, and has highlighted the benefit from using firefly algorithms in achieving optimal test cases, and suggested the use of ten firefly to reach the optimal selecting brightness to prioritize the generated test cases.

Furthermore, Expert F suggested that the comparison with the previous studies should highlight the achievement of the current study in achieving loop coverage, and demonstrate the improvement in minimization and prioritization, by stating that this study achieved higher coverage criteria with less number of test cases in comparison to other studies.

However, the experts had some comments on the full automation of the prototype. For example, Experts B and G suggested including UML statechart diagram upload function, and Expert D suggested adding an integrated drawing function for future work. Meanwhile, Expert A concluded that the prototype was working perfectly, and the proposed system could simplify the software testing process.

Table 5.19

Results for Expert Review Verification

Dimensions	**Expert A**	**Expert B**	**Expert C**	**Expert D**	**Expert E**	**Expert F**	**Expert G**
Practicality	Agree	Agree	Agree	Agree	Agree	Agree	Agree
Clarity	Agree	Agree	Agree	Agree	Agree	Agree	Agree
Completeness	Agree	Agree	Agree	Agree	Agree	Agree	Agree
Correctness	Agree	Agree	Agree	Agree	Agree	Agree	Agree
Effectiveness	Agree	Agree	Agree	Agree	Agree	Agree	Agree
Accuracy	Agree	Agree	Agree	Agree	Agree	Agree	Agree
Perceived Usefulness	Agree	Agree	Agree	Agree	Agree	Agree	Agree
Usability	Agree	Agree	Agree	Agree	Agree	Agree	Agree
Comprehensibility	Agree	Agree	Agree	Agree	Agree	Agree	Agree

Overall comments:

Expert A: The system is very useful in software testing for the industry field, and it can reduce the time and cost.

Expert B: The researcher is on the right track.

Expert C: The researcher is on the right track.

Expert D: The researcher improves the test case generation for UML statechart diagram by modifying and combining the current approaches. The current work enhances the generating of test cases by minimizing and prioritizing them.

Expert E: A beneficial study and can be improved by scoping the size of the tested systems. In addition, the use of firefly algorithm benefit in minimizing the number of test cases and prioritize them.

Expert F: The researcher is on the right track but needs to highlight the all-one-loop coverage comparison. In addition, this work generated fewer prioritized test cases with higher coverage criteria in comparison to other works.

Expert G: The researcher has implemented the technique correctly.

5.3 Summary

This chapter has discussed the evaluation of the proposed framework, which was conducted on the framework and algorithms using prototyping and comparison. In addition, an expert review was conducted by seven experts. Based on the feedback, the framework was improved.

CHAPTER SIX
CONCLUSION

6.1 Introduction

In this chapter, the conclusion of the present study is presented as explored and described in the thesis. The discussion begins a summary of the study in Section 6.2 followed by the contributions in Section 6.3. The limitations and future work of the study are described in Section 6.4. The chapter ends with the conclusions in Section 6.5.

6.2 Summarizing the Study

This aim was achieved through four objectives, which have been defined in Section 1.5. The study is summarized based on these objectives accordingly.

Objective 1: To investigate the current practices of software test case generation methods that use the UML diagrams as an input, to design the proposed framework.

The first objective is to investigate the current methods used to generate test cases based on UML diagrams by analysing the content of related past studies. This objective aims to use these previous studies to establish the comprehensive process in generating test cases in the proposed framework and to identify the existing methods in each process. Furthermore, the content analysis of past research indicated the UML statechart diagram is the most suitable UML diagram to generate the test cases from the design software lifecycle phase. Moreover, the content analysis showed that past studies lack coverage, particularly in SUT with transition and/or loop states, as

highlighted in the problem statement in Chapter 1. The present study proposed a framework that takes the UML statechart diagram as an input and generates test cases as an output. The proposed framework has clear and well defined processes: SRTs, STGs, test case path generation, consistency checking, test case path minimization, test case path prioritization, path pruning, and test case generation, as shown in Figure 3.2.

Objective 2: To identify the suitable coverage criteria, which are covered by the proposed framework generated test cases.

This objective was achieved by conducting content analysis on the previous studies to select suitable coverage criteria for the generated test cases from the UML statechart diagram. The content analysis of past research indicated that all-state and all-transition coverage are the most commonly used coverage criteria. However, two other coverage criteria were also selected, namely, all-transition pairs and all-one-loop coverage, for their importance to deal with parallel and loop path coverage. The selected coverage criteria for the generated test cases from the UML statechart diagram are all-state, all-transition, all-transition-pair, and all-one-loop coverage. This framework has proposed four test coverage criteria, while similar studies is this field such as Chimisliu and Wotawa (2012); Chimisliu and Wotawa (2013a, 2013b) are having less number of coverage criteria.

Objective 3: To develop an improved method that generates minimized and prioritized test cases using the proposed test case generation framework.

The present study has fulfilled this objective by improving the methods that generate optimized test cases by implementing them in the proposed test case generation

framework. This process is conducted by converting the inputted UML statechart diagram to a compatible SRT that can store all the relevant information in the database. In turn, this table is converted to a SRG to be traversed to generate all possible test paths using the TCGP algorithm. However, these paths will generate a large number of test cases that will be difficult to test in their current condition. To overcome this phenomenon, a modified firefly algorithm is proposed to minimize the number of test paths and overcome this liability. The minimized test paths are passed into the second phase of the firefly algorithm and prioritized. In the final phase, the test cases will be generated according to the proposed coverage criteria as identified in the second objective. This study is more comprehensive compare to similar studies such Chimisliu and Wotawa (2012); Chimisliu and Wotawa (2013a, 2013b); Swain et al. (2012a, 2012b); Swain et al. (2012c), as this study cover more coverage criteria, minimized, and prioritized test cases.

Objective 4: To evaluate the proposed framework using prototyping, comparison with existing work, and expert review.

The last objective was fulfilled by evaluating this study into three stages, which are prototyping, comparison with previous test case generation methods, and expert review. In the prototyping phase, five different UML statechart diagrams with loops and higher cyclomatic complicity have been used. As stated in the problem statement, studies such as Biswal (2010); Swain et al. (2012a, 2012b); Swain et al. (2012c) are using simpler graph. The examples were inserted into the developed prototype to generate the test cases automatically; then, coverage criteria for each example are calculated. The coverage criteria percentage results were compared with previous methods in the comparison phase as described in Section 5.2.1.2.

Practicality, clarity, completeness of the framework, correctness of the algorithm, effectiveness of the prototype and accuracy, perceived usefulness, and usability of the system were evaluated by the experts. Results from this stage revealed that the proposed framework is practical, clear, and complete. Similarly, the algorithms were implemented correctly. Likewise, the prototype is effective. The system was accurate, useful, and usable. However, some modifications were performed to organize the software processes. Further details on the expert review results are described in Section 5.2.2.

6.3 Contributions

In this thesis, the author has described the contribution of the proposed method. It starts with its vital contribution, which is to design a framework for automatic test case generation from the UML statechart diagram. The specific contributions are elaborated in the next subsection.

6.3.1 Test Case Generation Framework

The main contribution of this study is to produce a new test case generation framework. It was built based on the outcomes of the content analysis from previous studies and enhanced through exploratory studies. This framework maps comprehensive processes in converting UML diagrams to test cases, which can be used by researchers to generate test cases for similar diagrams. Existing test case generation processes only focus on parts of the process or do not achieve the appropriate coverage. Accordingly, the present study focuses on generation of test cases with the highest coverage and with the lowest number of possible test cases to overcome these limitations.

The framework consists of the following seven main components: construction of UML statechart diagram, SRT, SRG, test case path generation, test case path minimization, test case path prioritization, and test case generation. In addition, the present study has added two components, namely, consistency checking and path pruning.

6.3.2 Enhanced Consistency Checking of Test Paths

The consistency checking equation was proposed to be added to the test case generation framework phases to reduce human error for the UML statechart diagram illustration. This consistency checking equation (see Equation 4.6) is an improved version of the CC equation enhanced to support the loop coverage. This method aids in ensuring the reliability of the inputted diagram.

6.3.3 Improved Path Pruning

A large test case makes the diagnosis difficult because it has redundant information (Leitner, Oriol, Zeller, Ciupa, & Meyer, 2007); therefore, path pruning steps were developed based on UML statechart diagram components to be added to the test case generation framework phases. In genetic algorithm, pruning has been used to fasten the process time of results and produce the optimal solution (Hedjazi & Marjani, 2010). The proposed pruning steps has been developed based on the concept of pruning in genetic algorithm and by adapted state type method by Kundu and Samanta (2009). The proposed path pruning steps (see Figure 4.13) has been developed to prune the generated test path to generate smaller size test cases by reducing the unnecessary information it them.

6.3.4 Coverage Criteria for UML Statechart Diagram

A test case is a sequence of conditions that satisfy certain coverage criteria (Rhmann & Saxena, 2016). Therefore, coverage criteria were required to evaluate the generated test cases. A review on the related previous studies was conducted to select the most common coverage criteria for UML diagrams in general and UML statechart diagram in particular. Four coverage criteria were selected for UML statechart diagram test case generation according to their importance. In addition, it is important in achieving the aims of the present study in covering parallel paths and loop paths. The commonly used coverage criteria are all-state coverage, all-transition coverage, and all-transition-pair coverage. However, this study has added an additional coverage criterion that is all-one-loop coverage to highlight the inadequacy in state loop covering.

- **Coverage Criteria Calculation**

After the coverage criteria were selected, an accurate coverage criteria calculation method was needed to measure the percentage of criteria coverage to evaluate the accuracy or quality of test case generation. These methods use element coverage equation as basis and has been modified to accurately calculate each selected type of coverage criterion as shown in Section 4.4. This equations help in calculating the coverage for the selected coverage criterion automatically.

6.3.5 SRT Algorithm

To process the UML statechart diagram in a manner that the machine understands, the diagram should be converted to a table to be stored in the database. However, there are shortages of concepts as regards parsers that are capable of reading, extracting, and

interpreting artefacts from UML diagrams (Oluwagbemi & Asmuni, 2014). This study proposed an algorithm to extract and store UML statechart diagram information regardless of the diagram complexity. The SRT algorithm (see Figure 4.6) stores the UML statechart diagram in a table and highlights the relationships between the states and at the same time store the state and edge information to be used later in automatic test case generation. The SRT algorithm has been developed based on the previous studies intermediate tables. However, these tables have been modified to handle loop states and UML statechart diagram. In addition, this relation table can support the generation of the test paths for most of UML diagrams by using its approach rules.

6.3.6 TCGP Algorithm

A traversal algorithm is needed to generate the test path from the intermediate graph. The most common tree traversal algorithms are DFS and BFS. However, this study focuses on covering the loop path, while the two existing traversal algorithms depend on the tree graph. A tree is a special type of graph that contains no cycles. A tree is a set of vertices with one vertex designated as the root vertex and a list of edges connecting the vertices without creating cycles (Oluwagbemi & Asmuni, 2014). Therefore, the present study proposed a traversal algorithm to generate all possible paths according to the proposed coverage criteria. The proposed TCGP algorithm (see Figure 4.10) was developed to handle decision and loop state to generate the test paths that cover transition pairs and loop coverage. This algorithm solve parallel and loop problems as discussed in Section 4.2.1.

6.3.7 Path Minimization Method

Test case minimization mechanisms play a major role in reducing the number of test cases without affecting their quality. However, reducing the number of test cases especially in software systems is a major problem (Ahmed, 2016). Therefore, the present study adapted the firefly algorithm with adjacency matrix to minimize the generated test paths while maintaining the coverage criteria. However, the brightness increases rapidly; thus, the path weight was formed to direct the firefly algorithm. The path weight method was proposed to select the optimized test paths. The proposed method uses the firefly algorithm output to select the optimized paths with the help of the generated test path weight to minimize the test paths and reduce redundancy. Therefore, this method (refer to Section 4.3.5) can generate fewer paths because one path can cover more than one sequence of vertices.

6.3.8 Path Prioritization Method

Test case prioritization aims at ordering test cases to increase the rate of fault detection, which quantifies how fast faults are detected during the testing phase (Eghbali & Tahvildari, 2016). Therefore, the present study adapted the firefly algorithm as well as the information flow metric to increase the brightness of the important vertices to drive the algorithm to the important path and also to add the path weight method to the total brightness of each path to select the most prioritized paths. Consequently, this method (refer to Section 4.3.6) can choose the paths and prioritize them accordingly.

6.3.9 Test Case Generation Algorithm

The main goal is to generate the test cases, which will use the generated information up to this step to generate all the possible test cases that achieve the proposed coverage criteria. The TCG algorithm (see Figure 4.14) uses the data stored in the SRT with the output of the minimized and prioritized test paths and combined them with the path pruning rules to generate the minimized test cases with prioritization order.

6.3.10 Developed Prototype

A prototype was developed to validate the proposed framework and investigate its algorithm performance. By developing the prototype in a Web-based system, the seven main phases of the framework were successfully embedded in the proposed framework.

6.4 Limitations and Future Work

The present study could be further improved and extended in several aspects based on previous in-depth discussion and detailed analysis. The study limitations and future work to enhance this study are summarized as follows.

The proposed framework can be improved to be used for other UML diagrams beside UML statechart diagram, also for a combination between two or more diagrams so that the system is able to handle all type of errors (Khurana & Chillar, 2015). Furthermore, the test case minimization and prioritization methods using the firefly algorithm can be implemented in different test case paths since these paths can be generated from different diagrams. However, further testing is required. For future work, other metaheuristic algorithms can be adapted to minimize or prioritize the test cases, for

example Krill Herd, Charged System Search, Bat Algorithm, Cuckoo Search, Bee Algorithms, Ant Colony Optimization, and Particle Swarm Optimization

The developed prototype has few limitations because it was developed for evaluation and not for commercial purposes. One of its limitations is the manual inputting of the UML statechart diagram. An integrated drawing add-on will improve the prototype to become more user friendly, and its compatibility with other modelling tools such as Rational Rose, Magic Draw, and Microsoft Visio is worthy of investigation. In addition, report function and previous system testing evaluation statistics can enhance the prototype for commercial use.

Moreover, the test case generation framework is designed for small to medium systems. In future work, the proposed method can be modified and tested on enterprise systems.

REFERENCES

Abdurazik, A., & Offutt, J. (1999). *Generating test cases from UML specifications.* George Mason University.

Abdurazik, A., & Offutt, J. (2000). *Using UML collaboration diagrams for static checking and test generation.* Paper presented at the «UML» 2000 -The Unified Modeling Language.

Abdurazik, A., Offutt, J., & Baldini, A. (2004). A controlled experimental evaluation of test cases generated from UML diagrams: Technical Report, ISE-TR-04-03. George Mason University.

Aggarwal, M., & Sabharwal, S. (2012). *Test case generation from UML state machine diagram: A survey.* Paper presented at the Computer and Communication Technology (ICCCT), 2012 Third International Conference on.

Ahamed, S. (2010). Studying the feasibility and importance of software testing: An analysis. *Internatinal Journal of Engineering Science and Technology, 1*(3), 119-128.

Ahmad, J., & Baharom, S. (2017). A Systematic Literature Review of the Test Case Prioritization Technique for Sequence of Events. *International Journal of Applied Engineering Research, 12*(7), 1389-1395.

Ahmed, B. S. (2016). Test case minimization approach using fault detection and combinatorial optimization techniques for configuration-aware structural testing. *Engineering Science and Technology, an International Journal, 19*(2), 737-753.

Aichernig, B. K. (2001). *Systematic black-box testing of computer-based systems through formal abstraction techniques.* (PhD Dissertation), Graz University of Technology, Graz, Austria.

Al-kahlout, A., B. salha, B., & El-haddad, N. (2017). *E-Account APP*. University of Palestine, Gaza Strip, Palestine.

Al-Tarawneh, F. H. (2014). *A framework for cots software evaluation and selection for COTS mismatches handling and non-functional requirements.* Universiti Utara Malaysia.

Al Dallal, J., & Sorenson, P. (2006). Generating class based test cases for interface classes of object-oriented black box frameworks. *Transactions on Engineering, Computing and Technology, 16*, 90-95. doi: 10.1.1.193.4045

Alhroob, A. (2014). *Best Test Cases Selection Approach*. Paper presented at the Scientific Cooperations International Workshops on Electrical and Computer Engineering Subfields.

Alhroob, A. M. (2012). *Software test case generation from system models and specification. Use of the UML diagrams and High Level Petri Nets models for developing software test cases.* University of Bradford.

Ali, M. A., Shaik, K., & Kumar, S. (2014). Test case generation using UML state diagram and OCL expression. *International Journal of Computer Applications, 95*(12), 7 -11. doi: 10.5120/ijais2016451599

Ali, S., Briand, L. C., Hemmati, H., & Panesar-Walawege, R. K. (2010). A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on software engineering, 36*(6), 742-762. doi: 10.1109/TSE.2009.52

Ali, S., Briand, L. C., Rehman, M. J.-u., Asghar, H., Iqbal, M. Z. Z., & Nadeem, A. (2007). A state-based approach to integration testing based on UML models. *Information and Software Technology, 49*(11), 1087–1106. doi: 10.1016/j.infsof.2006.11.002

Ammann, P., & Offutt, J. (2008). *Introduction to software testing*. Cambridge, United Kingdom: Cambridge University Press.

Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., . . . McMinn, P. (2013). An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software, 86*(8), 1978–2001. doi: 10.1016/j.jss.2013.02.061

Avancena, A. T., & Nishihara, A. (2015). Usability and pedagogical assessment of an algorithm learning tool: a case study for an introductory programming course for high school. *Issues in Informing Science & Information Technology, 12*, 21-44.

Bahrin, Z. S. (2011). *Mobile game-based learning (mGBL) engineering model.* Universiti Utara Malaysia.

Baig, M. M. (2009). *New software testing strategy.* NED University of Engineering & Technology, Karachi.

Baudry, B., Fleurey, F., Jézéquel, J.-M., & Le Traon, Y. (2005). Automatic test case optimization: A bacteriologic algorithm. *IEEE software, 22*(2), 76-82.

Bell, D. (2003). UML basics Part III: The class diagram. *The Rational Edge Nov*.

Belli, F., & Hollmann, A. (2008). *Test generation and minimization with basic statecharts.* Paper presented at the Proceedings of the 2008 ACM symposium on Applied computing.

Belli, F., Hollmann, A., & Kleinselbeck, M. (2009). *A graph-model-based testing method compared with the classification tree method for test case generation.* Paper presented at the Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on.

Bentley, J. E. (2005). *Software testing fundamentals-concepts, roles, and terminology.* Paper presented at the Proceedings of SAS Conference.

Berardi, D., Calvanese, D., & De Giacomo, G. (2005). Reasoning on UML class diagrams. *Artificial Intelligence, 168*(1), 70-118. doi: 10.1016/j.artint.2005.05.003

Bernstein, P. A. (1996). Middleware: a model for distributed system services. *Communications of the ACM, 39*(2), 86-98.

Bertolino, A. (2003). *Software testing research and practice.* Paper presented at the Abstract State Machines 2003.

Bertolino, A. (2007). *Software testing research: Achievements, challenges, dreams.* Paper presented at the Future of Software Engineering.

Beynon-Davies, P., Carne, C., Mackay, H., & Tudhope, D. (1999). Rapid application development (RAD): an empirical review. *European Journal of Information Systems, 8*(3), 211-223. doi: 10.1057/palgrave.ejis.3000325

Bhat, S., & Prashanth, C. (2014). A study on automatic test case generation. *International Journal of Engineering Sciences & Research Technology, 3*(4), 4073-4079. doi: 10.1.1.682.1527

Binder, R. V. (2000). *Testing object-oriented systems: models, patterns, and tools.* Massachusetts, United States: Addison-Wesley Longman Publishing Co., Inc.

Biswal, B. N. (2010). *Test case generation and optimization of object-oriented software using UML behavioral models.*

Biswal, B. N., Nanda, P., & Mohapatra, D. P. (2008). *A novel approach for scenario-based test case generation.* Paper presented at the Information Technology, 2008. ICIT'08. International Conference on.

Blanco, R., Fanjul, J., & Tuya, J. (2010). Test case generation for transition-pair coverage using Scatter Search. *International Journal of Software Engineering and Its Applications, 4*(4), 37-56. doi: 10.1.1.233.765

Boehm, B., & Basili, V. R. (2005). Software defect reduction top 10 list. *Foundations of empirical software engineering: the legacy of Victor R. Basili, 426*(426-431). doi: 10.1109/2.962984

Boghdady, P., Badr, N., Hashem, M., & Tolba, M. (2012). *An enhanced technique for generating hybrid coverage test cases using activity diagrams.* Paper presented at the Informatics and Systems (INFOS), 2012 8th International Conference on.

Boghdady, P. N., Badr, N. L., Hashem, M., & Tolba, M. F. (2011a). A proposed test case generation technique based on activity diagrams. *International Journal of Engineering & Technology IJET-IJENS, 11*(03), 37-57. doi: 10.1.1.296.9294

Boghdady, P. N., Badr, N. L., Hashim, M. A., & Tolba, M. F. (2011b). *An enhanced test case generation technique based on activity diagrams.* Paper presented at the Computer Engineering & Systems (ICCES), 2011 International Conference on.

Booch, G. (2005). *The unified modeling language user guide*. London, United Kingdom: Pearson Education.

Bozeman, C., Ellsworth, A., Hogben, L., Lin, J. C.-H., Maurer, G., Nowak, K., . . . Strickland, J. (2015). Minimum rank of graphs with loops. *Electronic Journal of Linear Algebra, 27*(1), 1071. doi: 10.13001/1081-3810.2007

Bozkurt, M., Harman, M., & Hassoun, Y. (2013). Testing and verification in service-oriented architecture: a survey. *Software Testing, Verification and Reliability, 23*(4), 261-313.

Briand, L. C., Labiche, Y., & Cui, J. (2005). Automated support for deriving test requirements from UML statecharts. *Software & Systems Modeling, 4*(4), 399–423. doi: 10.1007/s10270-005-0090-5

Budnik, C. J., Subramanyan, R., & Vieira, M. (2008). Peer-to-Peer Comparison of Model-Based Test Tools. *GI Jahrestagung (1), 133*, 223-226.

Cain, A., Chen, T. Y., Grant, D., Poon, P.-L., Tang, S.-F., & Tse, T. (2003). An automatic test data generation system based on the integrated classification-tree methodology *Software Engineering Research and Applications* (pp. 225-238): Springer.

Calisir, F., & Calisir, F. (2004). The relation of interface usability characteristics, perceived usefulness, and perceived ease of use to end-user satisfaction with enterprise resource planning (ERP) systems. *Computers in Human Behavior, 20*(4), 505-515.

Carmel, E., & Becker, S. (1995). A process model for packaged software development. *Engineering Management, IEEE Transactions on, 42*(1), 50-61.

Cartaxo, E. G., Neto, F. G. O., & Machado, P. D. (2007). *Test case generation by means of UML sequence diagrams and labeled transition Systems.* Paper presented at the Systems, Man and Cybernetics.

Cavarra, A., Crichton, C., Davies, J., Hartman, A., & Mounier, L. (2002). *Using UML for automatic test generation.* Paper presented at the international symposium on software testing and analysis ISSTA.

Chan, E. P., & Lim, H. (2007). Optimization and evaluation of shortest path queries. *The VLDB Journal—The International Journal on Very Large Data Bases, 16*(3), 343-369.

Chavez, H. M., Shen, W., France, R. B., Mechling, B. A., & Li, G. (2016). An approach to checking consistency between UML class model and its Java implementation. *IEEE Transactions on software engineering, 42*(4), 322-344.

Chen, L., & Li, Q. (2010). *Automated test case generation from use case: A model based approach.* Paper presented at the Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on.

Chen, M., Mishra, P., & Kalita, D. (2008). *Coverage-driven automatic test generation for UML activity diagrams.* Paper presented at the Proceedings of the 18th ACM Great Lakes symposium on VLSI, Orlando, Florida, USA.

Chen, M., Qiu, X., Xu, W., Wang, L., Zhao, J., & Li, X. (2009). UML activity diagram-based automatic test case generation for Java programs. *Computer Journal, 52(5)*, 545-556.

Chen, T., Poon, P.-L., & Tse, T. (1999). *A new restructuring algorithm for the classification-tree method.* Paper presented at the Software Technology and Engineering Practice, 1999. STEP'99. Proceedings.

Chimisliu, V., & Wotawa, F. (2012). *Model based test case generation for distributed embedded systems*. Paper presented at the Industrial Technology (ICIT), 2012 IEEE International Conference on.

Chimisliu, V., & Wotawa, F. (2013a). *Improving test case generation from UML statecharts by using control, data and communication dependencies.* Paper presented at the Quality Software (QSIC), 2013 13th International Conference on.

Chimisliu, V., & Wotawa, F. (2013b). *Using dependency relations to improve test case generation from UML statecharts.* Paper presented at the Computer Software and Applications Conference Workshops (COMPSACW), 2013 IEEE 37th Annual.

Choudhary, K., Gigras, Y., & Rani, P. (2016). *Cuckoo Search in Test Case Generation and Conforming Optimality using Firefly Algorithm.* Paper presented at the Proceedings of the Second International Conference on Computer and Communication Technologies.

Chrissis, M. B., Konrad, M., & Shrum, S. (2011). *CMMI for development: guidelines for process integration and product improvement*. London, United Kingdom: Pearson Education.

Claude, J., & Thierry, J. (2002). TGV: theory, principles and algorithms: A Tool for the Automatic Synthesis of Conformance Test Cases for Non-Deterministic Reactive Systems. *Software Tools for Technology Transfer, 7*(4), 297-315.

Costagliola, G., Ferrucci, F., & Francese, R. (2002). Web engineering: Models and methodologies for the design of hypermedia applications. *Handbook of Software Engineering & Knowledge Engineering, 2*, 181-199.

Cruz-Lemus, J. A., Maes, A., Genero, M., Poels, G., & Piattini, M. (2010). The impact of structural complexity on the understandability of UML statechart diagrams. *Information Sciences, 180*(11), 2209-2220.

D'Souza, S., Rao, A., Sharma, A., & Singh, S. (2012). Modeling and verification of a multi-agent argumentation system using NuSMV. *arXiv preprint arXiv:1209.4330*.

Dahiya, S. S., Chhabra, J. K., & Kumar, S. (2010). *Application of artificial bee colony algorithm to software testing.* Paper presented at the Software Engineering Conference (ASWEC), 2010 21st Australian.

Daneva, M., & Ahituv, N. (2011). What practitioners think of inter-organizational ERP requirements engineering practices: focus group results. *International Journal of Information System Modeling and Design, 2*(3), 49-74.

Das, J. (2014). *Bengali digit recognition using adjacency matrix.* Jadavpur University Kolkata.

Dawson, M., Burrell, D. N., Rahim, E., & Brewster, S. (2010). Integrating software assurance into the Software Development Life Cycle (SDLC). *Journal of Information Systems Technology and Planning, 3*(6), 49-53.

Devroey, X., Perrouin, G., Legay, A., Cordy, M., Schobbens, P.-Y., & Heymans, P. (2014). *Coverage criteria for behavioural testing of software product lines.* Paper presented at the International Symposium On Leveraging Applications of Formal Methods, Verification and Validation.

Diestel, R. (2012). *Graph theory*. Berlin, Germany: Springer-Verlag Berlin Heidelberg.

Dinh-Trong, T. T., Ghosh, S., & France, R. B. (2006). *A systematic approach to generate inputs to test UML design models.* Paper presented at the Software Reliability Engineering, 2006. ISSRE'06. 17th International Symposium on.

Dix, A. (2009). *Human-computer interaction*. United States: Springer

Doungsa-ard, C. (2012). *Generation of software test data from the design specification using heuristic techniques. Exploring the UML state machine diagrams and GA based heuristic techniques in the automated generation of software test data and test code.* University of Bradford.

Doungsa-ard, C., Dahal, K., Hossain, A., & Suwannasart, T. (2008). GA-based automatic test data generation for UML state diagrams with parallel paths *Advanced Design and Manufacture to Gain a Competitive Edge* (pp. 147-156): Springer.

Drusinsky, D. (2011). *Modeling and verification using UML statecharts: a working guide to reactive system design, Runtime Monitoring and Execution-based Model Checking*: Elsevier.

Dssouli, R., Saleh, K., Aboulhamid, E., En-Nouaary, A., & Bourhfir, C. (1999). Test development for communication protocols: towards automation. *Computer Networks, 31*(17), 1835-1872.

Dubey, Y., Singh, D., & Singh, A. (2016). A parallel early binding recursive Ant Colony optimization (PEB-RAC) approach for generating optimized auto test cases from programming inputs. *International Journal of Computer Applications, 136*(3), 11-17.

Dustin, E., Garrett, T., & Gauf, B. (2009). *Implementing automated software testing: How to save time and lower costs while raising quality*. London, United Kingdom: Pearson Education.

Easterbrook, S., Singer, J., Storey, M.-A., & Damian, D. (2008). Selecting empirical methods for software engineering research *Guide to advanced empirical software engineering* (pp. 285-311): Springer.

Edvardsson, J. (1999). *A survey on automatic test data generation.* Paper presented at the Proceedings of the 2nd Conference on Computer Science and Engineering.

Eghbali, S., & Tahvildari, L. (2016). Test case prioritization using lexicographical ordering. *IEEE Transactions on software engineering, 42*(12), 1178-1195.

Elallaoui, M., Nafil, K., Touahni, R., & Messoussi, R. (2016). Automated model driven testing using AndroMDA and UML2 testing profile in scrum process. *Procedia Computer Science, 83*, 221-228.

Eshuis, R. (2006). Symbolic model checking of UML activity diagrams. *ACM Transactions on Software Engineering and Methodology (TOSEM), 15*(1), 1-38.

Eusuff, M., Lansey, K., & Pasha, F. (2006). Shuffled frog-leaping algorithm: a memetic meta-heuristic for discrete optimization. *Engineering Optimization, 38*(2), 129-154.

Fan, X., Shu, J., Liu, L., & Liang, Q. J. (2009). *Test case generation from uml subactivity and activity diagram.* Paper presented at the Electronic Commerce and Security, 2009. ISECS'09. Second International Symposium on.

Farooq, S. U., & Quadri, S. (2011). Evaluating effectiveness of software testing techniques with emphasis on enhancing software reliability. *Journal of emerging trends in Computing and Information Sciences, 2*(12), 740-745.

Felderer, M., & Herrmann, A. (2015). Manual test case derivation from UML activity diagrams and state machines: A controlled experiment. *Information and Software Technology, 61*, 1-15.

Felicie, A. L. (2012). *UML state machine*. Rhode Island, United States: Salve Regina University

Frantzen, L., Tretmans, J., & Willemse, T. A. (2006). A symbolic framework for model-based testing *Formal approaches to software testing and runtime verification* (pp. 40-54): Springer.

Fraser, G., & Wotawa, F. (2007). *Test-case generation and coverage analysis for nondeterministic systems using model-checkers.* Paper presented at the Software Engineering Advances, 2007. ICSEA 2007. International Conference on.

Garousi, V. (2010). Applying peer reviews in software engineering education: an experiment and lessons learned. *IEEE Transactions on Education, 53*(2), 182-193. doi: 10.1109/TE.2008.2010994

Ghai, S., & Kaur, S. (2017). Hill-Climbing Approach for Test Case Prioritization. *International Journal of Software Engineering and Its Applications, 11*(3), 13-20.

Gnesi, S., Latella, D., & Massink, M. (2004). *Formal test case generation for UML statecharts.* Paper presented at the Engineering Complex Computer Systems, 2004. Proceedings. Ninth IEEE International Conference on.

Gogolla, M., Hamann, L., Hilken, F., Sedlmeier, M., & Nguyen, Q. D. (2014). *Behavior modeling with interaction diagrams in a UML and OCL tool.* Paper presented at the Proceedings of the 2014 Workshop on Behaviour Modelling-Foundations and Applications.

Goodubaigari, A. (2013). A software test data generation tool for unit testing of C++ programs using control flow graph. *IJECS*, 2388-2392.

Gotlieb, A. (2012). TCAS software verification using constraint programming. *The Knowledge Engineering Review, 27*(03), 343-360.

Grant, E. S., & Datta, T. (2016). Modeling RTCA DO-178C Specification to Facilitate Avionic Software System Design, Verification, and Validation. *International Journal of Future Computer and Communication, 5*(2), 120.

Gries, D., & Schneider, F. B. (2005). *An integrated approach to software engineering*. Kanpur, India: Pankaj Jalote. Indian Institute of Technology

Gross, H.-G. (2005). *Component-based software testing with UML*: Springer.

Gulia, P., & Chillar, R. S. (2012). A new approach to generate and optimize test cases for UML state diagram using genetic algorithm: http://doi.acm.org/10.1145/180921.2180933. *SIGSOFT Softw. Eng. Notes, 37*(3), 1-5. doi: 10.1145/180921.2180933

Gulia, P., & Chugh, J. (2015). Comparative analysis of traditional and object-oriented software testing. *ACM SIGSOFT Software Engineering Notes, 40*(2), 1-4.

Gupta, J. (2014). *An investigation of test cases generation from activity diagram.* Thapar University Patiala.

Håkansson, J., & Mokrushin, L. (2004). *An analysis tool for UML models with SPT annotations.* Paper presented at the Nordic Workshop on Programming Theory.

Hallowell, M. R., & Gambatese, J. A. (2009). Qualitative research: Application of the Delphi method to CEM research. *Journal of construction engineering and management, 136*(1), 99-107.

Han, S.-H., & Kwon, Y.-R. (2008). An empirical evaluation of test data generation techniques. *Journal of Computing Science and Engineering, 2*(3), 275-300.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of computer programming, 8*(3), 231-274.

Hartmann, J., Imoberdorf, C., & Meisinger, M. (2000). *UML-based integration testing.* Paper presented at the ACM SIGSOFT Software Engineering Notes.

Hashim, N. L., & Salman, Y. D. (2011). *An improved algorithm in test case generation from UML activity diagram using activity path.* Paper presented at the Proceedings of the 3rd International Conference on Computing and Informatics, ICOCI, Bandung, Indonesia

Hashmi, A., Goel, N., Goel, S., & Gupta, D. (2013). Firefly algorithm for unconstrained optimization. *IOSR J Comput Eng, 11*(1), 75-78.

Hedjazi, S. M., & Marjani, S. S. (2010). *Pruned genetic algorithm.* Paper presented at the International Conference on Artificial Intelligence and Computational Intelligence.

Hessel, A. (2006). *Model-based test case selection and generation for real-time systems.* (PhD Dissertation), Uppsala University.

Heumann, J. (2001). Generating test cases from use cases. *The rational edge, 6*(1).

Holbrook, A. L., Krosnick, J. A., Moore, D., & Tourangeau, R. (2007). Response order effects in dichotomous categorical questions presented orally: The impact of question and respondent attributes. *Public Opinion Quarterly, 71*(3), 325-348.

Hong, H. S., & Ural, H. (2004). *Using model checking for reducing the cost of test generation.* Paper presented at the International Workshop on Formal Approaches to Software Testing.

Hooda, I., & Chhillar, R. (2014). A review: study of test case generation techniques. *International Journal of Computer Applications, 107*(16), 33- 37.

Hopper, G. M. (1981). The first bug. *Annals of the History of Computing, 3*(3), 285-286.

Ibrar, M. (2013). UML diagrams: an aid to database design specification: a review. *International Journal of Advanced Research in Computer Science and Software Engineering, 3*(3), 598 -602.

Inamdar, Y. (2015). Airport check-in of passenger. Retrieved Documents, 2016, from http://docslide.us/documents/airport-check-in-of-passenger.html

Ingle, S., & Mahamune, M. (2015). An UML based software automatic test case generation: survey. *International Research Journal of Engineering and Technology, 2*(2), 971-973.

Jain, E. S., & Sheikh, E. M. (2014). A novel test case generation method through metamorphic priority for 2-way testing method UMBCA implementation criteria. *International Journal of Engineering and Management Research, 4*(3), 157 -163.

Jain, R. (1990). *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. New Jersey, United States: John Wiley & Sons.

Javed, A. Z., Strooper, P. A., & Watson, G. (2007). *Automated generation of test cases using model-driven architecture.* Paper presented at the Automation of Software Test, 2007. AST'07. Second International Workshop on Automation of Software Test.

Javed, M., Ahmad, B., Abbas, Z., Nawaz, A., Abid, M. A., & Ullah, I. (2012). Decreasing defect rate of test cases by designing and analysis for recursive modules of a program structure: Improvement in test cases. *International Journal of Computer Science and Information Security, 10*(8).

Jena, A. K., Swain, S. K., & Mohapatra, D. P. (2014). *A novel approach for test case generation from UML activity diagram.* Paper presented at the Issues and Challenges in Intelligent Computing Techniques (ICICT), 2014 International Conference on.

Jena, A. K., Swain, S. K., & Mohapatra, D. P. (2015). Test case creation from UML sequence diagram: a soft computing approach *Intelligent Computing, Communication and Devices* (pp. 117-126): Springer.

Jia, X., & Liu, H. (2002). *Rigorous and automatic testing of web applications.* Paper presented at the Proceedings of the 6th IASTED International Conference on Software Engineering and Applications (SEA 2002).

Jia, X., Liu, H., & Qin, L. (2003). *Formal structured specification for web application testing.* Paper presented at the Midwest Software Engineering Conference.

Joo, S., Lin, S., & Lu, K. (2011). A usability evaluation model for academic library websites: efficiency, effectiveness and learnability. *Journal of Library and Information studies, 9*(2), 11-26.

Jorgensen, P. C. (2013). *Software testing: a craftsman's approach*. Boca Raton, Florida, United States: CRC press.

Jürjens, J. (2005). *Secure systems development with UML*. Berlin, Germany: Springer Science & Business Media.

Kaner, C., Falk, J., & Nguyen, H. Q. (1999). *Testing computer software*. India: Dreamtech Press.

Kaner, C., & Fiedler, R. L. (2013). *Foundations of Software Testing*. Massachusetts, United States: Context-Driven Press.

Kang, H., Lee, S., Lee, C., Yoon, C., & Shin, S. (2015). *SPIRIT: A framework for profiling SDN*. Paper presented at the Network Protocols (ICNP), 2015 IEEE 23rd International Conference on.

Kangas, K.-M. (2008). *Test automation of digital mammography device.* Helsinki Polytechnic Stadia.

Kansomkeat, S., Offutt, J., Abdurazik, A., & Baldini, A. (2008). *A comparative evaluation of tests generated from different UML diagrams.* Paper presented at the Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2008. SNPD'08. Ninth ACIS International Conference on.

Kansomkeat, S., & Rivepiboon, W. (2003). *Automated generating test case using UML statechart diagrams.* Paper presented at the Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology.

Kansomkeat, S., Thiket, P., & Offutt, J. (2010). *Generating test cases from UML activity diagrams using the condition-classification tree method.* Paper presented at the Software Technology and Engineering (ICSTE), 2010 2nd International Conference on.

Karambir, & Kaur, K. (2013). Performance analysis of Test Generation Techniques. *International Journal of Advanced Research in Computer Science and Software Engineering, 3*(7), 490-498.

Karambir, & Kuldeep, K. (2013). Survey of software test case generation techniques. *International Journal of Advanced Research in Computer Science and Software Engineering*, 937-942.

Kaur, A., & Harwinder, S. S. (2013). Automatic test case generation with SilK testing. *International Journal of Computer Applications, 79*(15), 32-34.

Kaur, G., & Singh, P. (2015). Test Case Generation Using UML Diagram. *International Journal of Emerging Technologies in Engineering Research, 1*(2), 23- 25.

Kaur, P., & Gupta, G. (2013). Automated model-based test path generation from UML diagrams via graph coverage techniques. *International Journal of Computer Science and Mobile Computing, 2*(7), 302-311.

Kavita, C., Shilpa, Yogita, G., Payal, R., & Akshath, G. (2015). A Survey Paper on Test Case Generation and Optimization: Cuckoo Search and Firefly Algorithm. *IJEDR, 3*(2), 584-589.

Keen, P. G. (1980). Decision support systems: a research perspective. *Decision Support Systems: Issues and Challenges*, 23-44.

Kelly, D. (1999). Software test automation and the product life cycle. *Mactech Magazine, 13*(10).

Kerlinger, F. N. (1986). *Foundations of behavioral research*. Orlando, Florida, United States: Holt, Rinehart and Winston.

Kernschmidt, K., & Vogel-Heuser, B. (2013). *An interdisciplinary SysML based modeling approach for analyzing change influences in production plants to support the engineering.* Paper presented at the Automation Science and Engineering (CASE), 2013 IEEE International Conference on.

Khandai, M., Acharya, A. A., & Mohapatra, D. P. (2011). A survey on test case generation from UML model. *International Journal of Computer Science and Information Technologies, 2*(3), 1164-1171.

Khurana, N., & Chillar, R. (2015). Test case generation and optimization using UML models and genetic algorithm. *Procedia Computer Science, 57*, 996-1004.

Kim, H., Kang, S., Baik, J., & Ko, I. (2007). *Test cases generation from UML activity diagrams.* Paper presented at the Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007).

Kim, J. M., Porter, A., & Rothermel, G. (2005). An empirical study of regression test application frequency. *Software Testing, Verification and Reliability, 15*(4), 257-279.

Kim, S., Lively, W. M., & Simmons, D. B. (2006). *An Effort Estimation by UML Points in Early Stage of Software Development.* Paper presented at the Software Engineering Research and Practice.

Kim, W. Y., Son, H. S., & Kim, R. Y. C. (2011). A study on test case generation based on state diagram in modeling and simulation environment *Advanced Communication and Networking* (pp. 298-305): Springer.

Kim, Y. G., Hong, H. S., Bae, D.-H., & Cha, S.-D. (1999). Test cases generation from UML state diagrams *IEE Proceedings-Software, 146*(4), 187-192. doi: 10.1049/ip-sen:19990602

Knaak, N., & Page, B. (2005). *UML ¾ as a Modelling Language in Discrete Event Simulation.* Paper presented at the 9th european conference on modelling and simulation.

Komuro, M., & Komoda, N. (2008). *An explanation model for quality improvement effect of peer reviews.* Paper presented at the International Conference on Computational Intelligence for Modelling Control & Automation.

Konstantinou, P. (2013). Rapid application development. *Retrieved April, 1*.

Koong, C.-S., Shih, C., Hsiung, P.-A., Lai, H.-J., Chang, C.-H., Chu, W. C., . . . Yang, C.-T. (2012). Automatic testing environment for multi-core embedded software - ATEMES. *Journal of Systems and Software, 85*(1), 43-60.

Korel, B. (1990). Automated software test data generation. *IEEE Transactions on software engineering, 16*(8), 870-879. doi: 10.1109/32.57624

Kosindrdecha, N., & Daengdej, J. (2010). A test generation method based on state diagram. *Journal of Theoretical and Applied Information Technology*, 28-44.

Kot, M. (2003). The state explosion problem. *Retrieved May, 18*, 2015.

Krishnachandra, M. (2016). A customized approach for automated test case generation and optimization for system based software testing. *international Journal of Emerging Trends & Technology in Computer Science, 5*(2), 36-39.

Kulkarni, N. J., Naveen, K. V., Singh, P., & Srivastava, P. R. (2011). Test case optimization using artificial bee colony algorithm. *Advances in Computing and Communications*, 570-579.

Kull, A. (2009). *Model-based testing of reactive systems.* (PhD Dissertation), Tallinn University of Technology, Tallinn, Estonia.

Kumar, V. K., & Mathew, S. (2014). Compiler based test case generation. *International Journal on Recent Trends in Engineering & Technology, 11*(1), 558.

Kumaran, U. S., Kumar, S. A., & Kumar, K. V. (2011). An approach to automatic generation of test cases based on use cases in the requirements phase *International Journal on Computer Science and Engineering, 3*(1), 102-113.

Kundu, D., & Samanta, D. (2009). A novel approach to generate test cases from UML activity diagrams. *Journal of Object Technology, 8*(3), 65-83.

Kusumoto, S., Matukawa, F., Inoue, K., Hanabusa, S., & Maegawa, Y. (2005). Effort estimation tool based on use case points method. *Osaka University*.

Kwiecień, J., & Filipowicz, B. (2012). Firefly algorithm in optimization of queueing systems. *Bulletin of the Polish Academy of Sciences: Technical Sciences, 60*(2), 363-368.

Lam, S. S. B., Raju, M. H. P., Ch, S., & Srivastav, P. R. (2012). Automated generation of independent paths and test suite optimization using artificial bee colony. *Procedia Engineering, 30*, 191-200.

Lammich, P., & Neumann, R. (2015). *A framework for verifying depth-first search algorithms.* Paper presented at the Proceedings of the 2015 Workshop on Certified Programs and Proofs.

Lange, C. F., Chaudron, M. R., & Muskens, J. (2006). In practice: UML software architecture and design description. *IEEE software, 23*(2), 40-46. doi: 10.1109/MS.2006.50

Lauder, A., & Kent, S. (2001). Statecharts for Business Process Modeling *Enterprise Information Systems II* (pp. 121-125): Springer.

Lavagno, L., Markov, I. L., Martin, G., & Scheffer, L. K. (2016). *Electronic Design Automation for Ic System Design, Verification, and Testing*. United States: CRC Press.

Leitner, A., Oriol, M., Zeller, A., Ciupa, I., & Meyer, B. (2007). *Efficient unit test case minimization.* Paper presented at the Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering.

Li, B.-L., Li, Z.-s., Qing, L., & Chen, Y.-H. (2007). *Test case automate generation from UML sequence diagram and OCL expression.* Paper presented at the Computational Intelligence and Security, 2007 International Conference.

Li, H., & Lam, C. P. (2005). *An ant colony optimization approach to test sequence generation for state-based software testing.* Paper presented at the Quality Software, 2005.(QSIC 2005). Fifth International Conference

Li, L., He, T., & Wu, J. (2012). Automatic test Generation from UML statechart diagram based on euler circuit. *International Journal of Digital Content Technology & its Applications, 6*(19), 129-136.

Li, L., Li, X., He, T., & Xiong, J. (2013a). Extenics-based test case generation for UML activity diagram. *Procedia Computer Science, 17*, 1186-1193.

Li, L., Li, X., Tan, S., & Xiong, J. (2013b). Generating test cases from UML statechart diagram based on extended context-free grammars. *International Journal of Digital Content Technology and its Applications, 7*(5), 1206.

Lilly, R., & Uma, G. V. (2010). Reliable Mining of Automatically Generated Test Cases from Software Requirements Specification. *IJCSI international journal of computer science issues, 7*(1), 87-91.

Lima, B., & Faria, J. P. (2016). *A Survey on Testing Distributed and Heterogeneous Systems: The State of the Practice.* Paper presented at the International Conference on Software Technologies.

Linzhang, W., Jiesong, Y., Xiaofeng, Y., Jun, H., Xuandong, L., & Guoliang, Z. (2004). *Generating test cases from UML activity diagram based on Gray-box method*. Paper presented at the Software Engineering Conference.

Lithner, J. (2008). A research framework for creative and imitative reasoning. *Educational Studies in Mathematics, 67*(3), 255-276.

Liu, J., & Zhang, L. (2014). Using Formal Methods and Aspect Oriented Techniques to Model Cyber Physical Systems. *International Information Institute (Tokyo). Information, 17*(5), 1729.

Lu, M.-S., & Tseng, L.-K. (2010). An integrated object-oriented approach for design and analysis of an agile manufacturing control system. *The International Journal of Advanced Manufacturing Technology, 48*(9), 1107-1122.

Makker, V., & Singh, V. (2011). An Approach for Test Case Generation Using UML State chart Diagram. *International Journal of Advanced Research in Computer Science, 2*(5), 567 - 571.

Mala, D. J., Kamalapriya, M., Shobana, R., & Mohan, V. (2009). *A non-pheromone based intelligent swarm optimization technique in software test suite optimization.* Paper presented at the Intelligent Agent & Multi-Agent Systems, 2009. IAMA 2009. International Conference on.

Mala, D. J., & Mohan, V. (2009). ABC tester-artificial bee colony based software test suite optimization approach. *International Journal of Software Engineering, 2*(2), 15-43.

Mala, D. J., & Mohan, V. (2010). Quality improvement and optimization of test cases: a hybrid genetic algorithm based approach. *ACM SIGSOFT Software Engineering Notes, 35*(3), 1-14.

Mala, D. J., Ruby, E., & Mohan, V. (2012). A hybrid test optimization framework-coupling genetic algorithm with local search technique. *Computing and Informatics, 29*(1), 133-164.

Mall, R. (2009). *Fundamentals of software engineering*. New Delhi, India: PHI Learning Pvt. Ltd.

Mani, P., & Prasanna, M. (2016). Test Case Generation for Embedded System Software Using UML Interaction Diagram. *Journal of Engineering Science and Technology, 12*(4), 860 - 874.

March, S. T., & Smith, G. F. (1995). Design and natural science research on information technology. *Decision support systems, 15*(4), 251-266.

Marijuán, P. C., & Westley, J. (1992). Enzymes as molecular automata: a reflection on some numerical and philosophical aspects of the hypothesis. *BioSystems, 27*(2), 97-113.

Martin, J. (1991). *Rapid application development*. Basingstoke, United Kingdom: Macmillan Publishing Company.

Mathur, A. P. (2008). *Foundations of Software Testing, 2/e*. London, United Kingdom: Pearson Education.

McCaffrey, J. D. (2009). *Generation of pairwise test sets using a genetic algorithm.* Paper presented at the Computer Software and Applications Conference, 2009. COMPSAC'09. 33rd Annual IEEE International.

McMinn, P. (2004). Search-based software test data generation: a survey. *Software Testing, Verification and Reliability, 14*(2), 105-156.

McQuillan, J. A., & Power, J. F. (2005). A survey of UML-based coverage criteria for software testing. *Department of Computer Science. NUI Maynooth, Co. Kildare, Ireland*, 1 -17.

Miller, T., Padgham, L., & Thangarajah, J. (2010). *Test coverage criteria for agent interaction testing.* Paper presented at the International Workshop on Agent-Oriented Software Engineering.

Mingsong, C., Qiu, X., Xu, W., Wang, L., Zhao, J., & Li, X. (2009). UML activity diagram-based automatic test case generation for Java programs. *THE COMPUTER JOURNAL, 52*(5), 545-556.

Mingsong, C., Xiaokang, Q., & Xuandong, L. (2006). *Automatic test case generation for UML activity diagrams.* Paper presented at the Proceedings of the 2006 international workshop on Automation of software test.

Mohamed, S. F. P. (2015). *A process based approach software certification model for agile and secure environment.* Universiti Utara Malaysia.

Mohi-Aldeen, S. M., Mohamad, R., & Deris, S. (2014). *Automatic test case generation for structural testing using negative selection algorithm.* Paper presented at the 1st International Conference of Recent Trends in Information and Communication Technologies.

Morell, L. J. (1984). *A Theory of Error-based Testing.* (PhD Dissertation), University of Maryland at College Park.

Morell, L. J. (1990). A theory of fault-based testing. *Software Engineering, IEEE Transactions on, 16*(8), 844–857.

Moret, B. M., & Shapiro, H. D. (2001). Algorithms and experiments: The new (and old) methodology. *Journal of Universal Computer Science, 7*(5), 434-446.

Muniz, L. L., Netto, U. S., & Maia, P. H. M. (2015). *TCG-a model-based testing tool for functional and statistical testing.* Paper presented at the ICEIS.

Murthy, P., Anitha, P., Mahesh, M., & Subramanyan, R. (2006). *Test ready UML statechart models.* Paper presented at the Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools.

Na, H.-S., Choi, O.-H., & Lim, J.-E. (2006). *A metamodel-based approach for extracting ontological semantics from UML models*. Paper presented at the International Conference on Web Information Systems Engineering.

Naik, K., & Tripathy, P. (2011). *Software testing and quality assurance: theory and practice*. New Jersey, United States: John Wiley & Sons.

Nayak, A., & Samanta, D. (2010). Automatic test data synthesis using UML sequence diagrams. *Journal of Object Technology, 9*(2), 75-104.

Ngah, A. (2012). *Regression test selection by exclusion.* (PhD thesis), Durham University.

Nidhra, S., & Dondeti, J. (2012). Blackbox and whitebox testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA), 2*(2), 29-50.

Norshuhada, S., & Shahizan, H. (2010). Design research in software development: Constructing and linking research questions, objectives, methods and outcomes: Sintok: Penerbit Universiti Utara Malaysia.

O'Neil, D. (2001). *Peer reviews, encyclopedia of software engineering*. New York, United States: Wiley.

Offermann, P., Levina, O., Schönherr, M., & Bub, U. (2009). *Outline of a design science research process.* Paper presented at the Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology.

Offutt, A. J. (1988). *Automatic test data generation.* (PhD), Georgia Institute of Technology, Atlanta, GA, USA.

Offutt, J., & Abdurazik, A. (1999). *Generating tests from UML specifications*: Springer.

Offutt, J., Liu, S., Abdurazik, A., & Ammann, P. (2003). Generating test data from state-based specifications. *Software Testing, Verification and Reliability, 13*(1), 25-53.

Oladejo, B. F., & Ogunbiyi, D. T. (2014). An empirical study on the effectiveness of automated test case generation techniques. *American Journal of Software Engineering and Applications, 3*(6), 95-101.

Olson, K. (2010). An examination of questionnaire evaluation by expert reviewers. *Field Methods, 22*(4), 295-318.

Oluwagbemi, O., & Asmuni, H. (2014). Development of a robust parser for extracting artifacts during model-based testing from UML diagrams. *International Journal of Software Engineering and Technology, 1*(2), 43-50.

Oluwagbemi, O., & Asmuni, H. (2015). Automatic generation of test cases from activity diagrams for UML based testing (UBT). *Jurnal Teknologi, 77*(13).

Omotunde, H., Ibrahim, R., Ahmed, M., Olanrewaju, R., Ibrahim, N., & Shah, H. (2016). A framework to reduce redundancy in android test suite using

refactoring. *Indian Journal of Science and Technology, 9*(46). doi: 10.17485/ijst/2016/v9i46/107107

Ooi, W., Shahrizal, I., Noordin, A., Nurulain, M., & Norhan, M. (2014). *Development of rural emergency medical system (REMS) with geospatial technology in Malaysia.* Paper presented at the IOP Conference Series: Earth and Environmental Science.

Ostrand, T. J., & Balcer, M. J. (1988). The category-partition method for specifying and generating fuctional tests. *Communications of the ACM, 31*(6), 676-686.

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

Pandey, B., & Jain, R. (2014). Importance of unified modelling language for test case generation in software testing. *Rituraj Jain et al, Int.J.Computer Technology & Applications, 5*(2), 345-350.

Panthi, V., & Mohapatra, D. (2015). Generating prioritized test sequences using Firefly optimization technique *Computational Intelligence in Data Mining-Volume 2* (pp. 627-635): Springer.

Panthi, V., & Mohapatra, D. P. (2012). *Automatic test case generation using sequence diagram.* Paper presented at the Proceedings of International Conference on Advances in Computing.

Parnami, S. (2013). Testing target path by automatic generation of test data using genetic algorithm. *International Journal of Information and Computation Technology, 3*(8), 825-832.

Patnaik, D., Acharya, A. A., & Mohapatra, D. P. (2011). Generating testcases for concurrent systems using UML state chart diagram *Information Technology and Mobile Communication* (pp. 100-105): Springer.

Patwa, S., & Malviya, A. K. (2014). Impact of coding phase on object oriented software testing. *Covenant Journal of Informatics and Communication Technology (CJICT), 2*(1), 57-67.

Paul, A., & Jeff, O. (2008). *Introduction to Software Testing*. New York, United States: Cambridge University Press.

Perry, W. E. (2007). *Effective Methods for Software Testing: Includes Complete Guidelines, Checklists, and Templates*. New Jersey, United States: John Wiley & Sons.

Pilskalns, O., Andrews, A., Ghosh, S., & France, R. (2003). Rigorous testing by merging structural and behavioral UML representations *The Unified Modeling Language. Modeling Languages and Applications* (pp. 234-248): Springer.

Pimenta, A. (2006). *Automated specification based testing of graphical user interfaces.* (PhD Thesis), Porto University, Porto, Portugal.

Pinheiro, A. C., Simão, A., & Ambrosio, A. M. (2014). FSM-based test case generation methods applied to test the communication software on board the ITASAT University Satellite: A Case Study. *Journal of Aerospace Technology and Management, 6*(4), 447-461.

Popp, R., Falb, J., Arnautovic, E., Kaindl, H., Kavaldjian, S., Ertl, D., . . . Bogdan, C. (2009). *Automatic generation of the behavior of a user interface from a high-level discourse model.* Paper presented at the System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on.

Prasanna, M., & Chandran, K. (2011). Automated Test Case Generation for Object Oriented Systems Using UML Object Diagrams *High Performance Architecture and Grid Computing* (pp. 417-423): Springer.

Prasanna, M., Chandran, K., & Suberi, D. B. (2011). Automatic test case generation for UML class diagram using data flow approach. *Academia Education*, 1-7.

Prasanna, M., Sivanandam, S., Venkatesan, R., & Sundarrajan, R. (2005). A survey on automatic test case generation. *Academic Open Internet Journal, 15*(6).

Presser, S., & Blair, J. (1994). Survey pretesting: Do different methods produce different results? *Sociological methodology, 24*, 73-104.

Priya, S. S., & Sheba, P. (2013). *Test case generation from UML models-a survey.* Paper presented at the Proc. International Conference on Information Systems and Computing (ICISC-2013), INDIA.

Rafi, D. M., Moses, K. R. K., Petersen, K., & Mäntylä, M. V. (2012). *Benefits and limitations of automated software testing: Systematic literature review and practitioner survey.* Paper presented at the Proceedings of the 7th International Workshop on Automation of Software Test.

Rapos, E. (2012). *Understanding the effects of model evolution through incremental test case generation for UML-RT models.* Queen's University.

Rhmann, W., & Saxena, V. (2016). Optimized and prioritized test paths generation from UML activity diagram using firefly algorithm. *International Journal of Computer Applications, 145*(6), 16-22.

Rice, R. W. (2010). STBC: the economics of testing. http://www.riceconsulting.com/public_pdf/STBC-WM.pdf.

Robinson, H. (1999). *Graph theory techniques in model-based testing.* Paper presented at the International Conference on Testing Computer Software.

Rothermel, G., Untch, R. H., Chu, C., & Harrold, M. J. (2001). Prioritizing test cases for regression testing. *IEEE Transactions on software engineering, 27*(10), 929-948.

Rumbaugh, J., Jacobson, I., & Booch, G. (2004). *Unified Modeling Language Reference Manual, The*: Pearson Higher Education.

Rungi, K., & Matulevičius, R. (2013). Empirical analysis of the test maturity model integration (TMMi) *Information and Software Technologies* (pp. 376-391): Springer.

Ruohonen, K. (2013). *Graph theory*. Tampere, Finland: Tampere University of Technology.

Sahoo, R. K., Mohapatra, D. P., & Patra, M. R. (2016a). A firefly algorithm based approach for automated generation and optimization of test cases. *International Journal on Computer Science and Engineering, 4*(8), 54-58.

Sahoo, R. K., Ojha, D., Mohapatra, D. P., & Patra, M. R. (2016b). Automated test case generation and optimization: a comparative review. *International Journal of Computer Science & Information Technology, 8*(5), 19-32.

Saifan, A. A., & Mustafa, W. B. (2015). Using formal methods for test case generation according to transition-based coverage criteria. *Jordanian Journal of Computers and Information Technology, 1*(1), 15-30.

Saini, E. S., & Srivastava, E. V. (2015). Case Generation from the Combination of UML Class and Activity Diagrams. *International Journal Of Modern Engineering Research, 5*(7), 10-13.

Salah, D., Paige, R., & Cairns, P. (2014). *An evaluation template for expert review of maturity models.* Paper presented at the International Conference on Product-Focused Software Process Improvement.

Salman, Y. D., & Hashim, N. L. (2014). An improved method of obtaining basic path testing for test case based on UML state chart. *Science International, 26*(4), 1607 - 1610.

Salman, Y. D., & Hashim, N. L. (2016). Automatic Test Case Generation from UML State Chart Diagram: A Survey *Advanced Computer and Communication Engineering Technology* (pp. 123-134): Springer.

Salman, Y. D., & Hashim, N. L. (2017). Test Case Generation Model for UML Diagrams. *Journal of Telecommunication, Electronic and Computer Engineering (JTEC), 9*(2-2), 171-175.

Salman, Y. D., Hashim, N. L., Rejab, M. M., Romli, R., & Mohd, H. (2017). Coverage Criteria for UML State Chart Diagram in Model-based Testing. *Journal of Telecommunication, Electronic and Computer Engineering (JTEC), 9*(2-11), 85-89.

Samuel, P., Mall, R., & Bothra, A. K. (2008). Automatic test case generation using unified modeling language (UML) state diagrams. *IET software, 2*(2), 79-93.

Santiago, V., do Amaral, A. S. M., Vijaykumar, N., Mattiello-Francisco, M. F., Martins, E., & Lopes, O. C. (2006). *A practical approach for automated test case generation using statecharts.* Paper presented at the Computer Software

and Applications Conference, 2006. COMPSAC'06. 30th Annual International.

Santiago, V., Vijaykumar, N. L., Guimarães, D., Amaral, A. S., & Ferreira, É. (2008). *An environment for automated test case generation from statechart-based and finite state machine-based behavioral models.* Paper presented at the Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on.

Santosh, M., & Singh, R. (2013). Test Case Minimization By Generating Requirement Based Mathematical Equations. *International Journal of Engineering Research & Technology (IJERT), 2*(6), 1180 - 1188.

Sapna, P., & Mohanty, H. (2008). *Automated Scenario Generation Based on UML Activity Diagrams*. Paper presented at the International Conference on Information Technology.

Schligloff, H., & Roggenbach, M. (2002). Path testing. *Advanced Topics in Computer Science*.

Schroeder, P. J., Kim, E., Arshem, J., & Bolaki, P. (2003). *Combining behavior and data modeling in automated test case generation.* Paper presented at the Quality Software, 2003. Proceedings. Third International Conference on.

Schwarzl, C., & Peischl, B. (2010a). Static-and dynamic consistency analysis of UML state chart models *Model Driven Engineering Languages and Systems* (pp. 151-165): Springer.

Schwarzl, C., & Peischl, B. (2010b). *Test sequence generation from communicating UML state charts: An industrial application of symbolic transition systems.* Paper presented at the Quality Software (QSIC), 2010 10th International Conference.

Schweighofer, T., & Heričko, M. (2014). *Approaches for test case generation from UML diagrams.* Paper presented at the Third Workshop on Software Quality Analysis, Monitoring, Improvement and Applications.

Shah, S. A. A., Shahzad, R. K., Bukhari, S. S. A., Minhas, N. M., & Humayun, M. (2016). A Review of Class Based Test Case Generation Techniques. *Journal of Software, 11*(5), 464-480.

Shahzad, A., Raza, S., Azam, M. N., Bilal, K., & Shamail, S. (2009). *Automated optimum test case generation using web navigation graphs.* Paper presented at the Emerging Technologies, 2009. ICET 2009. International Conference on.

Shamshiri, S., Just, R., Rojas, J. M., Fraser, G., McMinn, P., & Arcuri, A. (2015). *Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t).* Paper presented at the Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on.

Shamsoddin-Motlagh, E. (2012). A review of automatic test cases generation. *International Journal of Computer Applications, 57*(13), 25 - 29.

Shanthi, A., & Kumar, G. M. (2012). Automated test cases generation from UML sequence diagram. *International Proceedings of Computer Science & Information Technology, 41*, 83 -89.

Sharma, P. (2014). Automated software testing using metahurestic technique based on improved ant algorithms for software testing. *International Journal on Recent and Innovation Trends in Computing and Communication, 2*(11).

Sharma, R., & PrakashSonwani, S. (2015). Programmed test case generation from simulink/stateflow model. *Indian Journal of Computer Science and Engineering (IJCSE), 6*(2), 45 - 51.

Sharp, H., Rogers, Y., & Preece, J. (2007). Interaction design: beyond human-computer interaction. *netWorker: The Craft of Network Computing, 11*(4), 34.

Shen, J., & Abraham, J. A. (2000). An RTL abstraction technique for processor microarchitecture validation and test generation. *Journal of Electronic Testing, 16*(1), 67-81.

Sherwood, C., & Rout, T. (1998). *A structured methodology for multimedia product and systems development.* Paper presented at the ASCILITE.

Shinde, V. (2013). *Software testing career package - a software tester's journey from getting a job to becoming a test leader!* : Software Testing Help.

Shiratuddin, N., Hassan, S., Hashim, N. L., Sarif, S. M., Bakar, A., & Shahbani, M. (2013). Focus group evaluation on IPTComKitTM commercialization model. *Recent Advances in Electrical and Computer Engineering*, 90-95.

Shirole, M., & Kumar, R. (2010). A hybrid genetic algorithm based test case generation using sequence diagrams *Contemporary Computing* (pp. 53-63): Springer.

Shirole, M., & Kumar, R. (2013). UML Behavioral Model Based Test Case Generation: A Survey. *ACM SIGSOFT Software Engineering Notes, 38*(4), 1-13.

Shirole, M., Suthar, A., & Kumar, R. (2011). *Generation of improved test cases from UML state diagram using genetic algorithm.* Paper presented at the Proceedings of the 4th India Software Engineering Conference.

Shneiderman, S. B., & Plaisant, C. (2005). *Designing the User Interface: Strategies for Effective Human-Computer Interaction (4th Edition).* United States: Pearson Addison Wesley.

Shull, F., Rus, I., & Basili, V. (2000). How perspective-based reading can improve requirements inspections. *Computer, 33*(7), 73-79.

Singh, R. (2014). *Test case generation for object-oriented systems: A review.* Paper presented at the Communication Systems and Network Technologies (CSNT), 2014 Fourth International Conference on.

Singh, S., & Shree, R. (2016). A combined approach to optimize the test suite size in regression testing. *CSI transactions on ICT, 4*(2-4), 73-78.

Society, I. C. (2014). *Guide to the software engineering body of knowledge (SWEBOK Version 3)*: IEEE.

Sommerville, I. (2011). *Software engineering* (9th ed.). Massachusetts, United States: Addison-Wesley.

Sood, B., & Rattan, D. (2016). An efficient method to generate automation scripts using selenium tool. *An International Journal of Engineering Sciences, 17*, 1 - 7.

Specification, O. A. (2007). OMG unified modeling language (OMG UML), Superstructure, V2. 1.2. *Object Management Group, 2*(12).

Sprague Jr, R. H., & Carlson, E. D. (1982). *Building effective decision support systems*. New Jersey, United States: Prentice Hall Professional Technical Reference.

Srikant, Y., & Shankar, P. (2007). *The compiler design handbook: optimizations and machine code generation*. Cambridge, United Kingdom: CRC Press.

Srivastav, S., & Gupta, S. (2016). Software design pattern static validation using cyclomatic complexity and UML approach. *International Journal, 4*(7), 89 - 97.

Srivastava, P. R., Baby, K., & Raghurama, G. (2009). *An approach of optimal path generation using ant colony optimization.* Paper presented at the TENCON 2009-2009 IEEE Region 10 Conference.

Srivastava, P. R., & Kim, T.-h. (2009). Application of genetic algorithm in software testing. *International Journal of Software Engineering and Its Applications, 3*(4), 87-96.

Srivatsava, P. R., Mallikarjun, B., & Yang, X.-S. (2013). Optimal test sequence generation using firefly algorithm. *Swarm and Evolutionary Computation, 8*, 44-53.

Srividhya, J., & Alagarsamy, K. (2014). A synthesized overview of test case optimization techniques. *Journal of Recent Research in Engineering and Technology, 1*(2).

Stecklein, J., Dabney, J., Dick, B., Haskins, B., Lovell, R., & Moroney, G. (2004). Error cost escalation through the project life cycle. *National Aeronautics and Space Administration*.

Stewart, D. W., & Shamdasani, P. N. (2014). *Focus groups: Theory and practice* (Vol. 20). California, United States: Sage Publications.

Sumalatha, V. M., & Raju, G. (2014). Model based test case optimization of UML activity diagrams using evolutionary algorithms. *Model Based Test Case Optimization of UML Activity Diagrams using Evolutionary Algorithms, 12*(11), 131-142.

Sung, P. W.-B., & Paynter, J. (2006). *Software testing practices in New Zealand.* Paper presented at the Proceedings of the 19th Annual Conference of the National Advisory Committee on Computing Qualifications.

Suri, B., Mangal, I., & Srivastava, V. (2011). Regression test suite reduction using an hybrid technique based on BCO and genetic algorithm. *Special Issue of International Journal of Computer Science & Informatics (IJCSI), 2*(2), 2231-5292.

Swain, R. K., Behera, P. K., & Mohapatra, D. P. (2012a). Generation and optimization of test cases for object-oriented software using state chart diagram. *International Journal*, 407- 424.

Swain, R. K., Behera, P. K., & Mohapatra, D. P. (2012b). Minimal testcase generation for object-oriented software with state charts. *International Journal of Software Engineering & Applications (IJSEA), 3*(4).

Swain, R. K., Panthi, V., Behera, P., & Mohapatra, D. (2012c). Automatic test case generation from UML state chart diagram. *International Journal of Computer Applications, 42*(7), 26-36.

Swain, S. K., Mohapatra, D. P., & Mall, R. (2010a). Test case generation based on state and activity models. *Journal of Object Technology, 9*(5), 1-27.

Swain, S. K., Mohapatra, D. P., & Mall, R. (2010b). Test case generation based on use case and sequence diagram. *International Journal of Software Engineering, 3*(2), 21-52.

Tan, R. P. (2003). *Programming language support for automated testing.* (PhD Dissertation), Virginia Tech.

Tewari, A., & Misra, A. K. (2015). An approach to Model Based Test case generation for Student Admission Process. *International Journal of Innovative Science, Engineering & Technology, 2*(10), 818 -825.

Theis, B., Frood, J., Nishri, D., & Marrett, L. D. (2002). Evaluation of a risk factor survey with three assessment methods. *Chronic Diseases and Injuries in Canada, 23*(1), 1 - 47.

Tomar, A., & Singh, P. (2016). Software testing with different optimization techniques. *International Journal of Emerging Technology and Advanced Engineering, 6*(4), 169-171.

Tripathy, A., & Mitra, A. (2012). *Test case generation using activity diagram and sequence diagram.* Paper presented at the International Conference on Advances in Computing.

Tsumaki, T., & Morisawa, Y. (2000). *A framework of requirements tracing using UML*. Paper presented at the Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific.

UML, O. M. G. (2004). UML 2.0 Infrastructure Specification. *OMG, Needham*.

Utting, M., & Legeard, B. (2007). *Practical model-based testing: a tools approach*. San Francisco, United States: Morgan Kaufmann.

Utting, M., & Legeard, B. (2010). *Practical model-based testing: a tools approach*. San Francisco, United States: Morgan Kaufmann.

Utting, M., Pretschner, A., & Legeard, B. (2006). A Taxonomy of Model-based Testing *Technical report*. Hamilton, New Zealand: The University of Waikato.

Vaziri, R., & Mohsenzadeh, M. (2012). A questionnaire-based data quality methodology. *International Journal of Database Management Systems, 4*(2), 55.

Verma, A., & Dutta, M. (2014). Automated Test case generation using UML diagrams based on behavior. *International Journal of Innovations in Engineering and Technology (IJIET), 4*(1), 31 - 39.

Vernotte, A., Dadeau, F., Lebeau, F., Legeard, B., Peureux, F., & Piat, F. (2014). *Efficient Detection of Multi-step Cross-Site Scripting Vulnerabilities*. Paper presented at the 10th International Conference on Information Systems Security Hyderabad, India.

Voloshin, V. I. (2009). *Introduction to graph theory*. New York, United States: Nova Science Publishers.

Waller, M. P., Dresselhaus, T., & Yang, J. (2013). JACOB: an enterprise framework for computational chemistry. *Journal of computational chemistry, 34*(16), 1420-1428.

Wei, Z., & Xiaoxue, W. (2010). *Graph theory model based automatic test platform design.* Paper presented at the Software Engineering and Data Mining (SEDM), 2010 2nd International Conference on.

Weilkiens, T. (2011). *Systems engineering with SysML/UML: modeling, analysis, design*. Massachusetts, United States: Morgan Kaufmann.

Weißleder, S. (2010). *Test models and coverage criteria for automatic model-based test generation with UML state machines.* Humboldt University of Berlin.

Weißleder, S., & Sokenou, D. (2010). ParTeG-a Model-Based Testing tool. *Softwaretechnik-Trends, 30*(2), 1 -2

Werner, E., & Grabowski, J. (2012). Mining test cases: optimization possibilities. *International Journal On Advances in Software, 5*(3 and 4), 200-211.

Wiegers, K., & Beatty, J. (2013). *Software requirements*. London, United Kingdom: Pearson Education.

Wiegers, K. E. (2002a). *Peer reviews in software: A practical guide*. Boston, United State: Addison-Wesley Boston.

Wiegers, K. E. (2002b). Seven truths about peer reviews. *Cutter IT Journal, 15*(7), 31-37.

Wu, Y.-C., & Fan, C.-F. (2014). Automatic test case generation for structural testing of function block diagrams. *Information and Software Technology, 56*(10), 1360-1376.

Xiong, J. (2011). *New software engineering paradigm based on complexity science: an introduction to NSE*. Berlin, Germany: Springer Science & Business Media.

Xu, S., Chen, L., Wang, C., & Rud, O. (2016). *A comparative study on black-box testing with open source applications.* Paper presented at the Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2016 17th IEEE/ACIS International Conference on.

Xu, Z., Kim, Y., Kim, M., Rothermel, G., & Cohen, M. B. (2010). *Directed test suite augmentation: techniques and tradeoffs.* Paper presented at the Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering.

Yadav, K., Patel, S., Arora, T., Uptu, U. P., & Jnu, J. (2016). Challenges in automatic test case generation. *International Journal of Communications, 1*, 99-102.

Yan, X.-T., Jiang, C., & Eynard, B. (2008). *Advanced design and manufacture to gain a competitive edge*: Springer.

Yang, X.-S. (2010). *Nature-inspired metaheuristic algorithms*: Luniver press.

Yang, X.-S., & He, X. (2013). Firefly algorithm: recent advances and applications. *International Journal of Swarm Intelligence, 1*(1), 36-50.

Yemul, M. S., Vhatkar, K., & Bag, V. (2014). Testing approach for automatic test case generation and Optimization using GA. *international Journal of Emerging Trends & Technology in Computer Science, 3*(5), 69 - 71.

Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability, 22*(2), 67-120.

Yu, Z., Martinez, M., Danglot, B., Durieux, T., & Monperrus, M. (2017). Test case generation for program repair: a study of feasibility and effectiveness. *ArXiv e-prints*, 1 -12.

Zaibon, S. B., & Shiratuddin, N. (2010). *Mobile game-based learning (mGBL) engineering model as a systematic development approach.* Paper presented at the Global Learn.

Zelkowitz, M. V., & Wallace, D. R. (1998). Experimental models for validating technology. *Computer, IEEE, 31*(5), 23-31. doi: 10.1109/2.675630

Zhang, C., Duan, Z., Yu, B., Tian, C., & Ding, M. (2016). A Test Case Generation Approach Based on Sequence Diagram and Automata Models. *Chinese Journal of Electronics, 25*(2), 234-240.

Zhang, W., & Liu, S. (2013). Supporting tool for automatic specification-based test case generation *Structured Object-Oriented Formal Language and Method* (pp. 12-25): Springer.

Zhu, H., Hall, P. A., & May, J. H. (1997). Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR), 29*(4), 366-427.

Appendix A

EXPERT EVALUATION FORM

EXPERT EVALUATION FORM

Dear Respected Respondent,

My name is Yasir Dawood Salman and I am currently pursuing my Ph.D. in Information Technology (IT). I am specializing in Software Testing at the School of Computing, College of Arts and Sciences, Universiti Utara Malaysia (UUM). My Ph.D. research entitled An Automated Test Case Generation Model for UML Statechart Diagram aims to develop a model and algorithms that can automatically generate test case from the UML statechart diagram.

Expert review is the verification method selected to evaluate this study. This study seeks your expertise in evaluating the proposed work. The information supplied will be treated as confidential and will be used for research purposes only and may be reported anonymously in academic publications. I humbly solicit for your kind assistance to participate in this research.

The main purpose of this verification is to verify the proposed model and its components, as well as other entities within the model, possesses a satisfactory range of accuracy, completeness, and consistency.

Kindly attach a copy of your CV after completion of this verification form for the proper documentation of this research.

If you have any questions regarding this research, please feel free to contact me by e-mail at yasir.dawod@gmail.com, phone number (+60169790922), or through my supervisor Dr. Nor Laily Hashim at laily@uum.edu.my.

Thank you for your time and assistance.

Instructions:

Please read the system review documents provided to you and go through the model, algorithms, and prototype carefully. Once this is done, please tick (✓) the most appropriate answer. You are advised to answer the questions based on your knowledge and experience and verify the items in Section B. This section on software quality dimensions is used to measure the originality and validity of the proposed system implementation for automatic test case generation of the UML statechart diagram. Section A is expert profile. This questionnaire is NOT intended to assess people, their work, or knowledge. Completing the questionnaire will take around 30–45 minutes. I will deeply appreciate if you could answer the questions carefully as the information you provide will influence the accuracy and success of this research.

Section A: Expert Profile

Name (First and Last) ……...

Employer/ Facility ……...

Position [ ] Professor [ ] Associate Professor [ ] Senior Lecturer [ ] Lecturer [ ] Others (Please specify)……...

……...

Fields of ……...

Specialization ……...

……...

Years of Experience in:

Algorithms	Software Development	Software Engineering	Software Testing
……	……	……	……

Research Interests ……...

……...

……...

E-mail ……...

Office Phone …… Mobile Phone ……...

Section B: Items for Review

Please validate and give comments on the below mentioned dimensions on the proposed system (framework, algorithms and prototype) implementation for an automatic test case generation:

DIMENSIONS	DESCRIPTIONS	COMMENTS/SUGGESTIONS
Practicality	The proposed framework of automatic test case generation from UML diagrams can practically be implemented in the real world.	☐ Agree ☐ Disagree Comments/ Suggestions: ---------- ---------- ----------
Clarity	As a whole, the framework is workable and the steps in the framework are easily followed.	☐ Agree ☐ Disagree Comments/ Suggestions: ---------- ---------- ----------
Completeness	The essential items of the proposed framework are complete, satisfactory, and suitable to generate test cases.	☐ Agree ☐ Disagree Comments/ Suggestions: ---------- ---------- ---------- ----------

Correctness	The algorithms: State Relationships Table (SRT), Test Cases Paths Generation (TCGP), minimization, prioritization, and Test Cases Generation (TCG), provide correct results and achieve its objectives.	☐ Agree ☐ Disagree Comments/ Suggestions: --- --- ---
Effectiveness	The prototype automatically generates the test cases from UML statechart diagram, for which it is intended.	☐ Agree ☐ Disagree Comments/ Suggestions: --- --- --- ---
Accuracy	The system provides correct test cases result to the inputted UML statechart diagram.	☐ Agree ☐ Disagree Comments/ Suggestions: --- --- --- ---
Perceived Usefulness	The proposed system is useful for the software tester in improving the coverage criteria quality of test case generation.	☐ Agree ☐ Disagree Comments/ Suggestions: --- --- ---

Usability	Using the proposed system would make generating the test cases easy for the software tester.	☐ Agree ☐ Disagree Comments/ Suggestions: ---------- ---------- ---------- ----------
Understand-ability	All documentations are clearly and simply written such that procedures, rules, and algorithms are readable and can be easily understood.	☐ Agree ☐ Disagree Comments/ Suggestions: ---------- ---------- ---------- ----------

Additional comments (if any):

..……..

……

……

……

Thank you.

…… Date……

(Signature & Official Stamp)

Appendix B

CONSENT FOR PARTICIPATION IN EXPERT VERIFICATION

I volunteer to participate in a research project conducted by Yasir Dawood Salman, Ph.D. student, in Information Technology (IT), School of Computing, College of Arts and Sciences, Universiti Utara Malaysia (UUM).

I understand that the expert verification form is designed to evaluate the proposed framework, algorithms, and prototype. I will be one of approximately eight people being interviewed for this research.

1. My participation in this project is voluntary. I may withdraw and discontinue participation at any time. If I decline to participate or withdraw from the study, no one on my campus will be told.
2. The interview will last approximately 30-45 minutes. Notes will be written during the interview. An audio tape of the interview and subsequent dialogue will be make. If I do not want to be taped, I will need to inform in advance.
3. I understand that the researcher will not identify me by name in any reports using information obtained from this interview, and that my confidentiality as a participant in this study will remain secure. Subsequent uses of records and data will be subject to standard data use policies, which protect the anonymity of individuals and institutions.
4. I have read and understand the explanation provided to me. I have had all my questions answered to my satisfaction, and I voluntarily agree to participate in this study.
5. I have been given a copy of this consent form.

Name of Participant	Date	Signature
Name of Researcher	Date	Signature

Appendix C

DETAILED MINIMIZATION AND PRIORITIZATION FOR SELECTED EXAMPLES

Section A: UML Statechart Diagram of an Online Shop

The process of minimize and prioritize the UML statechart diagram of an online shop example (see Section 5.2.2.1.2) is shown below.

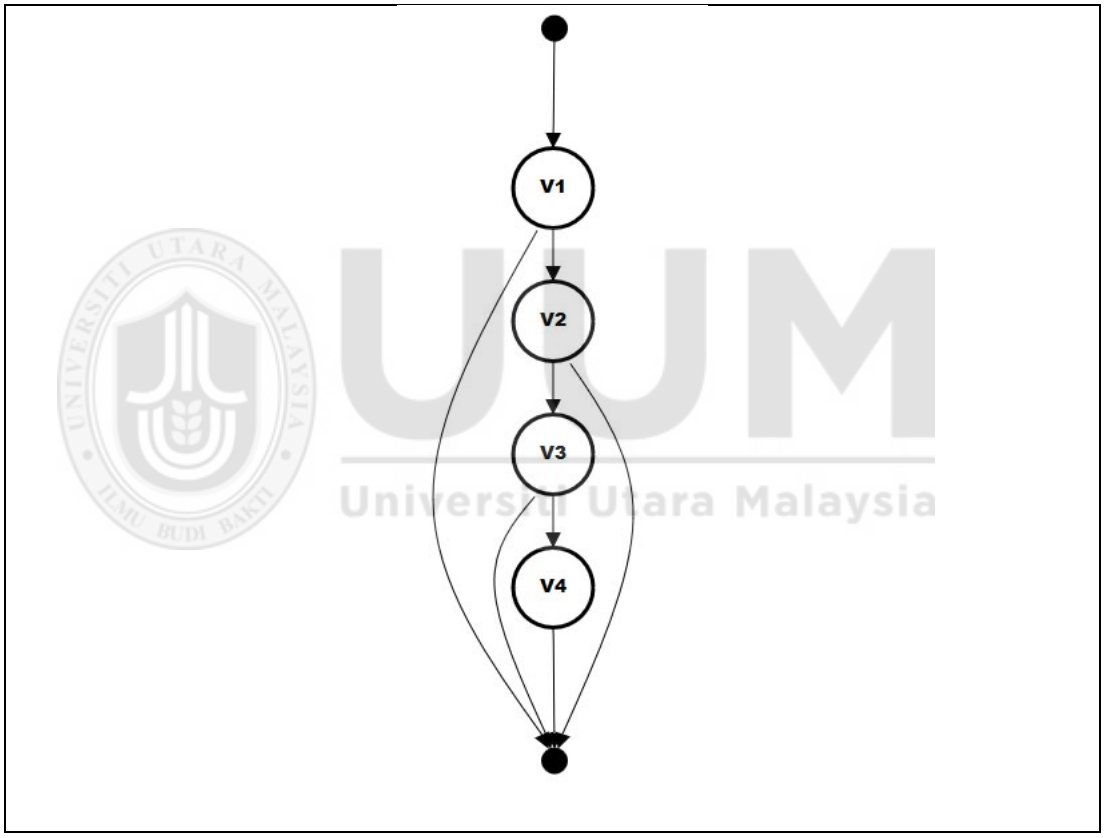

Figure B.1. Chart Relationship Graph for the UML Statechart Diagram of an Online Shop

The intermediate graph (Figure B.1) was converted to test paths using TCGP algorithm, and all the possible generated test paths from the intermediate graph is shown in Figure B.2.

TP 1: [S → 1 → 2 → 3 → 4 → E]
TP 2: [S → 1 → E]
TP 3: [S → 1 → 2 → E]
TP 4: [S → 1 → 2 → 3 → E]

Figure B.2 All Possible Test Paths for the UML Statechart Diagram of an Online Shop

Path weight was calculated for each tests path using Equation 4.7, as shown in Table B.1, to determine each path weight of transactions in the system

Table B.1

Path Weight for Each Path for the UML Statechart Diagram of an Online Shop

TC	S→1	1→2	1→E	2→3	2→E	3→4	3→E	4→E	E	W_v
1	1	1	0	1	0	1	0	1	5	0.83
2	1	0	1	0	0	0	0	0	2	0.66
3	1	1	0	0	1	0	0	0	3	0.75
4	1	1	0	1	0	0	1	0	4	0.8

After generate the path weight, next step start by calculate path coverage for each single path as shown in Table B.2.

Table B.2

Coverage Criteria for Each Path for the UML Statechart Diagram of an Online Shop

TP No	All-State	All-Transition	All-Transition-pairs	All-One-loop-paths
1	100%	62%	50%	-
2	50%	25%	16%	-
3	66%	37%	33%	-
4	83%	50%	50%	-

After generate the path weight and coverage criteria for each path the intermediate graph is converted to adjacency matrix, as showing in Table B.3. Then, this matrix is used to generate the guidance matrix for the graph.

Table B.3

Adjacency Matrix for the UML Statechart Diagram of an Online Shop

States	**0**	**1**	**2**	**3**	**4**	**5**
0	0	1	0	0	0	0
1	0	0	1	0	0	1
2	0	0	0	1	0	1
3	0	0	0	0	1	1
4	0	0	0	0	0	1
5	0	0	0	0	0	0

After creating adjacency matrix, it is then used to generate guidance matrix. In the example in Figure 1, the number of vertices is 6, and the number of edges is 8, therefore the Cyclomatic Complexity equal to 4. However, the Cyclomatic Complexity for each vertex need to be calculated using Equation 4.9 to be used to calculate the guidance value using Equation 4.8. The results are shown in Table 4.

Table B.4

Guidance Value for the UML Statechart Diagram of an Online Shop

States	**Cyclomatic Complexity CC**	**guidance value GF**
0	4	196
1	4	156
2	3	87
3	2	38
4	1	9
5	1,000 [END vertex infinity]	1,000 [finial state]

Guidance matrix (Table B.5) is just as a look-up/decision table of adjacency matrix with each guidance value corresponding to every edge.

Table B.5

Guidance Matrix for the UML Statechart Diagram of an Online Shop

States	0	1	2	3	4	5
0	0	156	0	0	0	0
1	0	0	87	0	0	1000
2	0	0	0	38	0	1000
3	0	0	0	0	9	1000
4	0	0	0	0	0	1000
5	0	0	0	0	0	0

Then the algorithm will generate the path sequences as:

Path 1= [0, 1, 2, 3, 4, 5],
Path 2= [1, 5],
Path 3= [2, 5],
Path 3= [3, 5].

To optimize the test cases, the algorithm will match each optimal path with paths in Figure B.2, and chose the lowest path weight W_v between the selected paths match paths. The minimized test paths are shown in Figure B.3

TP 1: [S → 1 → 2 → 3 → 4 → E]
TP 2: [S → 1 → E]
TP 3: [S → 1 → 2 → E]
TP 4: [S → 1 → 2 → 3 → E]

Figure B.3. Optimized Test Paths for the UML Statechart Diagram of an Online Shop

The combination use of these three paths lead to achieving: all-state coverage, all-transition coverage and, all-transition-pairs coverage as shown in Table B.6.

Table B.6

Coverage Criteria Percentage for the Minimized Paths for the UML Statechart Diagram of an Online Shop

TP No	All-State	All-Transition	All-Transition-pairs	All-One-loop-paths
1,2,3,4	100%	100%	100%	-

The ten generated fireflies for each state are showing in Table B.7.

Table B.7

Calculation of Brightness Values of 10 Fireflies

V	1		2		3		4		5		6		7		8		9		10	
	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i
0	6.9	3.5	6.8	3.55	6.7	3.6	6.6	3.65	6.5	3.7	6.4	3.76	6.3	3.82	6.2	3.88	6.1	3.94	6	4
1	5.9	2.07	5.8	2.11	5.7	2.15	5.6	2.18	5.5	2.2	5.4	2.26	5.3	2.3	5.2	2.35	5.1	2.39	5	2.44
2	4.9	2.83	4.8	2.89	4.7	2.95	4.6	3.01	4.5	3.08	4.4	3.14	4.3	3.22	4.2	3.29	4.1	3.37	4	3.45
3	3.9	4.1	3.8	4.2	3.7	4.31	3.6	4.42	3.5	4.55	3.4	4.67	3.3	4.81	3.2	4.95	3.1	5.1	3	5.26
4	2.9	14.71	2.8	15.15	2.7	15.63	2.6	16.13	2.5	16.67	2.4	17.24	2.3	17.86	2.2	18.52	2.1	19.23	2	20

Table B.8 shows the separate calculation for cyclomatic complexity and information flow for each vertex, then show the Firefly brightness for that specific vertex after including the random factor.

Table B.8

Objective Function

Vertex	Cyclomatic Complexity CC	Information Flow IF_i	Firefly brightness A_i
0	4	0	3.94
1	4	4	2.26
2	3	4	3.29
3	2	4	4.31
4	1	1	14.71

By calculating the mean of brightness at every path using Equation 4.15, the results are shown in Table 9.

Table B.9

Test Path Prioritization for the Minimized Paths for the UML Statechart Diagram of an Online Shop

Test ID	Test path	Brightness value
TP 1	$0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$	6.1010304355335
TP 4	$0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 5$	4.6998174561816
TP 2	$0 \rightarrow 1 \rightarrow 5$	4.5997256564649
TP 3	$0 \rightarrow 1 \rightarrow 2 \rightarrow 5$	4.4963083323801

In Table B.9 test paths mean of the brightness value is calculated for each generated optimized test path. From the table it is observed that optimized test path one has the highest brightness value and hence having high priority. Then the fourth path, the second path, and finely the third one.

Section B: UML Statechart Diagram of an Airline Check-in

The process of minimize and prioritize the UML statechart diagram of an airline check-in example (see Section 5.2.2.1.3) is shown below.

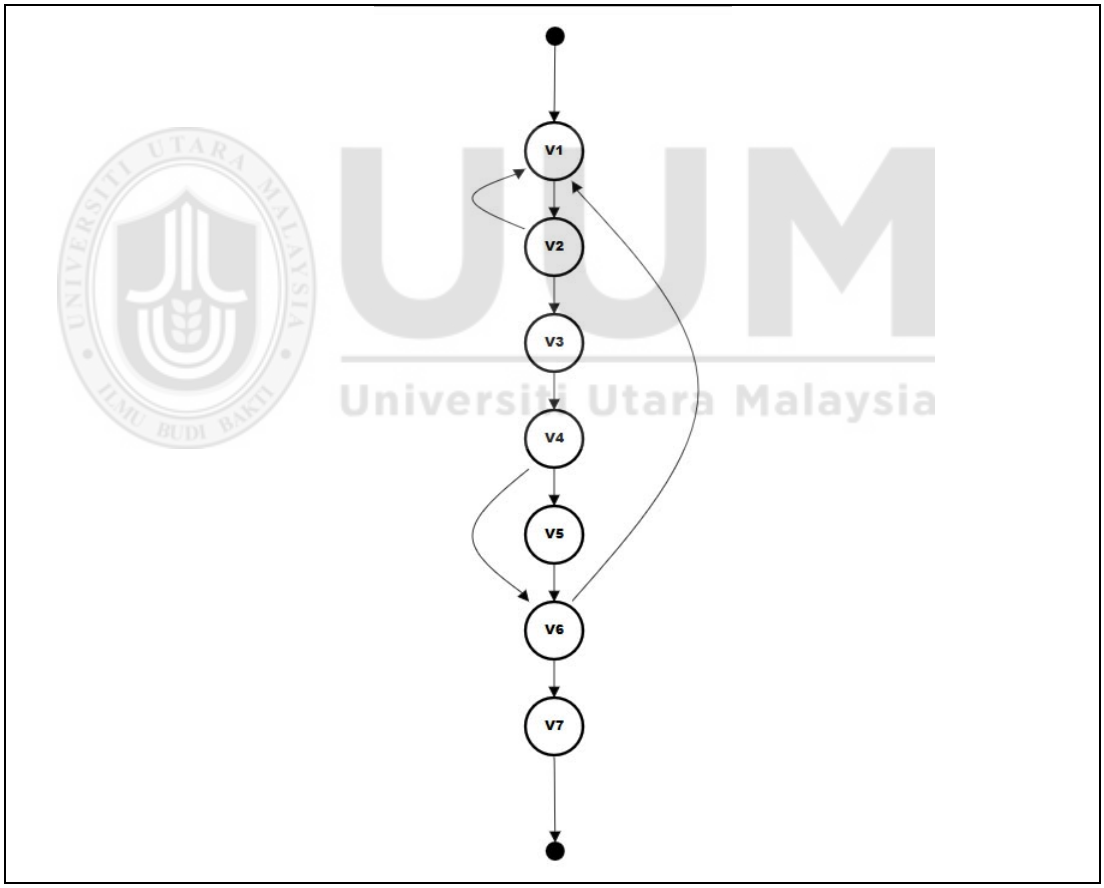

Figure B.4. Chart Relationship Graph of a UML Statechart Diagram of an Airline Check-in

The intermediate graph (Figure B.4) was converted to test paths using TCGP algorithm, and all the possible generated test paths from the intermediate graph is shown in Figure B.5.

TP 1: [S→1→2→3→4→5→6→7→E]
TP 2: [S→1→2→3→4→6→7→E]
TP 3: [S→1→2→3→4→5→6→1→2→3→4→5→6→7→E]
TP 4: [S→1→2→3→4→5→6→1→2→3→4→6→7→E]
TP 5: [S→1→2→3→4→6→1→2→3→4→5→6→7→E]
TP 6: [S→1→2→3→4→6→1→2→3→4→6→7→E]
TP 7: [S→1→2→1→2→3→4→5→6→7→E]
TP 8: [S→1→2→1→2→3→4→6→7→E]

Figure B.5. All Possible Test Paths of a UML Statechart Diagram of an Airline Check-in

Path weight was calculated for each tests path using Equation 4.7, as shown in Table B.10, to determine each path weight of transactions in the system

Table B.10

Path Weight for Each Path of a UML Statechart Diagram of an Airline Check-in

TC	S→1	1→2	2→3	2→1	3→4	4→5	4→6	5→6	6→7	6→1	7→E	E	W_v
1	1	1	1	0	1	1	0	1	1	0	1	8	0.88
2	1	1	1	0	1	0	1	0	1	0	1	7	0.87
3	1	1	1	0	1	1	0	1	1	1	1	9	0.6
4	1	1	1	0	1	1	1	1	1	1	1	10	0.71
5	1	1	1	0	1	1	1	1	1	1	1	10	0.71
6	1	1	1	0	1	0	1	0	1	1	1	8	0.61
7	1	1	1	1	1	1	0	1	1	0	1	9	0.81
8	1	1	1	1	1	0	0	0	1	0	1	7	0.7

After generate the path weight, next step start by calculate path coverage for each single path as shown in Table B.11.

Table B.11

Coverage Criteria for Each Path of a UML Statechart Diagram of an Airline Check-in

TP No	All-State	All-Transition	All-Transition-pairs	All-One-loop-paths
1	100%	72%	25%	0%
2	88%	63%	25%	0%
3	100%	81%	50%	50%
4	100%	90%	75%	50%
5	100%	90%	50%	50%
6	88%	72%	50%	50%
7	100%	81%	50%	50%
8	88%	63%	25%	50%

After generate the path weight and coverage criteria for each path the intermediate graph is converted to adjacency matrix, as showing in Table B.12. Then, this matrix is used to generate the guidance matrix for the graph.

Table B.12

Adjacency Matrix of a UML Statechart Diagram of an Airline Check-in

States	0	1	2	3	4	5	6	7	8
0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0
2	0	1	0	1	0	0	0	0	0
3	0	0	0	0	1	0	0	0	0
4	0	0	0	0	0	1	1	0	0
5	0	0	0	0	0	0	1	0	0
6	0	1	0	0	0	0	0	1	0
7	0	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0	0

After creating adjacency matrix, it is then used to generate guidance matrix. In the example in Figure B.4, the number of vertices is 9, and the number of edges is 11, therefore the Cyclomatic Complexity equal to 4. However, the Cyclomatic Complexity for each vertex need to be calculated using Equation 4.9 to be used to calculate the guidance value using Equation 4.8. The results are shown in Table B.13.

Table B.13

Guidance Value of a UML Statechart Diagram of an Airline Check-in

States	Cyclomatic Complexity CC	guidance value GF
0	4	316
1	4	276
2	4	236
3	3	147
4	3	117
5	2	58
6	2	38
7	1	9
8	1,000 [END vertex infinity]	1,000 [finial state]

Guidance matrix (Table B.14) is just as a look-up/decision table of adjacency matrix with each guidance value corresponding to every edge.

Table B.14

Guidance Matrix of a UML Statechart Diagram of an Airline Check-in

States	0	1	2	3	4	5	6	7	8
0	0	276	0	0	0	0	0	0	0
1	0	0	236	0	0	0	0	0	0
2	0	276	0	147	0	0	0	0	0
3	0	0	0	0	117	0	0	0	0
4	0	0	0	0	0	58	38	0	0
5	0	0	0	0	0	0	38	0	0
6	0	276	0	0	0	0	0	9	0
7	0	0	0	0	0	0	0	0	1000
8	0	0	0	0	0	0	0	0	0

Then the algorithm will generate the path sequences as:

Path 1= [0, 1, 2, 3, 4, 6, 7, 8],
Path 2= [2, 1],
Path 3= [4, 5, 6, 1].

To optimize the test cases, the algorithm will match each optimal path with paths in Figure B.5, and chose the lowest path weight W_v between the selected paths match paths. The minimized test paths are shown in Figure B.6

TP 2: [S→1→2→3→4→6→7→E]
TP 4: [S→1→2→3→4→5→6→1→2→3→4→6→7→E]
TP 8: [S→1→2→1→2→3→4→6→7→E]

Figure B.6. Minimized Test Paths of a UML Statechart Diagram of an Airline Check-in

The combination use of these three paths lead to achieving: all-state coverage, all-transition coverage, all-transition-pairs coverage, and all-one-loop coverage as shown in Table B15.

Table B.15

Coverage Criteria Percentage for the Minimized Paths

TP No	**All-State**	**All-Transition**	**All-Transition-pairs**	**All-One-loop-paths**
2, 4, 8	100%	100%	100%	100%

The ten generated fireflies for each state are showing in Table B16.

Table B.16

Calculation of Brightness Values of the Ten Fireflies

V	**1**		**2**		**3**		**4**		**5**		**6**		**7**		**8**		**9**		**10**	
	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i
0	8.9	2.73	8.8	2.76	8.7	2.79	8.6	2.82	8.5	2.86	8.4	2.89	8.3	2.92	8.2	2.96	8.1	2.99	8	3.03
1	7.9	0.96	7.8	0.98	7.7	0.99	7.6	1	7.5	1.02	7.4	1.03	7.3	1.04	7.2	1.06	7.1	1.07	7	1.09
2	6.9	1.78	6.8	1.81	6.7	1.83	6.6	1.86	6.5	1.89	6.4	1.92	6.3	1.95	6.2	1.98	6.1	2.01	6	2.04
3	5.9	4.07	5.8	4.13	5.7	4.2	5.6	4.27	5.5	4.35	5.4	4.42	5.3	4.5	5.2	4.59	5.1	4.67	5	4.76
4	4.9	2.83	4.8	2.89	4.7	2.95	4.6	3.01	4.5	3.08	4.4	3.14	4.3	3.22	4.2	3.29	4.1	3.37	4	3.45
5	3.9	7.87	3.8	8.06	3.7	8.26	3.6	8.47	3.5	8.7	3.4	8.93	3.3	9.17	3.2	9.43	3.1	9.71	3	10
6	2.9	5.43	2.8	5.62	2.7	5.81	2.6	6.02	2.5	6.25	2.4	6.49	2.3	6.76	2.2	7.04	2.1	7.35	2	7.69
7	1.9	20.83	1.8	21.74	1.7	22.73	1.6	23.81	1.5	25	1.4	26.32	1.3	27.78	1.2	29.41	1.1	31.25	1	33.33

Table B.17 shows the separate calculation for cyclomatic complexity and information flow for each vertex, then show the Firefly brightness for that specific vertex after including the random factor.

Table B.17

Objective Function

Vertex	Cyclomatic Complexity CC	Information Flow IF_i	Firefly brightness A_i
0	4	0	2.82
1	4	9	1.04
2	4	4	2.04
3	3	1	4.59
4	3	4	2.89
5	2	1	8.47
6	2	16	6.76
7	1	1	31.25

By calculating the mean of brightness at every path using Equation 4.15, the results are shown in Table B.18.

Table B.18

Test Path Prioritization of a UML Statechart Diagram of an Airline Check-in

Test ID	Test path	Brightness value
TP 2	0→1→2→3→4→6→7→0	8.3417877259468
TP 8	0→1→2→1→2→3→4→6→7→8	6.8306759195254
TP 4	0→1→2→3→4→5→6→1→2→3→4→5→7→8	6.7065188982599

In Table B.19 test paths mean of the brightness value is calculated for each generated optimized test path. From the table it is observed that optimized test second path has the highest brightness value and hence having high priority. Then the eighth path, and finely the fourth one.

Section C: UML Statechart Diagram for a Retail Point of Sale

The process of minimize and prioritize the UML statechart diagram for a retail point of sale example (see Section 5.2.2.1.4) is shown below.

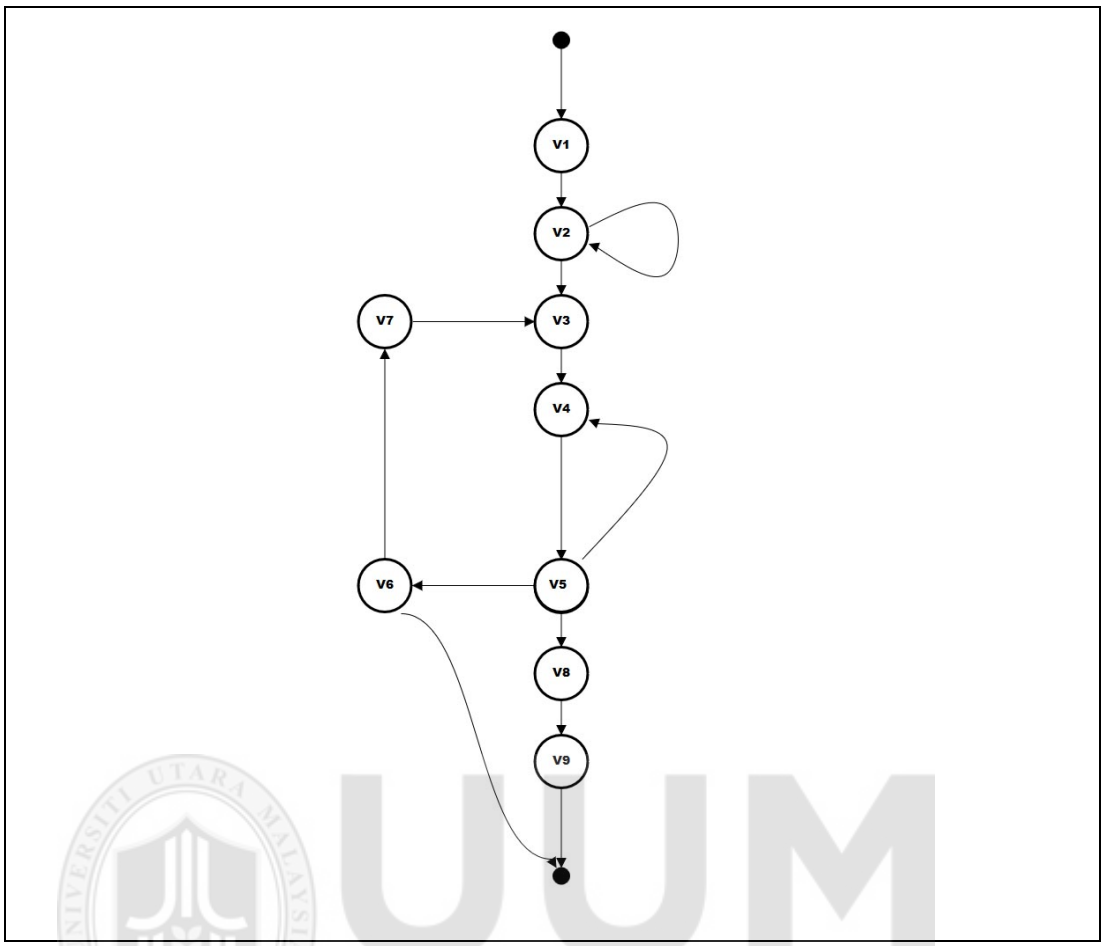

Figure B.7. Chart Relationship Graph for UML Statechart Diagram for a Retail Point of Sale

The intermediate graph (Figure B.7) was converted to test paths using TCGP algorithm, and all the possible generated test paths from the intermediate graph is shown in Figure B.8.

TP 1: [S→1→2→3→4→5→8→9→E]
TP 2: [S→1→2→3→4→5→6→E]
TP 3: [S→1→2→3→4→5→6→7→3→4→5→8→9→E]
TP 4: [S→1→2→3→4→5→6→7→3→4→5→6→E]
TP 5: [S→1→2→3→4→5→4→5→8→9→E]
TP 6: [S→1→2→3→4→5→4→5→6→E]
TP 7: [S→1→2→2→3→4→5→8→9→E]
TP 8: [S→1→2→2→3→4→5→6→E]

Figure B.8. All Possible Test Paths for UML Statechart Diagram for a Retail Point of Sale

Path weight was calculated for each tests path using Equation 4.7, as shown in Table B.19, to determine each path weight of transactions in the system

Table B.19

Path Weight for Each Path for UML Statechart Diagram for a Retail Point of Sale

TC	S → 1	1 → 2	2 → 3	2 → 2	3 → 4	4 → 5	5 → 6	5 → 4	5 → 8	6 → 7	6 → E	7 → 3	8 → 9	9 → E	E	W_v
1	1	1	1	0	1	1	0	0	1	0	0	0	1	1	8	0.88
2	1	1	1	0	1	1	1	0	0	0	1	0	0	0	7	0.87
3	1	1	1	0	1	1	1	1	1	1	0	1	1	1	12	0.85
4	1	1	1	0	1	1	1	1	0	1	1	1	0	0	10	0.76
5	1	1	1	0	1	1	0	1	1	0	0	0	1	1	9	0.81
6	1	1	1	0	1	1	1	1	0	0	1	0	0	0	8	0.8
7	1	1	1	1	1	1	0	0	1	0	0	0	1	1	8	0.9
8	1	1	1	1	1	1	1	0	0	0	1	0	0	0	8	0.88

After generate the path weight, next step start by calculate path coverage for each single path as shown in Table B.20.

Table B.20

Coverage Criteria for Each Path for UML Statechart Diagram for a Retail Point of Sale

TP No	All-State	All-Transition	All-Transition-pairs	All-One-loop-paths
1	81%	57%	28%	0%
2	72%	50%	42%	0%
3	100%	85%	42%	50%
4	81%	71%	57%	50%
5	81%	64%	42%	50%
6	72%	57%	57%	50%
7	81%	57%	42%	50%
8	72%	57%	57%	50%

After generate the path weight and coverage criteria for each path the intermediate graph is converted to adjacency matrix, as showing in table B.21. Then, this matrix is used to generate the guidance matrix for the graph.

Table B.21

Adjacency Matrix

States	0	1	2	3	4	5	6	7	8	9	10
0	0	1	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0	0
2	0	0	1	1	0	0	0	0	0	0	0
3	0	0	0	0	1	0	0	0	0	0	0
4	0	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	1	0	1	0	1	0	0
6	0	0	0	0	0	0	0	1	0	0	0
7	0	0	0	1	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	0	0	1
10	0	0	0	0	0	0	0	0	0	0	0

After creating adjacency matrix, it is then used to generate guidance matrix. In the example in Figure B.7, the number of vertices is 11, and the number of edges is 14, therefore the Cyclomatic Complexity equal to 5. However, the Cyclomatic Complexity for each vertex need to be calculated using Equation 4.9 to be used to calculate the guidance value using Equation 4.8. The results are shown in Table B.22.

Table B.22

Guidance Value

States	Cyclomatic Complexity CC	guidance value GF
0	5	495
1	5	*445*
2	5	395
3	4	276
4	4	236
5	4	196
6	3	117
7	3	87
8	1	19
9	1	9
10	1,000 [END vertex infinity]	1,000 [finial state]

Guidance matrix (Table B.23) is just as a look-up/decision table of adjacency matrix with each guidance value corresponding to every edge.

Table B.23

Guidance matrix

States	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	445	0	0	0	0	0	0	0	0	0
1	0	0	395	0	0	0	0	0	0	0	0
2	0	0	395	276	0	0	0	0	0	0	0
3	0	0	0	0	236	0	0	0	0	0	0
4	0	0	0	0	0	195	0	0	0	0	0
5	0	0	0	0	236	0	117	0	19	0	0
6	0	0	0	0	0	0	0	87	0	0	1000
7	0	0	0	276	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	9	0
9	0	0	0	0	0	0	0	0	0	0	1000
10	0	0	0	0	0	0	0	0	0	0	0

Then the algorithm will generate the path sequences as:

Path 1= [0, 1, 2, 3, 4, 5, 8, 9, 10],
Path 2= [2, 2],
Path 3= [5, 6, 7, 3],
Path 4= [5, 4],
Path 5= [6, 10].

To optimize the test cases, the algorithm will match each optimal path with paths in Figure B.8, and chose the lowest path weight W_v between the selected paths match paths. The minimized test paths are shown in Figure B.9.

TP 1: [S→1→2→3→4→5→8→9→E]
TP 8: [S→1→2→2→3→4→5→6→E]
TP 4: [S→1→2→3→4→5→6→7→3→4→5→6→E]
TP 6: [S→1→2→3→4→5→4→5→6→E]

Figure B.9. Minimized Test Paths for UML Statechart Diagram for a Retail Point of Sale

The combination use of these three paths lead to achieving: all-state coverage, all-transition coverage, all-transition-pairs coverage, and all-one-loop coverage as shown in table B.24.

Table B.24

Coverage Criteria Percentage for the Minimized Paths

TP No	All-State	All-Transition	All-Transition-pairs	All-One-loop-paths
1, 8, 4, 6	100%	100%	100%	100%

The ten generated fireflies for each state are showing in Table B.25.

Table B.25

Calculation of Brightness Values of the Ten Fireflies

V	1		2		3		4		5		6		7		8		9		10	
	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i	d_i	A_i
0	11.9	1.65	11.8	1.67	11.7	1.68	11.6	1.69	11.5	1.71	11.4	1.72	11.3	1.74	11.2	1.75	11.1	1.77	1	1.79
1	10.9	1.51	10.8	1.52	10.7	1.53	10.6	1.55	10.5	1.56	10.4	1.58	10.3	1.59	10.2	1.61	10.1	1.62	1	1.64
2	9.9	0.48	9.8	0.48	9.7	0.49	9.6	0.49	9.5	0.50	9.4	0.50	9.3	0.51	9.2	0.51	9.1	0.52	9	0.52
3	8.9	1.39	8.8	1.41	8.7	1.42	8.6	1.43	8.5	1.45	8.4	1.46	8.3	1.47	8.2	1.49	8.1	1.50	8	1.52
4	7.9	1.56	7.8	1.58	7.7	1.59	7.6	1.61	7.5	1.62	7.4	1.64	7.3	1.65	7.2	1.67	7.1	1.68	7	1.70
5	6.9	1.18	6.8	1.19	6.7	1.21	6.6	1.22	6.5	1.24	6.4	1.25	6.3	1.27	6.2	1.28	6.1	1.30	6	1.31
6	5.9	2.36	5.8	2.38	5.7	2.40	5.6	2.42	5.5	2.44	5.4	2.46	5.3	2.48	5.2	2.50	5.1	2.52	5	2.54
7	4.9	4.85	4.8	4.89	4.7	4.93	4.6	4.97	4.5	5.01	4.4	5.05	4.3	5.09	4.2	5.13	4.1	5.17	4	5.21
8	3.9	11.36	3.8	11.63	3.7	11.91	3.6	12.20	3.5	12.50	3.4	12.82	3.3	13.16	3.2	13.51	3.1	13.89	3	14.29
9	2.9	14.71	2.8	15.15	2.7	15.63	2.6	16.13	2.5	16.67	2.4	17.24	2.3	17.86	2.2	18.52	2.1	19.23	2	20.00

Table B.26 shows the separate calculation for cyclomatic complexity and information flow for each vertex, then show the Firefly brightness for that specific vertex after including the random factor.

Table B.26

Objective Function

Vertex	Cyclomatic Complexity CC	Information Flow IF_i	Firefly brightness A_i
0	5	0	1.74
1	5	1	1.61
2	5	16	0.5
3	4	4	1.52
4	4	4	1.58
5	4	9	1.1
6	3	4	2.4
7	3	1	5.75
8	1	1	11.36
9	1	1	14.71

By calculating the mean of brightness at every path using Equation 4.15, the results are shown in Table B.27.

Table B.27

Test Path Prioritization for UML Statechart Diagram for a Retail Point of Sale

Test ID	Test path	Brightness value
TP 1	S→1→2→3→4→5→8→9→E	4.9599705586331
TP 4	S→1→2→3→4→5→6→7→3→4→5→6→E	2.6504048295212
TP 8	S→1→2→2→3→4→5→6→E	2.3697922355156
TP 6	S→1→2→3→4→5→4→5→6→E	2.3482365027468

In Table B.27 test paths mean of the brightness value is calculated for each generated optimized test path. From the table it is observed that optimized test path 1 has the highest brightness value and hence having high priority. Then the fourth path, the eighth path, and finely the sixth one.