



A Test Case Generation Technique
Based on User Satisfaction

By

Ms. Nicha Kosindrdech

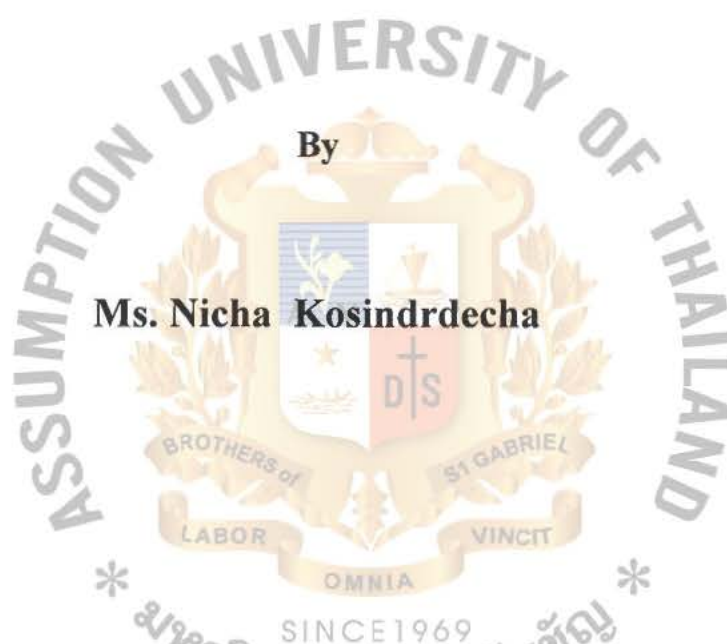
Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy
in Information Technology
Assumption University

November, 2010

A Test Case Generation Technique Based on User Satisfaction

By

Ms. Nicha Kosindrdecha



**Submitted in Partial Fulfillment of the
Requirement for the Degree of
Doctor of Philosophy
in Information Technology
Assumption University**

November, 2010

The Faculty of Science and Technology


Dissertation Approval


Dissertation Title A Test Case Generation Technique Based on User Satisfaction


By Ms. Nicha Kosindrdecha
Dissertation Advisor Asst. Prof. Dr. Jirapun Daengdej
Academic Year 2/2010


The Department of Information Technology, Faculty of Science and Technology of Assumption University has approved dissertation final report of the **thirty six** credits course. **IT9000 Dissertation**, submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Information Technology.

Approval Committee:

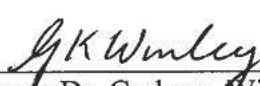

(Asst. Prof. Dr. Jirapun Daengdej)
Advisor

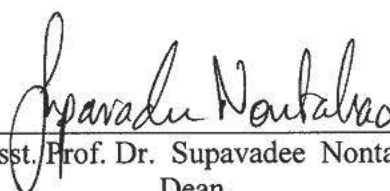

(Professor, Dr. Graham Winley)
Committee Member


(Asst. Prof. Dr. Vichit Avatchanakorn)
Committee Member


(Assoc. Prof. Dr. Surapong Euwatanamongkol)
Commission of Higher Education
University Affairs

Faculty Approval:


(Professor, Dr. Graham Winley)
Program Director


(Asst. Prof. Dr. Supavadee Nontakao)
Dean

November / 2010

The Faculty of Science and Technology

Declaration

This is to certify that the work presented in this thesis was carried out by the author in the Department of Information Technology at Assumption University, Thailand and is the result of original research conducted by the author, except where formally acknowledged and/or referenced, and has not been submitted for a degree to any other university or institution.

นิชา โคสินทร์เดช

(Nicha Kosindrdecha)



ABSTRACT

Software testing phase has been proven that it is one of the most critical and important phases in software development life cycle. In general, software testing phase takes around 40-70% of effort, time, and cost. Test case generation approaches are the most critical and widely-researched activities over a long period of time in the software testing. Many researchers propose effective test case generation techniques, such as specification-based, model-based and source code-based test case generation techniques. Large amount of attentions in literature has so far been given to model-based test cases generation. Despite the size of these efforts invested, outstanding problems for methods that derive tests from use cases are: lack of requirement prioritization before test generation, unable to systematically determine which test cases should be removed, and large number of tests is still generated due to size of alternate paths. Therefore, this dissertation proposes a marketing-driven prioritization method, along with WOW factors and cost model to classify and prioritize requirements. The study shows that there are a relationship between a return on investment (ROI) and a requirement complexity. This dissertation discovers that the high ROI requirements with less complexity are desirable. Furthermore, this thesis introduces alternate path points and risk-driven formulas to minimize a number of tests during a test generation process. The evaluation reveals that proposed methods can lead to smaller number of tests while covering higher critical requirements. In brief, the contributions of this dissertation are to: (a) propose a requirement prioritization based on customer satisfaction during a test case generation process (b) introduce alternative path point formula to minimize a number of test cases generated from UML use case diagram (c) discover a correlation between ROI and a complexity of requirement and (d) enhance the alternative path point formula by adding a retain score.

ACKNOWLEDGMENTS

First of all, I would like to express my gratitude to my advisor, Dr. Jirapun Daengdej, who gave me the valuable advice and support throughout the entire thesis process. He provided the continuous encouragement, sound advice, good teaching, and lots of good ideas. I have learned various things, from him, such as the way of thinking, the way of proceeding, and so on. With his enthusiasm, his inspiration, and his great efforts to explain things clearly and simply, he helped me through many problems. This thesis would never have succeeded without him.

I wish to express sincere appreciation to Dr. Graham Winley, who is the director of doctor of philosophy of information technology, for recommending the thesis guidance and for support he has provided during the preparation of this thesis. His patience, despite my many, many questions, is greatly appreciated. His suggestions regarding the choice of the initial conditions were truly invaluable.

However, it would have been simply impossible to start, continue, and complete without the support of my family, both financially and emotionally throughout my degree. I wish to thank my parents who, wholeheartedly, made the resources of the family available for me, I always consider myself extremely fortunate to have had the great opportunity in my life. Thanks for providing a loving environment for me. Special thanks go to my brother, for the computer problem solving and for everything he has done for me. My thanks also go to other family members for always being my inspiration.

The most important, I wish to thank two people, my grandparents, without whom none of this would have been even possible. Words cannot truly express my deepest appreciation. I own the greatest debt of gratitude to them for grow me up, for their compassion and understanding of the many turns my life has taken, and for the unconditional freedom, support and love that they have given me as I pursue my dreams. For all this and much more, I dedicate this thesis to them.

Lastly, I would like to express many thanks to my husband, Pol, who has been a great source of strength all through this work. I would have been lost without him. Thank you for the great encouragement, support, and helping me get through the difficult times. I would like to thank him for his loves, and everything he has done for me and being beside me always. In particular, his caring and understanding shown by are greatly appreciated.

PUBLICATION

The followings are a list of my publications produced during my PhD study.

- Nicha Kosindrdecha and Jirapun Daengdej, "A Black-Box Test Case Generation Method", *International Journal of Computer Science and Information Security*, USA, October 2010.
- Nicha Kosindrdecha and Jirapun Daengdej, "A Test Case Generation Process and Technique", *Journal of Software Engineering*, USA, September 2010, Vol. 4.
- Nicha Kosindrdecha and Jirapun Daengdej, "A Test Case Generation Process and Technique", *Proceeding of First International Workshop on Evolution Support for Model-Based Development and Testing (EMDT's 2010)*, Ilmenau, Germany, September 2010.
- Nicha Kosindrdecha and Jirapun Daengdej, "A Test Generation Method Based on State Diagram", *Journal of Theoretical and Applied Information Technology*, August 2010, Vol. 18, No.2.
- Nicha Kosindrdecha, Siripong Roongruangsuwan and Jirapun Daengdej, "Reducing Test Cases Created by Path Oriented Test Case Generation", *Proceedings of the AIAA Conference and Exhibition*, Rohnert Park, California, USA: NASA AIAA, 2007.

TABLE OF CONTENTS

ACKNOWLEDGMENTS	i
LIST OF FIGURES	ii
LIST OF TABLES	iv
ABSTRACT	v
PUBLICATION	vi
CHAPTER 1: INTRODUCTION	1-14
1.1 Overview	1
1.2 Objectives of the Thesis	6
1.3 Principal Contributions	7
1.4 Definitions	7
1.5 Dissertation Organization	14
CHAPTER 2: LITERATURE REVIEW	15-90
2.1 Software Testing	17
2.2 Test Case Generation Technique	22
2.3 Test Data Generation Technique	51
2.4 Test Sequence Generation Technique	61
2.5 Test Case Generation Process	72
2.6 Related Works	75
CHAPTER 3: RESEARCH PROBLEMS	91-100
3.1 Research Issues	91
3.2 Problem Statement	97
CHAPTER 4: PROPOSED TECHNIQUES	101-137
4.1 Overview	101
4.2 Assumptions	109
4.3 Test Case Generation Process	109
4.4 Requirement Prioritization Based On User Satisfaction	113
4.5 Example of Requirement Prioritization	124
4.6 Test Case Generation Technique	128
4.7 Limitations	137

CHAPTER 5: EVALUATION	138-152
5.1 Experiments	138
5.2 Measurements	145
5.3 Results	146
5.4 Discussions	150
CHAPTER 6: CONCLUSION	153-157
6.1 Major Contributions	153
6.2 Discussion: The Most Suitable Approach	155
6.3 Future Research	157
REFERENCES	158-183



LIST OF FIGURES

Figure 1-1	A Common Failure Scenario of IT Projects	1
Figure 1-2	A Structure of UML 2.0 Diagrams	10
Figure 1-3	Example of UML Use Case Diagram	11
Figure 2-1	Software Development Life Cycle	17
Figure 2-2	Software Testing Process	19
Figure 2-3	A Classification of Test Case Generation Techniques	23
Figure 2-4	Specification-Based Test Case Generation Techniques	24
Figure 2-5	Sketch Diagram-Based Test Case Generation Techniques	36
Figure 2-6	Source Code-Based Test Case Generation Techniques	48
Figure 2-7	A Classification of Test Data Generation Techniques	51
Figure 2-8	Specification-Based Test Data Techniques	52
Figure 2-9	Source Code-Based Test Data Techniques	56
Figure 2-10	A Classification of Test Sequence Generation Techniques	62
Figure 2-11	Specification-Based Test Sequence Techniques	63
Figure 2-12	Sketch Diagram-Based Test Sequence Techniques	64
Figure 2-13	Test Case Generation Process	72
Figure 2-14	ISO/IEC 9126-1 Model for Internal and External Quality	86
Figure 3-1	A Classification of Remaining Problems	95
Figure 3-2	Research Problems Motivated This Dissertation	97
Figure 3-3	Matrix Table between Test Case and Use Case	99
Figure 4-1	Overview of Proposed Test Case Generation Technique	101
Figure 4-2	Relationship for ROI, Req. Complexity and User Satisfaction	105
Figure 4-3	Relationship between Number of Test Cases and Complexity	107
Figure 4-4	Proposed Methods Relative to Research Problems	108
Figure 4-5	Traditional Test Case Generation Process	110
Figure 4-6	Compare Test Case Generation Process	111
Figure 4-7	Requirement Prioritization based on User Satisfaction	112
Figure 4-8	Overview of Requirement Prioritization	115
Figure 4-9	Kano Model Analysis	116
Figure 4-10	WOW Factors and Implementation Cost	119
Figure 4-11	Example of Requirement Prioritization	125
Figure 4-12	Overwhelm Alternative Paths	132

LIST OF TABLES

Table 1-1	Principle Contributions against Objectives	7
Table 2-1	Specification-based Test Case Generation Techniques	33
Table 2-2	Sketch Diagram-Based Test Case Generation Techniques	44
Table 2-3	Source code-based Test Case Generation Techniques	50
Table 2-4	Specification-Based Test Data Techniques	55
Table 2-5	Source Code-Based Test Data Techniques	60
Table 2-6	Activity Diagram-Based Test Sequence Generation Techniques	67
Table 2-7	State Diagram-Based Test Sequence Generation Techniques	70
Table 2-8	Sequence Diagram-Based Test Sequence Generation Techniques	71
Table 2-9	First Process in “2D-4A-4D” Test Case Generation Process	73
Table 2-10	Second Process in “2D-4A-4D” Test Case Generation Process	74
Table 2-11	ISO/IEC 9126-1 Characteristics	86
Table 2-12	Testing Metrics	89
Table 3-1	Test Case Generation Techniques and Issues	96
Table 3-2	Problem Statements and Objectives	100
Table 4-1	Definitions of Factors Normally Considered in Literatures	102
Table 4-2	Reasons Why Factors Are Selected in Dissertation	104
Table 4-3	Measuring Requirement Complexity	123
Table 4-4	Total Estimated Cost	125
Table 4-5	Total Charges to Customer	126
Table 4-6	ROI for Each Requirement	127
Table 4-7	Ratio between ROI and Requirement Complexity	127
Table 4-8	Example Fully Dressed Use Case	129
Table 4-9	Extracted to Use Case Scenarios	129
Table 4-10	Extract to Test Scenarios	130
Table 4-11	Extract to Test Cases	130
Table 5-1	Generate Random Requirements	140
Table 5-2	Generate Random Use Case Scenario	142
Table 5-3	Generate Random Alternative Paths for Use Cases	144
Table 5-4	Attributes of Test Cases	144
Table 5-5	A Comparison Result for Test Case Generation Methods	150

Figure 4-13	Matrix Table between Tests and Paths	133
Figure 4-14	Example of Test Steps Required for Path	136
Figure 5-1	Overview of Experiment	138
Figure 5-2	Comparison Result for a Number of Test Cases	147
Figure 5-3	Comparison Result for Requirement Coverage	148
Figure 5-4	Result of Test Case Generation Methods	149
Figure 5-5	A Comparison for a Number of Test Cases	151
Figure 5-6	A Comparison for a Number of Test Cases and Coverage	152



CHAPTER 1

INTRODUCTION

1.1 Overview

Many IT software projects fail to deliver the software product on time and within budget. Those projects fail when they are not managed well, insufficient control is exercised, the appropriate skills are missing and the testing is inadequate. A common failure scenario of IT software project can be shown as follows:



Figure 1-1 A Common Failure Scenario of IT Projects

From Figure 1-1, the primary reasons why IT software projects fail can be addressed as follows [173]:

1. Miscommunication of requirements, resources and timescales.
2. Poor management, planning and control.

3. Poor software quality and inadequate testing.
4. Unrealistic timescales.
5. Failure to manage user expectations and changes required.

In fact, poor software quality and inadequate testing are one of five primary causes of failure. In general, testing typically consumes 40 to 50 percent of development efforts, which positions the software testing phase to be one of the most important activities of development projects.

Testing is the process of executing a program or system with the intent of finding errors [120]. It involves any activities aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results [68]. Software processes are not, unlike other physical processes, inputs that are received and outputs that are produced. Where software differs is in the manner in which it fails. Most physical systems fail in a fixed set of ways. However, software can fail in many other peculiar ways. Detecting all of the different failure modes in software is generally infeasible.

There is a process called “software development life cycle”, or SDLC, for developing IT software projects. In general, the waterfall software development life cycle contains five phases as follows:

1. **Requirements.** This phase is to gather customers or users requirements.

Typically, customers have an abstract idea of what they want to do as an end result. They have no idea what software should do or look like. Therefore, the responsibilities of software engineers are: (a) gathering requirements and (b) analyzing those requirements for the implementation.

2. **Design.** This phase is to design the system by following the requirements. The main responsibility for software engineer is to ensure that the software system will meet the user requirements.
3. **Implementation.** This is the part of SDLC where software engineers actually write program or source code against the design.
4. **Testing.** This phase is an integral part of the SDLC. One of the main goals of software testing is to recognize defects or software bugs as early as possible.
5. **Maintenance.** This phase is necessary when software engineers discover a new requirement, fix bugs or changed requirements.

A lot of researchers [3][15][21][22][23][69][75][136][156][159][161] have proven that software testing is one of the most critical phases in software development life cycle and that it takes approximately half of the time and effort from the SDLC. Generally, software testing process contains the following steps [134][168]:

1. **Test Planning.** This step is to establish test strategy, produce test plan and define testing criteria.
2. **Test Generation.** This step is used to generate test cases including test steps and prepare test data.
3. **Test Execution.** This step is to execute the generated test cases along with the prepared test data. Also verify actual and expected results.
4. **Test Evaluation.** This step is to evaluate test results and create test reports.

The studies [3][15][21][22][23][69][75][136][156][159][161] present that test case generation has been proven to be one of the most important phases in the software testing process. This is because good test cases can help engineers to detect defects, to maximize a number of faults, to block premature product release, to help

with the decision of releasing or not releasing the software, to minimize technical support costs and to access conformance to specification [34].

In addition, the studies show that many test case generation techniques have been proposed over a long period of time. Those techniques are developed to effectively generate a set of test cases while minimizing a number of test cases and maximizing requirements coverage. Unfortunately, none of existing test case generation methods concentrates on a user satisfaction [4][14][123][178][197]. All of the existing test case generation methods are mainly developed to enhance the ability to generate test cases based on software testing perspectives only. Those methods fail to generate test cases that cover critical requirements, which have an impact on the user satisfaction. These studies [99][100][114][115][124] demonstrate that user satisfaction is a key to a project success, long-term relationship and maximum profits. Furthermore, the studies suggest that testing activities are one of the key factors to satisfy users. This is due to the fact that none of the users expects low quality of software.

The studies [4][14][24][124][125][178][197] explain that there are two major research problems in test case generation, (a) is being a large number of test cases and (b) it is the inadequate coverage of critical requirements. Due to the complexity of software development at present, software test engineers aim to generate a huge number of test cases in order to be able to verify and validate all requirements. Large number of test cases takes a greater amount of cost and effort. However, there are many approaches that have been proposed to minimize the number of tests, such as effective test case generation methods, [4][66][69][85][178], test case selection techniques during test execution and test case reduction methods [125]. In addition, these studies [14][99] reveal not only that there is an inadequate coverage for those

critical requirements but also that generated test cases may explicitly ignore critical requirements such as domain specific requirements and high return on investment functional requirements.

These studies [23][76] illustrate that there are two types of testing techniques: black-box¹ and white-box². This thesis concentrates on the black-box testing only. The reason for focusing on black-box testing only is based on the fact that earlier research [18][35][124] has proved (a) that testing activities should start at the beginning of the software development life cycle (b) one of the testing goals is to verify and validate requirements as early as possible [23] and (c) the cost of a defect resolution at the beginning is significantly less than the cost of fixing defects in later phases [35][124].

There are two groups of test case generation methods for black-box testing [124]: one group is to generate test cases from requirement specification document and the other is to derive test cases from model diagrams such as data flow diagram and UML (Unified Modelling Language) diagram [123]. The focus of this thesis is to generate test cases from UML use case diagram. The reasons for concentrating on the UML use case diagram is that it describes the behaviour of the system as well as being the first high-level diagram for development [24].

¹ Black box testing takes an external perspective of the test object to derive test cases. These tests can be functional or non-functional, though usually functional. The test designer selects valid and invalid inputs and determines the correct output. There is no knowledge of the test object's internal structure. This method of test design is applicable to all levels of software testing: unit, integration, functional testing, system and acceptance. The higher the level, and hence the bigger and more complex the box, the more one is forced to use black box testing to simplify. While this method can uncover unimplemented parts of the specification, one cannot be sure that all existent paths are tested.

² White box testing (a.k.a. clear box testing, glass box testing, transparent box testing, translucent box testing or structural testing) uses an internal perspective of the system to design test cases based on internal structure. It requires programming skills to identify all paths through the software. The tester chooses test case inputs to exercise paths through the code and determines the appropriate outputs.

There are many proposed techniques to generate test cases from UML use case diagram [24][69][85][105]. However, these previous investigations [123][124][197] seem to be insufficient. The outstanding problems are as follow: (a) lack of ability to classify and prioritize requirements before test case generation process (b) unable to determine which test cases should be removed during test case generation activities and (c) large number of test cases due to large number of alternate paths described in each use case. This thesis proposes the following to resolve these research issues: (a) requirement prioritization based on user satisfaction [89][115] (b) classify requirement from business' perspective [100][114] (c) remove test cases during test case generation process and (d) finally enhance ability to reduce a number of test cases.

1.2 Objectives of the Thesis

This section describes objectives of this dissertation. The dissertation concentrates on test case generation techniques, where the test cases are derived from UML use case diagram. The following are the objectives of this research:

1. Prioritize requirements based on user satisfaction prior to generate test cases in order to improve the ability to generate and select the most suitable test cases.
2. Propose an alternative path point formula in order to systematically select which test cases could be removed during test case generation activities.
3. Enhance ability to minimize a number of test cases by adding a complexity factor.

1.3 Principal Contributions

The following table shows all of the principle contributions of this dissertation against the above three objectives:

Table 1-1 Principle Contributions against Objectives

Objective	Principle Contribution
Objective #1 - Prioritize requirements based on user satisfaction prior to generate test cases in order to improve the ability to generate and select the most suitable test cases.	Investigate and propose a marketing-driven requirement prioritization technique based on user satisfaction during test case generation process. According to this method, software test engineers can classify and prioritize requirements before generating test cases.
Objective #2 - Propose an alternative path point formula in order to systematically select which test cases could be removed during test case generation activities.	Introduce an alternative path point formula along with a correlation between return on investment (ROI) and a complexity of requirements. This method is developed for selecting the test cases that should be removed.
Objective #3 - Enhance ability to minimize a number of test cases by adding a complexity factor.	Propose a removable score that combine both of alternative path point formula and a complexity factor to minimize a number of test cases.

1.4 Definition

This section provides definitions used in this dissertation. The following are the definition of terminologies used in this thesis:

Cem [36] defined *test scenario* as follows:

"Test Scenario is a software testing activity that uses scenario tests, or simply scenarios, which are based on a hypothetical story to help a person think through a complex problem or system for a testing environment."

Mealy [163] defined Mealy Machine diagram as follows:

"Mealy machine is a finite state transducer that generates an output based on its current state and input. This means that the state diagram will include both an input and output signal for each transition edge. In contrast, the output of a

Moore finite state machine depends only on the machine's current state; transitions are not directly dependent upon input. However, for each Mealy machine there is an equivalent Moore machine."

John [133] presented test case as:

"Test case is a document that defines a test item and specifies a set of test inputs or data, execution conditions, and expected results. The inputs/data used by a test case should be both normal and intended to produce a 'good' result and intentionally erroneous and intended to produce an error. A test case is generally executed manually but many test cases can be combined for automated execution."

Weyuker [157] defines test data as:

"Test Data are data which have been specifically identified for use in tests, typically of a computer program. Some data may be used in a confirmatory way, typically to verify that a given set of input to a given function produces some expected result. Other data may be used in order to challenge the ability of the program to respond to unusual, extreme, exceptional, or unexpected input. Test data may be produced in a focused or systematic way (as is typically the case in domain testing), or by using other, less-focused approaches (as is typically the case in high-volume randomized automated tests). Test data may be produced by the tester, or by a program or function that aids the tester. Test data may be recorded for re-use, or used once and then forgotten."

Brucker [182] defines test sequence as:

"Test sequence can also be used for specifying the test of a transition function under test, which takes some input of type and some state of type and can produce a successor state."

Cem [158] defined *test oracle* as follows:

"An oracle is a mechanism used by software engineers for determining whether that product has passed or failed a test. It is used by comparing the output(s) of a product for a given test case input to the outputs that the oracle determines that product should have. Oracles are always separate from the product under test."

Black [160] defined *system testing* as follows:

"System testing of software or hardware is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System testing falls within the scope of black box testing, and as such, should require no knowledge of the inner design of the code or logic."

As a rule, system testing takes, as its input, all of the "integrated" software components that have successfully passed integration testing³ and also the software system itself integrated with any applicable hardware system(s). The purpose of integration testing is to detect any inconsistencies between the software units that are integrated together (called assemblages) or between any of the assemblages and the hardware. System testing is a more limiting type of testing; it seeks to detect defects both within the "inter-assemblages" and also within the system as a whole."

Grady [161] defined *UML diagram* as follows:

³ Integration testing is the activity of software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before system testing.

“Unified Modeling Language (UML) is a standardized general-purpose modeling language in the field of software engineering. UML includes a set of graphical notation techniques to create abstract models of specific systems. UML offers a standard way to write a system's blueprints, including conceptual components such as actors, business processes, system's components, activities, programming language statements, database schemas and reusable software components.”

Armin [162] defined UML 2.0 diagrams as follows:

“UML 2.0 has 13 types of diagrams divided into three categories. Six diagram types represent the structure application, seven represent general types of behavior, including four that represent different aspects of interactions.”

Armin [200] classified UML 2.0 diagrams as follows:

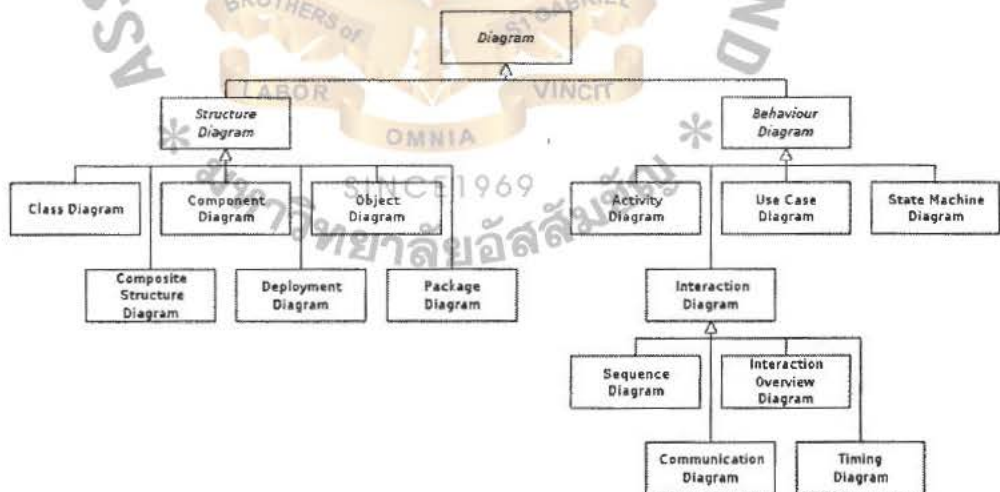


Figure 1-2 A Structure of UML 2.0 Diagrams

Grady [60] defined UML use case diagram as follows:

“Use Case diagram in the Unified Modeling Language (UML) is a type of behavioral diagram defined by and created from a Use-case analysis. Its purpose is to present a graphical overview of the functionality provided by a

4465 e 1

system in terms of actors, their goals (represented as use cases), and any dependencies between those use cases."

The example of UML use case diagram can be shown as follows:

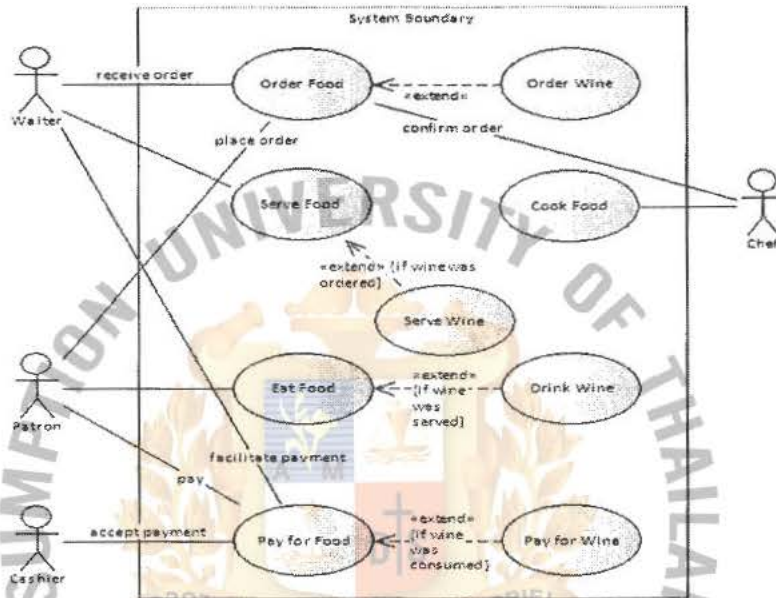


Figure 1-3 Example of UML Use Case Diagram

Alistair [40][151] divided use case into three categories: (a) brief use case (b) casual use case and (c) fully dressed use case, as follows:

"Brief use case contains the followings: use case name, use case number and purpose of use case. Casual use case contains the followings: use case name, use case number, purpose of use case and summary description. Fully dress use case contains the followings: use case name, use case number, purpose of use case, summary description, actors, stakeholder, pre-conditions, post-conditions, basic event, alternative events, business rules, notes, version, author and date."

Percy [136] adapted the *error* terminology recommended by the IEEE Computer Society as:

"An error is made by somebody. A good synonym is mistake. When people make mistakes during coding, we call these mistakes bugs⁴. A fault is a representation of an error. As such it is the result of an error. A failure is a wrong behavior caused by a fault. A failure occurs when a fault executes"

Karthikeyan [177] defined *traceability matrix* as follows:

"A traceability matrix is a table that correlates any two baseline documents that require a many to many relationship to determine the completeness of the relationship. It is often used with high-level requirements (sometimes known as marketing requirements) and detailed requirements of the software product to the matching parts of high-level design, detailed design, test plan, and test cases."

Nicha [124] defined *users* as follows:

"A user is a person who uses a computer or Internet service. A user may have a user account that identifies the user by a username (also user name), screen name (also screen name). To log in to an account, a user is typically required to authenticate himself/herself/itself with a password or other credentials for the purposes of accounting, security, logging, and resource management."

⁴ A software bug is an error, flaw, mistake, failure, or fault in a computer program that prevents it from behaving as intended (e.g., producing an incorrect or unexpected result). Most bugs arise from mistakes and errors made by people in either a program's source code or its design, and a few are caused by compilers producing incorrect code. A program that contains a large number of bugs, and/or bugs that seriously interfere with its functionality, is said to be buggy. Reports detailing bugs in a program are commonly known as bug reports, fault reports, problem reports, trouble reports, change requests, and so forth.

Meuter [115] defined *customer satisfaction* as follows:

"Customer satisfaction, a business term, is a measure of how products and services supplied by a company meet or surpass customer expectation. It is seen as a key performance indicator within business and is part of the four of a Balanced Scorecard. In a competitive marketplace where businesses compete for customers, customer satisfaction is seen as a key differentiator and increasingly has become a key element of business strategy."

Nicha [124] defined *market driven requirement prioritization* as follows:

"Market-driven requirement prioritization is a requirement prioritization based on user satisfaction."

Tokman [100] defined *WOW factors* as follows:

"WOW factors contain three levels of user satisfaction, which are: basic, surprise and extraordinary."

1.5 Dissertation Organization

Chapter 1 introduces an overview of the dissertation along with objectives and scope of research, contributions and definitions. This chapter discusses why this dissertation is important together with the background of software testing. Research problems are also given.

Chapter 2 discusses and includes a significant literature reviews in software testing, test case generation, test data generation and test sequence generation area. The literature survey includes problems and limitations of each technique. This chapter is concluded with a literature review that breaks down tasks in test case generation techniques.

Chapter 3 discusses all research problems in software testing area. This chapter also discusses the outstanding research problems which are the motivation of this dissertation.

Chapter 4 introduces a requirement prioritization based on user satisfaction. Also, Chapter 4 introduces an effective model-based test case generation method for black-box testing along with alternate path points and retain score.

Chapter 5 presents an experiment design and measurement metrics in order to determine the most recommended automated test case generation techniques. Also, it discusses an evaluation result of the experiments.

Chapter 6 is the outcomes of this dissertation along with limitations of the studied. Also, this chapter provides a direction of the future research into test case generation techniques.

Finally, the last chapter contains all references used in this thesis.

CHAPER 2

LITERATURE REVIEW

This chapter discusses and includes the literature reviews for this research. It describes the following topics:

1. **Software Testing.** John [22] claimed that software testing is one of the most critical and important phases in software testing. For instance, *"In June 1996 the first flight of the European Space Agency's Ariane 5 rocket failed shortly after launching, resulting in an uninsured loss of \$500,000,000. The disaster was traced to the lack of exception handling for a floating-point error when a 64-bit integer was converted to a 16-bit signed integer"*. This has proven that software testing is one of the most critical phases that cannot be ignored.
2. **Test Case Generation.** Bertolino [23] proved that *"Test case generation is a most challenging and an extensively researched activity"*. Many test case generation techniques have been proposed in order to increase the ability to generate and prepare test cases, such as Antonio [136], Offutt [3] and Heumann [69]. In addition, Kaner [34] gave the purposes of test cases. For instance, find defects, maximizing bug count and help managers make go / no-go decision. This has proven that test cases and methods are one of the most challenging processes during software testing phase. Also, those researches presents that there are many methods to generate and prepare some parts in each test case such as input data (also known as test data generation), output data (also known as test data generation) and test steps (also known as test sequence generation).

3. **Test Data Generation.** Beizer [21] mentioned that “*Software testing accounts for 50% of the total cost of software development*”. Many researchers [15][27][39][53][67][74][80][81][82][94][107][116][122][156][161]

mentioned that automated test data generation is one of the approaches to reduce cost and prepare data values for each test case. In fact, test data is one of the components for test case format. This is concluded that test data generation is one of the interesting topics under software testing field.

4. **Test Sequence Generation.** According to above Beizer’s statement, another approach for reducing cost is to generate automatically test sequences that are parts of test cases. In fact, test sequences are steps described in each test case. Many methods have been proposed to identify the sequence of test case, such as Rayadurgam’s work [159], Hyungchoul’s work [75] and Frohlich’s work [137]. This shows that test sequence generation is one of other interesting topics.

5. **Related Works.** Apart from the above most challenging topics in software testing, there are other interesting topics that have been investigated in this study. For example, requirement prioritization field, how to design practical test case format, the international quality standard and software testing metrics.

Eventually, this chapter is concluded with an overall test case generation process.

The following sections describe the above topics in details.

2.1 Software Testing

This section discusses and includes the software development life cycle, software testing process including reasons why software testing is important and example of test case and test data.

Typically, according to the waterfall software development life cycle below, there are five phases in the life cycle, which are: (a) requirements (b) design (c) implementation (also known as development) (d) verification (also known as software testing) and (e) maintenance.

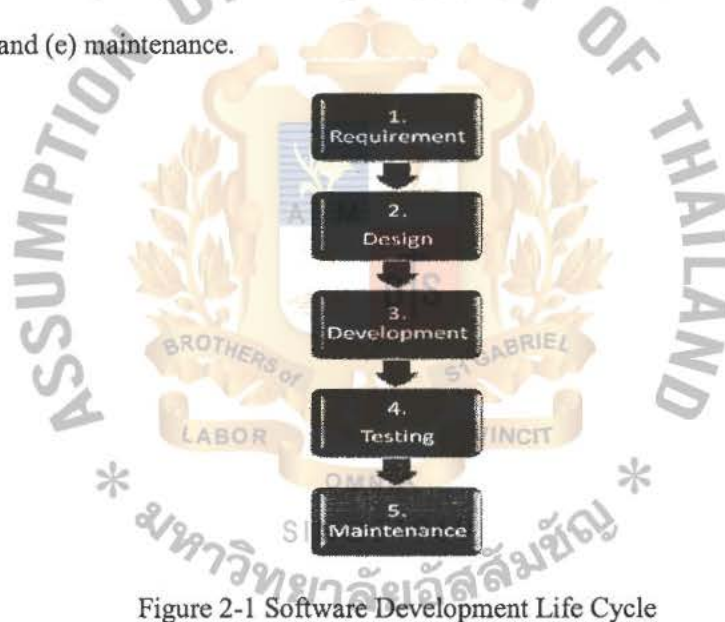


Figure 2-1 Software Development Life Cycle

From Figure 2-1, software testing phase is the process of executing a program or system with the intent of finding errors [120]. It involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results [68]. Software is not unlike other physical processes where inputs are received and outputs are produced. Where software differs is in the manner in which it fails. Most physical systems fail in a fixed (and reasonably small) set of ways. By contrast, software can fail in many bizarre ways. Detecting all of the different failure modes for software is generally infeasible.

Obviously, software testing is an essential activity in software development life cycle. In the simplest terms, it amounts to observing the execution of a software system to validate whether it behaves as intended and identify potential malfunctions. Testing is widely used in industry for quality assurance: indeed, by directly scrutinizing the software in execution, it provides a realistic feedback of its behavior and as such it remains the inescapable complement to other analysis techniques. Earlier studies estimated that testing can consume fifty percent, or even more, of the development costs [21], and a recent detailed survey in the United States [127] quantifies the high economic impacts of an inadequate software testing infrastructure.

The following are the list of examples why software testing is one of the most critical and important phases in software development life cycle [22].

1. *"In February 2003 the U.S. Treasury Department mailed 50,000 Social Security checks without a beneficiary name. A spokesperson said that the missing names were due to a software program maintenance error."*
2. *"In July 2001 a "serious flaw" was found in off-the-shelf software that had long been used in systems for tracking U.S. nuclear materials. The software had recently been donated to another country and scientists in that country discovered the problem and told U.S. officials about it."*
3. *"In October 1999 the \$125 million NASA Mars Climate Orbiter—an interplanetary weather satellite—was lost in space due to a data conversion error. Investigators discovered that software on the spacecraft performed certain calculations in English units (yards) when it should have used metric units (meters)."*
4. *"In June 1996 the first flight of the European Space Agency's Ariane 5 rocket failed shortly after launching, resulting in an uninsured loss of \$500,000,000."*

The disaster was traced to the lack of exception handling for a floating-point error when a 64-bit integer was converted to a 16-bit signed integer."

This is concluded that the impact of inadequate testing can be root-cause problems of: (a) increasing failures due to a poor quality (b) increasing software development costs (c) increasing time to market due to inefficient testing and (d) increasing market transaction costs [127]. Due to the above examples, software testing phase has proven that it is one of the most critical phases in SDLC.

Next paragraphs describe a general process of running software testing activities. This study includes the software testing process provided by Ian [168], who is the author of well-known software testing books, and Pan [134] from Carnegie Mellon University, as follows.

Ian [168] describes the software testing process as follows:

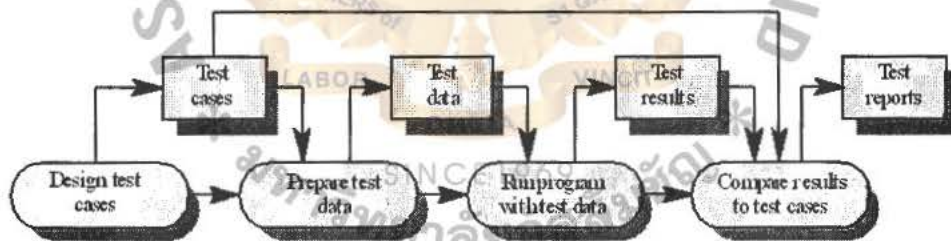


Figure 2-2 Software Testing Process

Figure 2-2 describes a general software testing process, which consists of four processes. Those processes are: (a) design test cases (b) prepare test data (c) run program with test data and (d) compare results to test cases. Each process has its own outcomes. There are four outcomes during these processes, which are: (a) a set of test cases (b) a set of test data (c) test results and (d) test reports. More detailed information in each process can be shown as follows:

1. **Design test cases.** The purpose of this step is to generate and prepare a set of test case. Therefore, the outcome of this step is a set of test cases. A set of test cases may represent as excel format, words document or database.
2. **Prepare test data.** The purpose of this step is to generate and prepare test data for each test case. The outcome of this step is a set of test data.
3. **Run program with test data.** This is an execution test step. Test case and test data will be run in this step. The result of this step is actual results.
4. **Compare results to test cases.** This step is used to compare the previous actual results and expected results design in test case. The milestone of this step is a test report of running test case and test data.

In addition, Pan [134] defines the typical life cycle of testing as follows:

- **Requirements analysis:** Software testing should begin in the requirements phase of the SDLC. Software testing engineer should play a major role during the requirement phase. During the design phase, software testing engineers work with developers in determining what aspects of a design are testable and with what parameters those tests work.
- **Test planning:** Test strategy¹, test plan, testbed creation. A testbed is a platform for experimentation for large development projects. Testbeds allow for rigorous, transparent and replicable testing of scientific theories,

¹ A test strategy is an outline that describes the testing portion of the software development cycle. It is created to inform project managers, testers, and developers about some key issues of the testing process. This includes the testing objective, methods of testing new functions, total time and resources required for the project, and the testing environment. In the test strategy is described how the product risks of the stakeholders are mitigated in the test levels, which test types are performed in the test levels, and which entry and exit criteria apply. The test strategy is created based on development design documents. The system design document is the main one used and occasionally, the conceptual design document can be referred to. The design documents describe the functionalities of the software to be enabled in the upcoming release. For every set of development design, a corresponding test strategy should be created to test the new feature sets.

computational tools, and other new technologies. There are many activities carried out during software testing process. Therefore, test planning is a must.

- **Test development:** In this step, it contains the following activities: develop test procedures, design test scenarios, produce test cases, prepare test datasets, and build test scripts to use in testing software.
- **Test execution:** Once test plan and test case, including test data, are already generated and prepared, software testing engineers can execute the software based on the plans and tests and report any errors found to the development team.
- **Test reporting:** When running test cases is completed, software testing engineers generate metrics and make final reports on their test effort² and whether or not the software tested is ready for release.
- **Test result analysis (also known as defect analysis):** This step is done by the testing team. It is usually done along with the client, in order to decide what defects should be treated, fixed, rejected (i.e. found software working properly) or deferred to be dealt with at a later time.
- **Retesting the resolved defects.** When a defect has been resolved with by the development team, re-testing those defects is a desirable.
- **Regression testing:** In general, it is common to have a small test program built of a subset of tests, for each integration of new, modified or fixed software, in order to ensure that the latest delivery has not ruined anything. Additionally, this step ensures that the software product as a whole is still working correctly.

² In software development, test effort refers to the expenses for (still to come) tests. There is a relation with test costs and failure costs (direct, indirect, costs for fault correction). Some factors which influence test effort are: maturity of the software development process, quality and testability of the test object, test infrastructure, skills of staff members, quality goals and test strategy.

- **Test Closure:** When the test meets the exit criteria, the activities such as capturing the key outputs, lessons learned, results, logs, documents related to the project are archived and used as a reference for future projects.

2.2 Test Case Generation Technique

This section describes test case generation techniques in details. Also, it discusses a limitation of each existing technique which has been researched in the literature.

Test case generation has always been fundamental to the testing process. Bertolino [23] articulated that the test case generation step is one of the most challenging and extensively researched activities during software testing phases. Many techniques have been proposed for test case generation, mainly random, source code-based technique (also known as path-oriented technique), goal-oriented and sketch diagram-based methods (also known as model-based approaches).

Random techniques determine a set of test cases based on assumptions concerning fault distribution. Path-oriented techniques generally use control flow graph to identify paths to be covered and generate the appropriate test cases for those paths. Goal-oriented techniques identify test cases covering a selected goal such as a statement or branch, irrespective of the path taken. There are many researchers and practitioners who have been working in generating a set of test cases based on the specifications. Modeling languages are used to get the specification and generate test cases. Since UML is the most widely used language, many researchers are using UML diagrams such as use case diagrams, activity diagram and sequence diagrams to generate test cases and this is called model-based test case generation techniques.

Due to the fact that there are many test case generation techniques, the studies and what have been found in the literature propose the following classification for

existing test case generation techniques. The study classifies those techniques based on source information from where test cases can be derived.

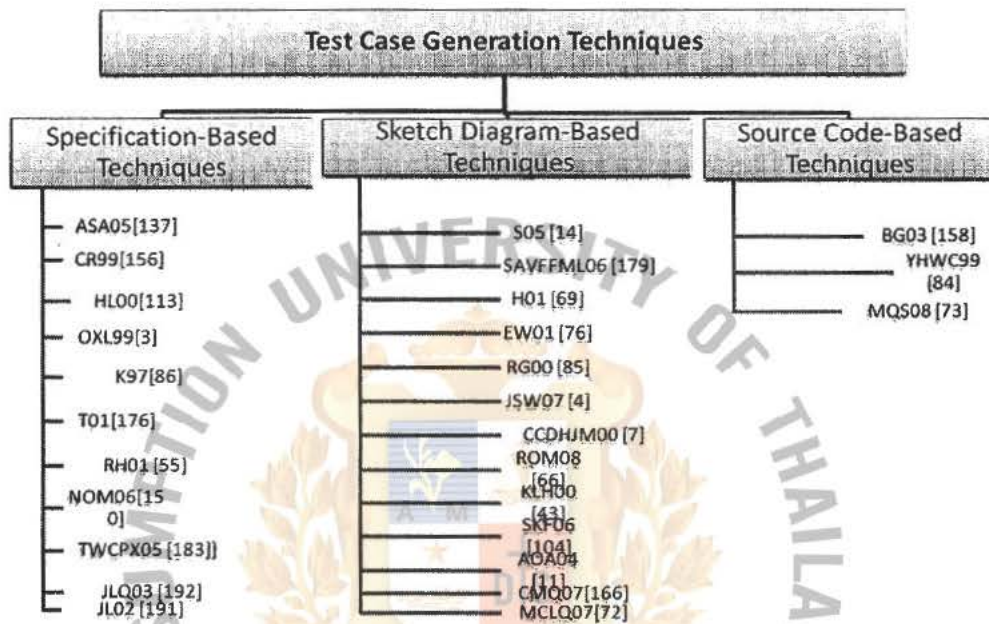


Figure 2-3 A Classification of Test Case Generation Techniques

Figure 2-3 presents that there are three categories, which are: (a) specification-based test case generation techniques (b) sketch diagram-based test case generation techniques and (c) source code-based test case generation techniques. There are three sources from where test cases can be derived: (a) requirement specification (b) model diagrams and (c) source code or program.

The following discusses the above techniques in details.

2.2.1 Specification-Based Test Case Generation Techniques

This section discusses an overview of how this technique works and provides a comprehensive survey of existing specification-based techniques.

An overview of this technique can be found as follows:

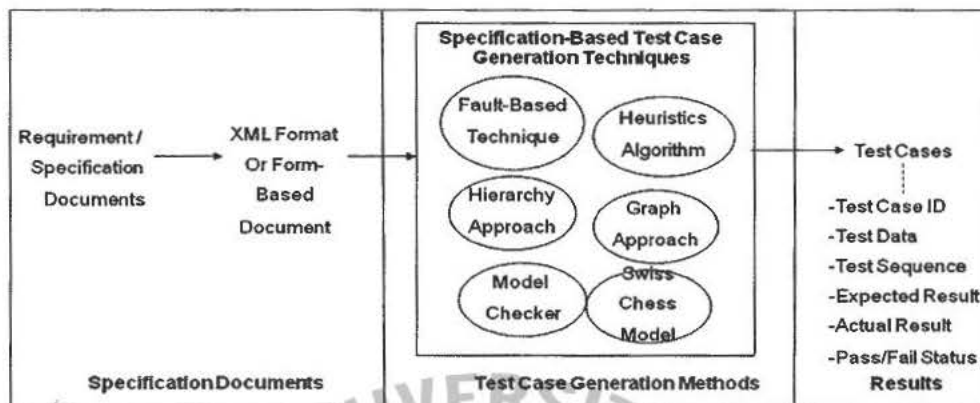


Figure 2-4 Specification-Based Test Case Generation Techniques

Specification-based techniques are methods to generate a set of test cases from specification documents such as a formal requirements specification [150][156][160][176][183], Z-specification [113][191][192] and object constraint language (OCL) specification [137].

In fact, the specification precisely describes what the system is without describing how to do it. Thus, the software test engineer has important information about the software's functionality without having to extract it from unnecessary details. The advantages of this technique include that the specification document can be used to derive expected results for test data, and that tests may be developed concurrently with design and implementation. The latter is also useful for breaking "Code now test later" practices in software engineering, and for helping develop parallel testing activities for all phases [113].

The specification requirement document can be used as a basis for output checking, significantly reducing one of the major costs of testing. Specifications can also be analyzed with respect to their testability [15]. The process of generating tests from the specifications will often help the test engineer discover problems with the specifications themselves. If this step is done early, the problems can be eliminated early, saving time and resources. Generating tests during development also allows

testing activities to be shifted to an earlier part of the development process, allowing for more effective planning and utilization of resources. Test generation can be independent of any particular implementation of the specifications [3].

Furthermore, the specification-based technique offers a simpler, structured, and more formal approach to the development of functional tests than non-specification based testing techniques do. The strong relationship between specification and tests helps find faults and can simplify regression testing. An important application of specifications in testing is to provide test oracles³.

The drawbacks of the specification-based technique with formal methods are: (a) the difficulty of conducting formal analysis and the perceived or actual payoff in project budget. Testing is a substantial part of the software budget, and formal methods offer an opportunity to significantly reduce testing costs, thereby making formal methods more attractive from the budget perspective [39] and (b) there is greater manual effort or processes in generating test cases, compared with techniques involving automatic generation processes.

This research reveals that many techniques have been proposed such as heuristics algorithms [86][156], model checkers [39][150][155] and hierarchy approaches [113][191][192]. The following paragraphs describe existing specification-based techniques that have been proposed for traditional and web-based application since 1997.

Percy Antonio [137] presented the underlying theory by providing a set of test cases with formal semantics and translated this general testing theory to a constraint satisfaction problem. A prototype test case generator serves to demonstrate the

³ A test oracle is a mechanism used by software engineers for determining whether the product has passed or failed a test. It is used by comparing the output(s) of a product for a given test case input to the outputs that the oracle determines that product should have. Oracles are always separate from the product under test [87].

automation of the method. It works on Object Constraint Language (OCL) specifications. The OCL is part of the UML⁴ 2.0 standard. It is a language allowing the specification of formal constraints in context of a UML model. Constraints are primarily used to express invariants of classes, pre-conditions and post-conditions of operations. These invariants become elements of test cases. In their work, they aimed to generate test-cases focusing on possible errors during the design phase of software development. Examples of such errors might be a missing or misunderstood requirement, a wrongly implemented requirement, or a simple coding error. In order to represent these errors, they introduced faults into formal specifications. The faults are introduced by deliberately changing a design, resulting in wrong behavior possibly causing a failure. They focused dedicatedly on the problem of generating test cases from a formal specification. The problem can be represented as a Constraint Satisfaction Problem (CSP). A CSP consists of a finite set of variables and a set of constraints. Each variable is associated with a set of possible values, known as its domain. A constraint is a relation defined on some subset of these variables and denotes valid combinations of their values. A solution to a constraint satisfaction problem is an assignment of a value to each variable from its domain, such that all the constraints are satisfied. Formally, the conjunction of these constraints forms a predicate for which a solution should be found. To resolve the above problem, they proposed to embed the test generation problem modeled as a CSP into a specially designed and implemented Constraint System. But this is not a novelty because this

⁴ Unified Modeling Language (UML) is a standardized general-purpose modeling language in the field of software engineering. UML includes a set of graphical notation techniques to create abstract models of specific systems. UML offers a standard way to write a system's blueprints, including conceptual components such as actors, business processes, system's components, activities, programming language statements, database schemas and reusable software components.

approach has been widely explored and implemented. The novelty in their approach is the relation that they formalized between fault-based testing and constraint solving.

Huaikou [113] presented a framework based on Phil and David's work [133][172]. They defined a test class using an object-oriented concept instead of Phil Stock's test template in the framework. Phil's test template defines test data only. The benefit of their test framework for Z specifications is that the test data and oracles are defined in a test class which also contains the information of before states and after states for an operation. The test framework is therefore a dynamic system involving state change, containing three components: (a) valid input space & output space (b) test class & test state space and (c) test class hierarchy & instantiation. These elements [113] can be described shortly as follows:

First, the valid input space (VIS) is the subset of the input space for which the operation is defined, and is also the subset of the input space satisfying the precondition of the operation. The valid input space can be derived directly from the formal specification of an operation, and it can be an automated process. The valid output space (VOS) can be defined similarly to the valid input space. It is the subset of output space satisfying the post-condition of the operation. The post-condition in VOS does not contain the predicate involving input variables. VOS is the source of all expected expressions. Second, the central concept of the framework is the Test Class (TC), which is the basic unit for defining a test case. A test case comprises test data and a test oracle. In a formal specification, the relationship between input states and output states is specified precisely. This means that the specification can serve as a test oracle. The simplest oracle is a comparison of the actual output for some input against a pre-calculated expected output for the same input. From the formal specification, it is simple to derive the description of expected output for given input.

Third, the structure approach is used to build a hierarchy of test classes. The hierarchy is similar to Phil's test template hierarchy [94][148]. The difference is that the nodes in Miao's hierarchy graph are test classes, not test templates.

Offutt [3] presented a model for developing test inputs from state-based specifications, and formal criteria for test case selection. For state-based specification technique, their paper used the term specification-based testing in the narrow sense of using specifications as a basis for deciding what tests to run on software. The proposed approach is related to Blackburn's state-based functional specifications of the software, expressed in the language, T-Vec [102]. It is used to derive disjunctive normal form constraints, which are solved to generate tests. Also, the approach is related to Weyuker [49] who presented a test case generation method from Boolean logic specifications. Moreover, they introduced several criteria for system level testing. These criteria are expected to be used both to guide the testers during system testing and to help the testers find rational, mathematical-based points at which to stop testing. In those criteria, tests are generated as multi-part, multi-step and multi-level artifacts. The multi-part aspect means that a test case is composed of several components: test case values, prefix values, verify values, exit commands, and expected outputs. The multi-step aspect means that tests are generated in several steps from the functional specifications by a refinement process. The functional specifications are first refined into test specifications, which are then refined into test scripts. The multi-level aspect means that tests are generated to test the software at several levels of abstraction.

Prasad [86] used a form of specification-based testing that employs the use of an automated theorem prover to generate test cases. A similar approach was developed using a model checker on state-intensive systems. The method applies to

systems with functional rather than stat-based behaviors. The approach allows for the use of incomplete specifications to aid in generation of tests for potential failure cases. He suggested a new method of testing software based on the formal specification. He used the Prototype Verification System (PVS) and its in-built theorem prover to derive test cases corresponding to the properties stated in the requirements.

Cunning [156] were interested in the model-based co-design of real-time embedded systems. It relies on system models at increasing levels of fidelity in order to explore design alternatives and to evaluate the correctness of these designs. As a result, the tests that they desire should cover all system requirements in order to determine if all requirements have been implemented in the design. The set of generated tests is maintained and applied to system models of increasing fidelity and to the system prototype in order to verify the consistency between models and physical realizations. In the co-design method, test cases are used to validate system models and prototypes against the requirements specification. In the paper, they presented continuing research toward automatic generation of test cases from requirements specifications for event-oriented, real-time embedded systems. They used a heuristic algorithm to automatically generate test cases in their works. The heuristic algorithm uses the greedy search method followed by a distance based search if needed. The algorithm with pseudo code is addressed in their paper [156].

Tran [176] focused on existing research in using model checking to generation test cases. He touched on several areas, like the methodology of properly testing software, the use of model checking to generate tests suits and specialization of specification to suit the needs of test generation. A model checker is used to analyze a finite-state representation of a system for property violations. If the model checker analyzes all reachable states and detects no violations, then the property holds.

However, if the model checker finds a reachable state that violates the property, it returns a counterexample – a sequence of reachable states beginning in a valid initial state and ending with the property violation. In his technique, the model checker is used as a test oracle to compute the expected outputs and the counterexamples it generates are used as test sequences. In summary, his approach is used to generate test cases by applying mutation analysis. Mutation analysis is a white-box method for developing a set of test cases which is sensitive to any small syntactic change to the structure of a program.

Rayadurgam [155] presented a method for automatically generating test cases to structural coverage criteria. They showed how, given any software development artifact that can be represented as a finite state model, a model checker can be used to generate complete test cases that provide a predefined coverage of that artifact. He provided a formal framework that is: (a) suitable for defining the test-case generation approach and (b) easily used to capture finite state representations of software artifacts such as program code, software specifications, and requirements models. They showed how common structural coverage criteria can be formalized in their framework and expressed as temporal logic formulae used to challenge a model checker to find test cases. Finally, they demonstrated how a model checker can be used to generate test sequences for modified condition and decision (MC/DC) coverage. Their approach to generating test cases involves using the model-checker as the core engine. A set of properties called trap properties [2] is generated and the model-checker is asked to verify the properties one by one. These properties are constructed in such a way that they fail for the given system specification.

Nilsson [150] has proposed a model based method for generating test cases to test timeliness by using heuristic driven simulation. Their approach is perfectly suited

to generating test cases for small real-time systems that contain shared resources, precedence constraints and few sporadic tasks. Conversely, in dynamic real-time systems there are many sporadic tasks, making model-checking impractical. For these dynamic real-time systems, they proposed an approach where a simulation of each mutant model is iteratively run and evaluated using genetic algorithms⁵ with application specific heuristics. By using a simulation-based method instead of model-checking for execution order analysis, the combinatorial explosion of full state exploration is avoided. Furthermore, they conjectured that it is easier to modify a system simulation than a model-checker, to correspond to the architecture of the system under test. In their paper, they focused on genetic algorithms. They included three types of functions needed to solve the specific search problem. Those three functions are: (a) a genome mapping function (b) heuristic cross-over functions and (c) fitness function.

Additionally, the literature shows that a few specification-based techniques for web-based application have been proposed. Those techniques can be described below.

Tsai [183] presented a framework that assures the trustworthiness of web services. New assurance techniques are developed within the framework, including specification verification via completeness and consistency checking, test case generation, and automated web services testing. Traditional test case generation methods only generate positive test cases that verify the functionality of software. The proposed “Swiss Cheese” test case generation method is designed to generate both positive and negative test cases that also reveal the vulnerability of web services. He

⁵ Genetic algorithms are one of the best ways to solve a problem for which little is known. They are a very general algorithm and so will work well in any search space. All you need to know is what you need the solution to be able to do well, and a genetic algorithm will be able to create a high quality solution. Genetic algorithms use the principles of selection and evolution to produce several solutions to a given problem.

presented that the first step of the development process before testing is to create a web service specification. The next step is to perform specification check. He focused on the completeness and consistency analysis for the specification which is then applied to their automated test cases generation technique. He also applied the verification patterns technique to generate many test cases by recognizing patterns in system behavior and generate the corresponding test cases by composition.

Jia [191][192] addressed limitations in web application testing, especially in testing the overall functionality of a web application. He believed that web application testing is a new area. Therefore, he proposed a new approach for rigorous and automatic testing of web applications using formal specifications. He applied Z notation, one of the best known formal methods, in their approach. The formal specification based approach is powerful, extensible, and versatile. It aims to address testing of various aspects of web applications, including functionality, security, and performance. He has developed a prototype tool based on the proposed approach, which accepts formal specifications in XML syntax. The approach covers functionality testing, page structure testing, security testing and performance testing (they classify performance testing as non-functional testing in their paper.). He used the formal specification language to specify the specification of functionality, security and performance of web application.

In conclusion, the above specification-based techniques can be summarized as the following table.

Table 2-1 Specification-based Test Case Generation Techniques

Author / Reference	Type of Application	Type of Testing Technique	Type of Specification	Method	Limitation
ASA05 [137]	Traditional Application	Black Box	Object Constraint Language	Fault-based Technique	<ol style="list-style-type: none"> 1. Lack of test case sequencing generation. 2. The input model can be extended. 3. Ignore the negative test case⁶.
CR99 [156]	Traditional Application	Black Box	Formal Requirement Specification	Heuristic Algorithm & Greedy Approach	<p>Their approach is limited to embedded systems only. The requirements model beyond FSMs can be improved for various systems.</p>
HL00 [113]	Traditional Application	Black Box	Z Specification	Hierarchy approach	<p>The analysis of the effectiveness of testing strategies and their tool, TCGS⁷ can be enhanced. The assessing testing strategies in their framework is very difficult and unclear.</p>
OXL99 [3]	Traditional Application	Black Box	State-based Specification	Graph approach	<ol style="list-style-type: none"> 1. Their technique is limited and they are looking forward to evaluating their technique for industrial applications.

⁶ Negative test case is a test case that report when a test fails. Meanwhile, positive test case is a test case that report when a test successes.

⁷ TCGS is a test case generation system, which is a sub system of Huaikou and Ling system, called Z User Studio.

Author / Reference	Type of Application	Type of Testing Technique	Type of Specification	Method	Limitation
					2. Their tool contains various restrictions on the form of the specifications
K97 [86]	Traditional Application	Black Box	Form-based specification	Heuristic Algorithm	Users can't select test templates with a specific property.
T01 [176]	Traditional Application	Black Box	Formal Requirement Specification	Model Checker by using mutation analysis ⁸	1. Model checking technology is not fully utilized in software testing. There is a lot of potential for model checking in automated test generation. 2. Model checkers have not gained acceptance in the software industry.
RH01 [155]	Traditional Application	Black Box	Formal Requirement Specification	Model Checker Technique	1. The problem of state space explosion affects the search for counter-examples. 2. The environment specification is always a difficult issue

⁸ Mutation analysis is a white-box technique to develop test cases which are sensitive to any small changes to the structure of a program.

Author / Reference	Type of Application	Type of Testing Technique	Type of Specification	Method	Limitation
					in modeling systems.
NOM06 [150]	Traditional Application	Black Box	Formal Requirement Specification	Model Checker by using mutation analysis	Their genome mapping function in their applied genetic algorithm is limited to small class of Timed Automata with Tasks (TAT) automata templates.
TWCPX05 [183]	Web Application	Black Box	Formal Requirement Specification	Swiss Cheese Model	Their approach is limited to two parameters: the Hamming distance and the boundary count.
JLQ03 [192]	Web Application	Black Box	Z specification	Hierarchy approach	<ol style="list-style-type: none"> 1. Their prototype tool is limited to simple web applications. 2. There are limitations to their simple complete test specifications to specify functionality security and performance of a web application.
JL02 [191]	Web Application	Black Box	Z specification	Hierarchy approach	<ol style="list-style-type: none"> 1. Their prototype tool is limited to simple web applications. 2. There are limitations to their simple complete test specifications to specify functionality,

Author / Reference	Type of Application	Type of Testing Technique	Type of Specification	Method	Limitation
					security and performance of a web application.

2.2.2 Sketch Diagram-Based Test Case Generation Techniques

This section discusses an overview of how this technique works and provides a comprehensive survey of existing sketch diagram-based techniques.

An overview of the sketch diagram-based technique can be found as follows:

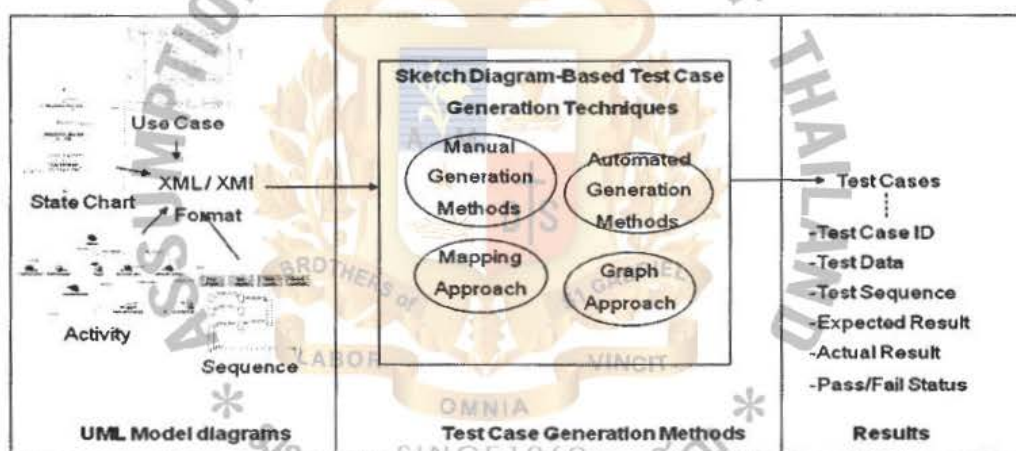


Figure 2-5 Sketch Diagram-Based Test Case Generation Techniques

From Figure 2-5, sketch diagram-based techniques are methods to generate test cases from model diagrams like UML Use Case diagram [69][85][105], UML Sequence diagrams [4] and UML State diagrams [7][11][14][43] [66][76][104][179]. The following paragraphs survey current sketch diagram-based test case generation techniques that have been proposed for traditional and web-based application for a long time.

A major advantage of model-based V&V⁹ is that it can be easily automated, saving time and resources. Other advantages are shifting the testing activities to an earlier part of the software development process and generating test cases that are independent of any particular implementation of the design [4].

The following paragraphs describe existing specification-based techniques that have been proposed for traditional and web-based application since 2000.

Heumann [69] presented how using use cases to generate test cases can help launch the testing process early in the development lifecycle and also help with testing methodology. In a software development project, use cases define system software requirements. Use case development begins early on, so real use cases for key product functionality are available in early iterations. According to the Rational Unified Process (RUP), a use case is used to fully describe a sequence of actions performed by a system to provide an observable result of value to a person or another system using the product under development. Use cases tell the customer what to expect, the developer what to code, the technical writer what to document, and the tester what to test. He proposed three-step process to generate test cases from a fully detailed use case: (a) for each use case, generate a full set of use-case scenarios¹⁰ (b) for each scenario, identify at least one test case and the conditions that will make it execute and (c) for each test case, identify the data values with which to test.

Ryser [85] raised the practical problems in software testing as follows: (a) lack of planning/time and cost pressure, (b) lack of test documentation, (c) lack of tool support, (d) formal language/specific testing languages required, (e) lack of measures,

⁹ V&V stand for verification and validation. Verification and validation is the process of checking that a product, service, or system meets specifications and that it fulfills its intended purpose

¹⁰ A use-case scenario is an instance of a use case, or a complete "path" through the use case. End users of the completed system can go down many paths as they execute the functionality specified in the use case.

measurements and data to quantify testing and evaluate test quality and (f) insufficient test quality. Their proposed approach to resolve the above problems is to derive test cases from scenarios or UML use cases and state diagrams. In their work, the generation of test cases is done in three stages: (a) preliminary test case and test preparation during scenario creation (b) test case generation from Statechart and dependency charts and (c) test set refinement by application dependent strategies (intuitive, experience-based testing).

Nilawar [105] was interested in testing web based applications. Web based applications are of growing complexity and it is a serious business to test them correctly. They focused on black box testing which enables the software testing engineers to derive sets of input conditions that will fully exercise all functional requirements. They believed that black box testing is more generally suitable and more necessary for web applications than other types of application. Furthermore, they proposed four steps to generate test cases, based on J. Heumann's four-steps [69], as follows: (a) prioritize use cases based on the requirement traceability¹¹ matrix (b) generate tentatively sufficient use cases and test scenarios (c) for each scenario, identify at least one test case and the conditions and (d) for each test case, identify test data values. They also presented that the test cases contains: a set of test inputs, execution conditions and expected results developed for a particular objective.

Sinha [14] described a new model based testing technique developed to identify critical domain requirements. The new technique is based on modeling the

¹¹ Requirements traceability is a sub-discipline of requirements management within software development and systems engineering. Requirements traceability is concerned with documenting the life of a requirement. It may be possible to find the origin of each requirement and track every change which was made to this requirement. For this purpose, it may be necessary to document every change made to the requirement.

system under test using a strongly typed domain specific language (DSL)¹². In the new technique, information about domain specific requirements of an application are captured automatically by exploiting properties of the DSL and are subsequently introduced in the test model. The new technique is applied to generate test cases for the applications interfacing with relational databases and the example DSL. Test suites generated using the new techniques are enriched with tests addressing domain specific implicit requirements.

Santiago [179] focused on test sequence generation from a specification of a reactive system, space application software, in Statecharts [63] and the use of PerformCharts [180]. In order to adapt PerformCharts to generate test sequences, it has been associated to a test case generation method, switch cover, implemented within the Condado tool [9]. Condado is a test case generation tool for FSM. The algorithm implemented in Condado is known as sequence of “de Bruijn”. The steps in the algorithm are: (a) a dual graph is created from the original one, by converting arcs into nodes (b) by considering all nodes in the original graph, where there is an arc arriving and another arc leaving, an arc is created in the dual graph (c) the dual graph is transformed into a “Eulerized” graph by balancing the polarity of the nodes and (d) finally, the nodes are traversed registering those that are visited.

El-Far [76] was interested in model-based testing and generating test cases from finite state machines. The difficulty of generating test cases from a model depends on the nature of the model. Models that are useful for testing usually possess properties that make test generation effortless. Sometimes generation processes can be

¹² A domain-specific language (DSL) is a small, usually declarative language that offers expressive power focused on a particular problem domain [22]. Through suitable abstractions, through embedded types and through specific library functions, the DSL imports domain knowledge into any application. Information about domain specific requirements can be captured automatically by exploiting properties of the DSL.

automated. For some models, one must go through combinations of conditions described in the model. In the case of finite state machines, it is as simple as implementing an algorithm that randomly traverses the state transition diagram. The sequences of arc labels along the generated paths are, by definition, tests.

Cavarra [7] described a modeling architecture for the purposes of model based verification and testing. Their architecture contains two components. The first component of the architecture is the system model, written in UML; this is a collection of class, state, and object diagrams: the class diagram identifies the entities in the system; the state diagrams explain how these entities may evolve; the object diagram specifies an initial configuration. The second component, again written in UML, is the test directive; this consists of particular object and state diagrams: the object diagrams are used to express test constraints and coverage criteria; the state diagrams specify test purposes. The system model and the test directives can be constructed using any of the standard toolsets, like Rational Rose.

Reza [66] discussed a model-based testing method for web applications that utilizes behavioral models of the software under the test (SUT) from Statechart models originally devised by Harel [62][63]. Statechart models can be used both for modeling and generating test cases for a web application. The main focus of their work is on the front end design and testing of a web application. As such, they utilize the syntax of the web pages to guide the specification of the Statecharts. Their approach is a systematic way to test the front-end functionality of a web application. For the most parts, they are concerned with verifying that the links, forms, and images in the web application under test function according to the specification documents. Furthermore, they address how to model the web application with Statechart diagrams in their work. To generate test cases from Statechart diagram, they defined 5 test

coverage criteria: (a) all-blobs, (b) all-transitions, (c) all-transition-pairs, (d) all-conditions and (e) all-paths.

Kung [43] presented a methodology that uses an Object-Oriented Web Test Model, called WTM, to support web application testing. The WTM captures test related artifacts of a web application and represents the artifacts from three different aspects: (a) the object aspect, which models the entities of a web application as objects and describes their dependent relationships (b) the behavior aspect, which depicts the navigation and state-dependent behaviors of a web application and (c) the structure aspect, that describes the control flow and data flow information of a web application.

From the WTM, the structural and behavioral test cases can be derived automatically to support the test processes. To facilitate web application testing, the structural and behavioral test artifacts of a web application are represented in the WTM from three aspects: the object, the behavior, and the structure perspectives. For the object perspective, the entities of a web application are depicted by an object relation diagram in terms of objects and their dependent relationships. For the behavior perspective, the navigation behavior of a web application is described using a page navigation diagram, while the state-dependent behavior of interacting objects is represented using a set of object state diagrams. For the structure perspective, a set of flow graphs are used to describe the control flow and data flow information of the scripts and functions in a web application. Furthermore, the WTM also employs textual test constraints so that special testing concerns for objects can be expressed.

There are many limitations for performance testing in web application. Those limitations are related to several requirements with respect to synthetic workloads. Firstly, to reach reliable conclusions based on the results of a performance test, the

synthetic workloads used must be representative of real workloads. Secondly, since it is very difficult to know precisely what a real workload's characteristics will be, a performance testing methodology must provide the flexibility to conduct a controlled sensitivity analysis on the characterizations of the workload model's attributes. Furthermore, since the scripts developed are system-specific they need to be modified when changes are made to a system (e.g., changes in inter-request dependency, addition of new functionality). Shams [104] proposed a model-based approach that addresses these limitations. Their approach uses an application model that captures the application logic of a session-based system under study. Essentially, the application model can be used to obtain a large set of user request sequences that satisfy the correct inter request dependencies for the system under study. This set of sequences is used to automatically construct a synthetic workload with desired characteristics.

Andrews [11] addressed the problem of test case generation for web applications. They were interested in proposing a new approach to improve the effectiveness and efficiency of test case generation for web applications. They proposed a system-level testing approach to combine test generation based on finite state machines with constraints in order to test the function of a web application. They proposed to use a hierarchical approach to model potentially enterprise scale web based applications. The approach builds Finite State Machines (FSMs) that model subsystems of the web applications, and then generates test requirements as subsequences of states in the FSMs. Their approach contains two phases: (a) to build a model of the web application and (b) to generate test cases from the model defined in the previous phase.

The model in the first phase can be done in four steps: (a) the web application is partitioned into subsystems and components (b) logical web pages are defined (c) a partition FSM is built for each subsystem or component and (d) an aggregation FSM is built for the web application.

Traditional testing approaches are no longer adequate for web applications. Although there is much established work in the validation and verification of traditional software [17][31][39][43][194][200], systematic as well as flexible and extensible testing approaches and intelligent tools are in urgent demand. In addition, software testing in general and web application testing in particular are knowledge-driven, labor intensive activities, which require automatic software methods and techniques. Brim [95] proposed a model of Component-Interaction automata to model component interactions. The model is designed to preserve all the interaction properties to provide a rich base for their further research. In their paper, they combine Logical Components (LCs) with component interaction and an agent to assist automatically generating test cases to test web applications. Chen [72][166] proposed to generating test cases proceeds in four steps. Firstly, test sequences of logical components (LC) are generated. Secondly, each LC is modeled by an automaton. Thirdly, a final automaton modeling each whole test sequence of LCs can be achieved by iterative composition of automata of pair-wise LCs in sequence. Lastly, after mapping actions of output and input into actual operations, and adding test data to the final automaton, final test cases can be generated automatically.

Javed [4] proposed a model-driven approach to test software applications using sequence diagrams. Sequence diagrams are behavioral elements of a UML design that describe dynamic interactions among the components of a system. They play an important role in the software development processes that are use-case driven,

such as the Rational Unified Process. Since these descriptions of behavior are constructed at an early stage, testing based on them can start verification and validation (V&V) activities early in the software life cycle. The model-driven approach that they use for generating unit test cases consists of two steps. In the first step, they modeled a sequence diagram as a sequence of methods calls (SMC) which is then automatically transformed into an xUnit model by applying model-to-model transformations using Tefkat. Tefkat is an eclipse modeling framework-based model transformation engine which is available as an Eclipse plug-in. In the second step, JUnit test cases are generated from the xUnit model by applying model-to-text transformations using MOFScript. MOFScript is a model-to-text transformation language generating textual outputs from models based on meta-models, and is available as an Eclipse plug-in.

In conclusion, the above sketch diagram-based techniques can be summarized as the following table.

Table 2-2 Sketch Diagram-Based Test Case Generation Techniques

Author / Reference	Type of Application	Type of Testing Technique	Type of Specification	Method	Limitation
S05 [14]	Traditional Application	Black Box	Extended Finite State Machine Diagram	Function	<ol style="list-style-type: none"> 1. The costs of designing, implementing and maintaining a Domain Specific Language (DSL). 2. The costs of education of DSL users. 3. The limited availability of DSL. 4. The difficulty of

Author / Reference	Type of Application	Type of Testing Technique	Type of Specification	Method	Limitation
					balancing between domain specificity and general-purpose programming language constructs. 5. The potential for a tower of Babel, a potential language for every other domain.
SAVFFM L06 [179]	Traditional Application	Black Box	Statechart diagram	Function	1. Their approach is limited to only one component in the system, not entire software. 2. Their technique is not applicable and effective for dynamic behavior modeling in Statechart diagrams.
H01 [69]	Traditional Application	Black Box	Use Case Diagram	Function	Lack of the integration of UML 2.0 standard specification.
EW01 [76]	Traditional Application	Black Box	Finite State Machines Diagram	Function	1. Their approach is randomly traversed in the finite state machines diagram. 2. Their approach

Author / Reference	Type of Application	Type of Testing Technique	Type of Specification	Method	Limitation
					requires a lot of skills for testers (e.g. formal language, automata theory, graph theory and elementary statistics.)
RG00 [85]	Traditional Application	Black Box	Use Case and Statechart diagram	Function	<ol style="list-style-type: none"> 1. Need to improve the integration of non-functional requirements with scenarios and Statechart diagram. 2. Limit to the data annotations and performance requirements in deriving test cases from annotated state-charts.
JSW07 [4]	Traditional Application	Black Box	Sequence diagram	Function	Lack of an automated test case generation tool.
CCDHJM 00 [7]	Traditional Application	Black Box	Class, Object and State diagram	Function	Limited to branch coverage only.
ROM08 [66]	Web Application	Black Box	Statechart diagram	Function	Cannot support tests involving concurrent access of web application by multiple users.
KLH00 [43]	Web Application	Black Box	Object Relation Diagram and Object State Diagram	Function	Limited to a few test artifacts to facilitate regression testing and maintenance of

Author / Reference	Type of Application	Type of Testing Technique	Type of Specification	Method	Limitation
					web applications.
SKF06 [104]	Web Application	Black Box	Extend Finite State Machine (EFSM)	Non-Function (Performance)	Lack of flexibility for varying workload characteristics in a controlled manner.
ND03 [105]	Web Application	Black Box	Use Case Diagram	Function	<ol style="list-style-type: none"> 1. Manual process of assigning priorities test cases. 2. Limit to the functionality of web application. Their approach cannot support the relationship between the navigation and functionality of web pages (e.g. page testing and hyperlink testing).
AOA04 [11]	Web Application	Black Box	Finite State Machine	Function	Test case generation effort is too manual.
CMQ07 [166]	Web Application	Black Box	Page-Flow diagram	Function	Limited to only the interaction of components in web application.
MCLQ07 [72]	Web Application	Black Box	Page-Flow diagram	Function	Limited to only the interaction of components in web application.

2.2.3 Source Code-Based Test Case Generation Techniques

This section discusses an overview of how this technique works and provides a comprehensive survey of existing source code-based techniques.

An overview of this technique can be found as follows:

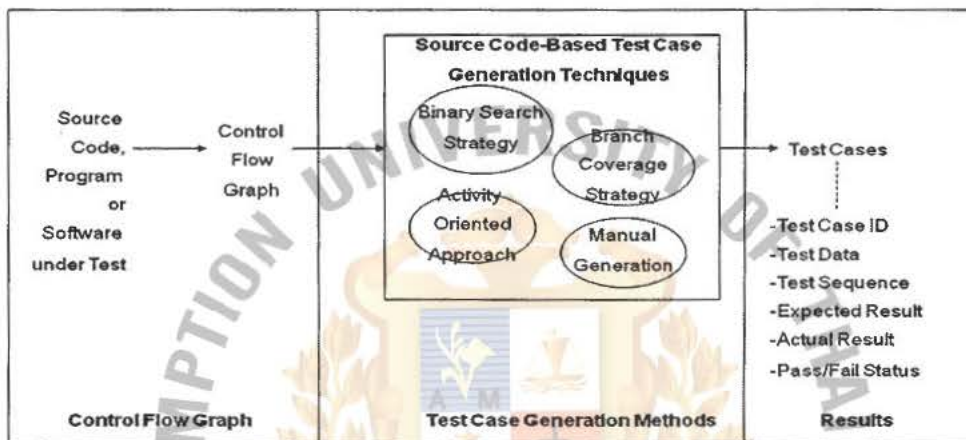


Figure 2-6 Source Code-Based Test Case Generation Techniques

From Figure 2-6, source code-based techniques generally use control flow information to identify a set of paths to be covered and generate appropriate test cases for these paths. The control flow graph can be derived from source code. The result is a set of test cases with the following format: (a) test case ID (b) test data (c) test sequence (also known as test steps) (d) expected result (e) actual result and (f) pass or fail status.

The following paragraphs describe the source code-based techniques that have been proposed for traditional and web-based applications.

Beydeda [158] presented a novel approach to automated test case generation. Several approaches have been proposed for test case generation, mainly random, source code-based, goal-oriented and intelligent approaches [151]. Random techniques determine test cases based on assumptions concerning fault distribution, e.g. [6]. Source code-based techniques generally use control flow information to

identify a set of paths to be covered and generate appropriate test cases for these paths. These techniques can further be classified as static or dynamic. Static techniques are often based on symbolic execution e.g. [29] whereas dynamic techniques obtain the necessary data by executing the program under test e.g. [94]. Goal-oriented techniques identify test cases covering a selected goal such as a statement or branch, irrespective of the path taken e.g. [151]. Intelligent techniques of automated test case generation rely on complex computations to identify test cases e.g. [127]. Another classification of automated test case generation techniques can be found in [127]. Their algorithm proposed in this article can be classified as a dynamic path-oriented one. Its basic idea is similar to that in [94]. The path to be covered is considered step-by-step, i.e. the goal of covering a path is divided into sub-goals, test cases are then searched to fulfill them. The search process, however, differs substantially. In Bogdan's work [94], the search process is conducted according to a specific error function. In their approach, test cases are determined using binary search, which requires certain assumptions but allows efficient test case generation.

Yang [84] presented a web application architecture to support testing of the web application. The architecture covers application model extraction, test execution automation, and test design automation. In addition, practitioners normally use a graph-based application model to represent the behavior of web-based applications. They are interested in extending the control flow graph (e.g. nodes, branches, and edges) to model web applications. The nodes in the control flow graph represent a programming module (e.g. single file such as .html, .cgi and .asp). The branch could be the user branch and application branch. The user branch represents the user selecting one of the hyperlinks from the browsed document in the browser. The application branch represents the current programming module forwarding control to

other programming modules for further processing based on application logic. The extended model is further used to generate test cases by applying the traditional flow-based test cases generation technique. They adopt two path testing strategies: statement and branch coverage for their environment. The IEEE software testing standard regards statement coverage as the minimum testing requirement. Real world, practical program testing requires both the statement and branch coverage. They declared four major steps for their testing activities in their framework: (a) application model construction (b) test case construction and composition (c) test case execution and (d) test result validation and measurement.

Bad web applications can have far-ranging consequences on businesses, economies, scientific progress, health and so on. Web application testing will play a more and more relevant part for ensuring requested software quality. Many aspects regarding web application testing have not been sufficiently investigated yet, and many open questions still need to be addressed, both in the technological and in the methodological field. Miao [73] proposed an approach to generating test paths for web applications. The intention is to help web application testers to ensure a reasonably comprehensive set of tests. The overall approach is simple and convenient. The main steps are: (a) construct a digraph from the web application schema (b) add an imaginary sink node for the default pages or pages leading nowhere (c) build a regular expression characterizing the digraph and (d) extract individual source-to-sink sequences from the regular expression.

In conclusion, the above techniques can be summarized as the following table.

Table 2-3 Source code-based Test Case Generation Techniques

Author / Reference	Type of Application	Type of Testing Technique	Method	Limitation
BG03 [158]	Traditional Application	White Box	Binary Search Strategy	It must have a total order existing in the

Author / Reference	Type of Application	Type of Testing Technique	Method	Limitation
				binary path.
YHWC99 [84]	Web Application	White Box	Branch Coverage Strategy	Limited to statement and branch coverage.
MQS08 [73]	Web Application	White Box	Control Flow Graph Approach	One of the most important problems of test generation is adequacy criteria.

2.3 Test Data Generation Technique

This section describes test data generation techniques in details. Also, it discusses a limitation of each existing technique which has been researched in the literature.

Through the years a number of different methods for generating test data have been presented such as Jon's studies [50], Grindal's work [107] and Hayes's works [67]. This dissertation classifies the existing test data generation techniques, based on source information from where test data can be derived, as follows:

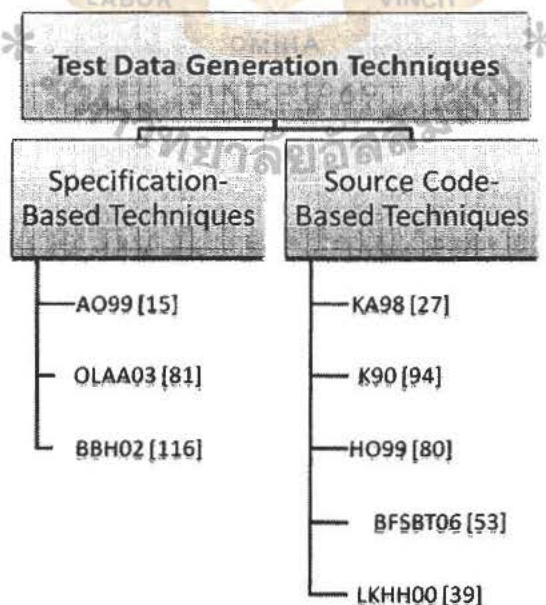


Figure 2-7 A Classification of Test Data Generation Techniques

Figure 2-7 presents that there are two groups of test data generation techniques: (a) specification-based test data generation techniques and (b) source code-based test data generation techniques. These techniques can be described in details as follows:

2.3.1 Specification-Based Test Data Generation Techniques

This section discusses an overview of how this technique works and provides a comprehensive survey of existing specification-based techniques.

An overall of this technique can be found as follows:

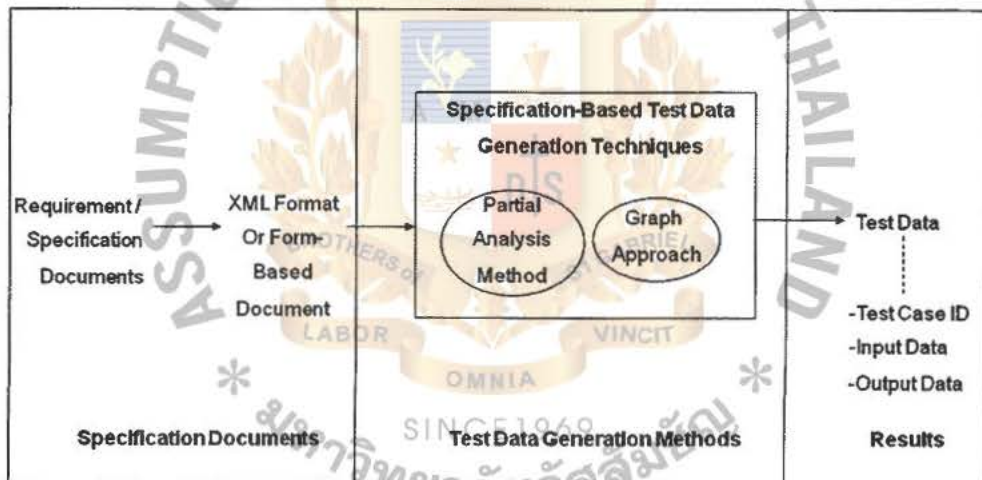


Figure 2-8 Specification-Based Test Data Techniques

From Figure 2-8, the specification-based techniques are methods to generate test data from specification documents such as state-based specification [15][81], OCL and test specification language (TSL) [116]. Eventually, those techniques generate a set of test data with the following format: (a) test case ID (b) input data and (c) output data.

The following paragraphs survey current specification-based test data generation techniques that have been proposed for a long time.

Previous attempts to automate the test generation process have been limited, having been constrained by the size and complexity of software, and the basic fact that in general, test data generation is an un-decidable problem. Meta-heuristic search techniques offer much promise in regard to these problems. Meta-heuristic search techniques are high-level frameworks, which utilize heuristics to seek solutions for combinatorial problems at a reasonable computational cost [111].

Abdurazik [15][81] defined the following definition in their work: Test requirements are specific things that must be satisfied or covered during testing; e.g., reaching statements are the requirements for statement coverage. Test specifications are specific descriptions of test cases including test data, often associated with test requirements or criteria. For statement coverage, test specifications are the conditions necessary to reach a statement. A testing criterion is a rule or collection of rules that impose test requirements on a set of test cases. A testing technique guides the tester through the testing process by including a testing criterion and a process for creating test data values. A test case is a general software artifact that includes test data input values, expected outputs, and any inputs that are necessary to put the software system into the state that is appropriate for the test input values. A TSL is a language that can be used to describe all components of a test case including input and output data. The components that they consider are test data values, pre x values, verify values, exit commands, and expected outputs. Test data values directly satisfy the test requirements, and the other components supply supporting values. A test data value is the essential part of a test case, the values that come from the test requirements. It may be a command, user inputs, or software function and values for its parameters. In state-based software, test data values are usually derived directly from triggering events and preconditions for transitions. A test data prefix value includes all inputs

necessary to reach the pre-state and to give the triggering event variables their before-values. Any inputs that are necessary to show the results are verify values, and exit commands depend on the system being tested. Expected outputs are created from the after-values of the triggering events and any post-conditions that are associated with the transition. In fact, the papers [15][81] presented a technique, which use Offut's state-based specification test data generation model to generate test data from UML state charts diagram.

Offutt [81] presented general criteria for generating test inputs from state-based specifications. The criteria include techniques for generating tests at several levels of abstraction for specifications (transition predicates, transitions, pairs of transitions and sequences of transitions). These techniques provide coverage criteria that are based on the specifications, and are made up of several parts, including test prefixes that contain inputs necessary to put the software into the appropriate state for the test values. The test generation process includes several steps for transforming specifications to tests. These criteria have been applied to a case study to compare their ability to detect seeded faults.

In object-oriented modeling, object constraint language (OCL) is used in the UML Semantics document to specify the well-formedness rules of the UML meta-model. OCL is a pure expression language and can be used to specify invariants, precondition, post-condition, and other kind of constraint (when the expressive power of the notation is not enough). The aim is often to constrain classes and types, to define pre- and post- conditions on operations and methods, to describe guards, and constraints on navigation. Despite its limitations, OCL seems to be now the main used language to formally constrain object-oriented models. Benattou [116] presented partition analysis concept, on which their approach for generating test data is based,

and they show by an example how to generate data from an OCL specification. The paper [116] had chosen to use the System Process Scheduler to illustrate partition analysis from OCL specification for two reasons: First, the specification of the system is very simple and second, they want to compare the results given in the context of the Vienna Development Method (VDM) specification with the object context of OCL.

The above current techniques can be summarized as follows:

Table 2-4 Specification-Based Test Data Techniques

Author / Reference	Type of Application	Type of Testing Technique	Type of Specification	Method	Limitation
AO99 [15]	Traditional Application	Black Box	State-based Specification and TSL	Graph Approach	Their approach is limited to software cost reduction (SCR) specifications that have only one mode class and UML specifications that have only one class with a statechart.
OLAA03 [81]	Traditional Application	Black Box	State-based Specification	Graph Approach	It is not clear that their approach is based on which UML standard specification.
BBH02 [116]	Traditional Application	Black Box	OCL specification	Partial Analysis Technique	<ol style="list-style-type: none"> 1. Their approach does not support inheritance concept in UML diagram. 2. The specification of the characteristics they are using in their approach is not completed.

2.3.2 Source Code-Based Test Data Generation Techniques

This section discusses an overview of how this technique works and provides a comprehensive survey of existing source code-based techniques.

An overall of this technique can be found as follows:

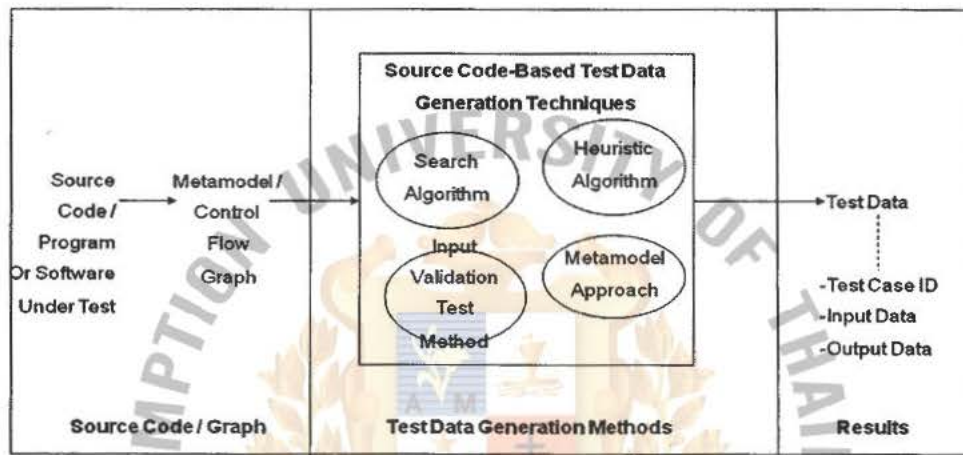


Figure 2-9 Source Code-Based Test Data Techniques

From Figure 2-9, the source code-based test data generation techniques are techniques to generate and prepare test data from control flow graph. The control flow graph can be derived from source or binary code. There are a few researchers who have researched this technique. Eventually, those techniques generate a set of test data with the following format: (a) test case ID (b) input data and (c) output data.

The following paragraphs survey current path-oriented test data generation techniques that have been proposed for traditional and web-based application for a long time.

Korel [27] presented a novel approach for automated regression testing. The main goal of this approach is to generate test data for a modified program such that each test data reveals a fault(s). The approach concentrates on testing automatically the common functionality of the original program and its modified version, i.e., it is used for programs whose functionality is unchanged after modifications. This is

achieved by utilizing the original version of the program in the process of test data generation. Specifically, this approach attempts to automatically generate an input data on which the original program and its modified version yield a different result (output). If such an input is found then an error(s) has been uncovered because both versions are expected to produce the same result. This error might be in the original program, the modified program, or in both programs. However, the error is most likely located in the modified program because the original program was well tested and previously used without problems.

Additionally, Korel [94] presented an alternative approach of test data generation, referred to as a dynamic approach of test data generation, which is based on actual execution of a program under test, dynamic data flow analysis, and function minimization methods. Test data are developed using actual values of input variables. When the program is executed on some input data, the program execution flow is monitored. If, during program execution, an undesirable execution flow at some branch is observed then a real-valued function is associated with this branch. This function is positive when a branch predicate is false and negative when the branch predicate is true. Function minimization search algorithms are used to automatically locate values of input variables for which the function becomes negative. In addition, dynamic data flow analysis is used to determine input variables which are responsible for the undesirable program behavior, leading to significant speed-up of the search process. In the paper's approach [94], arrays and dynamic data structures can be handled precisely because during program execution all variables values, including array indexes and pointers, are known; as a result, the effectiveness of the process of test data generation can be significantly improved.

Pringsulaka [140] proposed a technique called Coverall algorithm, which is based on a conventional attempt to reduce cases that have to be tested for any given software. The approach utilizes the advantage of Regression Testing where fewer test data would lessen time consumption of the testing as a whole. The technique also offers a means to perform test case generation automatically. Compared to most of the techniques in the literature where the tester has no option but to perform the test case generation manually, the proposed technique provides a better option. As for the test data reduction, the technique uses simple algebraic conditions to assign fixed values to variables (maximum, minimum and constant variables). By doing this, the variables values would be limited within a definite range, resulting in fewer numbers of possible test data to process. The technique can also be used in program loops and arrays. After a comparative assessment of the technique, it has been confirmed that the technique could reduce number of test data by more than 99%. As for the other features of the technique, automatic test data generation, all four step of test data generation in the proposed technique have been converted into an operational program.

Pringsulaka [140] resolved the following problems in order to improve the test performance: (a) reducing the number of test data (b) automatic test case generation and (c) minimum number of test runs. The purposes of the technique are:

1. To reduce number of all test data. Generally, the larger the input domain, the more exhaustive the testing would be. To avoid this problem, a minimum set of test data needs to be created using an algorithm to select a subset that represents the entire input domain. In addition, when test data are larger, the testing itself would take longer to run, particularly for regression testing where every change in the program demands repeat

testing. Therefore, reducing number of the test data does have advantage in efficiency

2. To find the technique for automatic generation of test data. To reduce the high cost of manual software testing while increasing reliability of the testing processes, IT researchers and technicians have found methods to automate the reduction process. With the automatic process, the cost of software development could be significantly reduced.
3. To keep a minimum number of test runs. The best technique must be able to generate

Hayes [80] was interested in the input validation testing (IVT) technique. The IVT technique has been developed to address the problem of statically analyzing input command syntax as defined in English textual interface and requirements specifications and then generating test data for input validation testing. The technique does not require design or code, so it can be applied early in the lifecycle. Input validation testing (IVT) focuses on the specified behavior of the system and uses a graph of the syntax of user commands. IVT incorporates formal rules in a test criterion that includes a measurement and stopping rule. Several grammar analysis techniques have been applied as part of the static analysis of the input specification.

This discusses the four major aspects of the IVT method:

1. How to specify the format of specifications
2. How to analyze a user command specification
3. How to generate valid test data for a specification
4. How to generate error test data for a specification.

Brottier [53] were interested in the automatic generation of test models, being given a meta-model describing the input domain of a model transformation. An

algorithm is defined to automate test model generation. The algorithm takes a meta-model and fragments of models as an input and produces a set of test models. The model fragments are either provided by the tester or derived from the meta-model. They specified parts of the meta-model that should be instantiated with particular values that are interesting for testing. The algorithm then consists in combining model fragments and completing them to build valid instances of the meta-model. The various strategies used to combine and complete a model to make it conformant to its meta-model are presented as well as the limitations of this algorithm.

Chien-Hung [39] extended traditional data flow testing techniques to web applications. Several data flow issues for analyzing HTML documents in web applications are discussed. A test model that captures data flow test artifacts of web applications is presented. In the test model, each component of a web application is modeled as an object. The dataflow information of the web application is captured using flow graphs. From the test model, dataflow test data for the web application then can be derived based on the intra-object, inter-object, and inter-client perspectives.

The above current techniques can be summarized as follows:

Table 2-5 Source Code-Based Test Data Techniques

Author / Reference	Type of Application	Type of Testing Technique	Method	Limitation
KA98 [27]	Traditional Application	White Box	Heuristics Approach	1. Limit to the functionality testing. 2. Their approach is used for programs where functionality is unchanged after modifications, during regression testing.
K90 [94]	Traditional Application	White Box	Heuristics Approach & Function minimization	Their approach is limited to local optimization for test data generation.

Author / Reference	Type of Application	Type of Testing Technique	Method	Limitation
			on search algorithms	
HO99 [80]	Traditional Application	White Box	Input Validation Test Method	Their approach is limited to statically analyzing input command syntax as defined in English textual interface and requirement specifications.
BFSBT06 [53]	Traditional Application	White Box	Metamodel Transformation	Their approach can not deal with static constraints associated to the input meta-model.
LKHH00 [39]	Web Application	White Box	Heuristics Approach & Search Algorithm	Lack of automated test data generation tool.

2.4 Test Sequence Generation Technique

This section describes test sequence generation techniques in details. Also, it discusses a limitation of each existing technique which has been researched in the literature.

Several approaches have been proposed to identify the sequence of test case, such as Sanjai's work [159], Hyungchoul's work [75] and Frohlich's work [137].

This dissertation classifies the existing test sequence generation techniques, based on source information from where test data can be derived, as follows:

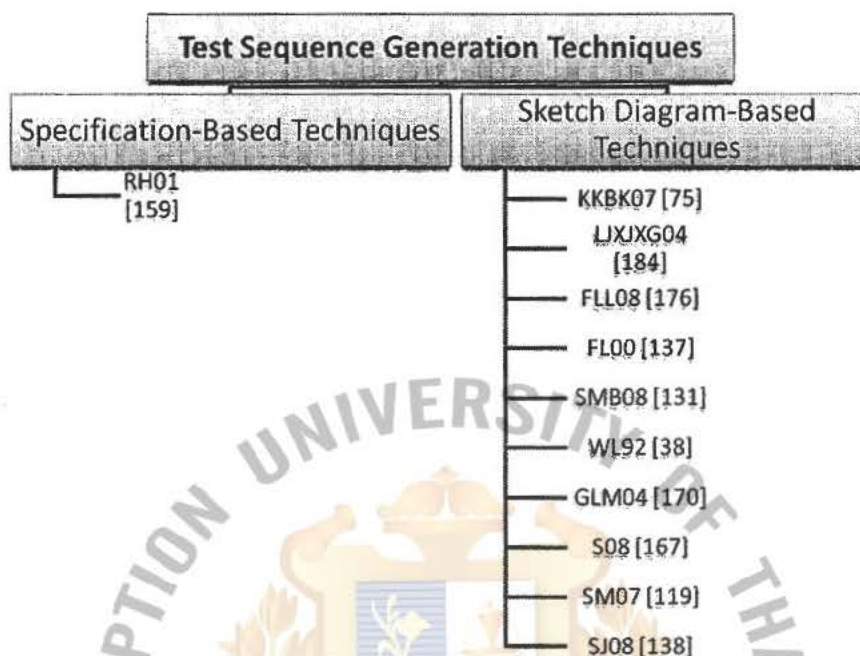


Figure 2-10 A Classification of Test Sequence Generation Techniques

Figure 2-10 presents that there are two groups of test sequence generation techniques: (a) specification-based test sequence generation techniques and (b) sketch diagram-based test sequence generation techniques. These techniques can be described in details as follows:

2.4.1 Specification-Based Test Sequence Generation Techniques

This section discusses an overview of how this technique works and provides a comprehensive survey of existing specification-based techniques.

An overall of this technique can be found as follows:

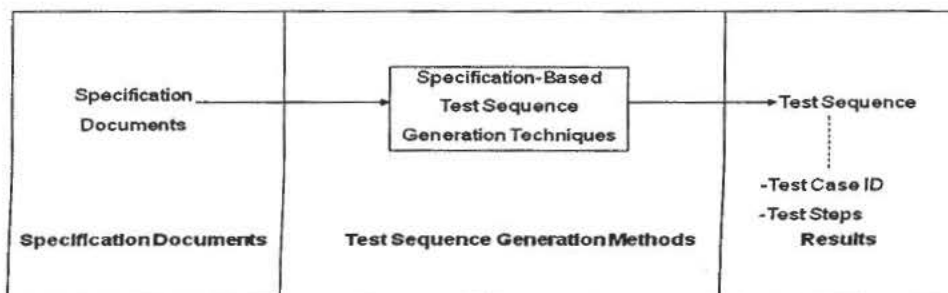


Figure 2-11 Specification-Based Test Sequence Techniques

From Figure 2-11, the specification-based techniques are methods to generate test sequence from specification documents. A few researchers have researched this area. Eventually, those techniques generate a set of test data with the following format: (a) test case ID and (b) test steps.

The following paragraphs survey current specification-based test sequence generation techniques that have been proposed for traditional and web-based application for a long time.

Rayadurgam [159] outlined a specification-centered approach to testing where they rely on a formal model of the required software behavior for test-case generation, as well as, an oracle to determine if the implementation produced the correct output during testing. Their work is based on the hypothesis that model checkers can be effectively used to automatically generate test sequences that provide a predefined structural coverage of a formal specification. Paper [154] defined formalism suitable for representing software engineering artifacts in which various structural test coverage criteria can be defined. Here, they show how this formal foundation can be used to generate structural tests from a formal specification of the required software behavior, using a small example from the avionics domain. To illustrate the approach, they define a set of structural coverage criteria that are applicable to requirements specified in RSML [186][199] or a similar formal language. While the specific criteria are indeed dependent on the specification language, the formal foundation is language independent and the underlying approach is equally applicable to any other language that can be model-checked. They show how the model can be translated into the input language of a model checker like SMV and how the coverage criteria can be captured as CTL or LTL properties. Test sequences are then generated by challenging

a model checker to find counter examples to the coverage criteria - such a counter example comprises a test sequence. This strategy has been used by [2] and [130].

2.4.2 Sketch Diagram-Based Test Sequence Generation Techniques

This section discusses an overview of how this technique works and provides a comprehensive survey of existing sketch diagram-based techniques.

An overall of this technique can be found as follows:

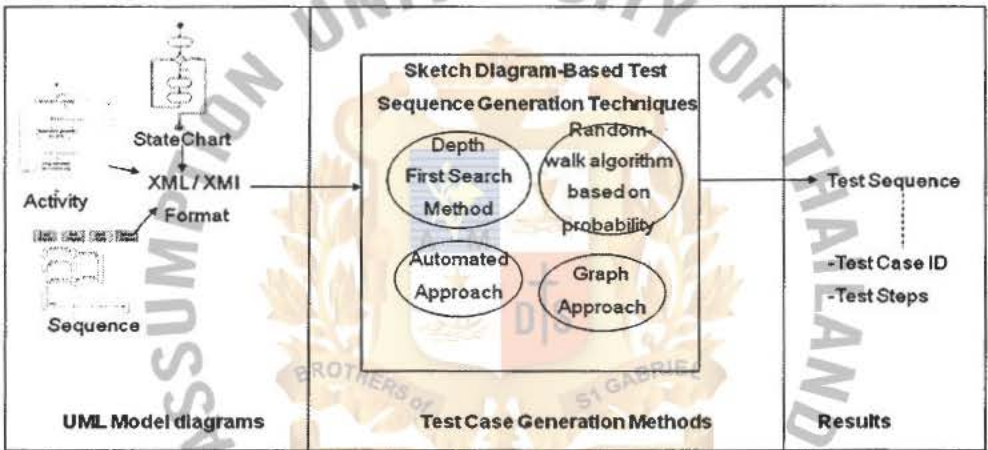


Figure 2-12 Sketch Diagram-Based Test Sequence Techniques

From Figure 2-12, there are many types of diagram used to generate test sequences, which are: 1) activity diagram 2) state diagram and 3) sequence diagram. Each method can be described as below.

1. Activity Diagram Based Technique

The following describes a test sequence generation technique, which prepare and generate test sequence from UML Activity diagram.

Kim [75] proposed a method to generate test sequence from UML activity diagrams that minimizes the number of test steps generated while deriving all practically useful tests. Their method first builds an input/output (I/O) explicit Activity Diagram from an ordinary UML activity diagram and then transforms it to a

directed graph, from which test steps for the initial activity diagram are derived. Their procedure for generating test sequences can be found as follows:

1. Derive a system of activity diagram from given specifications.
2. Derive IOAD Diagram Model Activity Diagram can be presented via specification writers and implementers).
 - a) Delete data objects and use them as input data
 - b) Delete implicit operations (e.g. read action and write action)
 - c) Leave send signal and accept event actions
3. Based on two principles, construct a graph from IOAD. They focus on the interrelation of subsystems from a stable state of a system to a stable state.
4. Traverse nodes based on all-paths test coverage criterion.
5. Generate test sequences.

Linzhang [184] proposed an approach to generate test sequences directly from UML activity diagram using gray-box method, where the design is reused to avoid the cost of test model creation. In their approach, test scenarios are directly derived from the activity diagram modeling an operation. Therefore, all the information such as test sequences or test data is extracted from each test scenario. At last, the possible values of all the input/output parameters could be generated by applying category-partition method, and test suite could be systematically generated to find the inconsistency between the implementation and the design. Gray-box testing¹³ method, which was

¹³

Kaner defines gray box testing as involving inputs and outputs, but test design is educated by information about the code or the program operation of a kind that would normally be out of view of the tester. Gray box testing can be seen as the blending of structural and functional testing methods throughout the entire testing procedure. Gray-box testing examines the activity of back-end components during test case execution. There are two types of problems that can be encountered during gray-box testing. The first is when a component encounters a failure of some kind, causing the operation to be aborted. For example, an edit check to allow dollars does not accept dollar amounts, i.e. "AAA". The second is when the test executes in full, but the content of the results is incorrect. Example: calculations - produces a number but it is incorrect.

proposed in [141] in the designer's viewpoint, generates test sequences based on high level design models which represent the expected structure and behavior of the software under test (SUT)¹⁴. The design specifications are the intermediate artifact between requirement specification and final code. They preserved the essential information from the requirement, and are the basis of the code implementation. Gray box method combines the white box method and the black box method. It extends the logical coverage criteria of white box method and finds all the possible paths from the design model which describes the expected behavior of an operation. Then it generates test sequences which can satisfy the path conditions by black box method. It can find problems which used to be ignored by both black and white method. Gray-box method could systematically generate test sequences directly from the activity diagrams which can be used to test the system at code level. Firstly, it parses the activity diagram and derives the set of test scenarios to satisfy the basic path coverage criteria. Then, each test scenario is processed. The input and output parameters are extracted from the action sequence. The constraint conditions are extracted from the guard conditions in each transition of the test scenario sequence. The object method sequence which represents the internal behavior of the software in runtime is extracted from activity states and corresponding objects. At last they use category partition method [174] to generate proper combination of values of input and output parameters to satisfy the condition constraints. So the input sequence, expected object method call sequence and expected output. And they could also generate all test sequences to form the test suite for the activity diagrams.

¹⁴ SUT refers to software that is being tested for correct operation. The term is used mostly in software testing. A special case of a software is an application which, when tested, is called an application under test. The term SUT means also a stage of maturity of the software; because a software test is the successor of integration test in the testing cycle.

Farooq [176] presented a novel control-flow based test sequence generation technique using UML 2.0 Activity Diagram, which is a behavioral type of UML diagram. Like other model-based techniques, this technique can be used in the earlier phases of the development process owing to the availability of the design models of the system. The Activity Diagram model is seamlessly converted into a Colored Petri Net. They proposed a technique that enables the automatic generation of test sequences according to a given coverage criteria from the execution of the Colored Petri Nets model.

The above current techniques can be summarized as follows:

Table 2-6 Activity Diagram-Based Test Sequence Generation Techniques

Author / Reference	Type of Application	Method	Limitation
KKBK07 [75]	Black-box Testing	Graph Approach	Too many manual efforts left in their approach.
LJXJXG04 [184]	Gray-box Testing	Depth First Search Method	Their approach does not utilize UML 2.0 Activity diagram specification. They do not mention any UML Activity diagram specifications.
FLL08 [176]	Black-box Testing	Random-walk algorithm based on probability	1. Their approach is limited to the intermediate level of UML 2.0 Activity diagram. 2. Their approach covers only control flow related aspects of the activity diagram.

2. State Chart Diagram Based Technique

The following describes a test sequence generation technique, which prepare and generate test sequence from UML State Chart diagram.

Frohlich [137] had recently shown how use cases can be systematically transformed into UML state charts considering all relevant information from a use case specification, including pre- and post conditions. The resulting state charts can have transitions with conditions and actions, as well as nested states (sub and stub states). The current paper outlines how test suites with a given coverage level can be

automatically generated from these state charts. They do so by mapping state chart elements to the STRIPS planning language. The application of the state of the art planning tool graph plan yields the different test sequences as solutions to a planning problem. The test sequences and test data can be used for automated or manual software testing on system level. Using state models to derive test sequences has been common practice in the software testing world for some time [106]. One of the model-based testing goals is to automate the test sequence generation from test models as much as possible. Their algorithm generates a set of valid test sequences, where the preconditions of all transitions are established either by previous actions or by properties of the test data. This is made possible by exploiting artifact intelligent (AI) planning techniques, which allow us to systematically search for paths in the state machine, which satisfy all preconditions of the transitions. In particular, they described the test generation problem as a STRIPS planning problem [143] and solve it with the graph plan tool [1]. The scope of their method is the generation of test sequences supplemented by constraints on the test data, as far as these can be derived from the information present in the state machine.

Samuel [131] proposed the automatic test sequence generation from state machine diagrams. In their approach there are three main steps in test sequence generation. The first step is to select a predicate. In this step, they select a predicate on a transition from a UML state machine diagram. The next step is to transform the selected predicate to a predicate function. In the third step, they generated test sequence corresponding to the transformed predicate function. The generated test sequences are stored for use with an automatic tester. Once the test sequences corresponding to a particular predicate are determined, they repeated these steps by

selecting the next predicate on the state machine diagram. The process is repeated until all predicates on the state machine diagram have been considered.

Wang [38] proposed an axiomatic test sequence generation method based on the extended finite state machine (EFSM) model [61], which can be easily translated from or to the normal specification form of Estelle. A program verification technique, called axiomatic semantics [133], is applied to the conformance testing area. When a protocol specification is verified, observable event sequences are recorded as candidate test sequences. By traversing a carefully chosen path in the EFSM, one can observe the effect produced by the path and confirm the correctness of the transitions in the path. No data flow graph is needed, and the test sequences are generated mechanically from the specification.

Gnesi [170] proposed a formal conformance testing relation and a test sequence generation algorithm for input enabled labeled transition systems over i/o-pairs (IOLTSs). IOLTSs are LTSs where each state has (at least) one outgoing transition for each element of the input alphabet of the transition system. Intuitively, such transition systems cannot refuse any of the specified input events, in the sense that they cannot deadlock when such events are offered to them by the external environment. Whenever a machine, in a given state, does not react on a given input, its modeling IOLTS has a specific loop-transition from the corresponding state to itself, labeled by that input and a special “stuttering” output-label. IOLTSs have been used as semantic model for a behavioral subset of UMLSCs [152].

Sokenou [167] presented an approach that combines UML components for class and integration testing of object-oriented programs. The main information is extracted from sequence diagrams, which is complemented by the use of state diagrams. State diagrams have two functions: initialization of participating objects in

a scenario and –in combination with object constraint language (OCL) constraints– serving as a test oracle (not shown in their work). Beyond the presented technique, they have developed the integration of the derived test oracles into the program under test using aspect-oriented programming techniques.

The above current techniques can be summarized as follows:

Table 2-7 State Diagram-Based Test Sequence Generation Techniques

Author / Reference	Type of Application	Limitation
FL00 [137]	Black Box	Limit to weak coverage criteria.
SMB08 [131]	Black Box	Evolutionary algorithms like genetic algorithms can provide globally optimal solutions but are likely to be computationally intensive.
WL92 [38]	Black Box	There is an exhaustive search for suitable paths.
GLM04 [170]	Black Box	Their approach does not cover UML 2.0 Statechart diagram specification.
S08 [167]	Black Box	Lack of the interaction of UML diagrams.

3. Sequence Diagram Based Technique

The following describes a test sequence generation technique, which prepare and generate test sequence from UML Sequence diagram.

Sarma [119] presented a novel approach of generating test sequences from UML design diagrams. They considered use case and sequence diagram in their test sequence generation scheme. Their approach consists of transforming a UML use case diagram into a graph called use case diagram graph (UDG) and sequence diagram into a graph called the sequence diagram graph (SDG) and then integrating UDG and SDG to form the System Testing Graph (STG). The STG is then traversed to generate test sequences. The test sequences thus generated are suitable for system testing and to detect operational, use case dependency, interaction and scenario faults.

Samuel [138] presented an approach to generate test sequences from UML 2.0 sequence diagrams. Sequence diagrams are one of the most widely used UML models in the software industry. Although sequence diagrams are used for modeling the dynamic aspects of the system, they can also be used for model based testing. Existing work does not encompass certain important features of UML 2.0 sequence diagrams. Their work considers many of the novel features of UML 2.0 sequence diagrams like alt, loop opt and break to generate test sequences. These are important features as far as testing are concerned.

The above current techniques can be summarized as follows:

Table 2-8 Sequence Diagram-Based Test Sequence Generation Techniques

Author / Reference	Type of Application	Limitation
SM07 [119]	Black Box	1. Lack of integration of UML diagrams. 2. Their approach does not cover UML 2.0 Sequence diagram specification.
SJ08 [138]	Black Box	Their approach does not support fully UML 2.0 Sequence diagram specification.

2.5 Test Case Generation Process

According to the literature review, the following shows the test case generation process:

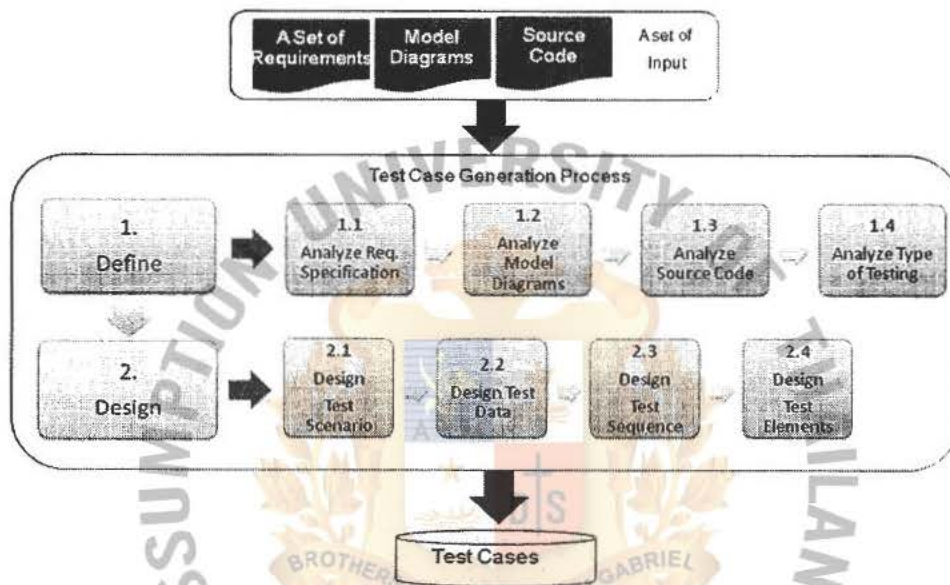


Figure 2-13 Test Case Generation Process

Figure 2-13 presents that there are two processes in the test case generation technique, which break down briefly as follows:

1. **Define.** This is a first process that allows software testing engineers to gather, analyze and define all pre-requisite and required information, such as requirements, constraints and type of testing. There are four sub-processes described shortly as follows:

Table 2-9 First Process in “2D-4A-4D” Test Case Generation Process

Sub-Process	Purpose	Description	Input	Output
Analyze Requirements Specification	<ul style="list-style-type: none"> To be able to perform black-box testing activities. To understand requirements or function specification document. To verify and validate between the requirements and system. 	Software test engineers need to walk through and understand all requirements or function in the specification.	Requirement or Function Specification Document.	Understanding of requirements, constraints and an overview of how to test in general.
Analyze Model Diagrams	<ul style="list-style-type: none"> To be able to perform black-box testing activities. To get better understand the design diagrams. To verify that the behavior of system is match to the design. 	Software test engineers have to analyze the detail design diagrams, such as UML Use Case diagram, UML Activity diagram and State Chart diagram.	Detailed design diagrams.	Understanding of information in the diagrams in order to be able to derive tests from them.
Analyze Program / Source Code	<ul style="list-style-type: none"> To be able to perform white-box testing activities. To be able to understand and help software developer to test program / source code. 	Software test engineers have to analyze and walk through program / source code in order to run a white-box testing activities.	Available of program or source code.	Understanding of the testing strategy / approach for which or how many line of code in the program should be tested. In addition, another output should be a control flow graph transformed from source code.
Analyze Type of Testing	<ul style="list-style-type: none"> To be able to identify which type of testing should be executed. To allow to design test 	Software test engineers need to analyze and identify which type of testing should be executed.	Requirement Specification and Diagrams.	Understanding a type of testing in order to prepare a proper testing strategy or plan.

Sub-Process	Purpose	Description	Input	Output
	strategy or plan for each testing type (i.e. functionality, performance and security).			

2. **Design.** This is a second process that aims to design, prepare and generate all elements in a set of tests, such as test data, test sequence and dependencies of each test case. This process contains the following sub-processes:

Table 2-10 Second Process in “2D-4A-4D” Test Case Generation Process

Sub-Process	Purpose	Description	Input	Output
Design Test Scenario	<ul style="list-style-type: none"> To design a high level scenario for testing. To be able to use as a reference to verify the requirements and testing scenario. 	Software test engineers have to design many testing scenarios to cover all requirements or function.	Requirement Specification, Diagrams and Source Code.	A set of test scenarios.
Design Input Data	<ul style="list-style-type: none"> To design a set of input data used during a test execution phase. To design a realistic input data, both of positive and negative data. To design a special case of input data (e.g. special characters or special combination of symbols and characters) 	Software test engineers have to design many sets of input data that are used for testing.	<ul style="list-style-type: none"> Requirement Specification and Source Code. A set of test scenarios 	Many sets of input data.
Design Test	<ul style="list-style-type: none"> To design a sequence of 	Software testing engineers have to	<ul style="list-style-type: none"> Requirement 	Many set of test sequences.

Sub-Process	Purpose	Description	Input	Output
Sequence	testing activities. <ul style="list-style-type: none"> To understand test steps of each test scenario 	design a set of test sequence or steps for each test scenarios	Specification and Detailed Diagrams <ul style="list-style-type: none"> A set of test scenarios 	
Design Other Elements	To complete designing a set of test cases	Software test engineers must complete a set of test cases by adding additional required elements, such as actual results, dependencies, business impact and defect id.	A set of test scenarios.	A complete set of test case.

The above process can help software test engineers to design, prepare and generate all elements in a set of test cases. It can ensure that all elements are well-prepared. In addition, this process contains all required important or critical elements that can be used in the general commercial industry, such as test scenario, test case, test data, test sequence and dependencies of each test case.

2.6 Related Works

This study reveals that there are additional topics related to test generation techniques, which are used in this research [8][54][88][93][144][192]. The following lists those related works.

1. **Prioritize Requirement.** Donald [54] argued that IT software projects cannot avoid the following facts during SDLC process:
 - a. All requirements are not equally important.
 - b. All projects have limited resources such as effort, time and cost.
 - c. Most project schedule is very tight and long

The above facts have proven that prioritizing requirements is a critical important part during SDLC process.

2. **ISO 9126 Standard.** ISO9126 standard is an international standard and well-known for the evaluation of software quality. Kilidar [8] used this standard to evaluate the quality of safety-critical systems where lives are at risk if software fails. This has proven that this standard is one of the most important standards for evaluating the software quality.
3. **Testing Metrics.** In addition to the above standard, many testing metrics have been proposed [88][93][144] to measure the quality of software. This has proven that metrics are one of the most interesting parts in software testing area.

Next sections described in detail the above related works for this study.

2.6.1 Prioritize Requirement

Donald [54] addressed the purpose of requirement prioritization as follows:

1. Determine the relative necessity of the requirements. Whereas all requirements are mandatory, some are more critical than others. For example, failure to implement certain requirements may have grave business ramifications that would make the system a failure, while others although contractually binding would have far less serious business consequences if they were not implemented or not implemented correctly.
2. Help programs through negotiation and consensus building to eliminate unnecessary potential “requirements” (i.e., goals, desires, and “nice-to-haves” that do not merit the mandatory nature of true requirements).
3. Schedule the implementation of requirements (i.e., help determine what capabilities are implemented in what increment).

Also, Donald [54] showed the significant benefits of requirement prioritization as below:

1. **Modify schedule.** When using an iterative incremental development cycle, it enables the project manager and customer to modify the project schedule to deal with the project realities of limited resources and fixed deadlines.
2. **Improved user satisfaction.** It improves user satisfaction by increasing the likelihood that the customer's most important requirements are implemented and delivered first.
3. **Lower risk of cancellation.** The project is less likely to be cancelled during SDLC. This is because valuable progress is being demonstrated with each increment. Even if the project must be cancelled before the delivery of the final increment, it is not a total loss because some important functionality has been implemented and delivered.
4. **Address all requirements.** Prioritizing requirements is a good approach to force stakeholders to address all requirements, particularly critical requirement.
5. **Estimate benefits.** Priorities provide management and engineering with a rough estimate of the benefit of the different requirements, which is useful when performing cost/benefit analyses of the requirements to determine where best to expend limited project resources in preparation for requirements negotiation.
6. **Prioritize investments.** The requirements prioritization techniques can help determine how to prioritize the investment of limited project resources. For example, the project can allocate most of its limited

resources for quality assurance and system testing according to the highest priority requirements.

Additionally, these researches [5][19][25][45][46][90][92][96][112][117][118][135][157][187] reveal that there are many requirement prioritization methods such as Binary Search Tree (BST), 100-point method and Analytic Hierarchy Process (AHP). Next paragraphs describe the existing methods as follows:

1. Binary Search Tree

Binary Search Tree is an algorithm that is typically used in a search for information and can easily be scaled to be used in prioritizing many requirements [5].

The basic approach for requirements is as follows:

1. Put all requirements in one pile.
2. Take one requirement and put it as root node.
3. Take another requirement and compare it to the root node.
4. If the requirement is less important than the root node, compare it to the left child node. If the requirement is more important than the root node, compare it to the right child node. If the node does not have any appropriate child nodes, insert the new requirement as the new child node to the right or left, depending on whether the requirement is more or less important.
5. Repeat steps 3-4 until all requirements have been compared and inserted into the BST.
6. For presentation purposes, traverse through the entire BST in order and put the requirements in a list, with the least important requirement at the end of the list and the most important requirement at the start of the list.

The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient. Binary search trees can choose to allow or disallow duplicate values, depending on the implementation. Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multi-sets, and associative arrays.

2. Numeral Assignment Technique

The Numeral Assignment Technique provides a scale for each requirement. Brackett proposed dividing the requirements into three groups: mandatory, desirable, and unessential. Participants assign each requirement a number on a scale of 1 to 5 to indicate its importance. The numbers carry the following meaning:

1. Does not matter (the customer does not need it)
2. Not important (the customer would accept its absence)
3. Rather important (the customer would appreciate it)
4. Very important (the customer does not want to be without it)
5. Mandatory (the customer cannot do without it)

The final ranking is the average of all participants' rankings for each requirement.

3. Planning Game

The planning game is a feature of extreme programming [19] and is used with customers to prioritize features based on stories. This is a variation of the Numeral Assignment Technique, where the customer distributes the requirements into three groups, “those without which the system will not function,” “those that are less essential but provide significant business value,” and “those that would be nice to have.”

4. 100-Point Method

The 100-Point Method [96] is basically a voting scheme of the type that is used in brainstorming exercises. Each stakeholder is given 100 points that he or she can use for voting in favor of the most important requirements. The 100 points can be distributed in any way that the stakeholder desires. For example, if there are four requirements that the stakeholder views as equal priority, he or she can put 25 points on each. If there is one requirement that the stakeholder views as having overarching importance, he or she can put 100 points on that requirement. However, this type of scheme only works for an initial vote. If a second vote is taken, people are likely to redistribute their votes to get their favorites moved up in the priority scheme.

5. Theory-W

Theory-W was initially developed at the University of Southern California in 1989 [25][135]. It is also known as "Win-Win." An important point is that it supports negotiation to solve disagreements about requirements, so that each stakeholder has a "win." It has two principles:

1. Plan the flight and fly the plan.
2. Identify and manage risks.

The first principle seeks to build well-structured plans that meet predefined standards for easy development, classification, and query. "Fly the plan" ensures that the progress follows the original plan. The second principle, "Identify and manage risks," involves risk assessment and risk handling. It is used to guard the stakeholders' "win-win" conditions from infringement. In win-win negotiations, each user should rank the requirements privately before negotiations start. In the individual ranking process, the user considers whether there are requirements that he or she is willing to

give up on, so that individual winning and losing conditions are fully understood.

Theory-W has four steps:

1. Separate the people from the problem.
2. Focus on interests, not positions.
3. Invent options for mutual gain.
4. Insist on using objective criteria.

6. Requirements Triage

Requirements Triage [46] is a multistep process that includes establishing relative priorities for requirements, estimating resources necessary to satisfy each requirement, and selecting a subset of requirements to optimize probability of the product's success in the intended market. This is clearly aimed at developers of software products in the commercial marketplace. Davis's more recent book [45] expands on the synergy between software development and marketing; he recommends that you read it if you are considering this approach. It is a unique approach that is worth reviewing, although it clearly goes beyond traditional requirements prioritization, considering business factors as well.

7. Wiegers' Method

This method relates directly to the value of each requirement to a customer [187]. The priority is calculated by dividing the value of a requirement by the sum of the costs and technical risks associated with its implementation [187]. The value of a requirement is viewed as depending on both the value provided by the client to the customer and the penalty that occurs if the requirement is missing. This means that developers should evaluate the cost of the requirement and its implementation risks, as well as the penalty incurred if the requirement is missing. Attributes are evaluated on a scale of 1 to 9.

8. Requirements Prioritization Framework

The requirements prioritization framework and its associated tool [117][118] includes both elicitation and prioritization activities. This framework is intended to address the following:

1. Elicitation of stakeholders' business goals for the project
2. Rating the stakeholders using stakeholder profile models
3. Allowing the stakeholders to rate the importance of the requirements and the business goals using a fuzzy graphic rating scale
4. Rating the requirements based on objective measure
5. Finding the dependencies between the requirements and clustering requirements so as to prioritize them more effectively
6. Using risk analysis techniques to detect cliques among the stakeholders, deviations among the stakeholders for the subjective ratings, and the association between the stakeholders' inputs and the final ratings

9. Cost-Value Approach

A good and relatively easy to use method for prioritizing software product requirements is the cost-value approach. This approach was created by Joachim Karlsson and Kevin Ryan. The approach was then further developed and commercialized in the company Focal Point (that was acquired by Telelogic in 2005). Their basic idea was to determine for each individual candidate requirement what the cost of implementing the requirement would be and how much value the requirement has. The assessment of values and costs for the requirements was performed using the Analytic Hierarchy Process (AHP). This method was created by Thomas Saaty. Its basic idea is that for all pairs of (candidate) requirements a person assesses a value or a cost comparing the one requirement of a pair with the other. For example, a value of

3 for (Req1, Req2) indicates that requirement 1 is valued three times as high as requirement 2. Trivially, this indicates that (Req2, Req1) has value $\frac{1}{3}$. In the approach of Karlsson and Ryan, five steps for reviewing candidate requirements and determining a priority among them are identified. These are summed up below [90].

1. Requirement engineers carefully review candidate requirements for completeness and to ensure that they are stated in an unambiguous way.
2. Customers and users (or suitable substitutes) apply AHP's pair-wise comparison method to assess the relative value of the candidate requirements.
3. Experienced software engineers use AHP's pair-wise comparison to estimate the relative cost of implementing each candidate requirement.
4. A software engineer uses AHP to calculate each candidate requirement's relative value and implementation cost, and plots these on a cost-value diagram. Value is depicted on the y axis of this diagram and estimated cost on the x-axis.
5. The stakeholders use the cost-value diagram as a conceptual map for analyzing and discussing the candidate requirements. Now software managers prioritize the requirements and decide which will be implemented.

10. Analytic Hierarchy Process (AHP)

The Analytic Hierarchy Process (AHP) is a structured technique for dealing with complex decisions. Rather than prescribing a "correct" decision, the AHP helps the decision makers determine one that suits their needs, wants, and understanding of the problem. Based on mathematics and psychology, it was developed by Thomas L. Saaty in the 1970s and has been extensively studied and refined since then. The AHP

provides a comprehensive and rational framework for structuring a problem, for representing and quantifying its elements, for relating those elements to overall goals, and for evaluating alternative solutions. It is used in a wide variety of decision situations, in fields such as government, business, industry, healthcare, and education.

In other words, AHP is a method for decision making in situations where multiple objectives are present [90][92][157]. This method uses a “pair-wise” comparison matrix to calculate the relative value and costs of individual security requirements to one another. By using AHP, the requirements engineer can confirm the consistency of the result. AHP can prevent subjective judgment errors and increase the likelihood that the results are reliable. AHP is supported by a standalone tool, as well as by a computational aid within the SQUARE tool. There are five steps in the AHP method:

1. Review candidate requirements for completeness.
2. Apply the pair-wise comparison method to assess the relative value of each of the candidate requirements.
3. Apply the pair-wise comparison method to assess the relative cost of the candidate requirements.
4. Calculate each candidate requirement's relative value and implementation cost, and plots each on a cost-value diagram.
5. Use the cost-value diagram as a map for analyzing the candidate requirement.

2.6.2 ISO 9126 Standard

Software quality is fundamental to software product success [147]. Yet quality as a concept is difficult to define, describe and understand [140]. Quality has a strong subjective element. For example, a factor one person identifies as indicating good quality (e.g. interface simplicity and elegance), another person may regard as an

indicator of poor quality (e.g. lack of help for novice users). Examination of quality definitions, meanings and views in [147] describes quality as hard to define and measure but easy to recognize. However, quality experts including some with a software background have proposed models e.g. [81][116][162] not to measure quality itself but to measure surrogate attributes such that when combined can provide some notion of the quality of the product.

Many definitions have been introduced to define quality [67][74][80][81][104][116][140][147][162]. The International Standards Organization (ISO) defines quality as: *"the totality of features and characteristics of a product or service that bear on its ability to satisfy specified or implied needs"* [107]. The IEEE defines quality as *"the degree to which a system, component, or process meets specified requirements and customer or user needs or expectations"* [80]. Essentially, both definitions are focused on satisfying the customer's need for the software product. Fifteen different quality definitions are defined and views of quality models. This shows that there is no one encompassing definition or view of quality. The quality view taken in any given situation depends upon the context, the meaning assigned to the quality attributes and the relationships between those attributes within that context.

ISO and the International Electrical technical Commission (IEC) have developed the ISO/IEC 9126 Standards for Software Engineering – Product Quality [53][82][111][122] to provide a comprehensive specification and evaluation model for the quality of software products [67].

Part 1 of ISO/IEC 9126 contains a two-part quality model: one part of the quality model is applicable for modeling the internal and external quality of a software product, whereas the other part is intended to model the quality in use of a

software product. These different quality models are needed to be able to model the quality of a software product at different stages of the software lifecycle. Typically, *internal quality* is obtained by reviews of specification documents, checking models, or by static analysis of source code. *External quality* refers to properties of software interacting with its environment. In contrast, *quality in use* refers to the quality perceived by an end user who executes a software product in a specific context. These product qualities at the different stages of development are not completely independent, but influence each other. Thus, internal metrics may be used to predict the quality of the final product – also in early development stages.

For modeling internal quality and external quality, ISO/IEC 9126 defines the same model. This generic quality model can then be instantiated as a model for internal quality or for external quality by using different sets of metrics. The model itself is based on the six characteristics *functionality*, *reliability*, *usability*, *efficiency*, *maintainability*, and *portability*, as follows [172]:

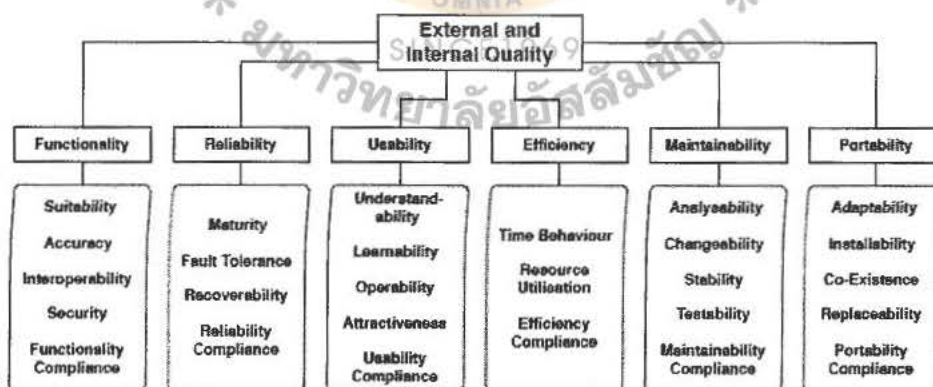


Figure 2-14 ISO/IEC 9126-1 Model for Internal and External Quality

The above characteristics in many papers [16][20][71][128][129][148] can be defined as follows:

Table 2-11 ISO/IEC 9126-1 Characteristics

Characteristic	Sub-Characteristic	Definition
Functionality		The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions [16][20][71][128][129][148].
	Suitability	The capability of the software product to provide an appropriate set of functions for specified tasks and user objectives.
	Accuracy	The capability of the software product to provide the right or agreed results or effects with the needed degree of precision.
	Interoperability	The capability of the software product to interact with one or more specified systems.
	Security	The capability of the software product to protect information and data so that unauthorized persons or systems cannot read or modify them and authorized persons or systems are not denied access to them.
	Functionality Compliance	The capability of the software product to adhere to standards, conventions or regulations in laws and similar prescriptions relating to functionality.
Reliability		The capability of the software product to maintain a specified level of performance when used under specified conditions [16][20][71][128][129][148].
	Maturity	The capability of the software product to avoid failure as a result of faults in the software.
	Fault Tolerance	The capability of the software product to maintain a specified level of performance in cases of software faults or of infringement of its specified interface.
	Recoverability	The capability of the software product to re-establish a specified level of performance and recover the data directly affected in the case of a failure.
	Reliability Compliance	The capability of the software product to adhere to standards, conventions or regulations relating to reliability.
Usability		The capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions [16][20][71][128][129][148].

Characteristic	Sub-Characteristic	Definition
	Understandability	The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use.
	Learnability	The capability of the software product to enable the user to learn its application.
	Operability	The capability of the software product to enable the user to operate and control it.
	Attractiveness	The capability of the software product to be attractive to the user.
	Usability Compliance	The capability of the software product to adhere to standards, conventions, style guides or regulations relating to usability.
Efficiency	The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions [16][20][71][128][129][148].	
	Time Behavior	The capability of the software product to provide appropriate response and processing times and throughput rates when performing its function, under stated conditions.
	Resource Utilization	The capability of the software product to use appropriate amounts and types of resources when the software performs its function under stated conditions.
	Efficiency Compliance	The capability of the software product to adhere to standards or conventions relating to efficiency.
Maintainability	The capability of the software product to be modified. Modifications may include corrections, improvements, or adaptation of the software to changes in environment, and in requirements and functional specifications [16][20][71][128][129][148].	
	Analyzability	The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified.
	Changeability	The capability of the software product to enable a specified modification to be implemented.

Characteristic	Sub-Characteristic	Definition
	Stability	The capability of the software product to avoid unexpected effects from modifications of the software.
	Testability	The capability of the software product to enable modified software to be validated.
	Maintainability Compliance	The capability of the software product to adhere to standards or conventions relating to maintainability.
Portability	The capability of the software product to be transferred from one environment to another [16][20][71][128][129][148].	
	Adaptability	The capability of the software product to be adapted for different specified environments without applying actions or means other than those provided for this purpose for the software considered.
	Installability	The capability of the software product to be installed in a specified environment.
	Co-existence	The capability of the software product to co-exist with other independent software in a common environment sharing common resources.
	Replaceability	The capability of the software product to be used in place of another specified software product for the same purpose in the same environment.
	Portability Compliance	The capability of the software product to adhere to standards or conventions relating to portability.

2.6.3 Testing Metrics

The literature reviews show that testing metrics are an important indicator of the effectiveness of a software testing process. The examples of current testing metrics can be found as follows [93]:

Table 2-12 Testing Metrics

Test Metric	Definition	Purpose	How to Calculate
-------------	------------	---------	------------------

Test Metric	Definition	Purpose	How to Calculate
Time to find a defect	The effort required to find a defect.	Shows how fast the defects are being found. This metric indicates the correlation between the test effort and the number of defects found.	Divide the cumulative hours spent on test execution and logging defects by the number of defects entered during the same period.
Test coverage	Defined as the extent to which testing covers the product's complete functionality.	This metric is an indication of the completeness of the testing. It does not indicate anything about the effectiveness of the testing. This can be used as a criterion to stop testing.	Coverage could be with respect to requirements (also known as requirement coverage), functional topic list, business flows, use cases, etc. It can be calculated based on the number of items that were covered vs. the total number of items.
Test case effectiveness	The extent to which test cases are able to find defects.	This metric provides an indication of the effectiveness of the test cases and the stability of the software.	Ratio of the number of test cases that resulted in logging remarks vs. the total number of test cases.
Number of defects	The total number of remarks found in a given time period/phase/test type that resulted in software or documentation modifications.	A more meaningful way of assessing the stability and reliability of the software than number of remarks. Duplicate remarks have been eliminated; rejected remarks have been done.	Only remarks that resulted in modifying the software or the documentation are counted.

CHAPTER 3

RESEARCH PROBLEMS

This chapter discusses outstanding research problems relative to test generation techniques. Also, this chapter is concluded with a discussion on problems addressed in this dissertation.

3.1 Research Issues

Every test case generation technique has its own weak and strong points, as addressed in the literature survey. In regard to using the test case generation techniques, there are a significant number of issues that need to be addressed (e.g. design test scenario, design test case format, generate test cases from model diagrams, etc). In general, referring to the literature review, the following lists are the main issues in test case generation, test data generation and test sequence generation area:

1. Problem of detecting bug delay [3][53][69][80][81][119][136][158][159]:

The process of generating tests from the requirement specifications will often help the test engineer discover problems with the specifications themselves; if this step is done early, the problems can be eliminated early, saving time and resources. Launch the testing process early in the development lifecycle and also help with testing methodology.

2. Inefficient requirement specification for functionality testing

[38][39][53][73][74][82][85][86][105][113][116][122][131][137][155][159]

[160][170][172][182][190][191]: Inefficient requirement specification can

lead to the problem of verify and validate the proposed system design against the functional requirements and to provide automatically measurable test case to support design alternative analysis.

3. **Inefficient test data** [15][49][81][116][160][175]: Many researchers are interested in identifying a collection of test data, such as input and output data, from state-based specification.
4. **Lack of ability to identify critical requirements** [14][149]: The current existing technique is lack of the capability to critical requirements, because those requirements are not explicitly discussed in the specification document. For example, Nilsson [149] proposed the technique to generate test cases for real-time system.
5. **Lack of a standard and formal specification** [178]: Incorrect interpretations of complex software from non-formal specification can result in incorrect implementations leading to testing them for conformance to its specification standard.
6. **Inefficient automated test case generation techniques** [4][7][11][13][27][38][43][51][72][74][76][83][84][94][97][98][116][119][131][137][138][140][147][156][161][162][165][167][170][176][181][182][184][190][191][198]: Improving the ability to automatically generate test cases, for example, enhancing the capability of automation tools, can be reduces time and cost for testing. Most of the time, testers perform the manual testing in the development life cycle.
7. **Lack of re-uses systematically of test case** [198]: Reusing test cases can reduce time and cost in software testing phase. Software test engineers do not need to generate a new set of test case every time they perform testing.
8. **Inefficient requirement specification for GUI testing** [12][13][83]: Improve the formal requirement specification in order to increase the capability of verify and validate user interactions against the system and to provide

automatically measurable test case to support design alternative analysis.

Systematically test cases can reduce cost and time of development.

- 9. Ignore unspecified input data [182]:** Current test case generation techniques ignore generating unspecified inputs or attacks in order to test the robustness of web application.

- 10. Inefficient requirement specification for security testing [191]:** Improve the formal requirement specification in order to increase the capability of verify and validate the proposed system design against the security requirements and to provide automatically measurable test case to support design alternative analysis. Systematically test cases can reduce cost and time of development.

- 11. Inefficient requirement specification for performance testing [104]:** Improve the formal requirement specification in order to increase the capability of verify and validate the proposed system design against the performance requirements and to provide automatically measurable test case to support design alternative analysis. Systematically test cases can reduce cost and time of development.

- 12. Unable to select suitable test cases due to a large number of test cases**

[4][7][11][13][27][38][43][51][72][74][76][83][84][94][97][98][116][119]

[131][137][138][140][147][156][161][162][165][167][170][176][181][182]

[184][190][191][196][198]: Software testing is one of the most forgettable.

Typically, software testing engineers have a few amounts of time, effort and cost to plan, design test case, run test cases and evaluate test cases respectively. Existing techniques are not effective for complex application with limited resources (e.g. time, effort, and cost) both of traditional and web application. The example of complex web application is the application with

dynamic behaviors, heterogeneous representations, novel control flow, and data flow mechanisms.

13. Lack of supporting multi-user interactions for web application [161]: An existing approach to using field data in testing web applications is user-session-based testing. Previous user-session-based testing approaches ignore state dependences from multi-user interactions. Current techniques are not effective, because they are lack of multi-user interaction issue.

14. Inefficient requirement specification for reliability testing [83][170]: Reliability is defined to be the probability of failure-free operations for web-based applications. There are many root-causes of problems in testing web application such as user interaction, information delivery between client and server (e.g. host, network or browser failures) and the correctness of functionality of web application. Communication protocols thus play a major role in today's distributed computing environments. To guarantee the reliability of a network, it is essential to ensure that protocol implementations are consistent with their specifications in various hardware and software environments.

15. Large number of test cases [51][75][140]: Most test case generation techniques are aiming to generate test cases in order to cover scenario as much as possible. Sometimes, they generate too big size of test cases and it is impossible to execute those cases with limited time and resources.

16. Lack of important information from diagrams [138][167]: It is difficult to generate automatically test cases from model diagrams (e.g. use case diagram, activity diagram and state diagram) if those diagrams are not completed.

17. Ignore concurrent program factor [181]: Test sequence generation for sequential program can be applied to concurrent programs, but they may not be efficient.

18. Ignore a modification of programs [27]: There are a lot of changes and modification during a regression testing phase. Most of existing test data generation techniques ignores the modification occurred during regression testing phase.

The above remaining problems can be classified aligned with the test case generation process mentioned in the Chapter 2 as follows:

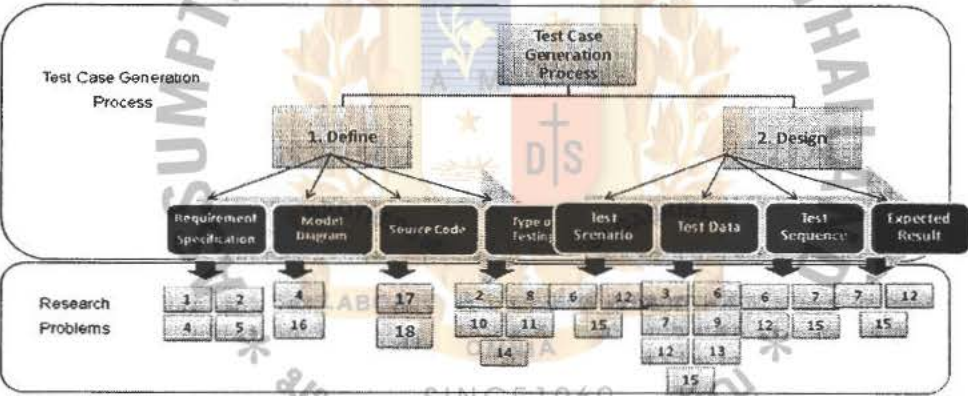


Figure 3-1 A Classification of Remaining Problems

Figure 3-1 represents a classification of remaining problems based on test case generation process. There are outstanding problems aligned with each sub-process. This dissertation does not aim to resolve all the above problems. It is nearly impossible to tackle with all problems within this study.

However, the study and what have been discovered present the following researchers who have investigated and proposed methods to resolve each problem.

Table 3-1 Test Case Generation Techniques and Issues

Authors	Issues																	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1. Test Cases Generation Techniques																		
ASA05 [137]	X																	
CR99 [156]		X																
HL00 [113]		X																
OXL99 [3]	X																	
K97 [86]		X																
T01 [176]			X															
RH01 [55]		X	X															
NOM06 [150]				X														
TWCPX05 [183]		X				X			X									
JLQ03 [192]		X				X				X								
JL [191]		X				X												
S05 [14]				X														
SAVFFML06 [179]					X													
H01 [69]	X																	
EW01 [76]						X												
RG00 [85]		X																
JSW07 [4]						X												
CCDHJM00 [7]						X												
ROM08 [66]																		
KLH00 [43]						X												
SKF06 [104]											X							
ND03 [105]		X																
AOA04 [11]						X												
CMQ07 [166]												X						
MCLQ07 [72]												X						
BG03 [158]	X					X												
YHWC99 [84]						X												
MQS08 [73]		X																
2. Test Data Generation Techniques																		
AO99 [15]			X															
OLAA03 [81]	X		X															
BBH02 [116]		X	X			X												
KA98 [27]						X												X
K90 [94]						X						X						
HO99 [80]	X																	
BFSBT06 [53]	X																	
LKHH00 [39]		X																
3. Test Sequence Generation Techniques																		
RH01 [159]	X	X																
KKBK07 [75]															X			
LJXJXG04 [184]						X												
FLL08 [176]						X												
FL00 [137]		X				X												
SMB08 [131]		X				X												
WL92 [38]		X				X												
GLM04 [170]		X				X							X					
S08 [167]																X		
SM07 [119]	X					X												
SJ08 [138]						X										X		

3.2 Problem Statement

Due to the fact that there are many challenges in software testing area, especially during test case generation step [123], the most challenges and interesting issues are: (a) large number of test cases and (b) lack of ability to cover critical requirements. These two problems can be presented based on Figure 3-1 as follows:

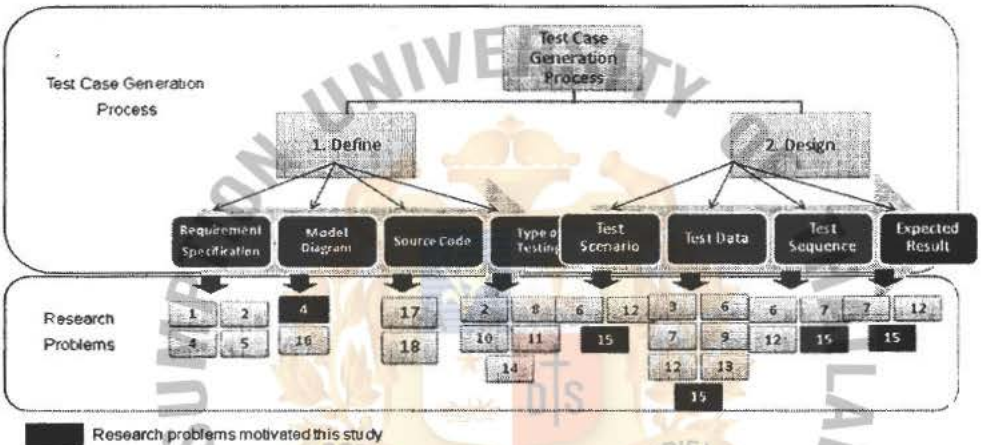


Figure 3-2 Research Problems Motivated This Dissertation

Figure 3-2 shows that there are two major research problems motivated this dissertation, which are: (a) large number of test cases (also referred as #15) and (b) lack of ability to cover critical requirements (also referred as #4). The literature shows that there are many test case generation methods proposed to resolve the above two problems. It discovers that there are two types of methods: (a) test case generation for black-box testing and (b) test case generation for white-box testing. This dissertation concentrates on the methods for black-box testing only. This is because:

1. The study shows that testing activities must be started at the beginning of software development life cycle.
2. (b) One of the testing goals is to verify and validate software with customer requirements.

3. The research discovers that cost to resolve defects in the later phase is by far greater than resolving at the earlier phase.

Furthermore, the study presents that there are two sources for test case generation techniques for black-box testing: (a) derive test cases from requirement documents and (b) derive test cases from model diagrams. This dissertation concentrates on an approach to generate test cases from UML use case diagram (which is part of the UML model diagram). This is due to the fact that:

1. UML use case diagram is used to describe a behavior or functions of systems.
2. One of important testing goals is to verify and validate function system with the customer requirements.

The following discusses outstanding research problems remaining from test case generation methods based on the UML use case diagram [57][69][85][105][197]:

The following list three problems motivated this study:

1. Lack of requirement prioritization before test case generation
2. Unable to identify which test cases can be removed during a test case generation process
3. Large number of test cases due to large number of alternative paths in each use case [197].

The following describes these problems in details.

The first problem reveals that existing test case generation methods ignore a requirement prioritization before generating test cases. Those methods explicitly assume that there are unlimited resources and cost to execute tests. Therefore, this thesis proposes the requirement prioritization activities during a test case generation process.

The second problem is to unable to identify how many and which test cases should be removed in order to generate and minimize a number of test cases. The following describes a testing matrix table between test cases and use cases. This matrix is developed to cross check whether test cases are generated and tested against all use cases.

Testing Matrix Table

	Use Case 1	Use Case 2	Use Case 3	Use Case 4
T1	X	X	X	X
T2	X	X		
T3		X	X	
T4				X

Figure 3-3 Matrix Table between Test Case and Use Case

Figure 3-3 shows that there are four test cases generated to be tested against four use cases. *Tn* represent as a test case. It also shows that *T1* cover *T2*, *T3* and *T4*. The problem here is that there are many choices to minimize a number of test cases. For example, the first choice is to remove only *T1* or the second choice is to remove *T2*, *T3* and *T4*. This is ambiguous which test cases should be removed.

The last problem is that there are a large number of test cases due to a large number of alternative paths or events in each use case. The study [197] shows that each use case can have overwhelm alternative paths. Therefore, if we can reduce those paths, a number of test cases should be reduced as well.

The following table shows the relationship between the problem statements in this section and the objectives of the thesis.

Table 3-2 Problem Statements and Objectives

Objective of the Thesis	Problem #1	Problem #2	Problem #3
Objective #1 - Prioritize requirements based on user satisfaction prior to generate test cases in order to improve the ability to generate and select the most suitable test cases.	X		
Objective #2 - Propose an alternative path point formula to be able to systematically determine which test cases could be removed during test case generation activities.		X	
Objective #3 - Enhance ability to minimize a number of test cases by adding a complexity factor.			X



CHAPTER 4

PROPOSED TECHNIQUES

This chapter introduces test case generation methods to resolve the research problems mentioned in the previous chapter. It is structured into five sections: (a) overview (b) assumption (c) requirement prioritization based on user satisfaction (d) test case generation technique and (e) limitations of the proposed method.

The following describes each section in details.

4.1 Overview

The following describes an overview of our proposed method to resolve outstanding research problem mentioned in the previous chapter.



Figure 4-1 Overview of Proposed Test Case Generation Technique

Figure 4-1 describes that there are many factors considered normally while generating test cases from UML use case diagram. The following shows definitions of each factor.

Table 4-1 Definitions of Factors Normally Considered in Literatures

Factor	Name	Description
A	Return on Investment	Return on investment (also known as ROI) is the ratio of money gained or lost (whether realized or unrealized) on an investment relative to the amount of money invested [201][202]
B	User Satisfaction	User / Customer satisfaction, a business term, is a measure of how products and services supplied by a company meet or surpass customer expectation. It is seen as a key performance indicator within business and is part of the four of a Balanced Scorecard. In a competitive marketplace where businesses compete for customers, customer satisfaction is seen as a key differentiator and increasingly has become a key element of business strategy [115]
C	Time-to-Market	Time-to-market (also known as TTM) is the length of time it takes from a product being conceived until its being available for sale. TTM is important in industries where products are outmoded quickly [163].
D	Test Case Complexity	A total number of test steps presented in each test case that are required to be executed [163].
E	A Number of Test Cases	A number of test cases can be represents as a total number of test cases in each test suite [69][85][105].
F	Requirement Complexity	Holly defines a meaning of complexity as follows: "(a) consisting of many different and connected parts and (b) not easy to analyze; complicated or intricate" [70].
G	Risk	A risk is a measurement that contains two metrics: (a) level of damage and (b) probability of failure [163].
H	Quality	Quality measures how well software is designed (quality of design), and how well the software conforms to that design (quality of conformance), although there are several different definitions [8].

This dissertation concentrates on the following factors in the proposed techniques:

1. **A Number of Test Cases** – This thesis proposes an effective test cases generation method to minimize a number of test cases. This is for the reason that large number of test cases consumes a great deal of effort, time and cost.
2. **Test Case Complexity** – This dissertation proposes to consider a complexity of test cases to remove test cases during a test case generation process. This is

due to the fact that the proposed method aims to reduce to a minimum the number of test cases.

3. **Requirement Complexity** – This study includes a requirement complexity to the proposed method in order to effectively prioritize requirements before generating test cases. Also, the study presents a correlation between requirement complexity and ROI. The ratio between these factors is applicable for requirement prioritization.
4. **User Satisfaction** – This study concentrates on generating test cases based on user satisfaction. It has been proven that user satisfaction is the most important factor for long-term project success and large profit. Currently, none of existing test case generation methods considers this factor during generation activities. This dissertation applies this factor to classify requirements during a test case generation process. Customers expect high quality systems, therefore, test cases should be generated to cover customer requirements, as it will have a significant impact on user satisfaction.
5. **Return on Investment (ROI)** – This study uses the ROI value for prioritizing requirements. ROI is the most important factor from business' perspective. It is a critical factor for project success and profitability.

The following shows reasons why the above five factors are selected:

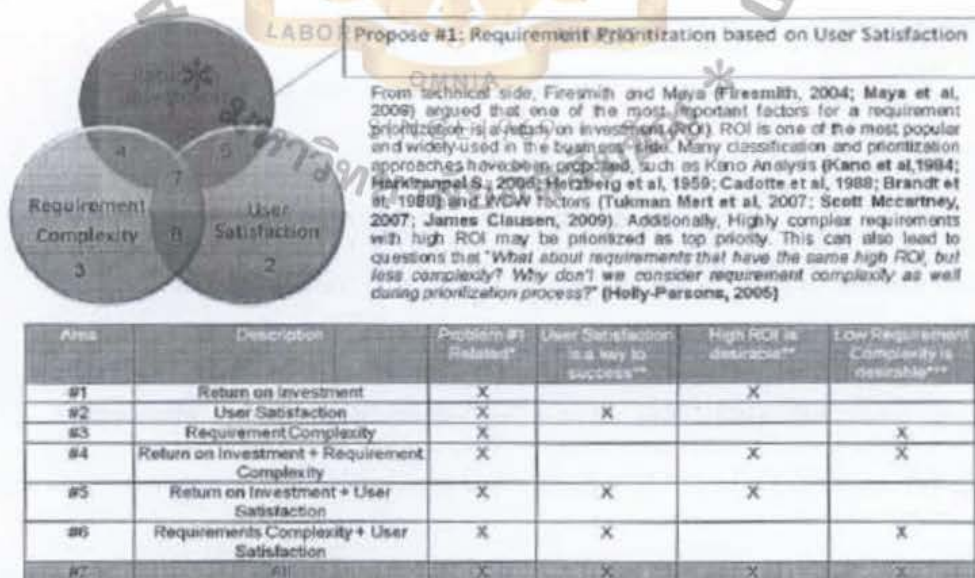
Table 4-2 Reasons Why Factors Are Selected in Dissertation

Factor	Name	Description	Problem 1 Related	Problem 2 Related
A	Return on Investment	Return on investment (also known as ROI) is the ratio of money gained or lost (whether realized or unrealized) on an investment relative to the amount of money invested [201][202]	X	
B	User Satisfaction	User / Customer satisfaction, a business term, is a measure of how products and services supplied by a company meet or surpass customer expectation. It is seen as a key performance indicator within business and is part of the four of a Balanced Scorecard. In a competitive marketplace where businesses compete for customers, customer satisfaction is seen as a key differentiator and increasingly has become a key element of business strategy [115]	X	
C	Time-to-Market	Time-to-market (also known as TTM) is the length of time it takes from a product being conceived until its being available for sale. TTM is important in industries where products are outmoded quickly [163].		
D	Test Case Complexity	A total number of test steps presented in each test case that are required to be executed [163].		X
E	A Number of Test Cases	A number of test cases can be represents as a total number of test cases in each test suite [69][85][105].		X
F	Requirement Complexity	Holly defines a meaning of complexity as follows: "(a) consisting of many different and connected parts and (b) not easy to analyze; complicated or intricate" [70].	X	
G	Risk	A risk is a measurement that contains two metrics: (a) level of damage and (b) probability of failure [163].		
H	Quality	Quality measures how well software is designed (quality of design), and how well the software conforms to		

Factor	Name	Description	Problem 1 Related	Problem 2 Related
		that design (quality of conformance), although there are several different definitions [8].		

Table 4-2 shows that there are many factors for the requirement prioritization from both, the business perspective and the software testing perspective this includes ROI, requirement complexity, a number of test cases, risk factors and user satisfaction [124]. The following factors: A, B, D, E and F are selected because: (a) A, B and F are relative to the first outstanding problem that is mentioned in the Chapter 3 and (b) D and E are related to the second outstanding problem mentioned in the Chapter 3.

Furthermore, previous research has proved [123][124] that there is a relationship among ROI, requirement complexity and user satisfaction, as shown below:



* (Firesmann, 2001; Ryser et al, 2000; Hilmour et al, 2003; A. Z. Javed et al, 2007)

** (Kano et al, 1984; Harkimpep S., 2006; Tukman Mert et al, 2007; Scott, 2007; James Clausen, 2009; Andreas Heitsmann, 2008)

*** (Holly, 2005)

Figure 4-2 Relationship for ROI, Req. Complexity and User Satisfaction

Figure 4-2 shows that there is a relationship among ROI, a complexity of requirement and user satisfaction. There are seven areas in the relationship.

1. Return on investment
2. User satisfaction
3. Requirement complexity
4. Return on investment and requirement complexity
5. Return on investment and user satisfaction
6. Requirement complexity and user satisfaction
7. All three factors

The first proposed method is in area 7. This is because the proposed method meets all the following criteria:

1. Problem 1 related – It is relative to the first outstanding problem that is mentioned in the Chapter 3.
2. User satisfaction is a key to success – It is to prioritize requirements based on user satisfaction that is a key to success.
3. High ROI is desirable – It is targeted to reserve high priority requirements with high return on investment.
4. Low requirement complexity is desirable – It aims to reserve low complexity of requirements as priority.

Additionally, previous research has proved [123][124] that there is a relationship between test case complexity and a number of test cases, as shown below:

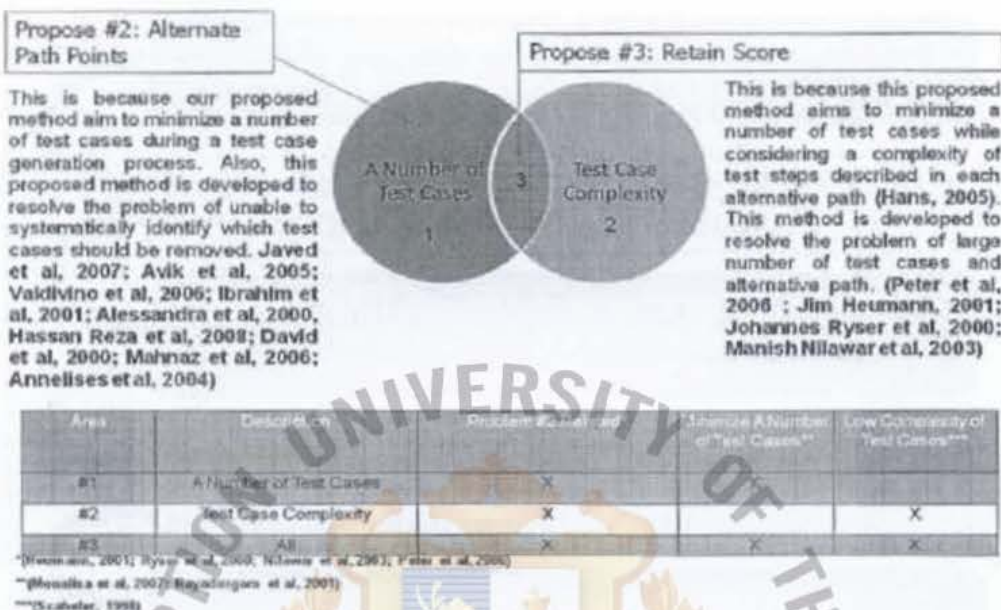


Figure 4-3 Relationship between Number of Test Cases and Complexity

Figure 4-3 shows that there is a relationship between a number of test cases and test case complexity. There are three areas in the relationship, as follows:

1. A number of test cases
2. Test case complexity
3. Both of two factors

The second proposed method is in the area 1. This is due to the fact that it aims to minimize a number of test cases during a test case generation process. Meanwhile, the last proposed method is in the area 3. This is because it included a test case complexity factor in the algorithm in order to reduce a large number of test cases. Both of these methods are relative to the outstanding problems mentioned in the Chapter 3.

The following presents an overview of our proposed techniques associated with resolving the above outstanding issues in previous section.

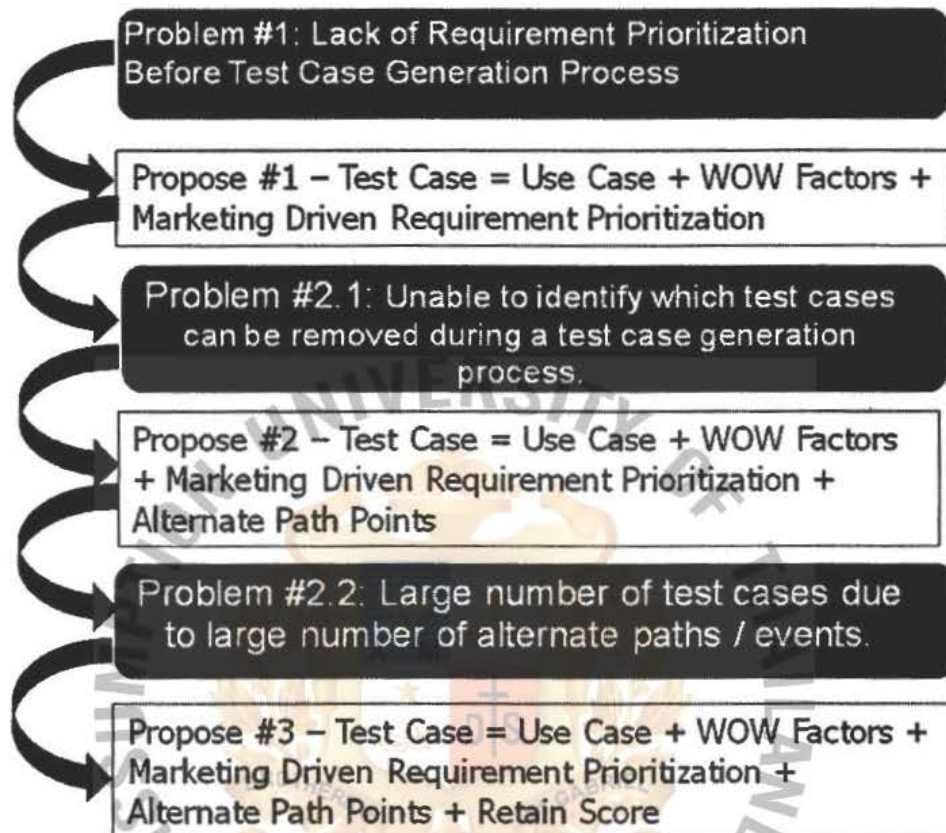


Figure 4-4 Proposed Methods Relative to Research Problems

Figure 4-4 shows that there are three proposes in this thesis. The first proposed method is to add the requirement prioritization process before a test case generation process. The second proposed method is developed to be able to identify how many and which test cases can be removed. The last method aims to remove test cases after the test case generation by using risk factors [10][139][163]. The objective of these methods is to minimize a number of test cases after test case generation process, while preserving critical higher priority requirements.

4.2 Assumptions

The following lists assumptions in this thesis:

1. In this dissertation, we assume that customers are the same people or users who can provide requirements.
2. Testing activity, which is to prioritize the requirement, is required to start in the requirement phase. Software test engineers play a major role as early as possible in SDLC.
3. Requirements are not prioritized before test case generation process.
4. UML use case diagram must be completed and must fully contain all required information (e.g. purpose, basic flow and alternative flow). No methods in this thesis are used to verify and qualify the completeness of diagrams.
5. Prioritizing requirements by test engineers are not required if there are a few requirements. For example, if there are 1-3 requirements to develop and implement software, there is no need to prioritize those requirements.
6. In order to generate test scenarios and test cases from UML use case diagram and respectively, the diagram with required information must be available.
7. Approaches to estimate cost for development and testing activities proposed in this dissertation such as COCOMO and Function Analysis, are out of scope.

4.3 Test Case Generation Process

This section introduces a recommended test case generation process by inserting a requirement prioritization based on user satisfaction. This dissertation proposes to prioritize requirements from business' perspective, prior to generate test cases, in order to maintain and increase user satisfaction [100][114].

Before discussing our proposed process, we would like to discuss existing test case generation process and point out exactly what is different between traditional and proposed process.

The following presents three types of test case generation process.

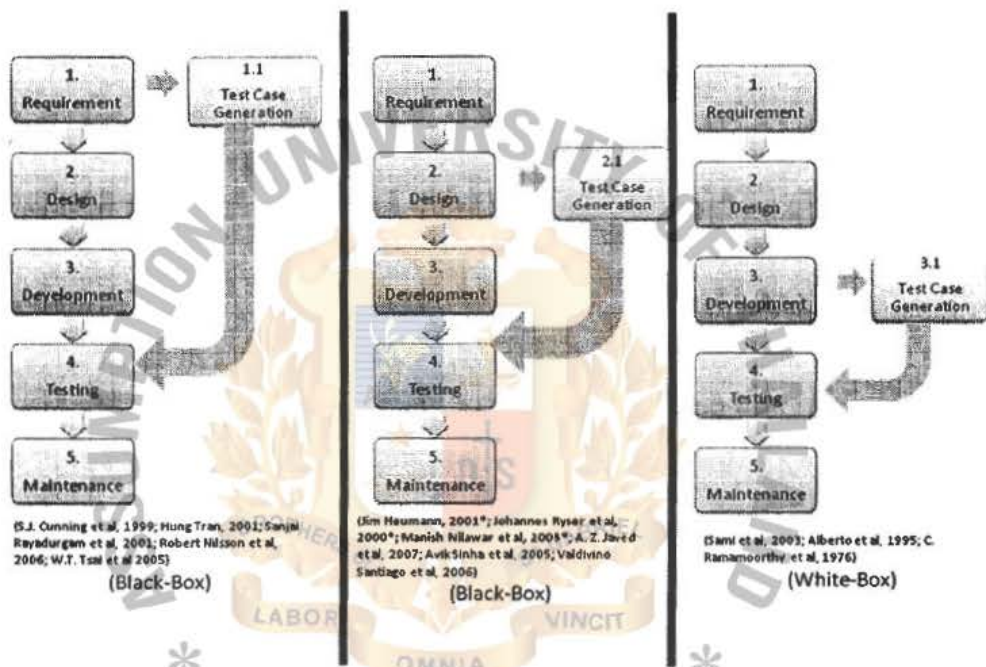
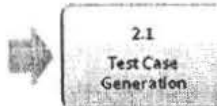
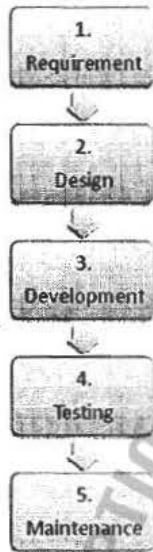


Figure 4-5 Traditional Test Case Generation Process

Figure 4-5 presents three test case generation processes. The first two processes are used for black-box testing. The last process is used for white-box testing. First, a test case generation process is occurred during a requirement phase. Second, the test case generation process is occurred during a design phase. Last, the test case generation process is happened during a development phase. Due to the fact that this dissertation concentrates on black-box testing and test cases derived from diagrams, therefore, the first and last process is ignored.

The following compares the second process and our proposed process.

Existing Test Case Generation Process



Proposed Test Case Generation Process

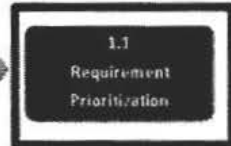


Figure 4-6 Compare Test Case Generation Process

Figure 4-6 compares test case generation process between traditional process that derive test cases from diagrams and our proposed process. The only different is that there is additional process, called "requirement prioritization", in the test case generation process.

The following depicts how the requirement prioritization activities can be aligned with the test case generation process.

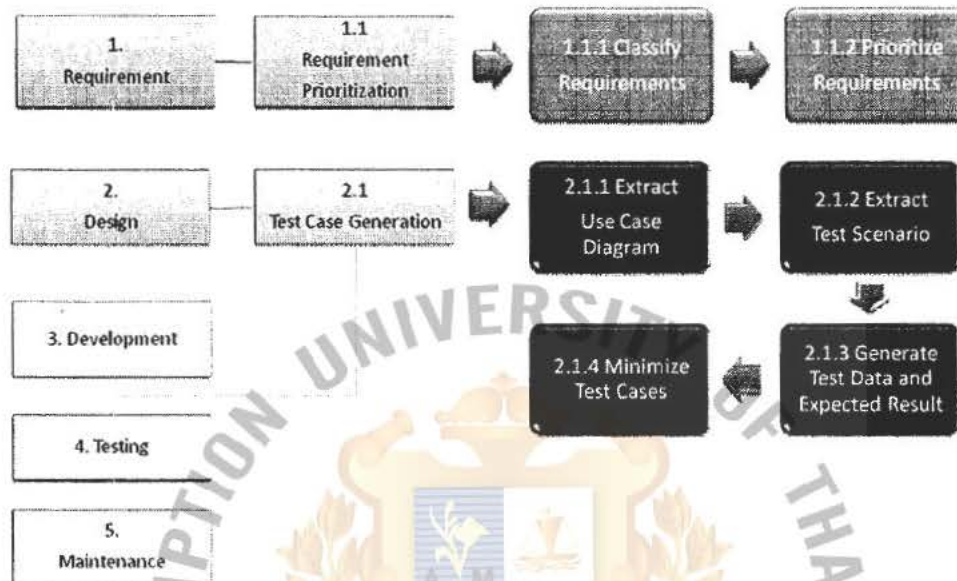


Figure 4-7 Requirement Prioritization based on User Satisfaction

Figure 4-7 shows that there are two test case generation processes: existing process and proposed process. Proposes shown on the right-hand side of Figure 4-7 add an additional requirement prioritization process before generating test cases. Traditional test case generation process does not include a requirement prioritization process. In fact, the requirement prioritization process aims to be able to effectively handle a large number of requirements. The objective of this process is to prioritize and organize requirements in an appropriate way in order to effectively design and prepare test cases [54][112][188]. There are two sub-processes: (a) classify requirements and (b) prioritize requirements.

In Figure 4-7, there are four sub-processes in the test case generation process: (a) extract use case diagram (b) extract test scenario (c) generate test data & expected result and (d) minimize test cases after generating.

Both of requirement prioritization (Refer to 1.1 in Figure 4-7) and test case generation (Refer to 2.1 in Figure 4-7) can be illustrated in details as follows:

4.4 Requirement Prioritization Based On User Satisfaction

Before discussing the procedure of the proposed requirement classification and prioritization technique, this dissertation begins a discussion with reasons why requirement prioritization and user satisfaction are important.

The explanations why requirement prioritization is important can be found as follows:

One of the most important testing goals is to generate a large number of test cases, in fact as many as possible, in order to cover and verify all customer requirements. At the present, requirements become more complex and difficult, particularly in the high competitive markets. Large number of complex requirements can lead to a huge number of test cases. Consequently, it takes longer and the project budget may be overrun due to those test cases.

A topic of many interesting researches has been prioritizing requirements. There are many techniques that have been proposed over a long period of time (as mentioned in Chapter 2). Donald [54] provides primarily benefits of requirement prioritization as follows:

1. **Modify schedule.** When using an iterative incremental development cycle, it enables the project manager and customer to modify the project schedule, to deal with the project realities of limited resources and to fix deadlines.
2. **Improved user satisfaction.** It improves user satisfaction by increasing the likelihood that the customer's most important requirements are implemented and delivered first.
3. **Lower risk of cancellation.** The project is less likely to be cancelled during SDLC. This is because valuable progress is being demonstrated with each increment. Even if the project must be cancelled before the delivery of the

final increment, it is not a total loss as some important functionality has been already implemented and delivered.

4. **Address all requirements.** Prioritizing requirements is a good approach to force stakeholders to address all requirements, particularly critical requirement.
5. **Estimate benefits.** Priorities provide management and engineering with a rough estimate of the benefit of the different requirements, which is useful when performing cost/benefit analyses of the requirements to determine where would be the best to expend limited project resources in preparation for requirements negotiation.
6. **Prioritize investments.** The requirements prioritization techniques can help determine how to prioritize the investment of limited project resources. For example, the project can allocate most of its resources for quality assurance and system testing according to the highest priority requirements.

The following shows a survey result why user satisfaction is important and matter to the business.

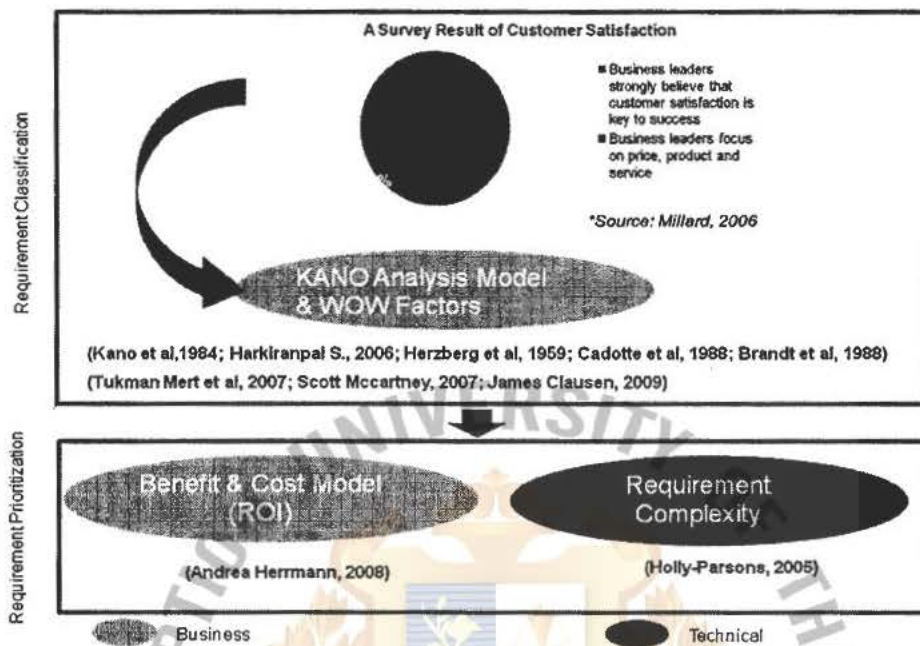


Figure 4-8 Overview of Requirement Prioritization

Figure 4-8 shows that 85 percent of business leader organizations focus on customer satisfaction (also known as “user satisfaction” in this dissertation) [114]. High user satisfaction can give a long-term success with the customer as well as higher profits. Therefore, this dissertation proposes to classify requirements based on business’ perspective [115]. Our proposed method is supported by using WOW factors [100][114].

This dissertation realizes that user satisfaction is the key to success. That is another reason why we propose a test case generation method based on user satisfaction. Our customers are a people who use the system that we develop and test. They expect high quality software that makes their life easier [45][115]. In order to achieve this, software testing can play a major role to ensure that software can meet their expectation and eventually satisfy them. Recall that test case generation activity is one of the most important and widely-researched activities in the software testing process. Finally, this dissertation proposes to classify and prioritize requirements

based on user satisfaction prior to generate test cases. The requirements classification and prioritization are to ensure that all requirements, that are able to highly satisfy customers, are addressed as priority.

These studies [28][32][57][89] show that a marketing perspective concentrates on two factors: customer's needs and customer satisfaction. We apply that perspective to the requirement prioritization and propose to build user satisfaction as shown below:

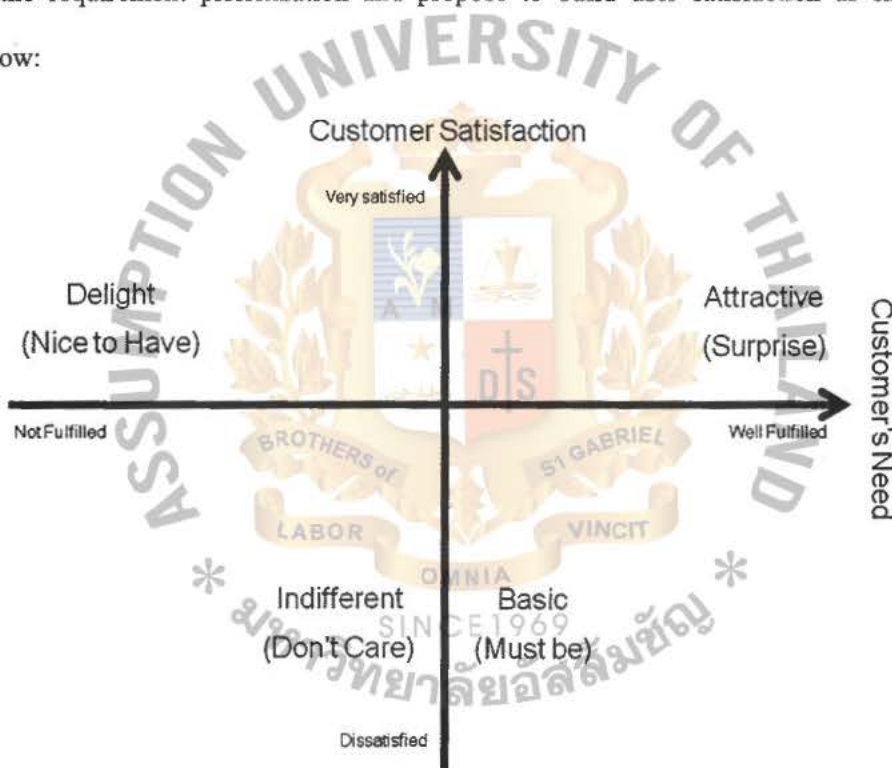


Figure 4-9 Kano Model Analysis

Figure 4-9 represents a KANO model proposed by Noriaki [89] to classify requirements based on customer's need and customer's satisfaction. The horizontal axis presents a customer's need while the vertical axis represents a customer satisfaction [89]. There are four groups of requirements based on those two factors: delight, attractive, indifferent and basic. First, the delight requirement is known as 'nice-to-have' requirement. If this requirement is well fulfilled, it will increase the

user satisfaction. Otherwise, it will not decrease the satisfaction. Second, the attractive requirement is called as 'surprise' or 'know your customer' requirement. This requirement can directly increase the user satisfaction if it is fulfilled. Marketers and sales [28] believe that if we can deliver this kind of requirement, it will impress customers and significantly improve the user satisfaction. Third, the indifferent requirement is the requirement that customer does not concentrate and it will not impress customers at all. In the competitive industry, this requirement may be fulfilled but there is not any impact to the user satisfaction. Last, the basic requirement is a mandatory requirement that customers basically expect. Therefore, if this requirement is well delivered, it will not increase the user satisfaction.

Our comprehensive of literature review shows that KANO model [89] is one of the best and highly recommended models to classify and prioritize requirements based on user satisfaction. In fact, it is a widely-used in the marketing and business sides. KANO model contains four types of requirements:

1. **Basic.** This is a basic requirement that customers expect to have. Customers are not surprised when this requirement is implemented, as they it is assumed that all basic requirements must be implemented. On the other hand, customers might be surprised if this requirement is not implemented. It can therefore be concluded that this requirement does not effect to the user satisfaction. For example, in the ATM system, the basic functionalities are withdraw, inquiry and transfer money.
2. **Indifferent.** This is an indifferent requirement that we may implement in order to be different from our competitors. Unfortunately, Noriaki claimed that this requirement does not satisfy customers at all. In fact, they do not care or acknowledge if this requirement is implemented. For example, in the ATM

system, the indifferent functionality is to book airline tickets from the machine. This type of requirement is excluded in the proposed method because this dissertation uses WOW factors, mentioned in Figure 4-10, to prioritize requirements based on user satisfaction. Our study [124] compares both of KANO model and WOW factors and we discover that the “indifferent” requirements are excluded in the WOW factors.

3. **Delight.** This is likely a nice-to-have requirement that can satisfy customers. Noriaki mentioned that the more nice-to-have requirements are included the higher the user satisfaction will be. ROI is one of the most important factors to determine to implement this requirement. Typically, high ROI requirements should have higher priority when it comes to implementation. For example, in the ATM system, nice-to-have requirements could be transferring money to other countries and mutual fund investment.
4. **Attractive.** This is beyond customer’s expectation. This requirement highly increases user satisfaction. Noriaki claimed that the more this requirement is implemented, the more project success can be. For example, in the ATM system, a surprise or attractive requirement could be withdraw, inquiry and transfer from multiple own accounts.

The literature review shows that the KANO model is widely-used in several of industries to classify and prioritize based on user satisfaction.

Apart from KANO model, this dissertation proposes to use WOW factors [100][114] to support an idea of classifying and prioritizing requirements based on user satisfaction. These factors and their implementation cost can be shown as follows:

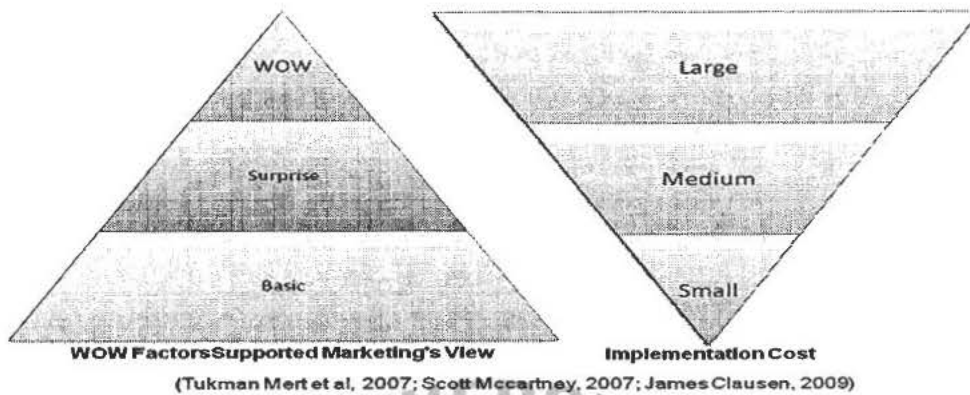


Figure 4-10 WOW Factors and Implementation Cost

Figure 4-10 shows that there are three groups of requirements from marketing's perspective: (a) basic (b) surprise and (c) WOW (or also known as extraordinary). These factors can be discussed in details as follows:

1. **Basic.** This is the same as a basic requirement introduced by Noriaki [89]. It is likely to be a must-have requirement. Tokman [100] and Milliard [114] studied and found that this requirement requires a small amount of cost to implement.
2. **Surprise.** This is a surprise requirement that has an impact on the user satisfaction factor. This is similar to a surprise requirement proposed by KANO model. This requirement requires higher cost than basic requirements.
3. **Extraordinary (also called "WOW").** This is by far beyond user's expectation. It takes a large amount of cost to implement this requirement.

Tokmann [100] claimed that these factors are perfectly suitable to reacquire lost customers. In the business' perspective, acquiring a new customer requires a significant amount of cost and time. Reacquiring lost existing customers is cheaper.

Additionally, the study shows that the implementation cost for a set of extraordinary requirements is the highest while the cost for basic requirements is the lowest.

When the requirements are classified, the next step is to prioritize requirement based on return on investment (ROI) and requirement complexity [70]. The proposed requirement prioritization method is built on the benefit and cost estimation [52][109].

The literature review shows that there are many techniques to calculate estimated efforts and cost for coding and testing (See *Eff* in (1)), such as COCOMO, Function Analysis and Cost-Value approach. Our previous work [124] discovers that one of the most widely-used techniques to estimate efforts and cost is the “*benefit and cost prediction and estimation*” approach introduced by Andrea [202].

Andrea and Maya [202] argue that “*Requirements prioritization based on importance has been a popular concept in software engineering for more than 30 years.*” They investigate and research how existing estimation approaches (e.g. COCOMO, Functional Analysis, Analytic Hierarchy Process, Planning Game, Binary Search Tree and traditional cost-value approach) are suitable for the requirement prioritization based on benefit and cost. They compare 15 requirement prioritization approaches in order to systematically determine which method is the highest recommended technique. Their evaluation result indicates that the most recommended prioritization technique based on importance is to simply calculate by using ROI.

Additionally, Richard [201] supports Andrea’s statement that requirement prioritization based on ROI is the most effective approach to prioritize requirements. He [201] claims that “*ROI is an effective approach for arguing the need for, or demonstrating the success of, process improvements and IT investments.*”

Therefore, this dissertation is built on Andrea’s experiment. We use the simple benefit and cost model in the proposed method. The following paragraphs describe a simple method to calculate ROI based on benefit and cost.

The following paragraphs describe the procedure step-by-step.

The first step is to compute a total estimated cost. The formula can be found as follows:

$$Cst = Eff * Cph \quad (1)$$

Where:

- *Cst* is a total estimated cost.
- *Eff* is a total number of efforts estimated for coding and testing.
- *Cph* is an employee cost per hour for coding and testing.

In order to compute *Cph*, we propose a cost-value approach based on WOW factors. For example, a cost of implementing “surprise” requirements is three times greater than the cost of “basic” requirements. It is assumed that the employee cost per hour is \$65 [201]. Richard [201] suggests that the average cost per employee per year is \$120,000 (Assumed that a number of working days are 230). This is equal to \$522 employee cost per day, which is equal to \$65 per hour. This dissertation use Richard’s average cost per employee per year in the example described in section 4.5. Therefore, the employee cost per hour for “surprise” requirements is equal to \$195 while the cost for “basic” requirements is \$65.

The second step is to compute the total cost for the customer. The formula can be found as follows:

$$Chg = Eff * Cgh \quad (2)$$

Where:

- *Chg* is a total number of fees that are charged to.
- *Eff* is a total number of efforts estimated for coding and testing.
- *Cgh* is the fee per hour charged for customers.

Richard [201] suggests that an average fee per hour charged for customers is \$100 for “Basic”, \$300 for “Surprise” and \$500 for ‘Extraordinary’ requirements. In

order to compute C_{gh} , this dissertation use Richard's average fee per hour in the example described in section 4.5.

The third step is to simply calculate ROI, as follows:

$$\%ROI = \frac{(Chg - Cst)}{Cst} * 100 \quad (3)$$

Where:

- $\%ROI$ is a percentage of return on investment.
- Chg is a total number of fees that are charged to customers.
- Cst is a total estimated cost.

In general, ROI is used to systematically determine which requirements (under surprise and extraordinary introduced by WOW concept) should be implemented. However, this dissertation discovers that ROI is not the only factor to estimate and determine which requirements could be developed and tested [109]. We find that a complexity of requirements is another factor needed to be taken into the account.

In fact, requirements with high ROI and less complexity are desirable. Our study [124] proposes to divide ROI by requirement complexity. This is because we want to prioritize high ROI and less complex requirements as top priority.

ROI is based on the estimation of development and testing. There is a chance that the estimation may be under estimated and cost is overrun when implementing high complex requirements.

In 2005, Hans [163] supported that the requirement complexity plays a major role for the requirement prioritization. He supported that "*complexity is the most important factor.*" He said that there are over 200 different complexity measurements to determine a complex of requirements. However, he suggested that 'a number of hours' is the simple and effective measurement to determine a complexity.

Additionally, Holly [70] argues that *“the number one reason for inability to complete a project as ‘incomplete requirements’”* The survey report in [70] confirmed that the above statement is true. Holly also claims that *“requirement complexity is well known paradigm within the software engineering domain”*. Holly defines a meaning of complexity as follows: *“(a) consisting of many different and connected parts and (b) not easy to analyze; complicated or intricate”*. In [70] there are many measurements defined to identify a complexity of requirements, such as time spent on project, a number and location of stakeholders and project resources. Eventually, the research [70] suggests to determine a requirement complexity based on a number of hours to implement the requirements. In fact, Holly proposes the following simple table to measure a requirement complexity.

Table 4-3 Measuring Requirement Complexity

Number of Hours	0-8	9-40	41-160	161-320	321-480
Complexity Number	1	2	3	4	5
*Note: 1 is very low, 2 is low, 3 is medium, 4 is high and 5 is very high.					

Table 4-3 guides a simple scale to measure requirement complexity. It defines 1 is very low, 2 is low, 3 is medium, 4 is high and 5 is very high complexity. This dissertation uses the above guideline to simply measure a requirement complexity. This is corresponding with we propose in (1), (2) and (3). In [70], the research claims that a complex of requirements is one of the most important factors we need to consider for prioritization. Our study [124] discovers that traditional requirement prioritization, based on benefit and cost, does not concentrate on the requirement complexity. Consequently, this can lead to a poor performance of prioritization. Highly complex requirements with high ROI may be prioritized as top priority. This can also lead to questions that *“What about requirements that have the*

same high ROI, but less complexity? Why don't we consider requirement complexity as well during prioritization process?"

To answer these questions, this dissertation proposes the following formula (4) in order to determine highly recommended priorities for each requirement. We normally prefer less complex requirements that have a high ROI. For example, there are two requirements that have the same ROI. The requirements with less complexity to implement and test should have higher priority.

$$Cor = \frac{\%ROI}{Cpx} \quad (4)$$

Where:

- *Cor* is a percentage ratio between ROI and a complexity of requirement.
- *Cpx* is a complexity of requirement.

The last step is to prioritize requirements based on the correlation ratio. Our study [124] reveals that the requirements with high ROI and high complexity are not desirable. From marketing's perspective, the complex requirements with high ROI may not be able to be implemented on time. Therefore, the requirements with high ROI and less complexity are preferred.

4.5 Example of Requirement Prioritization

This section provides an example of the above approach to classify and prioritize requirements. The following shows the example of 10 requirements, aligned with WOW factors, and implementation cost of each requirement.

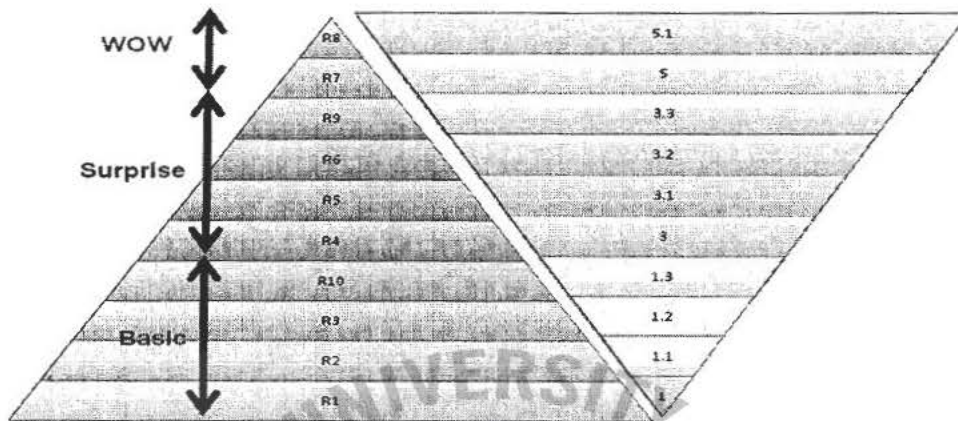


Figure 4-11 Example of Requirement Prioritization

Figure 4-11 shows that there are 10 requirements, classified with WOW factors, as follows:

1. Basic requirements – R_1 , R_2 , R_3 and R_{10} .
2. Surprise requirements – R_4 , R_5 , R_6 and R_9 .
3. WOW requirements – R_7 and R_8 .

The implementation cost of each requirement is assigned by weights as follows: $R_1 = 1$, $R_2 = 1.1$, $R_3 = 1.2$, $R_{10} = 1.3$, $R_4 = 3$, $R_5 = 3.1$, $R_6 = 3.2$, $R_9 = 3.3$, $R_7 = 5$ and $R_8 = 5.1$.

In order to prioritize above requirements, the proposed method can be explained as follows:

The first step is to compute total estimated cost for each requirement, which can be shown as follows:

Table 4-4 Total Estimated Cost

Req. Id	Classification	Estimated Efforts for Coding (Man-Hours) (a)	Estimated Efforts for Testing (Man-Hours) (b)	Total Efforts (Man-Hours) (c)	Cost-Value Assignment (d)	Employee Cost per Hour (e) $= (d) * 65$	Total Estimated Cost (f) $= (c) * (e)$
1	Basic	32	24	56	1.00	\$65.00	\$3,640
2	Basic	24	16	40	1.10	\$71.50	\$2,860
3	Basic	32	8	40	1.20	\$78.00	\$3,120
4	Surprise	88	80	168	3.00	\$195.00	\$32,760
5	Surprise	48	40	88	3.10	\$201.50	\$17,732
6	Surprise	96	88	184	3.20	\$208.00	\$38,272
7	Extra Ordinary	120	100	220	5.00	\$325.00	\$71,500
8	Extra Ordinary	220	190	410	5.10	\$331.50	\$135,915
9	Surprise	90	40	130	3.30	\$214.50	\$27,885
10	Basic	8	8	16	1.30	\$84.50	\$1,352

Table 4-4 presents total estimated cost calculated by using formula (1). In this example, the employee cost per hour is equal to \$65 [201]. Each requirement requires different cost per hour based on assigned weights. For example, R_1 requires \$65 ($=65*1$), R_4 requires \$195 ($=65*3$) and R_7 requires \$325 ($=65*5$).

Next, the second step is to calculate total charges to customer. This example assumes that charge per hour is \$100. Therefore, total charges for each requirement can be shown as follows:

Table 4-5 Total Charges to Customer

Req. Id	Classification	Total Efforts (Man-Hours) (a)	Cost-Value Assignment (b)	Total Estimated Cost (c)	Charge Customer per Hour (d)	Total Charges (e) = (a)*(d)
1	Basic	56	1.00	\$3,640	\$100	\$5,600
2	Basic	40	1.10	\$2,860	\$100	\$4,000
3	Basic	40	1.20	\$3,120	\$100	\$4,000
4	Surprise	168	3.00	\$32,760	\$300	\$50,400
5	Surprise	88	3.10	\$17,732	\$300	\$26,400
6	Surprise	184	3.20	\$38,272	\$300	\$55,200
7	Extra Ordinary	220	5.00	\$71,500	\$500	\$110,000
8	Extra Ordinary	410	5.10	\$135,915	\$500	\$205,000
9	Surprise	130	3.30	\$27,885	\$300	\$39,000
10	Basic	16	1.30	\$1,352	\$100	\$1,600

Table 4-5 presents total charges per requirement for customer by using formula (2). For example, the total charges for R_1 is equal to \$5,600 ($=56*100$), R_4 is equal to \$50,400 ($=168*300$) and R_7 is equal to \$110,000 ($=220*500$).

Afterward, the next step is to compute ROI using both of total estimated cost and total charges. The following shows ROI for each requirement.

Table 4-6 ROI for Each Requirement

Req. Id	Classification	Total Estimated Cost (a)	Total Charges (b)	Estimated Benefit (c) = (b)-(a)	ROI (%) (d) = ((c/a)*100
1	Basic	\$3,640	\$5,600	\$1,960	53.85%
2	Basic	\$2,860	\$4,000	\$1,140	39.86%
3	Basic	\$3,120	\$4,000	\$880	28.21%
4	Surprise	\$32,760	\$50,400	\$17,640	53.85%
5	Surprise	\$17,732	\$26,400	\$8,668	48.88%
6	Surprise	\$38,272	\$55,200	\$16,928	44.23%

Req. Id	Classification	Total Estimated Cost (a)	Total Charges (b)	Estimated Benefit (c) =(b)-(a)	ROI (%) (d) =((c/a)*100
7	Extra Ordinary	\$71,500	\$110,000	\$38,500	53.85%
8	Extra Ordinary	\$135,915	\$205,000	\$69,085	50.83%
9	Surprise	\$27,885	\$39,000	\$11,115	39.86%
10	Basic	\$1,352	\$1,600	\$248	18.34%

Table 4-6 presents ROI for each requirement using formula (3). However, we discover that requirements with high ROI and less complexity are desirable [70][124][163]. Recall that there are some arguments that requirement complexity factor is one of the most important factors for prioritization. Therefore, this dissertation proposes to use both of ROI and requirement complexity.

The last step shows a correlation ratio between ROI and requirement complexity, as follows:

Table 4-7 Ratio between ROI and Requirement Complexity

Req. Id	Classification	Total Efforts (Man-Hours) (a)	ROI (%) (b)	Requirement Complexity Factor (c)	Correlation Ratio (%) (d) =(b)/(c)
1	Basic	56	53.85%	3	17.95%
2	Basic	40	39.86%	2	19.93%
3	Basic	40	28.21%	2	14.10%
4	Surprise	168	53.85%	4	13.46%
5	Surprise	88	48.88%	3	16.29%
6	Surprise	184	44.23%	4	11.06%
7	Extra Ordinary	220	53.85%	4	13.46%
8	Extra Ordinary	410	50.83%	5	10.17%
9	Surprise	130	39.86%	3	13.29%
10	Basic	16	18.34%	1	18.34%

Table 4-7 shows a correlation ratio for each requirement. Our proposed method is to prioritize requirements based on the above ratio. Therefore, the prioritized requirements are: $R_2, R_{10}, R_1, R_5, R_3, R_4, R_7, R_9, R_6$ and R_8 .

4.6 Test Case Generation Technique

This section presents a test case generation method derived from fully dresses use case. After the requirements are classified and prioritized, we propose to generate test cases from those prioritized requirements that can be represented in the UML use case diagram. Alistair [40][151] classified the UML use case diagram into three categories: brief use case, casual use case and fully dressed use case.

The first category contains the following elements: use case name, use case number and goal. The second category is consists of: use case name, use case number, goal / purpose and summary. The last category is composed of all information, such as use case name, summary, conditions, basic path, alternative path and business rules. The proposed method concentrates on the last type only.

Our proposed method contains four steps, as follows: (a) extract use case diagram (b) extract test scenario, (c) generate test data and expected result and (d) minimize test cases. These steps can be shortly described as follows:

The first step is to extract the following information from fully dressed use cases: (a) use case number (b) purpose (c) summary (d) pre-condition (e) post-condition (f) basic event and (g) alternative events. This information is called use case scenario in this thesis. The example fully dressed use cases of ATM withdraw functionality can be found as follows [151]:

Table 4-8 Example Fully Dressed Use Case

Use Case Id	Use Case Name	Summary	Basic Event	Alternative Events	Business Rules
UC-001	Withdraw	To allow bank's customers to withdraw money from ATM machines anywhere in Thailand.	1. Insert Card 2. Input PIN 3. Select Withdraw 4. Select A/C Type 5. Input Balance 6. Get Money 7. Get Card	1. Select Inquiry 2. Select A/C Type 3. Check Balance	(a) Input amount \leq Outstanding Balance (b) Fee charge if using different ATM machines

The above use cases can be extracted into the following use case scenarios:

Table 4-9 Extracted to Use Case Scenarios

Scenario Id	Summary	Basic Scenario
Scenario-001	To allow bank's customers to withdraw money from ATM machines anywhere in Thailand.	1. Insert Card 2. Input PIN 3. Select Withdraw 4. Select A/C Type 5. Input Balance 6. Get Money 7. Get Card
Scenario-002	To allow bank's customers to withdraw money from ATM machines anywhere in Thailand.	1. Insert Card 2. Input PIN 3. Select Inquiry 4. Select A/C Type 5. Check Balance 6. Select Withdraw 7. Select A/C Type 8. Input Balance 9. Get Money 10. Get Card

The second step is to automatically extract test scenarios from the previous use case scenarios [69]. From the above table, the following test scenarios can be extracted.

Table 4-10 Extract to Test Scenarios

Test Scenario Id	Summary	Basic Scenario
TS-001	To allow bank's customers to withdraw money from ATM machines anywhere in Thailand.	1. Insert Card 2. Input PIN 3. Select Withdraw 4. Select A/C Type 5. Input Balance 6. Get Money 7. Get Card
TS-002	To allow bank's customers to withdraw money from ATM machines anywhere in Thailand.	1. Insert Card 2. Input PIN 3. Select Inquiry 4. Select A/C Type 5. Check Balance 6. Select Withdraw 7. Select A/C Type 8. Input Balance 9. Get Money 10. Get Card

The next step is to manually generate input data, expected result, actual result and "pass / fail" status for each test scenario. This example assumes that these elements can be created as follows:

Table 4-11 Extract to Test Cases

Test Case Id	Summary	Basic Scenario	Input Data	Expected Result	Actual Result*	Pass / Fail Status*
TC-001	To allow bank's customers to withdraw money from ATM machines anywhere in Thailand.	1. Insert Card 2. Input PIN 3. Select Withdraw 4. Select A/C Type 5. Input Balance 6. Get Money 7. Get Card	PIN, Amount, Balance	User gets money and the balance is calculated successfully.
TC-002	To allow bank's customers to withdraw money from ATM machines anywhere in Thailand.	1. Insert Card 2. Input PIN 3. Select Inquiry 4. Select A/C Type 5. Check Balance 6. Select Withdraw 7. Select A/C Type 8. Input Balance 9. Get Money 10. Get Card	PIN, Amount, Balance	The outstanding balance is displayed. User gets money and the balance is calculated successfully.

**Actual result and “pass/ fail” status can be fulfilled when test cases are executed.*

The last step is to minimize a number of test cases. The outstanding problem for reducing black-box tests is to unable to identify which test cases should be removed, as mentioned in Figure 4-1. Therefore, we propose to reduce a number of test cases based on alternative paths of use cases, called “ALT”. The study shows that there are many alternative paths in each use case [69][197].

Peter [197] argued that one requirement can have multiple use cases. Each use case must have basic path and has at least one alternative path. The relationship among requirements, use cases, basic paths, alternative paths and test cases can be shown as follows:

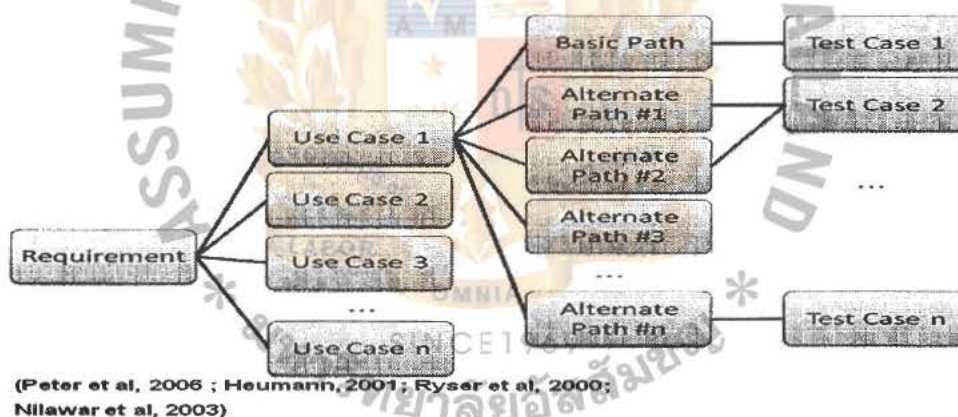


Figure 4-12 Overwhelm Alternative Paths

The studies [24][52][69][85][96][105][197] prove that both of basic and alternative paths, described in fully dressed use cases, play a major role to generate tests. In fact, each use case can have overwhelmed of those paths [197]. This means that some of duplicated or unnecessary paths could be removed. This is because duplicated paths can lead to larger number of test cases. One of our objectives is to minimize a number of test cases during test case generation process. Therefore, this dissertation proposes a formula, associated with alternative paths, to preserve high

coverage alternative paths. Note that the large number of unnecessary alternative paths can lead to greater deal amount of cost and time to generate and execute tests.

Therefore, we propose the following formula to identify which test cases should be removed.

$$Alt(TC_n) = Wgh(TC_n) * Pth(TC_n) \tag{5}$$

Where:

- $Alt(TC_n)$ is an alternate path point for each test case, TC_n .
- $Wgh(TC_n)$ is a weight factor that is calculated by using a number of paths in each use case, for each test case, TC_n , where:
 - $Weight = 1$ when a number of paths is less than or equal to 3.
 - $Weight = 2$ when a number of paths is greater than or equal to 4 and less than or equal to 7.
 - $Weight = 3$ when a number of paths is greater than 7.
- $Pth(TC_n)$ is a number of covered paths for each test case, TC_n .

The following shows an example of how to calculate the alternate path points.

	Path 1 of Use Case 1	Path 2 of Use Case 1	Path 1 of Use Case 2	Path 2 of Use Case 2	Path 3 of Use Case 2
T1	X	X	X	X	
T2	X				
T3	X	X			
T4		X	X	X	X
T5			X	X	
T6					X
T7	X				

Figure 4-13 Matrix Table between Tests and Paths

Figure 4-13 presents a testing matrix table described a relationship between test cases and alternate paths of use cases. There are seven test cases and two use cases. The first use case contains two alternate paths. The second use case consists of three paths.

The weight and value of the above test cases can be described as follows:

$$Wgh(T_1) = 2, Pth(T_1) = 4$$

$$Wgh(T_2) = 1, Pth(T_2) = 1$$

$$Wgh(T_3) = 1, Pth(T_3) = 2$$

$$Wgh(T_4) = 2, Pth(T_4) = 4$$

$$Wgh(T_5) = 1, Pth(T_5) = 2$$

$$Wgh(T_6) = 1, Pth(T_6) = 1$$

$$Wgh(T_7) = 1, Pth(T_7) = 1$$

The above test cases can be calculated the alternate path points as follows:

$$Alt(T_1) = 2 * 4 = 8$$

$$Alt(T_2) = 1 * 1 = 1$$

$$Alt(T_3) = 1 * 2 = 2$$

$$Alt(T_4) = 2 * 4 = 8$$

$$Alt(T_5) = 1 * 2 = 2$$

$$Alt(T_6) = 1 * 1 = 1$$

$$Alt(T_7) = 1 * 1 = 1$$

With the above alternate path points, T_2 , T_6 and T_7 are removed due to minimum points.

However, the further study [124][125][197] shows that alternate path points are not enough to reduce a number of black box tests. The above method ignores a risk of each test case. Risk contains two major factors [10][139][163]: (a) level of

damage and (b) probability of failure. In September of 2010, our previous work [124] proposed a risk-driven factor that contains both of:

1. **Level of Damage.** This factor indicates a level of damage if test cases are removed. There is a simple guideline to determine a level of damage proposed by Praveen [139].
2. **Probability of Failure.** This factor indicates a probability that test cases can be failed during a test execution process. Hans [163] and his work in 2005 presented that this factor can be represents as a complexity of test cases.

Fortunately, there was a suggestion during EMDT conference where our previous work is published [124] that a level of damage should be removed. This is because it is difficult to systematically determine a level of damage if test cases are removed. The simple guideline is ambiguous and inadequate. This is the first reason why we propose only a probability of failure.

Another reason why we propose to use a retain score associated with a complexity of test cases is that less complex test cases that are generated from alternative paths should be reserved. This is because less complex test cases can lead to a low probability of failure during the test execution [124][125][163]. The retain score allows to reduce a number of high complex test cases.

Eventually, we propose the following formula, called “RET”, to reduce a number of test cases by considering risk factor and alternate path points together.

$$Ret (T_n) = Cpx * Alt \quad (6)$$

Where:

- *Ret* is a retain score for each test case, T_n . Test cases with low score must be removed.
- *Cpx* represents as a total number of test steps in each test case.

- $Alt(TC_n)$ is an alternate path point for each test case, TC_n .

The following describes an example of how to compute retain scores.

Alternate Paths in Use Case	A Number of Test Steps in Test Case
Path 1 of Use Case 1	3
Path 2 of Use Case 1	3
Path 1 of Use Case 2	4
Path 2 of Use Case 2	2
Path 3 of Use Case 2	3

Figure 4-14 Example of Test Steps Required for Path

Figure 4-14 shows that each alternate path of each use case requires a number of test steps in the test case. For example, path 1 of use case 1 requires 3 steps in the test case.

In Figure 4-13 and 4-14, the complexity of each test case can be computed as follows:

$$Cpx(T_1) = 3+3+4+2 = 12$$

$$Cpx(T_2) = 3$$

$$Cpx(T_3) = 3+3 = 6$$

$$Cpx(T_4) = 3+4+2+3 = 12$$

$$Cpx(T_5) = 4+2 = 6$$

$$Cpx(T_6) = 3$$

$$Cpx(T_7) = 3$$

Afterward, the retain score for each test case can be computed as follows:

$$Ret(T_1) = 12*8 = 96$$

$$Ret(T_2) = 3*1 = 3$$

$$Ret(T_3) = 6*2 = 12$$

$$Ret(T_4) = 12 * 8 = 96$$

$$Ret(T_5) = 6 * 2 = 12$$

$$Ret(T_6) = 3 * 1 = 3$$

$$Ret(T_7) = 3 * 1 = 3$$

Finally, all test cases with a minimum retain score are removed. Thus, T_2 , T_6 and T_7 are removed.

4.7 Limitations

The following lists limitations of the proposed techniques.

1. The limitation of the proposed techniques is that both of input data and expected results require manual efforts to generate during a test case generation process.
2. In addition, the proposed techniques can generate test cases from fully dressed use case, which fully contains all required information only. The techniques are limited to brief and casual use case.
3. Alternative path points in the proposed method are not applicable when use cases have only basic path.
4. Our proposed method is limited to only fully dressed use case effectively written based on guidelines in [40][151]. In the commercial industry, it may be difficult to allow analysts to effectively write comprehensive information for use cases.

CHAPTER 5

EVALUATION

The chapter explains how the experiment has been designed, its measurements and the evaluation result, with the aim of determining which test generation method is the most recommended in terms of customer satisfaction. Also, this chapter discusses and compares the result in detail. The evaluation aims to proof that the proposed techniques can work well under circumstance. This dissertation does not argue that other test case generation techniques have poor performance.

5.1 Experiments

The section describes the experiment in details. The objective of the experiment is to provide an empirical support for our contributions mentioned in the Chapter 4. We design the experiment into three parts: (a) prepare data (b) generate test case and (c) evaluate a result. The following shows an overview of experiments.

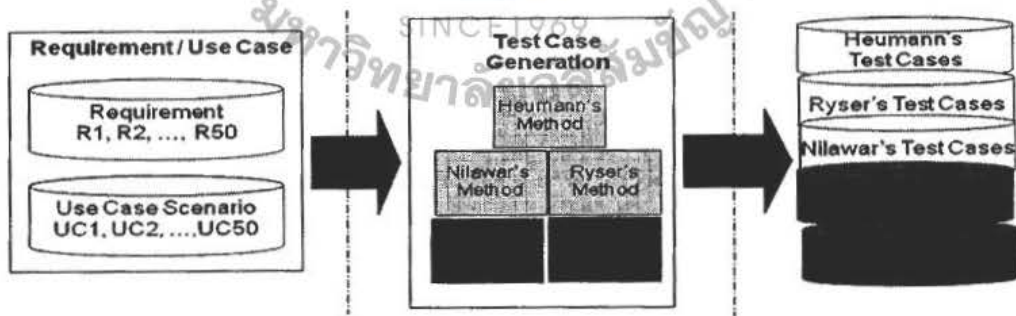


Figure 5-1 Overview of Experiment

Figure 5-1 can be explained in details, as follows:

1. Prepare Experiment Data. Before evaluating the proposed methods and other methods, preparing the experiment data is required. In this step, 50 requirements and 50 use case scenarios, associated with those requirements, are randomly

generated. The “dataset” term is used in the rest of this dissertation to represent the experiment data. This experiment is designed to randomly generate 10 datasets in order to determine an average value for each measurement.

2. Generate Test Case. A comparative evaluation method has been made among the proposed test generation algorithm, Heumann’s technique [69], Ryser’s method [85], Nilawar’s algorithm [105] and the proposed methods presented in the Chapter 4. This experiment aims to compare a performance of ALT and RET methods. This is because it is included a prioritization requirement algorithm prior to generate a set of test cases. The experiment includes a requirement prioritization based on user satisfaction steps in the ALT and RET methods respectively. Also, there is a link relationship between requirements that have been prioritized and use cases for those two proposed methods. There are 10 datasets randomly generated for requirements and use case scenarios. Therefore, this part aims to generate 10 sets of test cases as well.

3. Evaluate Results. In this part, comparative generation methods are executed by using 50 requirements and 50 use case scenarios. These methods are also executed for 10 datasets in order to find out the average percentage of a number of test cases and requirement coverage. In total, there are 500 requirements and 500 use case scenarios executed in this experiment. This part evaluates and compares results based on datasets and proposed measurements.

The following tables present how to randomly generate requirements and use case scenarios for each dataset respectively.

Table 5-1 Generate Random Requirements

Attribute	Description	Approach	Data Type
Requirement ID	A unique number to reference requirement.	Randomly generated from the following combination: <i>Req</i> + <i>Sequence Number</i> . For example, <i>Req1</i> , <i>Req2</i> , <i>Req3</i> , ..., <i>ReqN</i> .	String
Description	A description of requirement.	<p>Randomly generated from the followings:</p> <ol style="list-style-type: none"> 1. Supported protocols that are allowed via the access layer, service cells and core cells 2. Platform services should not require a specified start-up order in order to function properly 3. Platform Services will adhere to the ITIL based release management process and must issue a Release schedule which defines the frequency of major and minor releases 4. Any release/change must adhere to the following guidelines for maintenance windows 5. Services should use the Standard OS Builds provided by Shared Infrastructure Group 6. Automated installation procedures and automated software builds must be provided for servers via the automated scripts 7. Releases should be stored, packaged and delivered using Operations Management Infrastructure approved technology 8. Platform services must document their data retention times and 	String

Attribute	Description	Approach	Data Type
		<p>ensure the supporting processes are in place to meet them</p> <p>9. All regular activities that occur on platform must be scheduled via one of the Management Infrastructure job scheduling tools</p> <p>10. Content and data related changes on platform should be seamless, reliable, transparent and auditable</p>	
Type of Requirement	A type of requirement that contains four groups: Function, Performance, Security and Operational.	<p>Randomly selected from the following values:</p> <ol style="list-style-type: none"> 1. Functional 2. Performance 3. Security 4. Operational 	String
Classification	A classification of requirement based on WOW factors.	Randomly generated from: Basic, Surprise and Extra-ordinary.	String
Estimated Efforts for Coding	An estimated effort for coding.	Randomly generated from 1 to 480 hours.	Numeric
Estimated Efforts for Testing	An estimated effort for testing.	Randomly generated from 1 to 480 hours.	Numeric
Cost-Value Assignment	A cost-value assigned for each requirement.	Randomly generated from 1 to 5 such as 1, 1.1, 1.2, 1.3, 3, 3.1, 3.2, 3.3, 5, 5.1, 5.2 and 5.3.	Numeric
ReqComplex	A complexity of requirements.	Apply Holly's guideline in Holly's work [70].	Numeric
Correlation	A correlation ratio between ROI and requirement complexity.	This attribute is calculated by using ROI and requirement complexity as mentioned in the proposed method.	Numeric

Attribute	Description	Approach	Data Type
Num Use Case		Randomly generate from 1 to 10.	Numeric

Table 5-2 Generate Random Use Case Scenario

Attribute	Description	Approach	Data Type
Use case ID	A unique number to reference use case.	Randomly generated from the following combination: <i>uCase</i> + <i>Sequence Number</i> . For example, <i>uCase₁</i> , <i>uCase₂</i> , ..., <i>uCase_n</i> .	String
Purpose	A detail to explain a purpose of use case.	Randomly generated from the followings <ol style="list-style-type: none"> 1. Remote Procedure Calls (RPC) are not permitted through firewalls 2. The use of any protocol that dynamically allocates ports, rather than operating through set, known ports, is not permitted 3. As part of Management Infrastructure the patch management solution has been chosen as the product for security patch management 4. The Network Infrastructure Components used in platform must adhere to the management team's recommended technical standard 5. Applications statistics must be created in line with the formatting standards specified by the global management tool and transferred to the PAWZ Server on a daily basis 6. Each service/capability must have an associated 	String

Attribute	Description	Approach	Data Type
		<p>Site Failover (SFO) recovery plan</p> <p>7. The table below describes the baseline standard password security controls for internal applications and products</p> <p>8. Services should be built on approved technology platforms (hardware and OS)</p> <p>9. Events should be sent and formatted using the standards specified by Management Infrastructure</p> <p>10. The BSV should represent the status of the service/capability from an end-user client perspective</p>	
Pre-condition	A pre-requisite condition that must be done before executing use case.	Randomly generated from the following combination: <i>pCon</i> + <i>Sequence Number same as Use case ID</i> . For example, <i>pCon</i> ₁ , <i>pCon</i> ₂ , ..., <i>pCon</i> _n .	String
Basic Path	A basic path of use case.	Randomly generated from the following combination: <i>uCase</i> + <i>Sequence Number</i> . For example, <i>basic</i> ₁ , <i>basic</i> ₂ , ..., <i>basic</i> _n .	String
No Alternate Paths	A number of alternative paths in use case.	Random a number of paths from 1 to 10	Numeric

Table 5-3 Generate Random Alternative Paths for Use Cases

Attribute	Description	Approach	Data Type
Use case ID	A unique number to reference use case.	Randomly generated from the following combination: <i>uCase</i> + <i>Sequence Number</i> . For example, <i>uCase₁</i> , <i>uCase₂</i> , ..., <i>uCase_n</i> .	String
AltID	A unique number to reference an alternative path.	Randomly generated from the following combination: <i>Sequence Number</i> . For example, 1, 2, ..., <i>n</i> .	Numeric
No_Steps	A number of steps in each alternative path.	Randomly generated from 1 to 10.	Numeric
Steps	Details of steps in each alternative path.	Randomly generated from the following combination: <i>Step₁</i> + "+" + <i>Step₂</i> + "+" + ... <i>Step_n</i> ; <i>n</i> is a <i>No_Steps</i> number. For example, Step1 + Step2 or Step 1+ Step2 + Step3.	String

The following presents an example of test cases used in the experiment.

Table 5-4 Attributes of Test Cases

Attribute	Description	Data Type
Test Case ID	A unique number of test cases.	Numeric
Purpose	Detail information describing what the purpose of test case is.	String
Pre-Conditions	Pre-requisite conditions that must be done before executing test cases.	String
Test Steps	A detail information describing steps to execute test case.	String
Input Data	An input data using during a test execution	String
Expected Result	An expected result of test case that describes what the result should be.	String
Actual Result	An actual result of test case that describes what the result is after execution.	String
Status	A status of test case that contains "pass" or "fail"	Boolean

The following presents an example of data used in the experiment.

Table: Requirement						
Req ID	Description	Type Req	Classification	Est Eff Code	Est Eff Test	Cost Value
1	Supported pre: Function		Surprise	120	80	3.1

Table: Use Case				
Use Case ID	Purpose	Pre-Condition	Basic Path	Num Alt Pat
1.4.The Network Infrastructure Component	PreCon1		Basic1	2

Table: Step-Use Case		Table: Alternative Path			
Req ID	Use Case ID	AltPath ID	Num Steps	Steps	Use Case ID
1	1	1	3	Step1 + Step2 + Step3	1
		2	4	Step1 + Step2 + Step3 + Step4	1

Table: Test Case						
Test Case ID	Purpose	Pre-Condition	Test Step	Input Data	Expected Re	Actual Result
1.4.The network infrastructure PreCon1		Basic1	Step1	Input1	Expect1	<input type="checkbox"/>
2.4.The network infrastructure PreCon1		Step1 + Step2 + Step3	Step1	Input1	Expect1	<input type="checkbox"/>
3.4.The network infrastructure PreCon1		Step1 + Step2 + Step3 + Step4	Step1	Input1	Expect1	<input type="checkbox"/>

Figure 5-2 Example of Generated Random Data

5.2 Measurements

The section lists the metrics used in the experiment. This thesis proposes to use the following metrics, which are: (a) a number of test cases and (b) percentage of critical requirement coverage. This is because these measurements are selected to proof that the proposed method can generate small number of test cases while maintaining requirements coverage.

The followings describe these measurements in details.

1. A Number of Test Cases: This is the total number of generated test cases, expressed as a percentage, as follows:

$$\% Size = \frac{\#Size}{\# of Total} * 100 \quad (7)$$

Where:

- *% Size* is a percentage of the number of test cases.
- *# of Size* is a number of test cases.
- *# of Total* is the maximum number of test cases in the experiment, which is assigned 1,000.

2. Requirement Coverage: This is an indicator to identify the number of requirements covered in the system [14]. Due to the fact that one of the goals of software testing is to verify and validate requirements covered by the system, this metric is a must. Therefore, a high percentage of critical requirement coverage is desirable.

It can be calculated using the following formula:

$$\% CRC = \frac{\# Critical}{\# of Total} * 100 \quad (8)$$

Where:

- $\% CRC$ is the percentage of critical requirement coverage.
- $\# of Critical$ is a number of critical requirements covered.
- $\# of Total$ is the total number of requirements.

In 2005, Avik [14] used the following guideline in the experiment.

- If correlation ratio of requirement is greater than 80%, it shows that requirement is one of top priority requirements.
- If it is greater than 50%, it shows that requirement is one of medium priority requirements.

5.3 Results

This section shows comparison results of the above experiment. There are two types of comparison graph results:

1. Comparison based on each dataset randomly generated in each round by the approach in section 5.1.
2. Comparison of all measurements mentioned in section 5.2 among test case generation techniques.

5.3.1 Compare based on Dataset

This section presents three graphs that compare the latest proposed method against other three existing test case generation techniques, based on dataset generated randomly. Those three techniques are: (a) Heumman’s method [69] (b) Ryser’s work [85] and (c) Nilawar’s approach [105]. There are two dimensions in the following graph, (a) horizontal and (b) vertical axis. The horizontal represents a percentage value whereas the vertical axis represents a number of dataset.

The following graph compares a number of test cases based on each dataset generated as explained above.

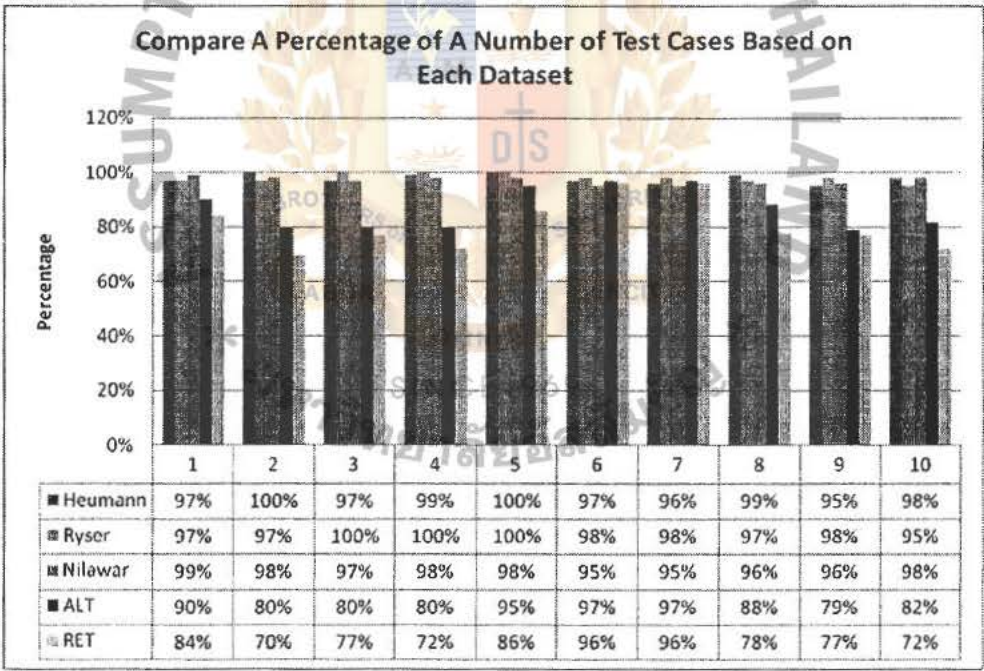


Figure 5-3 Comparison Result for a Number of Test Cases

Figure 5-3 presents that the proposed methods generate a number of test cases slightly smaller than other methods. Meanwhile, other three methods have a similar number of test cases. This is because the propose method has reduced a number of test cases during a test case generation process.

Secondly, the following compares requirement coverage based on each dataset generated as explain above.

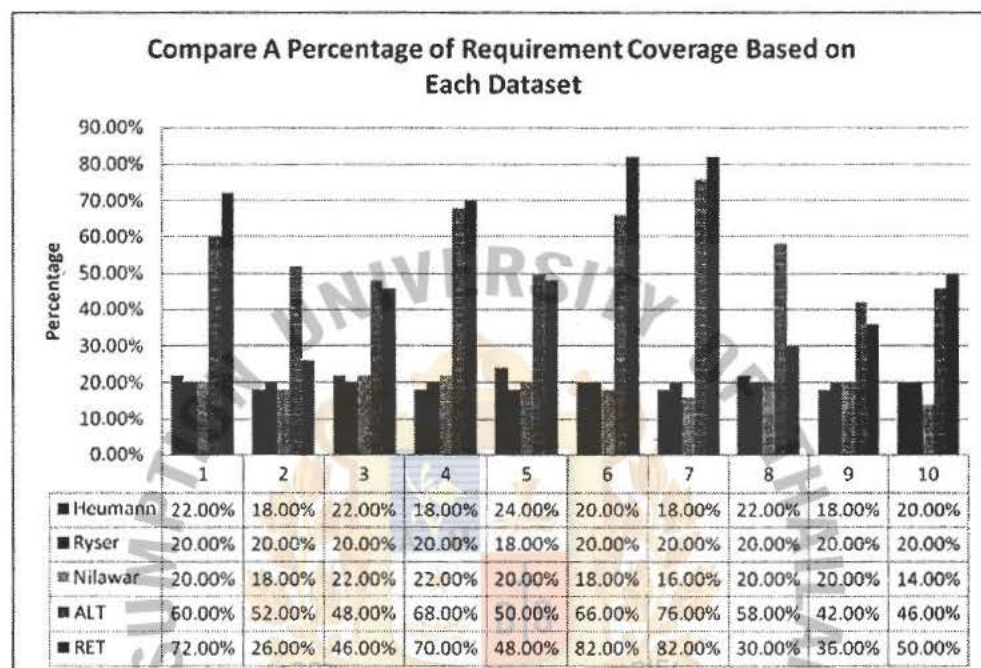


Figure 5-4 Comparison Result for Requirement Coverage

Figure 5-4 shows that the proposed methods have a high percentage of critical requirement coverage and it is by far greater than other methods. This is because the proposed method classifies and prioritizes requirements before generating test cases.

5.3.2 Compare based on Measurements

This section presents a graph that compares the latest proposed method against other three existing test case generation techniques, based on the following measurements: (a) an average of number of test cases and (b) an average of critical requirements coverage. Those three techniques are: (a) Heumann's method [69] (b) Ryser's work [85] and (c) Nilawar's approach [105]. There are two dimensions in the following graph: (a) horizontal and (b) vertical axis. The horizontal represents two measurements whereas the vertical axis represents the percentage value.

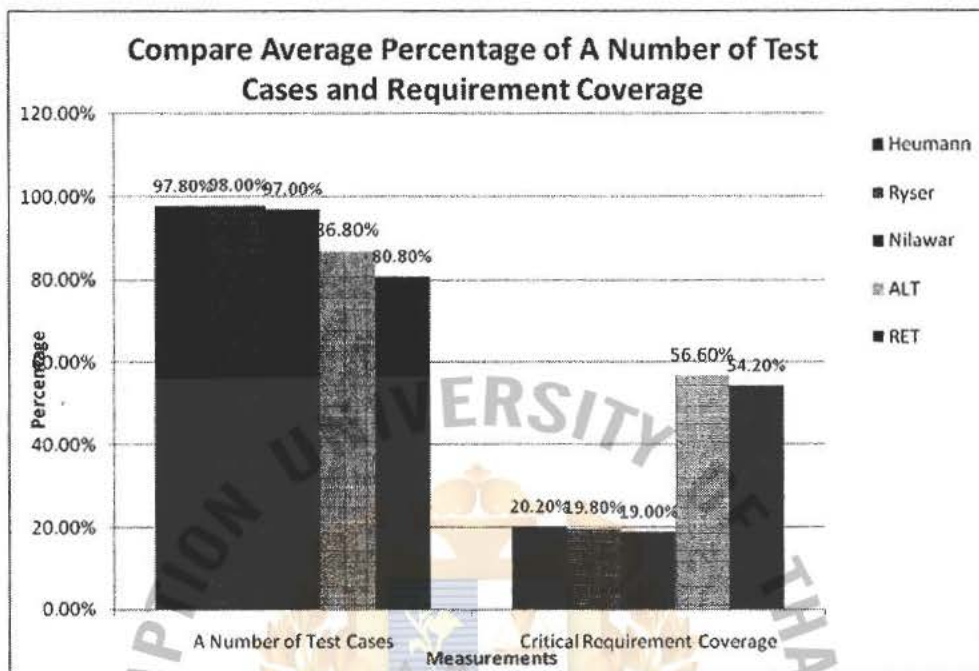


Figure 5-5 Result of Test Case Generation Methods

Figure 5-5 shows that the above proposed method generates the smallest set of test cases. It is calculated as 80.80% where as the other techniques is computed over 97%. Those techniques generated a bigger set of test cases, than a set generated by the proposed method. The literature review reveals that the smaller set of test cases is desirable. Finally, the graph presents that the proposed methods are the most recommended techniques to coverage high priority requirements. Its percentage is much greater than other techniques' percentage, more than 30%.

From Figure 5-5, this study determines and ranks the above comparative methods into five ranking: 5-Excellent, 4-Very good, 3-Good, 2-Normal and 1-Poor. This study uses a maximum and minimum value to find an interval value for ranking those methods.

For a number of test cases, the maximum and minimum percentage is 98% and 80.80%. The different between maximum and minimum value is 17.2%. An interval value is equal to a result of dividing the different values by 5. As a result, the interval

value is approximately 3.4. Thus, it can be determined as follows: 5-Excellent (since 80.80% to 84.2%), 4-Very good (between 84.2% and 87.6%), 3-Good (between 87.6% and 91%), 2-Normal (between 91% and 94.4%) and 1-Poor (from 94.4% to 97.8%).

To cover requirement, the maximum and minimum percentage is 53.20% and 19%. The different value is 34.2%. The interval value is 6.84. Therefore, it can be determined as follows: 5-Excellent (since 46.36% to 53.2%), 4-Very good (between 39.52% and 46.36%), 3-Good (between 32.68% and 39.52%), 2-Normal (between 25.84% and 32.68%) and 1-Poor (from 19% to 25.84%).

Therefore, the experiment result of those comparative methods can be shown below:

Table 5-5 A Comparison Result for Test Case Generation Methods

Algorithm	A Number of Test Cases	Requirement Coverage
Heumann	1	1
Ryser	1	1
Nilawar	1	1
ALT	5	5
REP*	5	5

In conclusion, the proposed method is the most recommended technique to generate the smallest size of test cases with the maximum requirement coverage. However, this dissertation does not claim that other techniques are poor.

5.4 Discussions

This section discusses the above evaluation results. This dissertation does not claim that other comparative test case generation methods are worst or have a poor performance during test case generation activities. In fact, the evaluation aims to prove that the proposed methods in this dissertation work as expected.

Our experiment found that our proposed method is the most recommended test case generation technique to minimize a number of test cases. Also, our experiment showed that our method is the best method comparing to other methods, like Heumann, Ryser and Nilawar. Those methods generate larger number of test cases. The following shows a comparison result in terms of numbers of test cases:

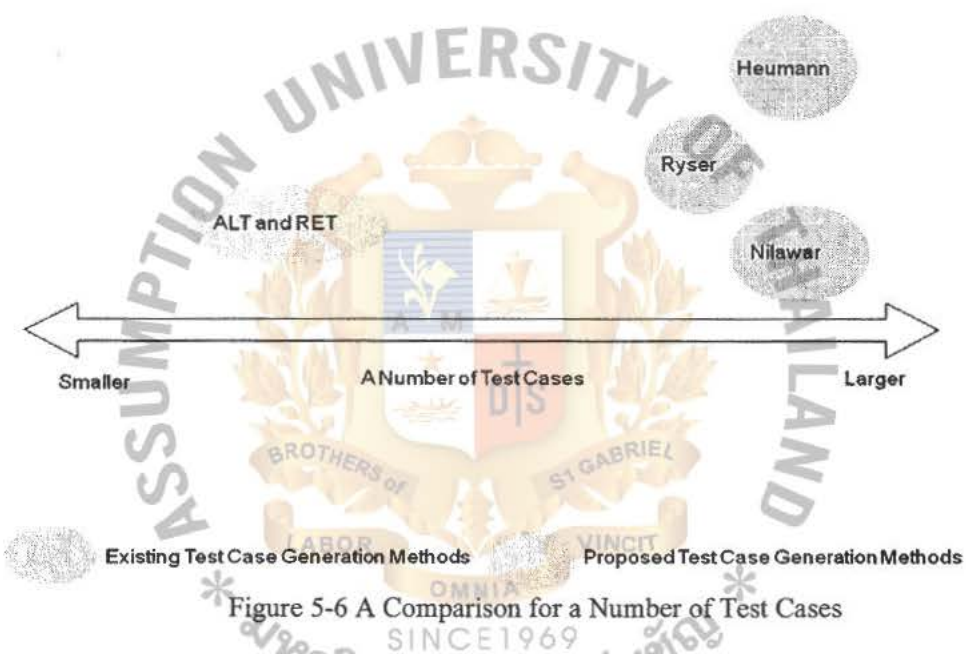


Figure 5-6 A Comparison for a Number of Test Cases

Figure 5-6 compares four test case generation techniques in terms of numbers of test cases. The horizon axis represents a number of test cases. The proposed methods are by far better than the other three methods. Generally, test case generation methods with the smallest number of test cases are desirable.

The following represents a comparison between a number of test cases and coverage of high priority requirements. The horizon axis presents a number of test cases while the vertical gives a percentage of requirement coverage.

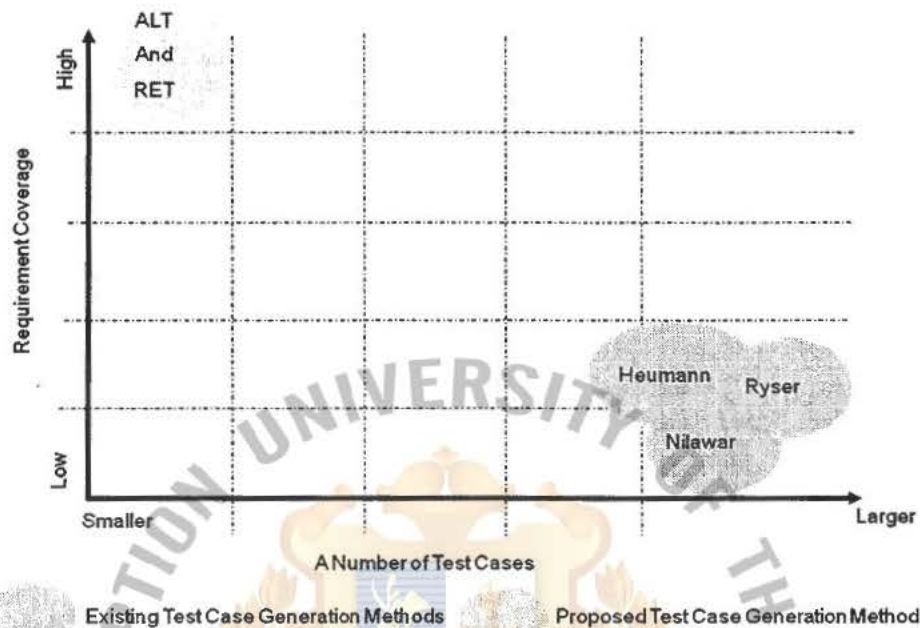


Figure 5-7 A Comparison for a Number of Test Cases and Coverage

Figure 5-7 shows that our proposed methods generate and minimize a number of test cases while preserving high capability to cover high priority requirements. Also, it shows that our methods are by far better than other existing test case generation methods in terms of number of tests and coverage.

CHAPTER 6

CONCLUSION

This chapter provides three sections. The first section concludes major contribution of this dissertation. The second section discusses the most suitable test case generation to tackle with research problems. The last section describes future researches.

6.1 Major Contributions

In the conclusion, software testing phase has been proven that it is one of the most critical phases in software development life cycle. Typically, it takes around 40-50% of effort and cost of developing software [21]. Many researchers have investigated to reduce time, effort and cost of testing activities. The literature reviews reveal that test case generation is one of the most important phases in software testing phase. Therefore, this proposal concentrates on test case generation techniques.

The outstanding research problems remaining from test case generation methods based on UML use case diagram, which motivated this study, are: lack of requirement prioritization before test case generation, unable to identify which test cases can be removed during a test case generation process and large number of test cases due to large number of alternative paths in each use case.

In order to resolve the above problems, this dissertation proposes an effective test case generation method, derived from UML Use Case diagram, along with marketing-driven requirement prioritization for black-box testing [89][115]. This dissertation introduces WOW factors based on user satisfaction to support the requirement classification and prioritization [100][114]. This is because critical requirements must have higher priority. Typically, a general requirement

prioritization uses only return on investment (ROI) to prioritize requirements. Unfortunately, our study reveals that ROI is not the only factor for a requirement prioritization. The research presents that a complexity of requirement is one of the critical factors to give a priority. This thesis introduces a relationship between ROI and the requirement complexity [70]. The high ROI requirements with less complexity are desirable.

In addition, this dissertation improves a sketch diagram-based test case generation method by minimizing a number of test cases during the process. The proposed method aims to reduce a number of test cases derived from UML use case diagram by considering alternate paths of use cases. It introduces alternate path points for removing unnecessary test cases. Unfortunately, the research shows that the remaining problem of considering only paths is that a number of test cases are still large due to overwhelm alternate paths.

Eventually, the dissertation proposes a retain score in order to enhance ability to remove test cases. It introduces a probability of failure, as a part of risk value, represented as a complexity of test case. The complexity factor is a total number of test steps in each test case. The high complex test cases generally have a lot of test steps. Consequently, these high complex test cases can lead to high probability to failure.

Furthermore, the research conducts an evaluation experiment with a random requirements and fully described use cases. The evaluation result reveals that the proposed method is the most recommended test case generation methods for maximizing critical requirement coverage. Also, the result presents that the proposed method is one of the most recommended test case generation methods to minimize a number of test cases and cover critical requirements based on user satisfaction.

6.2 Discussion: The Most Suitable Approach

This section provides a discussion on which test case generation method is best suited to the following problems: (a) lack of requirement prioritization during test case generation process that can lead to low rate of critical requirement coverage and poor user satisfaction (b) unable to systematically determine, which test cases for black-box testing should be removed, that can lead to a large number of test cases and (c) still large number of test cases due to large number of alternative paths described in use cases, that is resulted in consuming a greater amount of time and cost.

In this dissertation, we consider the following:

6.2.1 Requirement Prioritization based on Customer Satisfaction

This dissertation proposes a marketing-driven requirement prioritization technique with WOW factors to classify requirement. This is likely to classify requirements based on user satisfaction, along with the implementation cost. Additionally, the research introduces a correlation between ROI and requirement complexity to effectively prioritize requirements. In fact, this dissertation inserts the requirement prioritization process prior to generate test case. We conduct the experiment to determine which test case generation is best suited to resolve the above research problems. Our evaluation result shows that the proposed technique can increase the ability of requirement coverage based on user satisfaction during test case generation activities.

The following lists advantages and disadvantages when using the requirement prioritization technique prior to test case generation activities.

Advantages:

1. Increase requirement coverage based on user satisfaction.

2. Provide an easy method in prioritizing large number of requirements based on user satisfaction.
3. Indirectly reduce a number of test cases to be generated for low priority requirements that are not relatively to the satisfaction factor.
4. Raise high priority for requirements with high ROI and less complexity.

Disadvantages:

1. Requires several involvements with customers to identify and classify requirements.
2. Requires large number of requirements in order to classify and prioritize.
3. Difficult to systematically identify which requirements can extraordinary increase the user satisfaction.
4. Difficult to systematically determine a complexity of requirements.

6.2.2 Test Case Generation Technique

This dissertation proposes to reduce a number of test cases during a test case generation technique, which derive tests from fully dressed use cases. There are two major proposes: (a) alternative path point formula and (b) a retain score. They are important and be part of our proposed methods to remove test cases. This is due to that use cases have overwhelm alternative paths and it can eventually lead to large number of test cases. None of existing sketch diagram based generation techniques, derived from UML use case diagram, removes test cases and concentrates on alternative paths. In case that there are a large number of alternative paths that can be optimized for generating test cases, this dissertation suggests our proposed method.

The following lists advantages and disadvantages when reducing test cases during test case generation activities.

Advantages:

1. Reduce a number of test cases during test case generation process.
2. Reserve test cases with less complexity and a few steps in the test case.
3. Able to systematically determine which test cases should be removed during test case generation process.

Disadvantages:

1. Not applicable for use cases that have only basic paths.
2. Requires fully dressed use cases only.
3. Manually generate test data and expected result.

6.3 Future Research

The problems that occur when using the above approach need future investigation. In brief, they are:

1. Difficult to systematically determine and classify requirements

Recall that one of the weaknesses of the proposed method is to difficult to systematically determine and classify requirements based on user satisfaction prior test case generation process. One of the interesting areas for future research is finding a systematic approach to identify user satisfaction and relative to a requirement engineering field.

2. Manual generate test data and expected result

The proposed technique manually generates test data and expected result that cannot reduce time and cost as much as expected. The future research should concentrate on incorporating other diagrams or techniques to automatically generate both of test data and expected result.

REFERENCES

- [1] A. Blum and M. Furst. "Fast Planning Through Planning Graph Analysis." *Artificial Intelligence* 90 (1997): 281-300.
- [2] A. Gargantini and C. Heitmeyer. "Using model checking to generate tests from requirements specifications." *Software Engineering Notes* 24, no. 6 (November 1999): 146-162.
- [3] A. Jefferson Offutt, Yiwei Xiong and Shaoying Liu. "Criteria for Generating Specification-based Tests." *Proceedings of the 5th IEEE International Conference on Engineering of Complex Computer Systems, 1999 (ICECCS '99)*. Las Vegas, NV, USA: IEEE Computer Society, 1999. 119-129.
- [4] A.Z. Javed, P.A. Strooper and G.N. Watson. "Automated Generation of Test Cases Using Model-Driven Architecture." *Second International Workshop on Automation of Software Test (AST'07)*. Minneapolis, USA: IEEE Computer Society, 2007. 3.
- [5] Ahl, V. *An Experimental Comparison of Five Prioritization Methods*. Master's Thesis, Ronneby, Sweden: Blekinge Institute of Technology, 2005.
- [6] Alberto Avritzer and Elaine J. Weyuker. "The automatic generation of load test suites and the assessment of the resulting software." *IEEE Transactions on Software Engineering* 21, no. 9 (1995): 705-716.
- [7] Alessandra Cavarra, Charles Crichton, Jim Davies, Alan Hartman, Thierry Jeron and Laurent Mounier. "Using UML for Automatic Test Generation." *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000)*. Oxford University Computing Laboratory, 2000.
- [8] Al-Kilidar, H., Cox, K. and Kitchenham, B. "The use and usefulness of the ISO/IEC 9126 quality standard." *Proceedings of the International Symposium*

- on Empirical Software Engineering. Noosa Heads, Australia: IEEE Computer Society, 2005. 7.
- [9] Amaral. A.S.M.S. Test case generation of systems specified in Statecharts. M.S. thesis , Laboratory of Computing and Applied Mathematics, INPE, Brazil: Laboratory of Computing and Applied Mathematics, 2006.
 - [10] Andrea Hermann and Barbara Paech. "Practical Challenges of Requirements Prioritization Based on Risk Estimation." Journal of Empirical Software Engineering (Kluwer Academic Publishers) 14, no. 6 (2009): 644 - 684.
 - [11] Annelises A. Andrews, Jeff Offutt and Roger T. Alexander. "Testing Web Applications." Software and Systems Modeling 4, no. 1 (2005): 326--345.
 - [12] Atif M. Memon, Martha E. Pollack and Mary Lou Soffa. "Hierarchical GUI Test Case Generation Using Automated Planning." IEEE Transactions on Software Engineer (IEEE Press) 27, no. 2 (February 2001): 144-155.
 - [13] Atif M. Memon, Martha E. Pollack and Mary Lou Soffa. "Using a Goal-driven Approach to Generate Test Cases for GUIs." Proceedings of the 21st international conference on Software engineering (ICSE'1999). Los Angeles, CA, USA: ACM, 1999. 257-266.
 - [14] Avik Sinha. Domain Specific Test Case Generation Using Higher Ordered Typed Languages for Specification. Ph. D. Dissertation, College Park, MD, USA : University of Maryland , 2005.
 - [15] Aynur Abdurazik and Jeff Offutt. "Generating Test Cases from UML Specifications." Proceedings of the 2nd International Conference on the Unified Modeling Language (UML'99). Fort Collins, CO, USA, 1999.

- [16] AZUMA, Motoei. Applying ISO/IEC 9126-1 Quality Model to Quality Requirements Engineering on Critical Software. Waseda University, USA: Waseda University, 2004.
- [17] B.M. Subraya and S.V. Subrahmanya. "Object driven performance testing in Web applications." Proceedings of the First Asia-Pacific Conference on Quality Software (APAQs'00). Hong Kong: IEEE Computer Society, 2000. 17-26.
- [18] Barry Boehm and Victor R. Basili. "Software Defect Reduction Top 10 List." Computer (IEEE Computer Society Press) 34, no. 1 (January 2001): 135-137.
- [19] Beck, K. and Andres, C. Extreme Programming Explained: Embrace Change. Boston, MA: Addison-Wesley, 2004.
- [20] Bee Bee Chua and Laurel Evelyn Dyson. Applying the ISO 9126 model to the evaluation of an e-learning system. Sydney, Australia: University of Technology, 2004.
- [21] Beizer, B. Software Testing Techniques. New York, USA: Van Nostrand Reinhold, Inc, 1990.
- [22] Bentley, John E. Software Testing Fundamentals – Concepts, Roles and Terminology. USA: SAS Institute, TechRepublic, 2005.
- [23] Bertolino, A. "Software Testing Research and Practice." Proceedings of the 10th International Workshop on Abstract State Machines (ASM'03). Taormina, Italy: Springer-Verlag Berlin, Heidelberg, 2003. 1-21.
- [24] Bill Hasling, Helmut Goetz and Klaus Beetz. "Model Based Testing of System Requirements using UML Use Case Models." Proceedings of International Conference on Software Testing Verification and Validation (ICST'08). Lillehammer, Norway: IEEE Computer Society, 2008. 367-376 .

- [25] Boehm, B. and Ross, R. "Theory-W Software Project Management: Principles and Examples." IEEE Transactions on Software Engineering 15, no. 4 (July 1989): 902-916.
- [26] Boehm, B. "Industrial Metrics Top 10 List." IEEE Softwar, 1987: 84-85.
- [27] Bogdan Korel and Ali M. Al-Yami. "Automated Regression Test Generation." Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'98). Clearwater Beach, FL, United States: ACM, 1998. 143-152.
- [28] Brandt, D. Randall. "How service marketers can identify value-enhancing service elements." Journal of Services Marketing 2, no. 3 (1988): 35-41.
- [29] C. Ramamoorthy, S. Ho, and W. Chen. "On the automated generation of program test data." IEEE Transactions on Software Engineering 2, no. 4 (1976): 293-300.
- [30] C.-H. Liu, D. C. Kung, P. Hsia, and C.-T. Hsu. "An object based data flow testing approach for web applications." International Journal of Software Engineering and Knowledge Engineering 11, no. 2 (April 2001): 157-179.
- [31] C.H. Liu, D.C. Kung, P. Hsia and C.T. Hsu. "Structural testing of Web applications." Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'00). San Jose, USA: IEEE Computer Society, 2000. 84-96.
- [32] Cadotte, Ernest R. and Turgeon, Normand. "Dissatisfiers and Satisfiers: Suggestions from Consumer Complaints and Compliments." Journal of Consumer Satisfaction, Dissatisfactions and Complaining Behavior 1 (1988): 74-79.

- [33] Carl Adam Petri and Wolfgang Reisig. Petri net. USA: Scholarpedia Publisher, 2008.
- [34] Cem Kaner, J.D., Ph.D. "What Is a Good Test Case?" Proceeding of Software Testing Analysis & Review. Florida, USA: STAR East, 2003.
- [35] Cem Kaner, James Bach and Bret Pettichord. Lessons Learned in Software Testing: A Context-Driven Approach. New York: Wiley, 2002.
- [36] Cem, Kaner. An Introduction to Scenario Testing. Florida, USA: Florida Tech, 2003.
- [37] Chaffee, Alex. "What is a web application (or "webapp")?" 2008.
- [38] Chang-Jia Wang and Ming T. Liu. "Axiomatic Test Sequence Generation for Extended Finite State Machines." Yokohama , Japan Proceedings of the 12th International Conference on Distributed Computing Systems, 1992. Yokohama , Japan: IEEE Computer Society, 1992. 252 - 259.
- [39] Chien-Hung Liu, David C. Kung, Pei Hsia and Chih-Tung Hsu. "Object-Based Data Flow Testing of Web Applications." Proceedings of the First Asia-Pacific Conference on Quality Software (APAQS'00). Hong Kong: IEEE Computer Society, 2000. 7-16.
- [40] Cockburn, Alistair. Writing Effective Use Cases. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc, 2001.
- [41] D.M. Cohen, and M.L. Fredman. "New Techniques for Designing Qualitatively Independent Systems." Journal of Combinational Designs 6, no. 6 (1998): 411-416.
- [42] D.M. Cohen, S.R. Dalal, and M.L. Fredman. "The AETG System: An Approach to Testing Based on Combinatorial Design." IEEE Trans on Software Engineering 23, no. 7 (1997): 437-444.

- [43] David C. Kung, Chien-Hung Liu and Pei Hsia. "An Object-Oriented Web Test Model for Testing Web Applications." Proceedings of the First Asia-Pacific Conference on Quality Software (APAQs'00). Los Alamitos, CA, USA: IEEE Computer Society, 2000. 111.
- [44] David Turner, Moonju Park, Jaehwan Kim and Jinseok Chae. "An Automated Test Code Generation Method for Web Applications using Activity Oriented Approach." Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. L'Aquila, Italy: IEEE Computer Society, 2008. 411-414.
- [45] Davis, A. Just Enough Requirements Management: Where Software Development Meets Marketing. New York: Dorset House, 2005.
- [46] Davis, A. "The Art of Requirements Triage." IEEE Computer 36, no. 3 (March 2003): 42-49.
- [47] Deursen, P. Klint and J. Visser. "Domain-specific Languages: An annotated bibliography." ACM SIGPLAN Notices 35, no. 6 (2000): 26-36.
- [48] E. Heatt, R. Mee, and G. Foster. "Testing the web application engineering internet." IEEE Software 19, no. 2 (April 60-65): 2002.
- [49] E. Weyuker, T. Goradia, and A. Singh. "Automatically generating test data from a boolean specification." IEEE Transactions on Software Engineering 20, no. 5 (1994): 353-363.
- [50] Edvardsson, Jon. "A Survey on Automatic Test Data Generation." Proceedings of the 2nd Conference on Computer Science and Engineering. Linköping, Sweden, 1999. 21-28.

- [51] Elizabeth M. Rudnick and Janak H. Patel. "Efficient Techniques for Dynamic Test Sequence Compaction." IEEE Transactions on Computers, Marh 1999: 323 - 330 .
- [52] Érika Regina Campos de Almeida, Bruno Teixeira de Abreu and Regina Moraes. "An Alternative Approach to Test Effort Estimation Based on Use Cases." Proceedings of the 2009 International Conference on Software Testing Verification and Validation (ICST'09). Denver, CO, USA: IEEE Computer Society, 2009. 279-288.
- [53] Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry and Yves Le Traon. "Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool." Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE'06). Raleigh, NC, USA: IEEE Computer Society, 2006. 85-94.
- [54] Firesmith, Donald. "Prioritizing Requirements," Journal of Object Technology 3, no. 8 (2004).
- [55] Flippo Ricca and Paolo Tonella. "Analysis and Testing of Web Applications." Proceedings of the 23rd International Conference on Software Engineering (ICSE'01). Toronto, Ontario, Canada: IEEE Computer Society, 2001. 25-34.
- [56] Flippo Ricca and Paolo Tonella. "Web Testing: a Roadmap for the Empirical Research." Proceedings of the 2005 Seventh IEEE International Symposium on Web Site Evolution (WSE'05). Budapest, Hungary: IEEE Computer Society, 2005. 63-70.
- [57] Frederick Herzberg, Bernard Mausner and Barbara Bloch Snyderman. The motivation to work. 2nd edition. New York: Wiley, 1959.

- [58] George, Mealy H. "A Method for Synthesizing Sequential Circuits." Bell Systems Technical Journal 34 (September 1955): 1045-1079.
- [59] Geraci, Anne. Compilation of IEEE standard computer glossaries. NJ, USA: IEEE Press Piscataway, 1991.
- [60] Grady Booch, Ivar Jacobson and Jim Rumbaugh. OMG Unified Modeling Language Specification. USA: OMG Organization, 2000.
- [61] Gregor V. Bochmann and Jan Gecsei. "A unified method for the specification and verification of protocols." Proceedings of the IFIP Congress'77. Toronto, Canada: North-Holland Publishing Company, 1977. 229-234.
- [62] Harel, D. "On visual formalisms." Communications of the ACM 31, no. 5 (1988): 514-530.
- [63] Harel, D. "Statecharts: A visual formalism for complex system." Journal of ACM (Elsevier North-Holland, Inc. Amsterdam, The Netherlands), June 1987: 231-274.*
- [64] Harrold, M. J. "Testing: A Roadmap." Proceedings of the International Conference on Software Engineering. Limerick, Ireland: ACM, 2000. 61-72.
- [65] Hasan Ural and Keqin Zhu. "Optimal Length Test Sequence Generation Using Distinguishing Sequences." IEEE Transaction on Networking, 1993: 358 - 371.
- [66] Hassan Reza, Kirk Ogaard and Amarnath Malge. "A Model Based Testing Technique to Test Web Applications Using Statecharts." Proceedings of the Fifth International Conference on Information Technology. Las Vegas, NV, USA: IEEE Computer Society, 2008. 183-188.

- [67] Hayes, Jane Huffman. Input Validation Testing: A Requirements-Driven, System Level, Early Lifecycle Technique. Ph.D. Thesis, George Mason University, Fairfax, VA, USA: George Mason University, 1999.
- [68] Hetzel, William C. The Complete Guide to Software Testing. MA, USA: QED Information Sciences, Inc. Wellesley, MA, USA , 1988.
- [69] Heumann, Jim. Generating Test Cases From Use Cases. IBM Rational Software, 2001.
- [70] Holly Parsons-Hann and Kecheng Liu. "Measuring Requirements Complexity to Increase the Probability of Project Success." Proceedings of International Conference on Enterprise Information Systems (ICEIS'05). Miami, USA, 2005.
- [71] Ho-Won Jung, Seung-Gweon Kim and Chang-Shin Chung. "Measuring Software Product Quality: A Survey of ISO/IEC 9126." Journal of IEEE Software (IEEE Computer Society) 21, no. 5 (September 2004): 88-92.
- [72] Huaikou Miao, Shengbo Chen, Huanzhou Liu and Zhongsheng Qian. "An Approach to Generating Test Cases for Testing Component-based Web Applications." Proceedings of the Workshop on Intelligent Information Technology Application (IITA'07). Zhang Jiajie, China: IEEE Computer Society, 2007. 264-269.
- [73] Huaikou Miao, Zhongsheng Qian and Bo Song. "Towards Automatically Generating Test Paths for Web Application Testing." Proceedings of the 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE'08). Nanjing, China: IEEE Computer Society, 2008. 211-218.

- [74] Hui Liu and Hee Beng Kuan Tan. "Automated Verification and Test Case Generation for Input Validation." Proceedings of the 2006 international workshop on Automation of software test, International Conference on Software Engineering. Shanghai, China: ACM, 2006. 29-35.
- [75] Hyungchoul Kim, Sungwon Kang, Jongmoon Baik and Inyoung Ko. "Test Cases Generation from UML Activity Diagrams." Proceedings of the Eighth ACIS International Conference on Software Engineering , Artificial Intelligence, Networking, and Parallel/Distributed Computing. Qingdao, China: Haier International Training Center, Qingdao, China , 2007. 556-561.
- [76] Ibrahim K. El-Far and James A. Whittaker. Model-based Software Testing. Publication Report, Encyclopedia of Software Engineering, Wiley, 2001.
- [77] Irith Pomeranz, Praveen K. Parvathala and Srinivas Patil. "Estimating the Fault Coverage of Functional Test Sequences Without Fault Simulation." Proceedings of the 16th IEEE Asian Test Symposium. San Francisco, California, USA: IEEE Computer Society, 2007. 25-32.
- [78] J. Dick and A. Faivre. "Automating the Generation and Sequencing of Test Cases from Model based specification." Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods . Odense, Germany: Springer-Verlag London, UK , 1993. 268-284.
- [79] Jalote, Pankaj. "An Integrated Approach to Software Engineer." 2005.
- [80] Jane Huffman Hayes and A. Jefferson Offutt. "Increased Software Reliability through Input Validation Analysis and Testing." Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE '99). Boca Raton, FL, USA : IEEE Computer Society, 1999. 199.

- [81] Jeff Offutt, Shaoying Liu, Aynur Abdurazik and Paul Ammann. "Generating Test Data from State-based Specifications." *The Journal of Software Testing, Verification and Reliability* 13, no. 1 (March 2003): 25-53.
- [82] Jeff Offutt, Ye Wu, Xiaochen Du and Hong Huang. "Bypass Testing of Web Applications." *Proceedings of 15th International Symposium on Software Reliability Engineering*, 2004. ISSRE 2004. Fairfax, VA, USA: IEEE Computer Society, 2004. 187-197.
- [83] Jeff Tian, Li Ma, Zhao Li and A. Gunes Koru. "A Hierarchical Strategy for Testing Web-Based Applications and Ensuring Their Reliability." *Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC'03)*. Dallas, Texas, USA: IEEE Computer Society, 2003. 702.
- [84] Ji-Tzay Yang, Jiun-Long Huang, Feng-Jian Wang and William C. Chu. "Constructing Control-Flow-Based Testing Tools for Web Application." *Proceedings of the 11th Software Engineering and Knowledge Engineering Conference (SEKE'99)*. Kaiserslautern, Germany, 1999.
- [85] Johannes Ryser and Martin Glinz. SCENT: A Method Employing Scenarios to Systematically Derive Test Cases for System Test. Technical Report: ifi-2000.03 , Zurich, Switzerland: University of Zurich , 2000.
- [86] Kancherla, Mani Prasad. Generating Test Templates via Automated Theorem Proving. Technical Report: NASA CR-207042, NASA, Washington, DC, USA: NASA Ames Research Center, 1997.
- [87] Kaner, Cem. "A Course in Black Box Software Testing." 2004.

- [88] Kaner, Cem. "Software Engineering Metrics: What Do They Measure and How Do We Know?" Proceedings of the 10th International Software Metrics Symposium. Chicago, IL, USA: IEEE Press, 2004.
- [89] Kano Noriaki, Nobuhiku Seraku, Fumio Takahashi and Shinichi Tsuji. "Attractive Quality and Must-Be Quality." Journal of the Japanese Society for Quality Control 14, no. 2 (1984): 39-48.
- [90] Karlsson, J. "A Cost-Value Approach for Prioritizing Requirements." Journal of IEEE Software, 1997: 67-74.
- [91] Karlsson, J. and Ryan, K. "A Cost-Value Approach for Prioritizing Requirements." IEEE Software (IEEE Computer Society) 14, no. 5 (September/October 1997): 67 - 75.
- [92] Karlsson, J. "Software Requirements Prioritizing." Proceedings of the Second International Conference on Requirements Engineering (ICRE'96). Colorado : IEEE Computer Society, 1996. 110-116.
- [93] Konda, Kalyana Rao. Measuring Defect Removal Accurately. USA: Software Testing PRo, 2005.
- [94] Korel, Bogdan. "Automated Software Test Data Generation." IEEE Transaction on Software Engineering (IEEE Press) 16, no. 8 (August 1990): 870-879.
- [95] L. Brim, I. Cerna, P. Varekova, and B. Zimmerova. "Component-interaction automata as a verification oriented component-based system specification." ACM SIGSOFT Software Engineering Notes (ACM) 31, no. 2 (March 2006): 31-38.
- [96] Leffingwell, D. and Widrig, D. Managing Software Requirements: A Use Case Approach. Boston, MA: Addison-Wesley, 2003.

- [97] Lei Xu and Baowen Xu. "Applying Agent into Intelligent Web Application Testing." Proceedings of the International Conference on Cyberworlds. Hannover, Germany: IEEE Computer Society, 2007. 61-65 .
- [98] Lei Xu, Baowen Xu and Jixiang Jiang. "Testing Web Applications Focusing on Their Specialties." ACM SIGSOFT Software Engineering (ACM) 30, no. 1 (January 2005): 10.
- [99] Lena Karlsson, Asa G. Dahlstedt, Johan Natt och Dag, Bjorn Regnell and Anne Persson. "Challenges in Market-Driven Requirements Engineering - an Industrial Interview Study." Proceedings of Eighth International Workshop on Requirements Engineering: Foundation for Software Quality. Essen, Germany, September 2002. 37-49.
- [100] M Tokman, LM Davis and KN Lemon. "The WOW factor: Creating value through win-back offers to reacquire lost customers." Journal of Retailing 83, no. 1 (2007): 47-64.
- [101] M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. "Model-Based Testing with AsmL.NET." Proceedings of the 1st European Conference on Model-Driven Software Engineering. Nuremberg, Germany: Microsoft Press, 2003. 11-12.
- [102] M. Blackburn and R. Busser. "T-VEC: A tool for developing critical systems." Proceedings of the 1996 Annual Conference on Computer Assurance (COMPASS'96). Gaithersburg, MD: IEEE Computer Society Press, 1996. 237-249.
- [103] M. Prasanna S.N. Sivanandam R.Venkatesan R.Sundarrajan. "A Survey on Automatic Test Case Generation." Academic Open Internet Journal 15 (2005).

- [104] Mahnaz Shams, Diwakar Krishnamurthy and Behrouz Far. "A Model-Based Approach for Testing the Performance of Web Applications." Proceedings of the Third International Workshop on Software Quality Assurance (SOQUA'06). Portland, Oregon, USA: ACM, 2006. 54-61.
- [105] Manish Nilawar and Dr. Sergiu Dascalu. A UML-Based Approach for Testing Web Applications. Master Thesis, Master of Science with major in Computer Science, University of Nevada, Reno, Nevada, USA: University of Nevada, 2003.
- [106] Marick, Brian. The Craft of Software Testing: Subsystem Testing Including Object-Based and Object-Oriented Testing. USA: Prentice Hall, 1995.
- [107] Mats Grindal, Jeff Offutt and Sten F. Andler. "Combination Testing Strategies: A Survey." The Journal of Software Testing, Verification and Reliability 15 (2005): 167-199.
- [108] Mats P.E. Heimdahl, Sanjai Rayadurgam, Willem Visser, Devaraj George and Jimin Gao. Auto-generating Test Sequences using Model Checkers: A Case Study. NASA Ames Research Center, USA: NASA Ames Research Center, 2003.
- [109] Maya Daneva and Andea Hermann. "Requirements Prioritization Based on Benefit and Cost Prediction: A Method Classification Framework." Proceedings of the 2008 34th Euromicro Conference Software Engineering and Advanced Applications. Parma, Italy: IEEE Computer Society, 2008. 240-247.
- [110] McConnell, Steve. Code Complete. California, USA: Microsoft Press, 2004.
- [111] McMinn, Phil. "Search-based Software Test Data Generation: A Survey." Software Testing, Verification & Reliability 14, no. 2 (June 2004): 105-156.

- [112] Mead, Nancy R. Requirements Prioritization Introduction. Software Engineering Institute, Carnegie Mellon University, USA: Carnegie Mellon University, 2008.
- [113] Miao Huaikou and Liu Ling. "A Test Class Framework for Generating Test Cases from Z Specifications." Proceedings of the 6th IEEE International Conference on Complex Computer Systems (ICECCS'00). Tokyo, Japan: IEEE Computer Society, 2000. 164.
- [114] Millard, N. "Learning from the 'wow' factor -- how to engage customers through the design of effective affective customer experiences." Journal of BT Technology 24, no. 1 (2006): 11-16.
- [115] ML Meuter, AL Ostrom, RI Roundtree and MJ Bitner. "Self-service technologies: understanding customer satisfaction with technology-based service encounters." Journal of Marketing 64 (2000): 50-64.
- [116] Mohammed Benattou, Jean-Michel Bruel and Nabil Hameurlain. "Generating Test Data from OCL Specification." 2002.
- [117] Moisiadis, F. "A Requirements Prioritisation Tool." Proceedings of the 6th Australian Workshop on Requirements Engineering (AWRE'01). Sydney, Australia, 2001.
- [118] Moisiadis, F. "Prioritising Scenario Evolution." Proceedings of the International Conference on Requirements Engineering (ICRE'00). Schaumburg, IL, USA : IEEE Computer Society, 2000. 85-94.
- [119] Monalisa Sarma and Rajib Mall. "Automatic Test Case Generation from UML Models." Proceedings of the 10th International Conference on Information Technology. Rourkela, India: IEEE Computer Society, 2007. 196-201.

- [120] Myers, Glenford J. The art of software testing. New York, USA: Wiley, 1979.
- [121] N. Kobayashi, T. Tsuchiya, and T. Kikuno. "A New Method for Constructing Pair-wise Covering Designs for Software Testing." *Information Processing Letters* (Elsevier North-Holland, Inc. Amsterdam, The Netherlands, The Netherlands) 81, no. 2 (January 2002): 85-91.
- [122] Neelam Gupta, Aditya P. Mathur and Mary Lou Soffa. "Automated Test Data Generation Using An Iterative Relaxation Method." *ACM SIGSOFT Software Engineering Notes (ACM)* 23, no. 6 (November 1998): 231-244.
- [123] Nicha Kosindrdecha and Jirapun Daengdej. "A Test Generation Method Based on State Diagram." *Journal of Theoretical and Applied Information Technology*, 2010.
- [124] Nicha Kosindrdecha and Jirapun Daengdej. "Test Case Generation Technique and Process." *Proceedings of First International Workshop on Evolution Support for Model-Based Development and Testing (EMDT2010)*. Ilmenau, German, 2010.
- [125] Nicha Kosindrdecha, Siripong Roongruangsuwan and Jirapun Daengdej. "Reducing Test Cases Created by Path Oriented Test Case Generation." *Proceedings of the AIAA Conference and Exhibition (AIAA'07)*. Rohnert Park, California, USA: American Institute of Aeronautics and Astronautics, Inc., 2007.
- [126] Nigel Tracey, John Clark, and Keith Mander. "Automated program flaw finding using simulated annealing." *ACM SIGSOFT Software Engineering Notes (ACM)* 23 (1998): 73-81.
- [127] NIST. The economic impacts of inadequate infrastructure for software testing. USA: National Institute of Standards and Technology, 2002.

- [128] Nyman, Matias. "Software Component Quality." 2004.
- [129] P. Botella, X. Burgués, J.P. Carvallo, X. Franch, G. Grau, J. Marco and C. Quer. "ISO/IEC 9126 in practice: what do we need to know?" Rome, Italy, 2004.
- [130] P. E. Ammann, P. E. Black, and W. Majurski. "Using model checking to generate tests from specifications." Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98). Brisbane, Australia: IEEE Computer Society, 1998. 46–54.
- [131] P. Samuel, R. Mall and A.K. Bothra. "Automatic Test Case Generation Using Unified Modeling Language (UML) State Diagrams." Journal of IET Software 2, no. 2 (April 2008): 79 - 93.
- [132] P. Stocks and David Carrington. "A Framework for Specification-Based Testing." IEEE Transaction on Software Engineering (IEEE Press Piscataway, NJ, USA) 22, no. 11 (November 1996): 777-793.
- [133] Paga, F. G. Formal Specification of Programming Languages: A Panoramic Primer. Australia: Rentice-Hall, Inc., 1981.
- [134] Pan, Jiantao. Software Testing (18-849b Dependable Embedded Systems). Electrical and Computer Engineering Department, Carnegie Mellon University, USA: Carnegie Mellon University, 1999.
- [135] Park, J.; Port, D.; and Boehm B. "Supporting Distributed Collaborative Prioritization for Win-Win Requirements Capture and Negotiation." Proceedings of the International Third World Multi-conference on Systemics, Cybernetics and Informatics (SCT'99). Orlando, FL: International Institute of Informatics and Systemic (IIIS), 1999. 578-584.

- [136] Percy Antonio, Pari Salas and Bernhard K. Aichernig. Automatic Test Case Generation for OCL: a Mutation Approach. Technical Report UNU-IIST Report No. 321, Tokyo, Japan: United Nations University, 2005.
- [137] Peter Frohlich and Johannes Link. "Automated Test Case Generation from Dynamic Models." Proceedings of the 14th European Conference on Object-Oriented Programming. Nottingham, UK: Springer-Verlag London, UK, 2000. 472 - 492.
- [138] Philip Samuel and Anju Teresa Joseph. "Test Sequence Generation from UML Sequence Diagrams." Proceedings of the Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing. Phuket, Thailand: IEEE Computer Society, 2008. 879 - 887 .
- [139] Praveen Ranjan Srivastva, Krishan Kumar and G Raghurama. "Test Case Prioritization Based on Requirements and Risk Factors." Journal of ACM SIGSOFT Software Engineer Notes (ACM) 34, no. 4 (2008): 1-5.
- [140] Preeyavis Pringsulaka and Jirapun Daengdej. "Coverall Algorithm for Test Case Reduction." Aurospace Conference. Big Sky, MT, USA: IEEE Computer Society, 2006. 8.
- [141] Q.Nguyen, Hung. Testing Application on the Web: Test Planning for Internet-Based Systems. USA: John Wiley & Sons, 2003.
- [142] R. A. DeMillo and E. H. Spafford. "The Mothra software testing environment." Proceedings of the 11th Nasa Software Engineering Laboratory Workshop. Dayton, OH : Goddard Space Center, 1989. 1555 - 1561.

- [143] R.E. Fikes and N.J. Nilsson. "STRIPS: a new approach to the application of theorem proving to problem solving." *Artificial Intelligence 2 (ACM) 2* (1971): 189-208.
- [144] Rajib. "Software Test Metric." QCON. USA: QCON Corporation Training Services, 2006.
- [145] Ramesh, B. and Jarke, M. "Toward Reference Models for Requirements Traceability." *IEEE Transactions on Software Engineering* 27, no. 1 (January 2001): 58 - 93.
- [146] Rex, Black. *Managing the Testing Process* (2nd Edition). USA: Wiley Publishing, 2002.
- [147] Richard A. DeMillo and A. Jefferson Offutt. "Constraint-Based Automatic Test Data Generation." *IEEE Transaction on Software Engineering (IEEE Press)* 17, no. 9 (September 1991): 900-910.
- [148] Rob Hendriks and Robert van Vonderen. "Measuring software product quality during testing." *Proceedings of the European Software Quality Week*. San Francisco, California, USA: Software Magazine, 2000.
- [149] Robert Nilsson, Jeff Offutt and Jonas Mellin. "Test Case Generation for Mutation-based Testing of Timeliness." *Proceedings of the 2nd International Workshop on Model Based Testing (MBT'06)*. Vienna, Austria, 2006.
- [150] Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck. "Test-data generation using genetic algorithms." *Journal of Wiley Software Testing, Verification And Reliability* 9, no. 4 (1999): 263-282.
- [151] S. Adolph, A. Cockburn and P. Bramble. *Patterns for Effective Use Cases*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc, 002.

- [152] S. Gnesi, D. Latella, and M. Massink. "Formal conformance testing UML Statechart Diagrams Behaviours: From theory to automatic test generation." ACM SIGSOFT Software Engineering Notes (Consiglio Nazionale delle Ricerche, Istituto CNUCE), 2002: 144-153.
- [153] S. R. Dalal and C. L. Mallows. "Factor-covering designs for testing software." *Technometrics* 40, no. 3 (August 1998): 234-243.
- [154] S. Rayadurgam and M. P. Heimdahl. "Coverage based test case generation using model checkers." *Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2001)*. Washington, DC, USA: IEEE Computer Society, 2001. 83-91.
- [155] S.J. Cuning and J.W. Rozenblit. "Automatic Test Case Generation from Requirements Specifications for Real-time Embedded Systems." *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics, 1999 (SMC'99)*. Tokyo, Japan: IEEE Computer Society, 1999. 784-789.
- [156] S.R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, C.M. Lott, G.C. Patton and B.M. Horowitz. "Model-Based Testing in Practice." *Proceedings of the International Conference on Software Engineering (ICSE'99)*. Los Angeles, CA, USA: ACM, 1999. 285-294 .
- [157] Saaty, T. L. *The Analytic Hierarchy Process*. New York: McGraw-Hill, 1980.
- [158] Sami Beydeda and Volker Gruhn. "BINTEST – binary search-based test case generation." *Proceedings of the Computer Software and Applications Conference (COMPSAC'03)*. Dallas, TX, USA: IEEE Computer Society, 2003. 28.
- [159] Sanjai Rayadurgam and Mats P. E. Heimdahl. "Test-Sequence Generation from Formal Requirement Models." *Proceedings of the 6th IEEE International*

- Symposium on High Assurance Systems Engineering (HASE'01). Boca Raton, FL, USA : IEEE Computer Society, 2001. 23-31.
- [160] Sanjai Rayadurgam and Mats P.E. Heimdahl. "Coverage Based Test-Case Generation using Model Checkers." Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'01). IEEE Computer Society, 2001. 83-91.
 - [161] Sara Sprenkle, Emily Gibson, Sreedevi Sampath and Lori Pollock. "A Case Study of Automatically Creating Test Suites from Web Application Field Data." Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications (TAV-WEB'06). Portland, Maine, USA: ACM, 2006. 1-9.
 - [162] Sasa Misailovic, Alekasandar Milicevic, Sarfraz Khurshid and Darko Marinov. "Generating Test Inputs for Fault-Tree Analyzers using Imperative Predicates." Proceedings of the Workshop on Advances and Innovations in Systems Testing (STEP'07). Memphis, TN, USA, 2007.
 - [163] Scafeher, Hans. "Risk-based Testing." Proceedings of STAR WEST'98. USA, 1998.
 - [164] Shammi, Marjana. "How can a QA help prevent rather than cure?" 2008.
 - [165] Shengbo Chen, Huaikou Miao and Zhongsheng Qian. "Automatic Generating Test Cases for Testing Web Applications." Proceedings of the International Conference on Computational Intelligence and Security Workshops (CISW'07). Heilongjiang, China: IEEE Computer Society, 2007. 881-885.
 - [166] Silktest User's Guide, Version 6.5. Lexington, MA: Segue Software Inc., 2002.

- [167] Sokenou, Dehla. "Generating Test Sequences from UML Sequence Diagrams and State Diagrams." Proceedings of Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing. Phuket, Thailand: IEEE Computer Society, 2008. 879-887.
- [168] Sommerville, Ian. Software Engineering 6th edition Chapter 20. USA: Powell, 2000.
- [169] StClair, Bill. "Efficient Testing Ensures Requirements Traceability and Verification." 2006.
- [170] Stefania Gnesi, Diego Latella and Mieke Massink. "Formal Test-case Generation for UML Statecharts." Proceedings of the Ninth IEEE International Conference on Engineering Complex Computer Systems Navigating Complexity in the e-Engineering Age. Florence, Italy: IEEE Computer Society, 2004. 75 - 84 .
- [171] Stocks, P. Applying Formal Methods to Software Testing. PhD thesis, University of Queensland, Queensland, Australia: University of Queensland, 1993.
- [172] Suet Chun Lee and Jeff Offutt. "Generating Test Cases for XML-based Web Component Interactions Using Mutation Analysis." Proceedings of the 12th International Symposium on Software Reliability Engineering . Washington, DC, USA : IEEE Computer Society, 2001. 200.
- [173] The Standish Group. Chaos Report: Why IT Project Fail. USA: Standish Group, 1994.

- [174] Thoms J. Ostrand and Marc J. Balcer. "The Category-Partition Method for Specifying and Generating Functional Tests." *Communication of ACM* 31, no. 6 (1988): 676 - 686.
- [175] Tran, Hung. "Test Generation using Model Checking." *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR'01)*. Lisbon, Portugal: IEEE Computer Society, 2001.
- [176] U. Farooq, C.P. Lam and H. Li. "Towards Automated Test Sequence Generation." *Proceedings of the 19th Australian Conference on Software Engineering*. Perth, WA, Australia: IEEE Computer Society, 2008. 441 - 450 .
- [177] V., Karthikeyan. StickyMinds article: Traceability Matrix. USA: StickyMinds Website.
- [178] Valdivino Santiago, Ana Silvia Martins do Amaral, N.L. Vijaykumar, Maria de Fatima, Mattiello-Francisco, Eliane Martins and Odnei Cuesta Lopes. "A Practical Approach for Automated Test Case Generation using Statecharts." *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, Chicago, IL, USA: IEEE Computer Society, 2006. 183-188.
- [179] Vijaykumar, N. L.; Carvalho, S. V. and Abdurahiman, V. "On proposing Statecharts to specify performance models." *International Transactions in Operational Research*, 2002: 321-336.
- [180] von Knethen, A. "Change-Oriented Requirements Traceability. Support for Evolution of Embedded Systems." *Proceedings of the International Conference on Software Maintenance*. Montreal, Canada: IEEE Computer Society, 2002. 482-485.

- [181] W. Eric Wong, Yu Lei and Xiao Ma. "Effective Generation of Test Sequences for Structural Testing of Concurrent Programs." Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05). Shanghai, China: IEEE Computer Society, 2005. 539 - 548.
- [182] W.T. Tsai, X. Wei, Y. Chen, R. Paul and B. Xiao. "Swiss Cheese Test Case Generation for Web Services Testing." IEICE Transactions (IEICET) 88, no. D(12) (2005): 2691-2698.
- [183] Wagner, F. Modeling Software with Finite State Machines: A Practical Approach. USA: Auerbach Publications, 2006.
- [184] Wang Linzhang, Yuan Jiesong, Yu Xiaofeng, Hu Jun, Li Xuandong and Zheng Guoliang. "Generating Test Cases from UML Activity Diagram based on Gray-Box Method." Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04). Busan, Korea: IEEE Computer Society, 2004. 284 - 291.
- [185] Weyuker, E.J. "The evaluation of program-based software test data adequacy criteria." ACM, 1998: 668-675.
- [186] Whalen, M. W. A formal semantics for RSML. Master's thesis, University of Minnesota, USA: University of Minnesota, 2000.
- [187] Wiegers, K. E. Software Requirements. Redmond, WA: Microsoft Press, 2003.
- [188] Wiegers, Karl E. "First Things First: Prioritizing Requirements." Journal of Object Technology, 1999.
- [189] Wolff, Achim D. Brucker and Burkhat. "Test-Sequence Generation with HOL-TestGen With an Application to Firewall Testing." Proceedings of the

1st international conference on Tests and proofs . Zurich, Switzerland : Springer-Verlag Berlin, Heidelberg , 2007. 149-168.

- [190] Xiaoping Jia and Hongming Liu. "Rigorous and Automatic Testing of Web Applications." Proceedings in the 6th IASTED International Conference on Software Engineering and Applications (SEA'02). Cambridge, MA, USA, 2002. 280-285.
- [191] Xiaoping Jia, Hongming Liu and Lizhang Qin. "Formal Structured Specification for Web Application Testing." Proceedings of the 2003 Midwest Software Engineering Conference (MSEC'03). Chicago, IL, USA, 2003. 88-97.
- [192] Yamaura, Tsuneo. "How to Design Practical Test Cases." Journal of IEEE Software (IEEE Computer Society) 15, no. 6 (November 1998): 30-36.
- [193] Yang, J.T., Huang, J.L., Wang, F.J. and Chu, W.C. "Constructing an object-oriented architecture for Web application testing." Journal of Information Science and Engineering 18, 2002: 59-84.
- [194] Ye Wu and Jeff Offutt. Modeling and Testing Web-based Applications. Technical Report, Information and Software Engineering Department, George Mason University, Fairfax, VA, USA: George Mason University, 2002.
- [195] Ye Wu, Jeff Offutt and Xiaochen. Modeling and Testing of Dynamic Aspects of Web Applications. Technical Report ISE-TR-04-01, Information and Software Engineering Department, George Mason University, Fairfax, VA, USA: George Mason University, 2004.
- [196] Yu Qi, David Kung and Eric Wong. "An Agent-based Testing Approach for Web Applications." Proceedings of the 29th Annual International Computer

- Software and Applications Conference (COMPSAC'05). Edinburgh, Scotland: IEEE Computer Society, 2005. 45-50.
- [197] Zeilcynski, Peter. Traceability from Use Cases to Test Cases. IBM Research, 2006.
- [198] Zhenyu Liu, Ning Gu and Genxing Yang. "An Automated Test Cases Generation Approach Using Match Technique." Proceedings of the 5th International Conference on Computer and Information Technology (CIT'05). Shanghai, China: IEEE Computer Society, 2005. 922-926.
- [199] Zhu, H., Hall, P. and May, J. "Software Unit Test Coverage and Adequacy." ACM Comp. Survey 29, no. 4 (1997): 366-427.
- [200] Zimmermann, Armin. Stochastic Discrete Event Systems: Modeling, Evaluation, Applications. USA: Springer, 2007.
- [201] Richard Denney. Calculating ROI On Your Investment. USA: August, 2006.
- [202] Andrea Herrmann and Maya Daneva. Requirement Prioritization Based on Benefit and Cost Prediction: An Agenda for Future Research. Proceedins of 16th IEEE International Requirements Engineering Conference. Kaiserslautern, Germany: December, 2008.

