ADOPTING A PARTICLE SWARM-BASED TEST GENERATOR STRATEGY FOR VARIABLE-STRENGTH AND T-WAY TESTING

By

BESTOUN S. AHMED

Thesis submitted in fulfillment of the requirements for the degree

of Doctor of Philosophy

December 2011

Acknowledgement

First and foremost, I would like to thank God, the merciful and the compassionate, for having made everything possible in this work by giving me the power to believe in myself and showing me inner peace.

It is a pleasure to offer my regards and thanks to those who made this thesis possible and those who helped me during the research. This thesis could not have been written without their help and support. Firstly, I am sincerely thankful to all my family, particularly to my dearest loving mother. Thank you for everything that you have done for me; and this thesis is for you. I am also thankful to my supervisor, Assoc. Prof. Dr. Kamal Z. Zamli, for his support and for giving me the opportunity to work under him. His suggestions and constructive comments have improved my thinking, research view, and writing skills. Thanks for teaching me to view the problem from different angles. I am most grateful to Prof. Dr. Chee Peng Lim, for his valuable discussions and suggestions on the background of Particle Swarm Optimization. I also thank Xiao Qu from University of Nebraska – Lincoln for providing me the "flex" TSL file of her published papers and Wayne Motycka from the SIR administrator, University of Nebraska – Lincoln for helping on the experimental installation required for my case study.

Last, but by no means least, I would like to thank the Dean and staff of the School of EE, USM for their valuable support. In particular, I would like to thank Pn Sarina who has helped me during my candidature period in USM. May Allah bless All of You. Amen.

Table of Contents

List of Tables

List of Figures and Illustrations

List of Abbreviations and Nomenclature

Abbreviation	Meaning
ACA	Ant Colony Algorithm
ACS	Ant Colony System
ACTS	Advanced Combinatorial Testing Suite
CPU	Central Processing Unit
AETG	Automatic Efficient Test Generator
AI	Artificial Intelligence
CA	Covering Array
CEG	Cause and effect graphing
CTE-XL	Classification-Tree Editor eXtended Logics
DDA	Deterministic Density Algorithm
GA	Genetic Algorithm
GUI	Graphical User Interface
IPO	In Parameter Order
IPOG	In-Parameter-Order-General
IPOG-D	In-Parameter-Order-General Double
IPO-s	In-Parameter-Order symmetry
ITCH	Intelligent Test Case Handler
LOC	Lines of Code
mAETG	Modified Automatic Efficient Test Generator
MCA	Mixed Covering Array
MOA	Mutual Orthogonal Array
OA	Orthogonal Array
PICT	Pairwise Independent Combinatorial Testing
PSO	Particle Swarm Optimization
PSTG	Particle Swarm-based Test Generator
RAM	Random Access Memory
SA	Simulated Annealing
SQA	Software Quality Assurance
TS	Tabu Search
TConfig	Test Configuration
TCG	Test Case Generator
TVG	Test Vector Generator
VSCA	Variable-Strength Covering Array

MENGGUNA PAKAI STRATEGI PENJANA UJIAN BERDASARKAN PARTIKEL KAWANAN UNTUK KEKUATAN PEMBOLEH UBAH DAN T-HALA

Abstrak

Kebelakangan ini, penyelidik telah mula untuk meneroka penggunaan algorithma kecerdasan buatan (AI) sebagai strategi t-hala (di mana t menunjukkan kekuatan interaksi) dan pemboleh ubah. Pelbagai strategi berasaskan kecerdasan buatan telah dibangunkan, seperti Koloni Semut, Simulasi Penyepuhlindapan, Algoritma Genetik, dan Pencarian Tabu.Walaupun berguna, strategi berasaskan kecerdasan buatan sedia ada mengguna pakai proses carian yang rumit dan memerlukan pengiraan pengkomputeran yang berat.Oleh yang demikian, strategi berasaskan kecerdasan buatan sedia ada terbatas kepada kekuatan interaksi kecil (iaitu, $t \leq 3$) dan konfigurasi ujian kecil.Kajian terkini menunjukkan keperluankekuatan $t = 6$ bagi mengesan kebanyakan kesilapan. Tesis ini membentangkan reka bentuk dan pelaksanaan strategi penjana ujian interaksi baru, yang dikenali sebagai Penjana Ujian Partikel Kawanan (PSTG), untuk menghasilkan ujian t-hala dan kekuatan pemboleh ubah.Tidak seperti strategi berasaskan kecerdasan buatan sedia ada, proses carian pengkomputeran yang lebih ringan membolehkan PSTG menyokong kekuatan interaksi yang tinggi sehingga $t = 6$. Prestasi PSTG dinilai dengan menggunakan beberapa set eksperimen penanda aras. Secara perbandingan, PSTG berjaya mengatasi strategi berasaskan kecerdasan buatan dan strategi-strategi lain sedia ada secara konsisten dari segi saiz ujian.Tambahan pula, kajian kesmenunjukkankeberkesananPSTGuntuk mengesankomponen interaksi kemasukan bermasalah.

ADOPTING A PARTICLE SWARM-BASED TEST GENERATOR STRATEGY FOR VARIABLE-STRENGTH AND T-WAY TESTING

Abstract

Recently, researchers have started to explore the use of Artificial Intelligence (AI)-based algorithms as *t*-way (where *t* indicates the interaction strength) and variable-strength testing strategies. Many AI-based strategies have been developed, such as Ant Colony, Simulated Annealing, Genetic Algorithm, and Tabu Search. Although useful, most existing AI-based strategies adopt complex search processes and require heavy computations. For this reason, existing AI-based strategies have been confined to small interaction strengths (i.e., $t \leq 3$) and small test configurations. Recent studies demonstrate the need to go up to $t=6$ in order to capture most faults. This thesis presents the design and implementation of a new interaction test generation strategy, known as the Particle Swarm-based Test Generator (PSTG), for generating *t*-way and variable-strength test suites. Unlike other existing AI-based strategies, the lightweight computation of the particle swarm search process enables PSTG to support high interaction strengths of up to $t=6$. The performance of PSTG is evaluated using several sets of benchmark experiments. Comparatively, PSTG consistently outperforms its AI counterparts and other existing strategies as far as the size of the test suite is concerned. Furthermore, the case study demonstrates the usefulness of PSTG for detecting faulty interactions of the input components.

CHAPTER 1

INTRODUCTION

1.1 Overview

Nowadays, our dependencies on software are gradually increasing, as many kinds of software have become a part of our daily lives. We use software in life-saving or safety applications, for example. With this increase of its applications in the last 20 years, software has grown tremendously in terms of size (i.e. line of codes (LOCs)), and functionality. In the old days, there was hardly any commercial software with more than 15K LOCs (Jones, 1998, DeMarco, 1995). Nowadays, such a phenomenon has changed completely. It is now common to have commercial software that has more than a million LOC (Tan et al., 2006). Moreover, the current trend in the software industries is to produce large integrated software systems for applications in our life, which consist of a set of interconnected parts with individually related programs, rather than individual parts with individual programs (Williams and Probert, 2002, Cohen et al., 2008).

Unlike the old days, the development lifecycle of these software systems passes through several stages and comprehends different activities that need to be harmonized carefully

in order to meet the required user's specifications. Generally, those activities can be classified into two important activities, which are activities to construct the software product and activities to check the quality of the produced software (Baresi and Pezzè, 2006). Although the construction of the product is important, however, checking the quality, which is called the "quality process", represents the most important part of the software development lifecycle as it spans through the whole cycle (Baresi and Pezzè, 2006).

In general, not only in the software systems, the concept of quality started from old days when people tried to find a satisfactory quality in every man-made object. With the explosion of the Internet in the past couple of decades, the quality revolution has started to disseminate throughout the world rapidly. Soon after that, companies realized that the success in the new global economy requires quality products to be developed. To improve the quality, at the first time, all efforts are concentrated on improving the quality of the product at the end of production. However, the recent approach to improve the quality is to extend the efforts to the whole stages of production from the analysis of the requirements until the delivery of the product to the customer to emphasize the detection and correction of defects. In other words, every single stage in the development cycle must be completed to the highest possible standard (Naik and Tripathy, 2008).

As a complex logic product, software may suffer from different source of faults and defects from the requirement analysis and specification preparation until the delivery to

the customer. To this end, Software Quality Assurance (SQA) activities have been defined to develop high quality software products by assuring that the produced software and the procedures, tools, and techniques used during the software development and modification are adequate (Iacob and Constantinescu, 2008).

The quality process gained importance because each development stage may suffer from different errors and faults, which must be detected as early as possible in order to prevent its propagation to the whole software and reduce the cost of verification (Baresi and Pezzè, 2006). In other words, quality engineers must involve from the beginning of the software development process to assure the required quality from the users and the industrial perspective (Schulmeyer, 2008).

Among those broader activities involved in SQA, testing plays an important role to attain and assess the quality of software and to standardize and demonstrate knowledge of the quality process (Iacob and Constantinescu, 2008, Patton, 2005). This testing process helps to deduce its correct operation as a logical consequence of the design decisions and provides a realistic and practical way to analyze and understand the behaviour of the produced software-under-test (Burnstein, 2003). Moreover, it assesses how good the produced software is, before shipping to the costumer, by repeating the cycle "test-find defects–fix" during the development (Naik and Tripathy, 2008).

Therefore, the role of the tester always is to manage or mitigate the faults, risk, and failure of the system and the undesirable effects they may have on the user (Hass,

2008). The effect can be tedious when a computer game doesn't work properly for example, or it can be fatal and lead to loss of life (Patton, 2005, Xinxing and Yixiang, 2010). This in turn explains why testing consume more than 40-50% of the development costs (Carroll, 2003b, Bertolino, 2007, Carroll, 2003a, Pendharkar, 2010).

The main responsibility of the software tester is to design tests that can reveal faults and defects in the software-under-test (Burnstein, 2003, Hass, 2008). To this end, a finite number of test cases must be selected by the tester from a large input domain. Due to the time and resource constraints, the tester must select the test cases properly and smartly to ensure an effective utilization of the resources and time allocated for the job. Using all possible inputs and exercise all possible software configurations (known as exhaustive testing) may enable the tester to detect all faults and defects. However, realistically and economically this is not a feasible technique for testing. Hence, there is a need to develop effective test cases that have good possibilities to reveal the faults and defects (Burnstein, 2003).

The use of effective test cases has many positive consequences like increasing the probability of detecting faults and using the organizational resources more efficiently (Burnstein, 2003). To help achieve these positive consequences, different test case design techniques have emerged. To detect deferent types of faults, normally, more than one technique is used because when a fault is detected by a technique, some other faults may not be detected by the same technique.

1.2 Problem Statements

With the tremendously growing of software systems, more often unwanted interactions among the components are to be expected, rendering increased possibility of faults. While traditional test case design techniques are useful for fault detection and prevention, they may not be sufficient to tackle faults due to interaction, especially for multiple input software systems (Cohen et al., 2007, Williams and Probert, 2001, Schroeder and Korel, 2000). Addressing this issue, many *t*-way strategies (whereby *t* indicates the interaction strength) have been developed in the literature in the last 15 years. Indeed, *t*-way strategies help to search and generate a set of tests, which forms a complete suite that covers the required interaction strength at least once from a typically large space of possible test values. This mechanism uniformly covers *t*-interactions of the system components to generate the test cases.

Although useful, most *t*-way strategies assume uniform or fixed interaction, which means that the same interaction among all components of the system is being tested. However, the assumption that all interactions are uniform is not always true in many real applications (Afzal et al., 2009, Yilmaz et al., 2004a). For instance, the typical system may have a 100% two-way (pairwise) interactions among the components, but a subset of the components may also have a 100% three-way (or higher) interactions. Therefore, the strength of interaction might vary and be non-uniform during the testing process of the system component values. Thus, variable-strength interactions need to be

considered because sometimes they represent a more practical and flexible approach than *t*-way testing (Nie and Leung, 2011).

Viewing both cases (i.e., *t*-way and variable-strength test suite construction) as a hard computational optimization problem (Seroussi and Bshouty, 1988, Yu and Tai, 1998, Yilmaz et al., 2004a, Kuliamin and Petukhov, 2011, Afzal et al., 2009), searching for the optimal set of test cases is NP hard (Lei et al., 2008, Yu and Tai, 1998), i.e., searching for an optimum set of test cases can be a painstakingly difficult task, and it is challenging to find a unified strategy that generates optimum results all the times. In addition, an increase in the parameter size causes an exponential increase in the computational time as well as in the degree of problem complexity (Yu and Tai, 1998, Danziger et al., 2009). In order to solve this problem, many artificial intelligence (AI)-based strategies have been developed to find near optimal solutions (Shiba et al., 2004, Nurmela, 2004, Yuan et al., 2011, Danziger et al., 2009).

Although useful, most existing AI-based strategies require complex computations, e.g., in terms of the need to deal with mutations, crossovers, and the local minima problem (Jarboui et al., 2007, Panda and Padhy, 2008, Afzal et al., 2009, Anagnostopoulos and Kotsikas, 2010). For example, due to the need to interact with both the peers and the environment in order to update and exchange information, adopting Genetic Algorithm (GA) or Ant Colony Algorithm (ACA) as a general strategy for interaction test suite generation appears computationally expensive (Jarboui et al., 2007, Panda and Padhy, 2008, Afzal et al., 2009), especially for high interaction strengths (*t*). In the case of high

interaction strength, there can potentially be large combinatorial values to be manipulated and search for (Afzal et al., 2009, Pargas et al., 1999). Strategies based on Simulated Annealing (SA) and Tabu Search (TS) on the other hand, appear particularly effective. These strategies often give optimal results for small test configurations; but they suffer from the local minima problem (Anagnostopoulos and Kotsikas, 2010, Osman, 1993). For these reasons, existing AI-based strategies have been confined to small interaction strengths (i.e., $t \leq 3$) or small test configurations (Cohen et al., 2003, Chen et al., 2009, Wang et al., 2008, Shiba et al., 2004). However, in order to be effective, recent studies and empirical evidence demonstrate the need to go up to $t=6$ to capture most faults (Kuhn and Reilly, 2002, Kuhn et al., 2004, Dalal et al., 1999, Yuan et al., 2011). As such, the exploration and implementation of other strategies with lightweight computation is essential.

Particle Swarm Optimization (PSO) has proven its effectiveness in many research areas (Ganjali, 2008, Windisch et al., 2007, Jarboui et al., 2007, Panda and Padhy, 2008, Afzal et al., 2009). The effectiveness of PSO is due to three main features, namely, recombination, mutation, and selection (Padhy, 2009). With regard to recombination, PSO does not have a direct recombination operator despite the stochastic acceleration of a particle toward its previous best position resembling the recombination procedure of other techniques. Instead, it manages information exchange only between the possession experience of the particle and the experience of the best particle in the swarm. In terms of mutation, the standard PSO method has the advantage of not using evolutionary operators such as crossover and mutation (Liang et al., 2006, Liu and Maghsoodloo,

2011), thereby enabling a lighter computational load. For selection, PSO does not use the survival of the fittest concept, denoting that it does not use direct selection. Therefore, during optimization, particles with lower fitness values can survive and likely visit any point of the search space (Padhy, 2009).

Due to such alluring prospects whilst complementing earlier work on *t*-way and variable-strength strategies, this thesis presents the design and implementation of a new strategy called (Particle Swarm-based Test Generator) PSTG. As the name suggests, PSTG generates the interaction test suites using Particle Swarm Optimization (PSO). Thus, it is the hypothesis that suggests the adoption of PSO is useful for *t*-way and variable-strength test suite generation.

1.3 Aim and Objectives

The aim of the research is to design, implement, and evaluate a new interaction testing strategy, called Particle Swarm Test Generator (PSTG), for constructing *t*-way and variable interaction strength test suites based on Particle Swarm Optimization. To realize this aim, the following objectives are adopted:

i. To investigate the application of Particle Swarm Optimization for PSTG's design and implementation in order to support *t*-way and variable-strength test suites construction.

ii. To investigate and evaluate the performance of the PSTG strategy against other computational and AI-based strategies in term of the generated test suite size.
iii. To investigate and evaluate the effectiveness of the test suites generated by the PSTG strategy for interaction fault detection.

1.4 Methodology of the Research

Overall, the research's methodology is divided mainly into three phases.

i. Literature review: in this phase, the literature survey is undertaken to establish the state-of-the-art on interaction testing. The literature starts by reviewing the importance of the software testing in the software quality assurance process. By establishing this importance, the existing sampling and test design techniques are reviewed also and the importance of the interaction testing as complementary technique in software test design is established. Then, the existing literature of interaction testing strategies is reviewed to identify the features and drawbacks of the strategies and techniques. Based on the literature review survey, the requirement of the research is established in this phase. From the requirement, how the PSO, *t*-way, and variable-strength algorithms will be implemented is decided here.

ii. Design and Implementation: here, the adoption of PSO is established and the required algorithms are decided. Then, the complete algorithms making up the PSTG strategy are designed, implemented, and optimized in this phase. In addition, the parameter tuning of the strategy is performed here also.

iii. Evaluation, Benchmarking, and Case study: experiments with well-known benchmarking configuration as well as a case study are undertaken in this phase to investigate and evaluate the performance and effectiveness of the strategy.

To illustrate how the aforementioned phases are related, Figure 1-1 summarizes the research's activities.

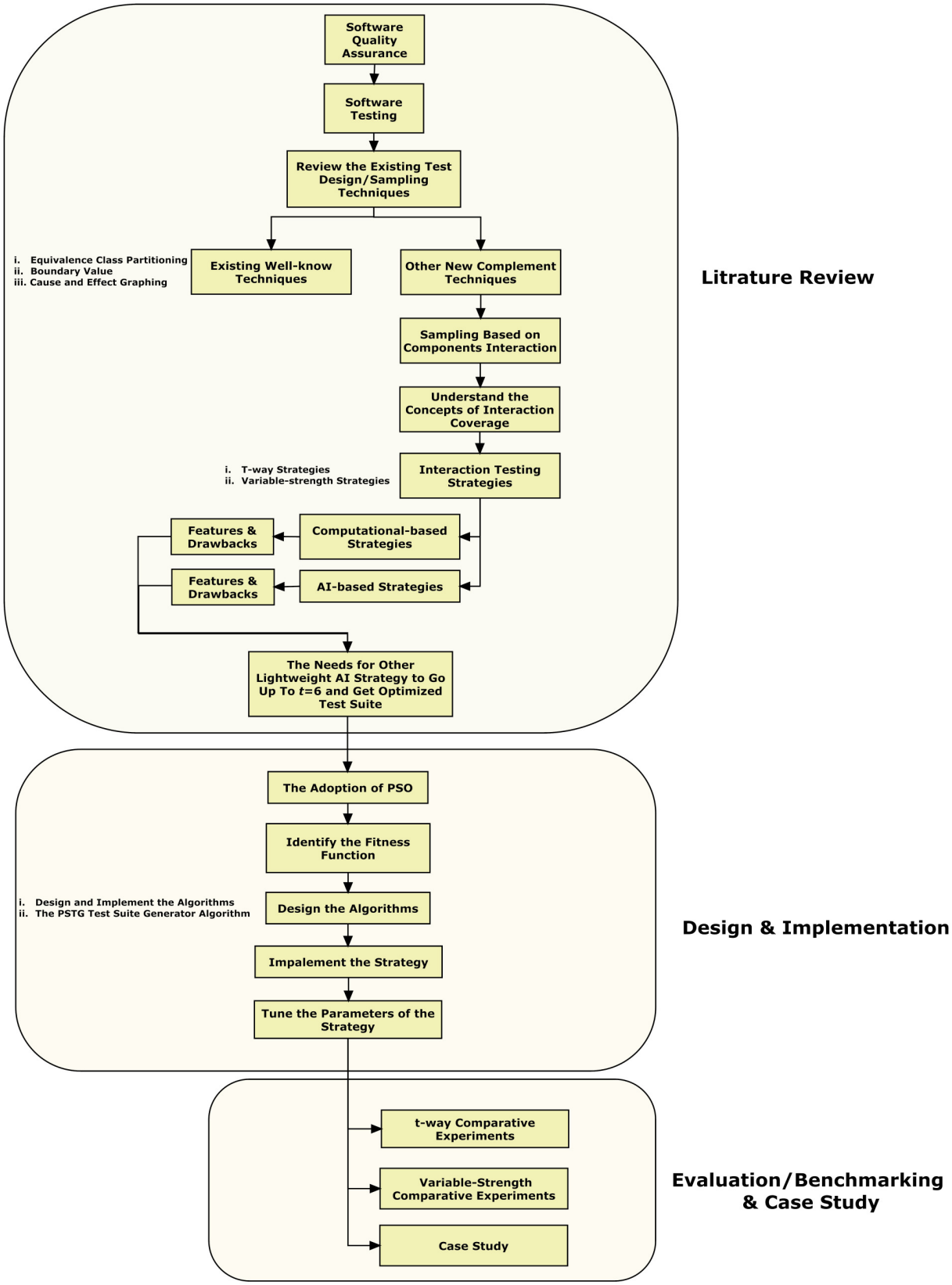

Figure 1-1 The Research's Activities and Flow

1.5 Thesis Organization

This thesis is organized into five chapters. The rest of the thesis is organized as follows. Chapter 2 reviews the state-of-the-art for the software interaction testing and test case generation strategies. The chapter starts by introducing the test case design techniques and illustrating how the test cases are selected in each technique. Then, a simple e-commerce model is given as a configurable software system to illustrate the *t*-way and variable-strength interaction testing. Thereafter, the chapter provides an extensive elaboration to understand how the interaction elements could be generated and how they could be covered. A theoretical background for interaction testing is then given. Finally, the chapter discusses and examines the existing literatures on *t*-way and variable-strength test suites generation strategies and different applications of interaction testing.

Chapter 3 outlines the design and implementation of the PSTG strategy, including its corresponding algorithms. The chapter illustrates each algorithm in brief with examples. The chapter also gives the justifications for using PSO and how to use it in the strategy. Finally, the chapter shows how the design parameters of the PSO are tuned to achieve the best possible result.

Chapter 4 highlights the evaluation of PSTG. Benchmarking of PSTG is undertaken to evaluate its competitiveness by comparing the results achieved with those published and publicly available well-known strategy implementations. In addition, the chapter presents a case study using a reliable artifact to demonstrate the applicability and

effectiveness of the test suites generated by PSTG. The presentation of the evaluation results in each stage and the case study is accompanied by the analysis.

Finally, in Chapter 5, the conclusions of the research are presented as well as the findings and contributions of the research are highlighted clearly. In addition, the chapter highlights the possible future works as a continuation of this work.

CHAPTER 2

LITERATURE REVIEW

In the previous chapter, the basic concepts of software quality assurance and software testing have been discussed. In addition, the importance of interaction testing as a compromise to exhaustive testing whilst complementing existing test case design techniques is elaborated.

Complementing the previous chapter, this chapter introduces the necessary background concepts and literature relating to interaction testing. To realize the usefulness of interaction testing, the chapter starts by reviewing three well-known test design techniques. Then, the problem of *t*-way and variable-strength interaction testing is identified using a modern e-commerce model. After that, a systematic example is given to illustrate how an interaction testing strategy works using the interaction elements generation and coverage mechanism. Next, the theoretical background, notations, and definitions for interaction testing are presented. Finally, the existing literature on *t*-way and variable-strength interaction is reviewed by identifying the generation strategies and the recent use of *t*-way and variable-strength interaction testing in different software testing applications.

2.1 Test Case Design Techniques

As mentioned previously, an important part of the testing process is the preparation of the appropriate test cases. To realize the importance of test case design techniques, the following sub sections highlight three well-known techniques and illustrate each technique with simplified examples.

2.1.1 Equivalence Class Partitioning

The equivalence class partitioning technique is used for designing test cases when the inputs and the outputs of the software-under-test are well-defined (Burnstein, 2003). In such a technique, the inputs are partitioned into classes that receive equivalent treatment. The test cases are selected by considering a test case for each class to be a representative of that class supposing that all the members of that class are processed equivalently by the software-under-test (Hass, 2008). In other words, the value of any test case in a class partition is supposed to be equivalent to any other test case in that partition (Sharma and B., 2010). Hence, if a fault is detected by a test case in the class partition, it supposed to be detected by the other test cases in the same partition (Myers, 2004). Figure 2-1 represents a simple example to illustrate this technique clearly.

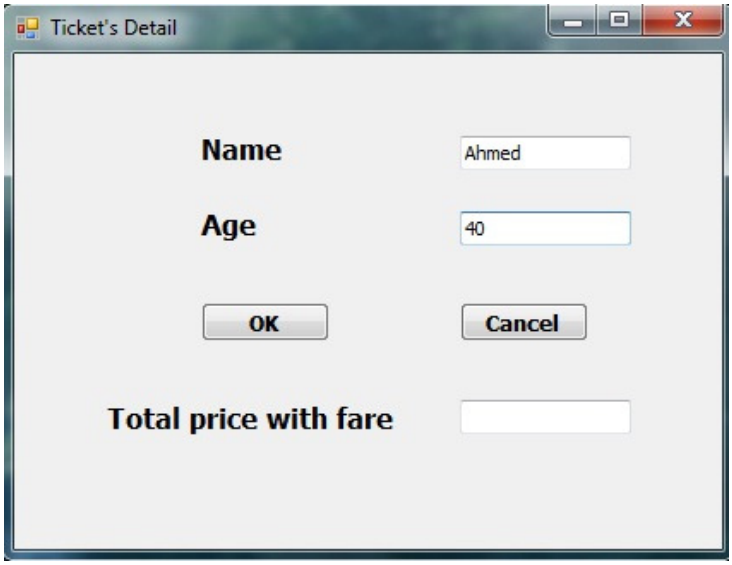

Figure 2-1 A Simple Application to Illustrate Equivalence Partitioning Technique

The example in Figure 2-1 represents a simple application to calculate the ticket's price after the fare considering the age of the costumer. The fare depends mainly on the age of costumer. For example, the tickets for children under 5 years old are free, and those aged between 5-15 years old will receive 50% discount, while those over 65 years old get a 25% discount. It should be noted here that no discount is given for age range between 16 to 64 years. Indeed, there are a finite number of age values that could be tested for the program. However, practically it is impossible to test all those values. Hence, equivalence class partitioning is used here to partition the values of the age into classes depending on the fare. Clearly, in this example, there are four classes of the age, those under the age of five years, those over the age 65 years, those between 5 and 15, and those between 16 and 64. To test the fare feature of the software, it is desirable to choose one value from each class partition, the middle value if possible. Therefore, 2, 10, 40, and 70 are needed to undertake the equivalence partition testing.

2.1.2 Boundary Value Testing and Analysis

While the equivalence class partitioning technique leads to select the test cases from the equivalence class, many faults could also occur exactly on, above, and below the edges of equivalence classes (Myers, 2004, Burnstein, 2003). Boundary value analysis is a test case design technique in which tests are selected to include representatives of boundary values.

For the example in Figure 2-1, the boundaries are at ages 0, 5, 15, and 65. Using the boundary value technique, the tester has to choose test values on and at either side of each boundary. Therefore, the tester must consider (-1, 0, 1), (4, 5, 6), (14, 15, 16), and (64, 65, 66) as test cases.

2.1.3 Cause and Effect Graphing

Cause and effect graphing (CEG) is another test case design technique used to validate a given software from its specification. In contrast with the aforementioned techniques, which are important for data processing intensive applications, the CEG is used with the control intensive application (Srivastava et al., 2009). The test cases in such a technique are designed to represent the input events (or causes) and the corresponding actions (or effects) (Naik and Tripathy, 2008). The cause is an input condition in the specification that may affect the output of the program, whereas the effect is the response of the program to any combination of input conditions (Srivastava et al., 2009).

With this technique, the tester first identifies the causes, effects, and constraints from the specification of the software-under-test. Then, the cause and effect graph is constructed as combinational logic network graph. The graph consists of nodes, which represent the causes and effects, constraints, and arcs with Boolean operators (and, or, not) between causes and effects. A unique identifier for each cause and effect is assigned then, and the relationship between causes and effects is marked on the cause and effect graph. Hereafter, the cause and effect graph is transformed to a decision table to prepare the test cases.

For the aforementioned example in Figure 2-1, suppose that the specification of the application states that the name field in the application should not contain any special characters or numbers and its length should not exceed a specific length. If the length of the name is out-of-range an error message will appear, and if the name contains special characters the message “not valid” will appear. Otherwise, the price of the ticket will appear. Hence, the input conditions, or causes are C1: The length of the name from 1 to 80, and C2: The name without special characters. Whereas, the output conditions or effects are E1: The name is out-of-range, E2: The price of the ticket, and E3: The name is not valid. Based on these causes and effects, the relationships of the CEG can be identified as shown in Figure 2-2.

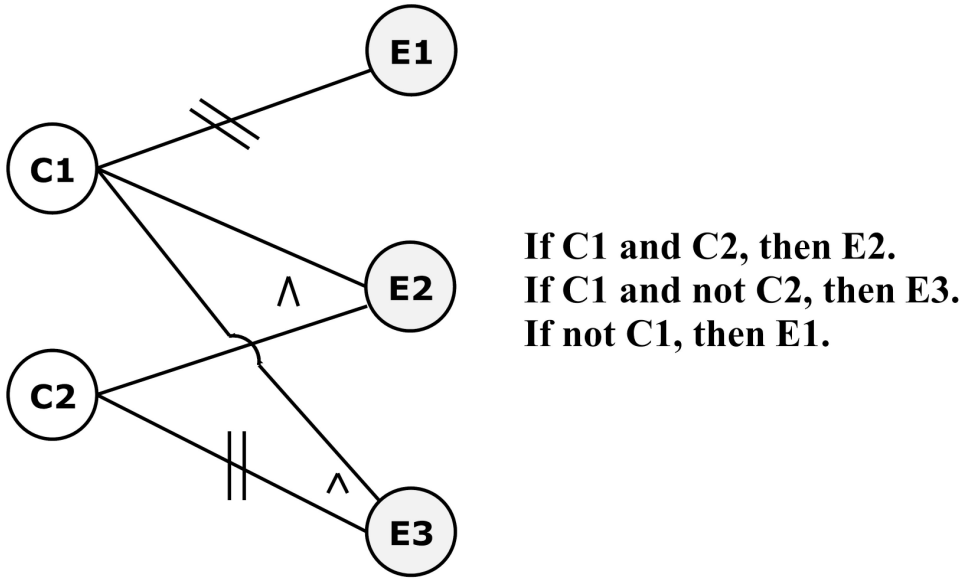

Figure 2-2 The CEG for the Example in Figure 2-1

The next step is to develop a decision table. Each row in the table is dedicated for a cause or an effect. The cells in the table can be filled by "0", "1", or "*". The "1" indicates the inclusion of a cause or effect, a "0" indicates the exclusion or absence of a cause or effect, while the "*" indicates a "don't care" value. Therefore, the decision table could be summarized in Table 2-1, where columns T1, T2, T3 represent the test cases.

Table 2-1 The Decision Table for the CEG in Figure 2-2

	T1	T2	T3
C1	1	1	0
C2	1	0	*
E1	0	0	1
E2	1	0	0
E3	0	1	0

2.2 A Problem Definition Model for Interaction Testing

A simple model to illustrate the *t*-way and variable-strength interaction is used here. Figure 2-3 represents the topology of a modern e-commerce configurable software system based on the Internet. The system may use different components or parameters. In this thesis, the term "parameter" (or P) is used to describe the components of the system similar to its usage in the literature. In this example, the system consists of five parameters. The client side has two parameters or two types of clients: those who use smart phones and those who use normal computers. There are different configurations in both cases. On the other side are different servers and databases.

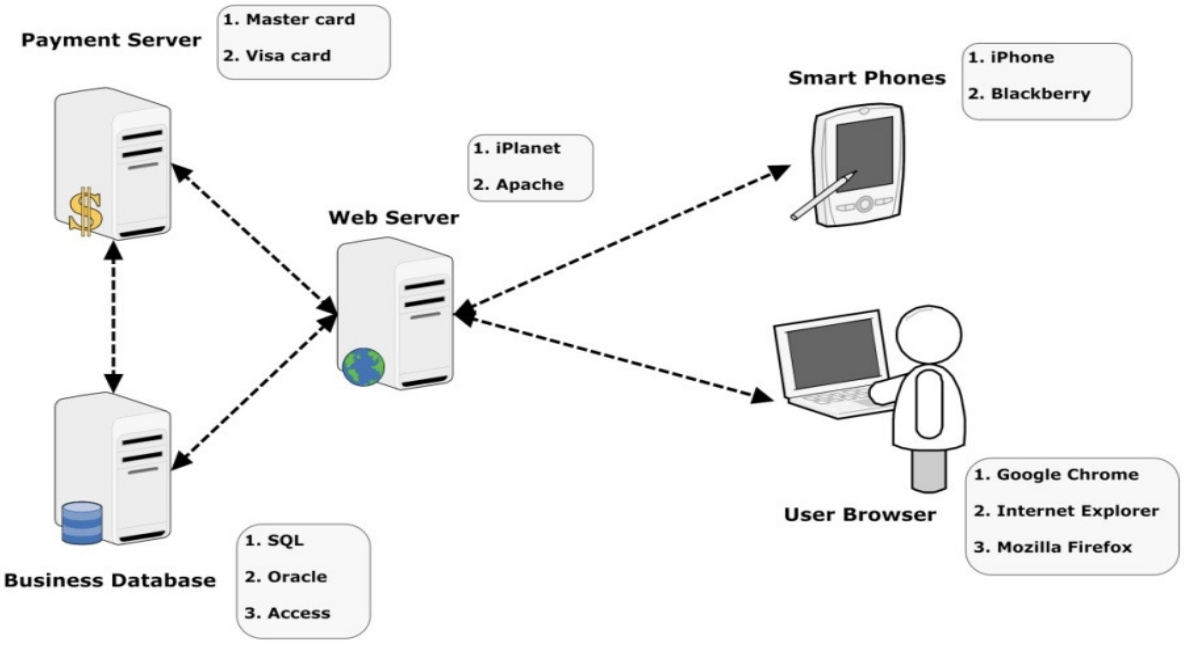

Figure 2-3 An e-Commerce Configurable Software System

The term “value” (or v) is used to describe the configuration of each component. Thus, the system in Figure 2-3 can be summarized as a five-parameter system with a combination of three parameters with two values, and two parameters with three values, as in Table 2-2.

Table 2-2 The e- Commerce System Components and Configurations

	Components or Parameters				
	Payment Server	Smart Phone	Web Server	User Browser	Business Database
Configurations or Values	Master Card	iPhone	iPlanet	Chrome	SQL
	Visa Card	Blackberry	Apache	Explorer	Oracle
				Firefox	Access

Different testing technique could be useful for this software system, but unexpected interactions between parameters is a common source of software fault (Williams and Probert, 2001). This risk increases when the number of parameters increases. To reduce this risk and ensure the quality of such software, manufacturers may need to test all combinations (i.e., exhaustive testing) or interactions among parameters, which requires 72 test cases (i.e., 2×2×2×3×3). However, all pairwise (two-way) interactions of the system parameters can only be tested using nine samples (i.e., test cases) as shown in the first nine test cases in Table 2-3. Although this will minimize the test cases and cover all pairwise interactions, testing the interactions between two servers (Web server and payment server) and the business database may be necessary to ensure that there are no interaction problems arise among the parameters. One way to achieve this is to compute the two-way test cases for all five parameters, then compute three-way test cases for all three parameters (Web server, payment server, and the business database),

which results in 21 test cases. However, this will require higher expenses due to the large amount of test cases, particularly in highly configurable systems. Another way to achieve two- and three-way test cases simultaneously is to combine them in one test suite. Hence, minimal coverage across the parameters is maintained and there is no need to run all 72 or 21 test cases. The test suite shown in Table 2-3 provides the variable-strength coverage for the system in Figure 2-3 with only 12 test cases by using the nine pairwise test cases and adding the last three test cases.

Table 2-3 The Pairwise and Variable-Strength Test Suite for the System in Table 2-1

Test No.	Payment Server	Smart Phone	Web Server	User Browser	Business Database
1	Master Card	Blackberry	Apache	Firefox	Oracle
2	Visa Card	iPhone	iPlanet	Explorer	Oracle
3	Visa Card	iPhone	Apache	Chrome	SQL
4	Master Card	Blackberry	iPlanet	Chrome	Access
5	Master Card	Blackberry	iPlanet	Explorer	SQL
6	Visa Card	iPhone	iPlanet	Firefox	Access
7	Master Card	iPhone	Apache	Explorer	Access
8	Visa Card	Blackberry	iPlanet	Chrome	Oracle
9	Visa Card	Blackberry	iPlanet	Firefox	SQL
10	Master Card	Blackberry	iPlanet	Explorer	Oracle
11	Master Card	iPhone	Apache	Firefox	SQL
12	Visa Card	Blackberry	Apache	Chrome	Oracle

As a result, it is essential to define and create test suites with *t*-way and variable-strength interactions, especially when the interactions grown. Finding efficient ways to construct the test suite within a minimum test size is also advantageous to save cost and time, especially for highly configurable systems.

2.3 Understanding the Interaction Coverage

In order to understand how the interaction test suites are constructed and how the interaction elements are covered, the following simplified example is considered. The example in Table 2-4 represents a system with three input parameters (P_1, P_2, and P_3) each of which having two values (0 and 1).

Table 2-4 A Simplified Example for a System with 3 Parameters 2 Values

	Input Parameters		
	P_1	P_2	P_3
Values	0	0	0
	1	1	1

Here, as illustrated previously, the exhaustive test suite or the full strength (i.e., t=3) of the system could be achieved using eight test cases (i.e., 2^3) as in the Figure 2-4.

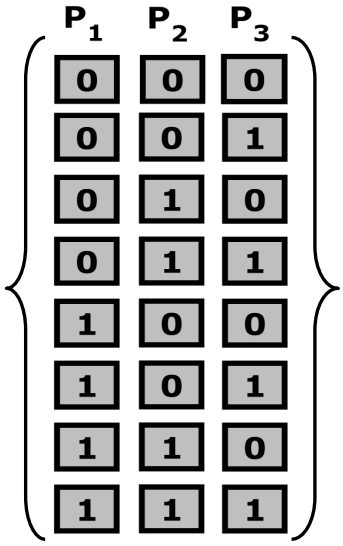

Figure 2-4 The Exhaustive Test Suite for the System in Table 2-4

When the number of parameters and values are grown, this exhaustive test suite grows exponentially. By considering pairwise interaction test suite as a compromise test suite, the number of test cases can be reduced significantly with full coverage of the interaction elements. The number of these interaction elements can be predicted mathematically using the following equation (Colbourn and Dinitz, 2006).

$$\text{Interaction Elements} = \binom{P}{t} v^t = \frac{P!}{t!(P-t)!} v^t \qquad (2\text{-}1)$$

The interaction elements were constructed by taking all the combination of system parameters then assign the related values for each combination. Figure 2-5 shows all the pairwise interaction elements for the system in Table 2-4.

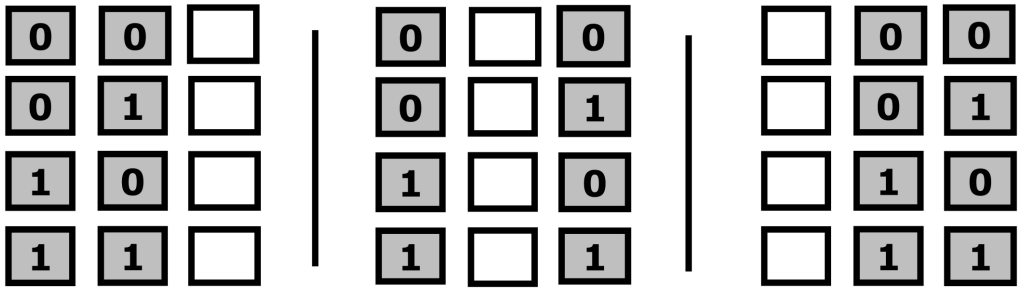

Figure 2-5 Total Pairwise Interaction Elements for the System in Table 2-4

The interaction test suite tends to cover all the interaction elements at least once to ensure that all the interactions are tested. The coverage of the interaction elements depends on how the test case is constructed. A test case could cover one or more interaction element depending on the arrangement of the test case itself. For the above example, Figure 2-6 illustrates how the test case can cover the interaction elements.

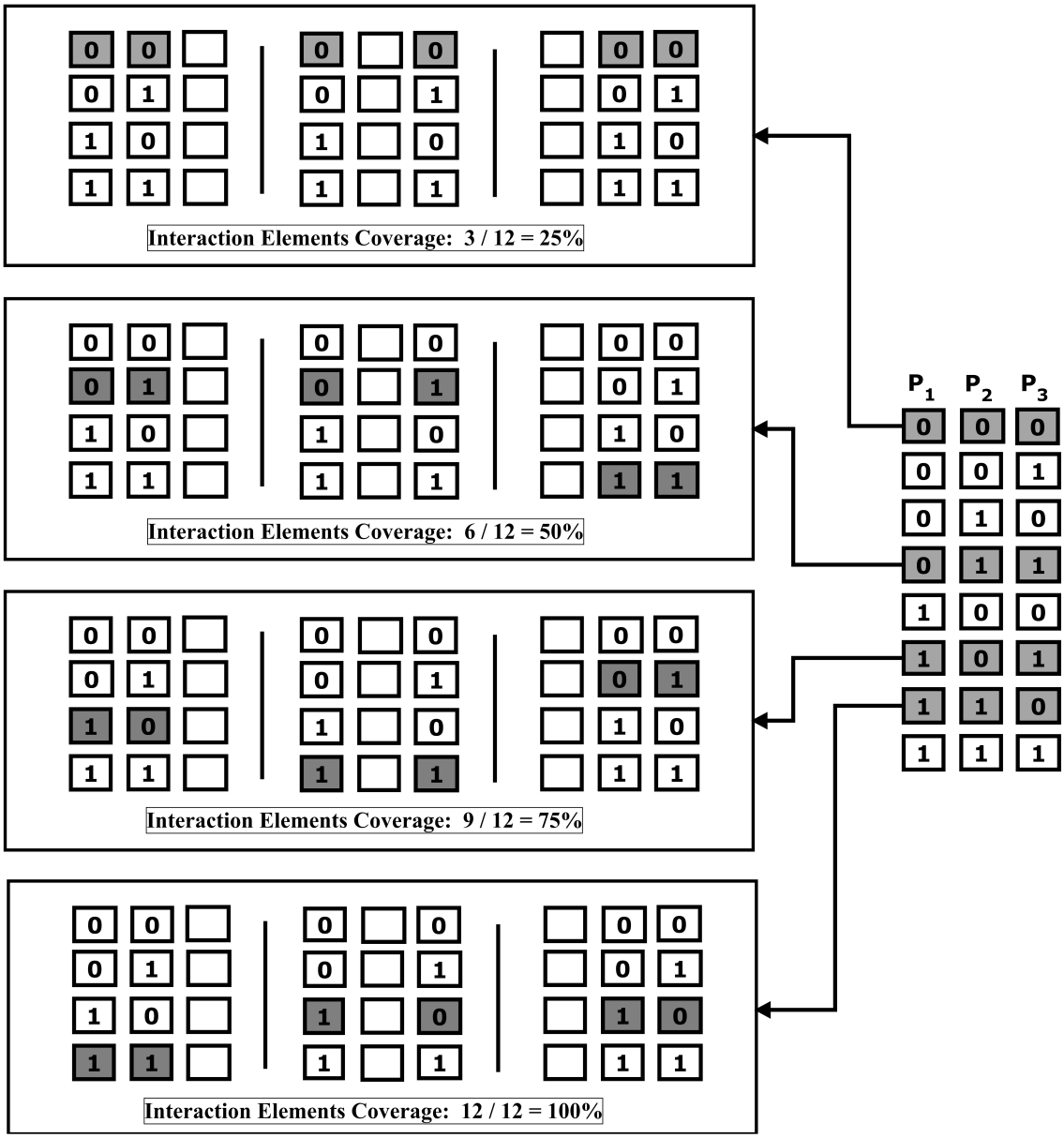

Figure 2-6 An Example to Illustrate the Interaction Elements Coverage for the System in Table 2-4

In the figure, 100% coverage of all the pairwise interactions could be achieved by only four test cases. However, it could be achieved by more test cases if other test cases are chosen rather than those test cases in the figure. As illustrated in Figure 2-7 clearly, the test case (0,0,0) covers 25% of the pairwise interaction elements when it is covering three interactions. By considering the next test case i.e., (0,1,1), three more interactions