P2R – A PAIRWISE TESTING STRATEGY SUPPORTING EXECUTION RESUMPTION

By

MOHAMMED H.S. HELAL

**Thesis submitted in fulfillment of the requirements
for the degree of
Master of Science**

February 2013

ACKNOWLEDGEMENT

First and foremost, I would like to express my deepest appreciation to this research supervisor Prof. Kamal Zuhairi for his limitless support and guidance. This research work would not reach completion without his attention and care.

To my father Prof. Hekmat Helal, your words of advice and times of support are priceless. Thank you for your tremendous efforts to support me and fund my study, I owe you all my life and I hope I will always make you proud. To my dearest person in my life, my mother Suhair, thank you for your genuine feelings. Thank you for supporting me all the time, for calling me almost every day and for praying for me every night. Thank you for taking part in my academic life since I was a child. To my brothers Muath, Hussein and Hamzah, and my little sister Ayah, thank you all for being my friends as well as siblings.

To the wonderful Malaysian people, thank you for your fine hospitality, for making me feel I am home. And thank you for being one of the biggest supporters for my beloved country Palestine. For the Malaysians who fix stickers of the Palestinian flags, my deepest thanks for your Islamic and humanitarian feeling toward the holy land. I hope the best for Malaysia, it has become my second home.

To all USM staff especial thanks to you, especially Madam Farida from IPS, thanks to everyone who helps the completion of this work, I wish you all the best.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES AND ILLUSTRATIONS

LIST OF ABBREVIATIONS AND NOMENCLATURE

Abbreviation	Meaning
ACA	Ant Colony Algorithm
AETG	Automatic Efficient Test Generator
AETGm	Automatic Efficient Test Generator Modified
CA	Covering Array
CIS	Complete Interaction Sequence
GA	Genetic Algorithm
GUI	Graphical User Interface
HDD	Hard Disk Drive
IEL	Interaction Elements List
IPO	In Parameter Order
IPOG	In Parameter Order General
NA	Not Applicable
SA	Simulated Annealing
V&V	Verification and Validation

P2R – SATU STRATEGI PENGUJIAN BERPASANGAN MENYOKONG PELAKSANAAN PENYAMBUNGAN SEMULA

ABSTRAK

Jaminan kualiti produk perisian memerlukan proses pengujian menyeluruh dan memakan masa terutama apabila melibatkan bilangan parameter dan nilai kemungkinan parameter input yang besar. Pengujian perisian berpasangan biasanya digunakan kerana kegunaannya telah terbukti bagi menyampel kes-kes ujian untuk perisian sebegitu sambil mencapai tahap liputan yang diterima untuk baris kod dan liputan fungsi. Walau bagaimanapun, proses ini mungkin mangambil masa berjam-jam malah berhari-hari lamanya kerana pengurangan saiz senarai yang dihasilkan memerlukan pengiraan yang memakan masa. Sekiranya gangguan berlaku semasa menjana sut ujian (contohnya disebabkan oleh kuasa kegagalan perkakasan, atau kerosakan perisian), proses itu mesti dimulakan dari awal, maka, banyak masa dan usaha akan sia-sia jika proses terganggu apabila ia hampir siap. Dalam usaha menangani isu yang disnyatakan di atas, beberapa skema penyambungan semula telah dikaji untuk dipadankan bagi strategi berpasangan. Projek itu memberi tumpuan kepada pembangunan berpasangan P2R strategi baru, yang boleh menjana sut ujian berpasangan yang kompetitif dari segi saiz sut ujian dan masa. Bagi mencapai matlamat ini, P2R menyediakan skema penyambungan semula yang terdiri dari gabungan periodik dan dinamik. Ekperimen membuktikan P2R berkebolehan untuk melakukan meneruskan semula ujian dan menghasilkan keputusan yang kompetitif.

P2R – A PAIRWISE TESTING STRATEGY SUPPORTING EXECUTION RESUMPTION

ABSTRACT

Achieving an accepted level of software product quality assurance requires exhaustive and time consuming testing process, especially when testing software with large number of parameters and large number of possible input values for its parameters. Pairwise software testing have been commonly employed since it proved its usefulness to sample lists of test cases for such software while achieving an accepted level of code line and function coverage. However this process might require hours even days, since reducing the size of the produced list require time consuming computations. In case interruption occurred while generating test suite (e.g. due to power failure, hardware or software malfunctions), the process must be restarted from the beginning, much time and effort will be wasted if process have been interrupted when it's near completion. In order to address the aforementioned issue, a number checkpointing schemes have been studied to refer the most suitable scheme to match pairwise test suite strategy. The project focuses on the development of a new pairwise strategy P2R, which can generate sufficiently competitive pairwise test suites in terms of test suite size and execution time. In order to achieve this target, P2R employed a new checkpointing scheme that is based on an amalgam of periodic and dynamic checkpointing schemes. Experiments have proven that P2R provides the ability to resume existing test while generating competitive results.

CHAPTER 1

INTRODUCTION

Our dependence on software is dramatically increasing over the years. Smart phones, cars, security systems, washing machines and most contemporary devices, often rely on software to achieve its functionality, control and performance. Such dependency on software raises the need for sufficiently extensive software testing in order to acquire an accepted level of reliability. While developing software, testing activities take action before marketing the product. This causes significant cost in time and resources. Testing software passes through many phases starting from test planning where test engineers sample test cases or test suites. Here, test cases are possible input combinations assigned to the suitable input parameters of the tested software, where such test cases are executed and the behaviour of the system is studied to determine its correctness. Often, test cases are required to cover an accepted percentage of tested system code lines and functions. Exhaustive test cases covering all possible input combinations will potentially catch all errors in the tested system. Although useful, it is not practical to perform such test since the generated test suite will be very large for systems with large configurations, and performing test execution may require years.

Many strategies have been developed in the past to sample test suites aiming at reducing test suite sizes, taking into consideration time needed to sample the lists. Pairwise testing is a commonly used testing strategy that has been proven to be efficient in sampling test suites for systems with large configurations. Pairwise testing ensures coverage of all possible pairwise input combinations for each pair of

parameters, resulting in an accepted percentage of line and functions coverage. As a result, many pairwise testing strategies have been developed in the last 15 years.

As mentioned earlier, software are increasing in term of complexity and parameter numbers and sizes (Danziger, Mendelsohn, Moura, & Stevens, 2009; Xun, M.B., & A.M., 2011). This scenario causes significant challenge for test engineers, that is, in terms of getting the required sample test cases. In this case, the process may take hours, days, or even weeks if the tested systems are large. If interruption occurs (i.e. caused by system failure or power failure) while generating test cases, restarting the process is often required. Restarting leads to undesired waste of time, effort and resources. Avoiding restarting the process in case of interruption can help avoid such wastes. Here, checkpointing can help achieve such objectives.

Motivated by these prospects, the development of a new pairwise testing strategy that is sufficiently competitive as far as test execution and size and address resumption capability is highly desirable. Continuing from these aforementioned discussions, the next section highlights overviews on software testing, problem statement discussion, thesis aim and objectives, research contributions. Finally, thesis outline will be elaborated in final section.

1.1 Overview on Software Testing

Software quality assurance is an essential phase during the overall software development process, as software needs to be analyzed to verify and evaluate its correctness (IEEE, 1008-1987 - IEEE Standard for Software Unit Testing, 1986)(IEEE, 610.12-1990 - IEEE Standard Glossary of Software Engineering

Terminology, 1990). Software quality assurance is basically achieved through testing throughout overall software development process (Beizer, 1995). Testing is often performed during all software development stages, each stage has different inputs and outcomes, thus testing each stage differ in aims and objectives.

In order to explain where software testing fits in the overall software development process, it is important to illustrate the phases of software development. According to Ammenn *et al*, software development process start with *requirements analysis* phase, at this phase the customer's requirements specifications are obtained and analysed. Then *architecture designed* is performed based on the requirements specifications, this phase defines the components and connectors that form the software architecture. After forming the architecture, *subsystem design* phase defines the behaviour of the subsystems and the functionality of each subsystem. After designing the subsystems, subsystems detailed design is performed at *detailed design* phase. At this phase, the structure and behaviour of each subsystem is designed preparing for *implementation phase* where software developers implement the design and start writing source codes (Ammann & Offutt, 2008).

While software development process is running, testing is often performed for each stage of the process. Tests associated with of each stage have different specifications, aims, information guiding the test and responsible staff to perform tests. A V-model of software development process associated with testing have been commonly presented in literature (Ammann & Offutt, 2008), V-model illustrates tests associated with each stage of the software development life cycle. Figure 1-1 bellow shows the V-model of software development and testing stages.

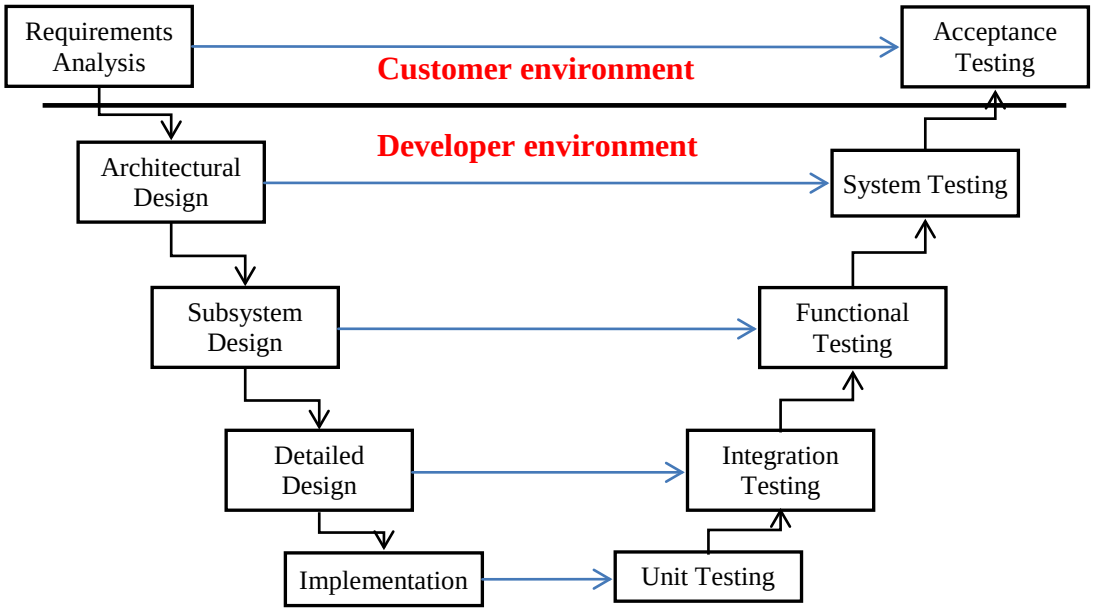

Figure 1-1Software development in V-model(Ammann & Offutt, 2008)

As shown in Figure 1-1, the software development stage starts with requirements analysis phase, then architectural design process. After that, subsystem design, detailed design, and then implementation phase. The figure shows five types of tests, each associated with one stage of development. The tests performed are unit testing, integration testing, functional testing, system testing, and acceptance testing.

The first test performed is unit testing, where testers examine the low structural level of the software. Unit testing is associated with the implementation phase and it is usually performed by software source code writers who implemented the software. The test verifies each function or class in the source code separately from the others

and regardless to the interaction between the software structure elements. Using selected input values, testers execute each function or class and observe the returned values. Returned values are required to match tester's expectations. Otherwise, changes on source code are required. After assuring that each component of the software is working correctly, integration testing examines that components work correctly together. Integration testing is associated with the implantation phase and usually performed by the same developing team.

As the functionality of subsystems is determined in the subsystem design phase, functional testing is associated with this phase to assure software's functionality correctness. After functional testing, system testing is performed to assure the software functions correctly on different platforms and environments, For example, different operating systems, hardware components, and network topography. Usually, functional and testing are performed by third or independent party.

Acceptance testing is what helps customer decide accepting or rejecting the software product. According to requirement specifications, customer examines the correctness of the software response. If faults are found, customer refers back to the developing company then changes are required to be done. Although acceptance testing are run in customer's environment, developing companies often assign testing teams to perform acceptance testing before presenting the software product to customer.

Not shown in Figure 1-1 is regression testing. This type of testing is performed when any of the five mentions tests result is finding faults. When a fault is found, changes

are performed on the software. Regression testing is performed after the changes are performed, and sometimes, regression testing is repeated until no faults are found.

Based on the difference in testing aims and the ability to access source code, Myers and many others have proposed classification of software testing strategies into two classes, the first is called white box testing and the other is called black box testing (Myers, 2004; Beizer, 1995). White box testing, or structural testing (IEEE, 1990), takes advantage of the ability to access software's source code, which enables testing teams to evaluate the internal structure of tested software. White box testing basically examines the logical structure of the software regardless to the customer's required specifications (Myers, 2004). As a result of dependence on the access on software's source code, white box testing is usually performed by the software programmer teams.

Black box testing basically depends on input/output relationship, without having access to the software's source code. Unlike white box testing, black box testing verifies the response of the software when executed with sets of test data, and then the response is studied and compared with the required specifications (Myers, 2004). Test is performed by developing executable files for the software, then series of suggested input combinations are chosen and the software is to be ran with the series test data. The software response is observed and if errors appear, modifications on the software are required (Myers, 2004). Usually, black box testing is performed by independent validation and verification teams.

The six testing types mentioned earlier differ in the ability to access source code, some of them are performed by developing teams while others are performed by independent party or the customer is self. Based on Myers's classification, Table 1-1 shows the difference between the six types of testing.

Table 1-1 Comparison between testing types

Testing Type	**Black box or white box**	**Testing performers**
Unit testing	White box	Developing team
Integration testing	White box	Developing team
Functional testing	Black box and white box	independent party and developing team
System testing	Black box	Independent party
Acceptance testing	Black box	Customer or independent party
Regression testing	Black box	Developing team and independent

According to Table 1-1, unit testing and integration testing use white box testing technique, functional testing use black box testing in test is performed by developing team while white box testing is used when functional testing is performed by independent validation and verification team. System testing, acceptance testing, and regression test use black box testing techniques.

According to Pressman, software testing activities involve four main steps, namely, test planning, test case design, test execution, and execution result monitoring and evaluation. Each test performed need to be planned at the early beginning stages of software development, the required resources for testing is assigned at test planning stage (Pressman, 2005). Moreover, testing scope, approach, schedule, features to be

tested and team performing test are decided at the test planning stage (Dustin, 2002). After that, test cases are to be designed according to the requirement specifications. Generated test cases are used to run the software at the test execution step. Finally, the outcome of software when test cases are executed is to be monitored in the execution result monitoring step, the results are compared to confirm matching the required specifications.

The chosen sets of test data are called *test cases* and a list of test cases is called *test suite* (Adrion, 1982). Generating more test cases might result in finding more errors, however useful, but causing more time overhead. Test suites are required to be large enough to satisfy an accepted level of reliability, and at the same time, small enough to reduce testing time overhead (Adrion, 1982).

Test cases are required to cover as many possible input combinations as possible, until the validity of the software is accepted. Testing all possible input combinations, and verifying software response when test is executed, will definitely reveal all bugs in the tested software. However useful and demanded, applying such exhaustive testing is not practical in terms of market deadline dates, resource constrains and budget(Yang & Chao, 1995).

Software testing phase plays a deciding role in software market price, as the test quality might cause either high cost or low quality. In order to help deciding when should software testing stop, many research work have been done on coverage criteria that helps testing teams to adopt testing plans during software testing. According to the nature of the tested software, its expected selling price, and its expected damage

and cost when fault occurrence, are to be considered when deciding when to terminate software testing. Researchers showed that a degree of reliability and coverage criteria is required when judging software to be qualified for release (Mathur, 2008). Mathur designed a model to illustrate how reliability and coverage criteria can help judge whether software is undesirable, risky, suspected or desirable (Mathur, 2008), as shown in Figure 1-2.

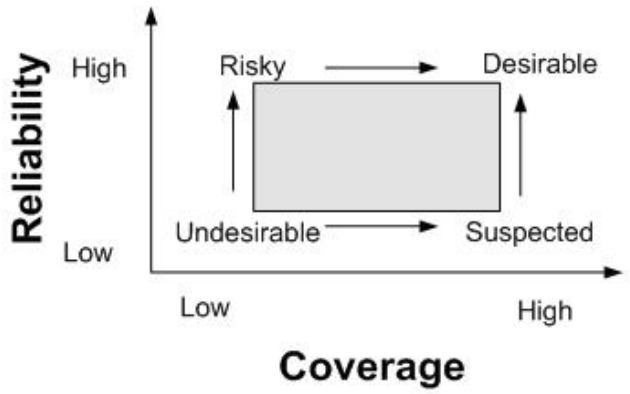

Figure 1-2 Software reliability vs. testing coverage (Mathur, 2008)

According to Figure 1-2, software that fails to achieve both high coverage and reliability is considered as undesirable, whereas risky software is highly reliable one but fails to achieve high level of coverage. A system that manages to achieve high level of coverage but with low reliability is considered as suspected. Finally, software achieving high level of coverage as well as high reliability is considered as desirable.

1.2 Importance of Software Testing

Software is involved in essential parts of our life, and sometime our human lives depend on software's reliability. For example, software used to manage modern cars, space shuttles, medical equipment, military applications and navigation systems. Faults can be deadly in such application while in other applications, faults might be less dangerous, but still cause massive loss of expenses. The importance of software testing can be illustrated by showing real life examples of faults that could have been avoided by a better testing technique. Highlighting the cost of these faults should explain the importance of software testing. The first example of lethal software fault was committed by Therac-25 a medical electron accelerator. The fault occurred six times between year 1985 and 1987 causing massive radial overdoses on patients, three of them died by direct effect of radiation(Leveson & Turner, 1993).

Another example of faults occurred in 1996 when a European space rocket fight 501 of Ariane 5 drifted away from its correct path 37 seconds after lunch spilling 370 million dollars. The fault was caused by variable overflow when moving data from 64-bit floating point variable to 16-bit integer (Le Lann, G., 1997; Dowson, 1997).

In 1999, software fault disconnected the popular online trading market eBay for 24 hours. According to The New York Time, the 24 hours disconnection dropped eBay's stock value from 182.6875 USD to 165.875 USD (The New York Times, 1999).

The examples mentioned earlier in this section illustrate the importance of software testing. Obviously, better testing techniques could have discovered the fault before their occurrence and avoid the unwanted losses.

1.3 Problem Statements

In order to maximize software reliability, it is often required to test all possible input combinations for tested software. Although useful, applying such exhaustive testing often causes high cost in terms of time and resources. Some software may contain large numbers of input parameters with large numbers of possible input values for their parameters, leading to large numbers of possible input combinations and consequently large numbers of test cases. In order to illustrate the possible size of the generated list, consider the form in Windows Internet Explorer shown in Figure 1-4. The form contains 62 checkboxes and two groups of radio buttons, one with two possible values and the other with three. Forming test suite to cover all possible input combination for this form (ignoring the other tabs) will result in a $27x10^{18}$ entry list (i.e. $2^{62}x2x3$). When executing the mentioned test cases, $89x10^{10}$ years may be needed to complete execution assuming one second is needed to execute each test case.

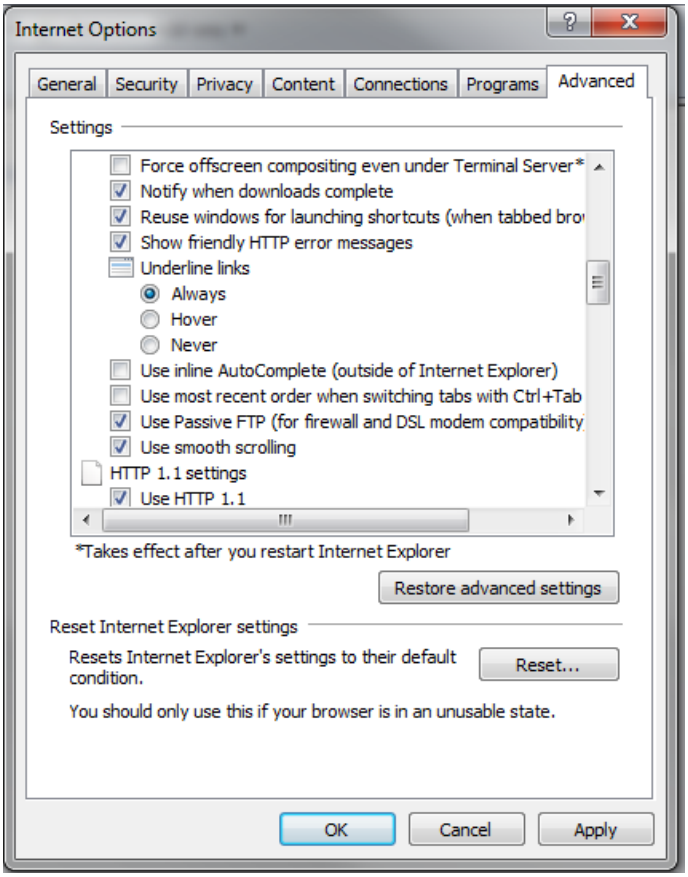

Figure 1-3Advanced option dialog for Microsoft Internet Explorer

Obviously, testing all possible input combinations is often impractical. Instead, generating test suites that cover all possible 2-way combinations (or pairwise testing) between all possible pairs of input parameters is often employed. Generating pairwise test suites will take a long time especially when the size of possible pairs increases dramatically with large configurations.

Here, the enormous growth in software functionalities and requirements cause massive increment in software line of codes, hence, affecting the parameters and size for testing consideration. As software applications size changed from kilobytes to gigabytes and terabytes, achieving high level of reliability will defiantly face the combinatorial explosion problem. This problem will show its effect while generating

pairwise test cases, more execution time (often, days) to generate them will be required since larger lists will need to be processed and significant time consuming computation will appear.

While generating pairwise test cases for systems with large configurations, power failure or system error might occur. If such interruption occurred, restarting the process from the beginning is often required. Generation process for pairwise test cases for some systems might require days or weeks; unwanted interruption can be costly especially near completion.

Motivated by this prospect, this research investigates the design of an efficient automatic pairwise test suite generation strategy that employs an efficient recovery system that suites test case generation strategy, namely P2R. P2R is a strategy to investigate the suitable recovery scheme to be applied on test case generation taking into consideration the time overhead of recovery.

1.4 Thesis Aim and Objectives

The main aim of this thesis is to investigate a suitable recovery scheme to be applied on an efficient pairwise test suite generation strategy. The main objectives of the work undertaken were:

- Investigate a new pairwise test suite generation strategy (P2R) that is able to accommodate a number of checkpointing schemes.
- Investigate P2R's efficiency in generating test suites as well as its overhead upon integrating the checkpointing schemes
- Evaluate P2R to confirm its correctness and effectiveness via experimental tests.

1.5 Research Scope

This research work focuses on pairwise test case generation process with resumption support. In doing so, the main concerns of the research involves in developing a pairwise strategy along with the suitable checkpointing schemes to address resumption.

1.6 Thesis Outline

This thesis is organized as follows:

Chapter 2 gives overviews on pairwise testing and checkpointing fundamentals, describing how pairwise testing and checkpointing work. Moreover, existing work on pairwise testing and checkpointing is highlighted.

Chapter 3 sets the scene on the research methodology, explaining the algorithms hired when generating test suites as well as when checkpointing. The chapter produces P2R strategy and justifies the use of some algorithms, and explains why these algorithms have been chosen and employed.

Chapter 4 illustrates the design and implementation of P2R, showing an illustrative figure for the whole process. The chapter also highlights the different checkpointing configurations that P2R provides along with their characteristics. Specifically, the chapter elaborates on the implementation of the algorithm for pairwise test generation as well as for its resumption support.

Chapter 5 involves evaluation and discussion. P2R is examined to prove its efficiency in generating test suites as well as its efficiency in recovery after interruption. Here, P2R will be subjected with a number of benchmark configurations. The correctness and the effectiveness of P2R will be elaborated in this chapter.

Chapter 6 draws the conclusion of this research work along with the scope for future work. Finally, the chapter ends with a closing remark..

CHAPTER 2

LITERATURE REVIEW

Chapter 1 has discussed the importance of pairwise software testing and the complications when generating test case during test planning for systems with large configuration. The combinatorial explosion problem and its effect on time to generate test cases are highlighted along with the problem statements.

Building for the materials in chapter 1, this chapter starts with an overview on pairwise testing and its objectives. In doing so, this chapter also describes how pairwise works through an illustrative example. Then, an overview on checkpointing will be illustrated. Finally, the chapter will also illustrate a literature review on both pairwise testing and checkpointing.

2.1 Pairwise Testing Fundamentals

Pairwise testing is a testing technique that assures coverage of all possible pairwise input combinations between all pairs of input parameters. In this section, basic fundamentals and definitions for pairwise testing will be highlighted. The section starts by presenting the objectives of pairwise testing then it shows how it work giving illustrative example. Then, time cost of pairwise test generation (which is one of the basic challenges for the generation process) will be discussed.

2.1.1 Overview on Pairwise Testing

Pairwise testing takes place while planning the test. Often, pairwise testing is highly effected by the size of tested software, large software (with large number of parameters and large number of possible vales for the parameters) yields large number of possible pairwise combinations. As a result of that, generating test suite for such system requires time consuming computations. For example: a software system with 20 parameters each parameter can have 10 possible input values result in 19000 possible tuples. Processing such large lists is not without a cost. Processing time is the main challenge for generating test suites for such systems as it may take hours or even days to complete.

Earlier studies suggest pairwise generation approaches into two: computational approach and algebraic approach (Nicola & Spanje, 1990). The first approach starts by generating *interaction element list (*or *tuples*), referring to the list of all possible 2-way input combinations (Younis *et. al.*, 2008). Then, the searching process starts to cover all interaction elements. Unlike the first approach, the second approach does need to generate interaction elements list, the approach use predefined algebraic formulas to directly generate test suite.

2.1.2 Illustrative Example

As an illustrative example showing how pairwise test suite generation works and how minimization of test suite size is done, consider the following example: a Microsoft Windows 7 form consisted of 4 checkboxes as input parameters (see Figure 2-1).

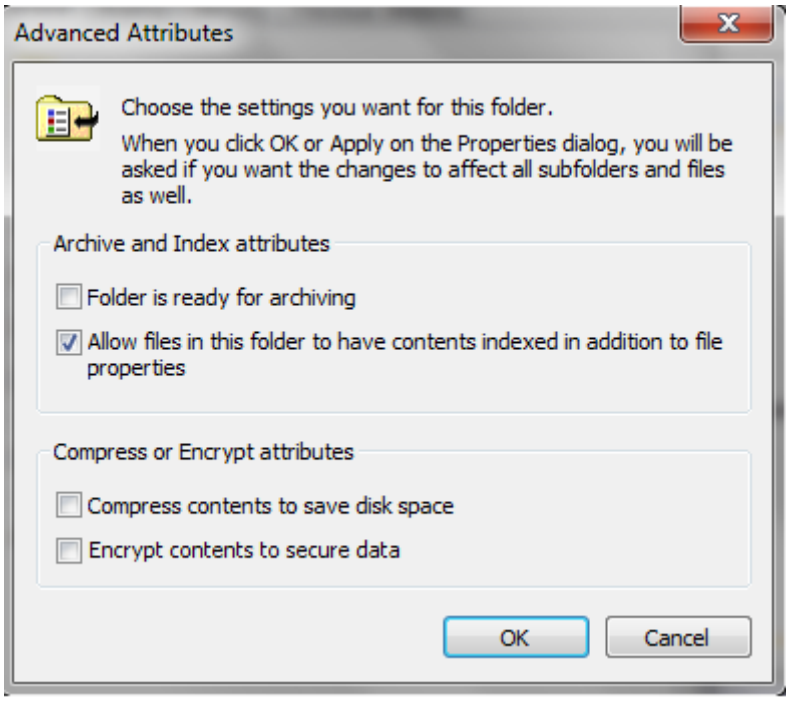

Figure 2-1Microsoft Windows form

From Figure 2-1, the input parameters are 4 each with two possible values (checked or unchecked). For the sake of simplicity, input parameters and their values are to be illustrated using symbols. The letter (A) represents the first parameters (folder is ready for archiving), A0 represents the checkpoint state when unchecked, while A1 represents the state when checked. For the rest of the parameters, the letters (B, C and D) are to represent them respectively. Whilst, B0 and B1 correspond to the values of the second parameter, C0 and C1 correspond to the third parameters and finally, D0 and D1 correspond to the fourth parameter. Table 2-1 shows possible input values represented by symbols and the actual value corresponds to each symbol.

Table 2-1 Possible input values

Parameter descriptions	**Parameter symbol**	**Parameter value**	**Parameter value symbol**
Folder is ready for archiving	A	Unchecked	A0
		Checked	A1
Allow files in this folder to have contents indexed in addition to file properties	B	Unchecked	B0
		Checked	B1
Compress contents to save disk space	C	Unchecked	C0
		Checked	C1
Encrypt contents to secure data	D	Unchecked	D0
		Checked	D1

Assembling test suite using exhaustive testing (i.e. testing all possible input combinations) yields 16 test cases (2 x 2 x 2 x 2 = 16). The number of generated combinations is calculated using the following equation (see Eq. 1).

$$\text{The number of possible combinations} = \prod_{i=1}^{P} V_i \qquad \text{Eq. 1}$$

Where:

P = number of parameters

V_i = number of possible values for parameter i

i = index for parameters.

In order to illustrate how pairwise testing work, it is important to show all possible pairwise combinations between all possible pairs of parameters. The possible pairs of parameters in our example are: AB, AC, AD, BC, BD, and CD. The pairwise combinations are shown in Figure 2-2.

Pair AB	Input Parameters			
	A	B	C	D
Possible Combinations	a1	b1	x	x
	a1	b2	x	x
	a2	b1	x	x
	a2	b2	x	x

Pair AC	Input Parameters			
	A	B	C	D
Possible Combinations	a1	x	c1	x
	a1	x	c2	x
	a2	x	c1	x
	a2	x	c2	x

Pair AD	Input Parameters			
	A	B	C	D
Possible Combinations	a1	x	x	d1
	a1	x	x	d2
	a2	x	x	d1
	a2	x	x	d2

Pair BC	Input Parameters			
	A	B	C	D
Possible Combinations	x	b1	c1	x
	x	b1	c2	x
	x	b2	c1	x
	x	b2	c2	x

Pair BD	Input Parameters			
	A	B	C	D
Possible Combinations	x	b1	x	d1
	x	b1	x	d1
	x	b2	x	d2
	x	b2	x	d2

Pair CD	Input Parameters			
	A	B	C	D
Possible Combinations	x	x	c1	d1
	x	x	c1	d2
	x	x	c2	d1
	x	x	c2	d2

Figure 2-2 All possible pairwise combinations for each pair of parameters

Pairwise strategies often suggest test case to cover as much pairwise combinations as possible reducing the generated test suite size in the process. Figure 2-2, shows all possible pairwise combinations for our example. It should be noted that the combinations between each pair of parameters is done while ignoring the values for the other parameters (as *Don't Care*) which do not contribute to the interaction. Here, the ignored values are marked as X.

Choosing values to replace the Xs in Figure 2-2 (i.e. generate test cases) has significant impact on the generated test suite size. In this case, the suggested test cases should cover as much pairwise interactions as possible. In order to illustrate how

choosing different values to replace X can affect the test suite sizes, two different suggestions with random values (first suggestion and second suggestion) are made in Figure 2-3 and Figure 2-4.

Pair AB	Input Parameters			
	A	B	C	D
Possible Combinations	a1	b1	c1	d1
	a1	b2	c2	d1
	a2	b1	c1	d2
	a2	b2	c2	d2

Pair AC	Input Parameters			
	A	B	C	D
Possible Combinations	~~a1~~	~~b1~~	~~c1~~	~~d1~~
	~~a1~~	~~b2~~	~~c2~~	~~d1~~
	~~a2~~	~~b1~~	~~c1~~	~~d2~~
	~~a2~~	~~b2~~	~~c2~~	~~d2~~

Pair AD	Input Parameters			
	A	B	C	D
Possible Combinations	~~a1~~	~~b1~~	~~c1~~	~~d1~~
	a1	b2	c1	d2
	a2	b1	c2	d1
	~~a2~~	~~b2~~	~~c2~~	~~d2~~

Pair BC	Input Parameters			
	A	B	C	D
Possible Combinations	~~a1~~	~~b1~~	~~c1~~	~~d1~~
	~~a2~~	~~b1~~	~~c2~~	~~d1~~
	~~a1~~	~~b2~~	~~c1~~	~~d2~~
	~~a2~~	~~b2~~	~~c2~~	~~d2~~

Pair BD	Input Parameters			
	A	B	C	D
Possible Combinations	~~a1~~	~~b1~~	~~c1~~	~~d1~~
	~~a2~~	~~b1~~	~~c1~~	~~d2~~
	~~a1~~	~~b2~~	~~c2~~	~~d1~~
	~~a2~~	~~b2~~	~~c2~~	~~d2~~

Pair CD	Input Parameters			
	A	B	C	D
Possible Combinations	~~a1~~	~~b1~~	~~c1~~	~~d1~~
	~~a2~~	~~b1~~	~~c1~~	~~d2~~
	~~a1~~	~~b2~~	~~c2~~	~~d1~~
	~~a2~~	~~b2~~	~~c2~~	~~d2~~

Figure 2-3 The first suggestion to replace X values

From Figure 2-3, the suggested values completed test cases causing redundancy in test case values. Here, redundant test cases are crossed with straight lines (i.e. strikethrough). The remaining uncrossed 6 test cases are the generated pairwise test suite. Table 2-2 shows the final pairwise test suite generated after the first suggestion of X values.

Table 2-2 Test suite resulted by the first suggestion

	Input Parameters			
	A	B	C	D
Test cases	a1	b1	c1	d1
	a2	b2	c2	d2
	a1	b2	c2	d1
	a2	b1	c1	d2
	a1	b2	c1	d2
	a2	b1	c2	d1

Pair AB	Input Parameters			
	A	B	C	D
Possible Combinations	a1	b1	c1	d1
	a1	b2	c2	d2
	a2	b1	c1	d1
	a2	b2	c2	d2

Pair AC	Input Parameters			
	A	B	C	D
Possible Combinations	~~a1~~	~~b1~~	~~c1~~	~~d1~~
	~~a1~~	~~b2~~	~~c2~~	~~d2~~
	~~a2~~	~~b1~~	~~c1~~	~~d1~~
	~~a2~~	~~b2~~	~~c2~~	~~d2~~

Pair AD	Input Parameters			
	A	B	C	D
Possible Combinations	~~a1~~	~~b1~~	~~c1~~	~~d1~~
	~~a1~~	~~b2~~	~~c2~~	~~d2~~
	~~a2~~	~~b1~~	~~c1~~	~~d1~~
	~~a2~~	~~b2~~	~~c2~~	~~d2~~

Pair BC	Input Parameters			
	A	B	C	D
Possible Combinations	~~a1~~	~~b1~~	~~c1~~	~~d1~~
	a2	b1	c2	d2
	a1	b2	c1	d1
	~~a2~~	~~b2~~	~~c2~~	~~d2~~

Pair BD	Input Parameters			
	A	B	C	D
Possible Combinations	~~a1~~	~~b1~~	~~c1~~	~~d1~~
	~~a2~~	~~b1~~	~~c2~~	~~d2~~
	~~a1~~	~~b2~~	~~c1~~	~~d1~~
	~~a2~~	~~b2~~	~~c2~~	~~d2~~

Pair CD	Input Parameters			
	A	B	C	D
Possible Combinations	~~a1~~	~~b1~~	~~c1~~	~~d1~~
	a2	b2	c1	d2
	a1	b1	c2	d1
	~~a2~~	~~b2~~	~~c2~~	~~d2~~

Figure 2-4 The second suggestion to replace X values

Referring to Figure 2-4, 8 test cases are generated after removing the redundant test cases. The generated test cases are shown in Table 2-3.

Table 2-3 Test suite resulted by the second suggestion

	Input Parameters			
Test cases	A	B	C	D
	a1	b1	c1	d1
	a1	b2	c2	d2
	a2	b1	c1	d1
	a2	b2	c2	d2
	a2	b1	c2	d2
	a1	b2	c1	d1
	a2	b2	c1	d2
	a1	b1	c2	d1

The comparison between Table 2-2 and Table 2-3 show slight differences in size of the generated test suites. The first suggestion of X values in Figure 2-2 resulted in 6 cases while the second suggestion resulted in 8 test cases although both cover all pairwise combinations between input values.

In our example, reduction (from 16 to 6 or 8 test cases) might not seem much impressive, however, it can be much more significant when applying pairwise checkpointing on system with large configuration such system with ten parameters each with ten possible input values. For such system, exhaustive testing requires executing 10^{10} test cases while some pairwise strategies managed to minimize the test suite size to 160. The saving here is significantly large with over 75 %.

2.1.3 Benefits of Pairwise Testing

Pairwise testing has proven its efficiency to detect fault among tested software or even hardware. In fact, many earlier works have focused on examining the effectiveness of pairwise testing. Cohen et al. have applied pairwise testing on 10

Unix commands in order to measure the block coverage. As a result of that, Cohen *et. al*. found out that the generated pairwise test suite managed to achieve more than 90 percent block coverage (Cohen *et. al.*, 1996). One year later, Cohen *et. al*. have redone the same experiment, but this time they compared the block coverage between randomly generated test suite and a pairwise test suite. As result of comparison, the pairwise test suite managed to achieve higher coverage than the randomly generated test suite (Cohen *et. al.*, 1997). Burr and Young applied 100 pairwise test cases on subset Nortel's e-mail system that requires 27 trillion exhaustive test cases. Their study showed that the 100 test cases managed to cover 97 percent of branches in the mail system (Burr & Young, 1998).

Although exhaustive testing might be the only way to confirm that all faults in tested software will definitely be found, performing such testing is often not practical. Dunietz *et. al*. have stated that pairwise testing can achieve block coverage that is comparable to exhaustive testing (Dunietz *et. al.*, 1997). In order to illustrate the time saved when applying pairwise testing, instead of exhaustive testing, Huller stated that companies using pairwise testing reduce testing schedule by 68 percent, and save labor costs around 67 percent (Huller, 2000).

Another relevant study, demonstrating the effectiveness of pairwise testing, has been reported by Wallace and Kuhn. The results showed that 98 percent of faults can be detected using pairwise testing for medical device software (Wallace & Kuhn, 2001).

Often, pairwise testing has been successfully applied in real life. Chaung et al. applied pairwise testing to benchmark between radio frequency identification components