DISTRIBUTED T-WAY TEST GENERATION STRATEGIES USING TUPLE SPACE APPROACH

ZAINAL HISHAM BIN CHE SOH

UNIVERSITI SAINS MALAYSIA

2013

DISTRIBUTED T-WAY TEST GENERATION STRATEGIES USING TUPLE SPACE APPROACH

by

ZAINAL HISHAM BIN CHE SOH

Thesis submitted in fulfillment of the requirements
for the degree of
Doctor of Philosophy

July 2013

ACKNOWLEDGEMENT

"All praises and thanks to ALLAH"

First and foremost, I would like to give Glory to God Almighty for His Grace and help in all my endeavors and for bringing me this far in my educational life.

I would like to express my sincere appreciation and heartfelt thanks to my supervisor, Prof. Dr. Kamal Zuhairi Zamli, for his creative guidance throughout this research work, his intellectual support and constructive criticisms greatly enhanced this thesis.

Also, I would like to thank my colleagues, En. Syahrul Afzal, Dr. M.I. Younis and Dr. Asari, for their constructive comments and invaluable advice.

Great thanks from my heart to my mother for her prayers, my wife for her patience and support, my sons and daughter for their patience too, and my father-in-law, mother-in-law, brothers, sisters, friends and UiTM colleagues for their encouragements.

Finally, I would like to thank all lecturers, administration and staff of Universiti Sains Malaysia and all academic and non-academic staff of the School of Electrical and Electronics for their help and support.

TABLE OF CONTENTS

LIST OF FIGURES

Figure No. **Title of Figure** **Page**

LIST OF TABLES

Table No. **Title of Table** **Page**

LIST OF ABBREVIATIONS

Abbreviation	Meaning
ACA	Ant Colony Algorithm
AETG	Automatic Efficient Test Generator
API	Application Programming Interface
CGPA	Cumulative Grade Point Average
CPU	Central Processing Unit
CTS	Combinatorial Test Service
DDA	Deterministic Density Algorithm
EDA	Event Driven Architecture
GA	Genetic Algorithm
GSC	Gigaspaces Service Container
GSM	Gigaspaces Service Manager
G_MIPOG	Grid_MIPOG
IDE	Integrated Desktop Environment
IMDG	In-Memory Data Grid
IPC	Inter-process communication
IPO	Input Parameter Order
IPOG	Input Parameter Order Generalized
ITCH	IBM's Intelligent Test Case Handler
JDK	Java Development Kit
JVM	Java Virtual Machine
MC_MIPOG	Multi_Core MIPOG
MIPOG	Modified Input Parameter Order Generalized
MPI	Message Passing Interface

Abbreviation	Meaning
PC	Personal Computer
POJO	Plain Old Java Object
PU	Processing Unit
PVM	Parallel Virtual Machine
RAM	Random Access Memory
RMI	Remote Method Invocation
SA	Simulated Annealing
SOA	Service Oriented Architecture
SUT	System Under Test
TCAS	Traffic Collision Avoidance System
TCG	Test Case Generator
TConfig	Test Configuration
TDIM	Test Data Input Modelling
TSE	Test Suite Execution
TSEV	Test Suite Evaluation
TSP	Test Suite Parser
TS_OP	Test Suite Generation One Parameter
TS_OT	Test Suite Generation One Test
TVG	Test Vector Generator
WAN	Wide Area Network
XAP	eXtreme Application Platform

STRATEGI-STRATEGI PENJANAAN UJIAN T-HALA TERAGIH MENGGUNAKAN PENDEKATAN RUANG TUPLE

ABSTRAK

Ketika menjana data ujian *t*-hala (di mana *t* merujuk kepada kekuatan interaksi) untuk sistem perisian yang besar dan kompleks, bilangan interaksi antara komponen-komponen perisian yang perlu dicari dan diliputi bagi *t*-hala yang lebih tinggi akan menjadi lebih besar dan boleh membawa ke arah masalah letupan bergabungan. Selain daripada menjadi masalah NP-lengkap, kerumitan pengiraan untuk menjana data ujian *t*-hala juga menjadi lebih sukar apabila nilai *t* bertambah. Bilangan kes ujian yang terhasil dalam data ujian juga meningkat secara mendadak apabila nilai kekuatan interaksi, *t* bertambah. Oleh itu, penjanaan data ujian untuk pengujian *t*-hala dengan pembolehubah masukan yang besar dan kekuatan interaksi yang tinggi akan memerlukan kuasa pengkomputeran dan ruang memori yang tinggi.

Baru-baru ini, pelbagai strategi penjanaan ujian *t*-hala yang berguna telah dilaksanakan mengunakan algoritma berjujukan di atas mesin tunggal. Walaupun membantu, kuasa pengkomputeran dan ruang memori mesin tunggal didapati tidak mencukupi apabila berurusan dengan pembolehubah masukan yang besar dan kekuatan interaksi yang tinggi. Selain daripada itu, kebanyakan strategi yang ada bagi penjanaan data ujian *t*-hala tidak dapat memanjangkan kerja pengiraan daripada mesin tunggal kepada persekitaran mesin pelbagai.

Dalam usaha menangani isu-isu di atas, penyelidikan ini berjaya membangunkan dua strategi pengujian *t*-hala teragih melalui dua pendekatan; Penjana Ujian Satu Parameter (TS_OP) berdasarkan kepada pendekatan "satu-parameter-pada-satu-masa" dan Penjana Ujian Satu Test (TS_OT) berdasarkan kepada pendekatan "satu-ujian-pada-satu-masa", diatas platform perkongsian memori

teragih yang mengunakan teknologi ruang tuple. Kedua-dua strategi mampu mengagihkan kerja pengkomputeran bagi penjanaan data ujian di kalangan rangkaian PC, seterusnya menyelesaikan masalah rumit tentang pengiraan dan sumber memori terhad semasa penjanaan data ujian.

Satu analisis kompleksiti untuk kedua-dua strategi menunjukkan bahawa pertumbuhan saiz ujian berkait rapat dengan rumusan teori dari segi bilangan parameter, *p*; bilangan nilai parameter, *v* dan kekuatan interaksi, *t*. Satu hasil kecepatan diperolehi bagi kedua-dua strategi itu menunjukkan keberkesanan menggunakan teknologi ruang tuple dalam mengagihkan kerja pengkomputeran bagi penjanan data ujian t-hala. Satu perbandingan penandaarasan dengan strategi yang sediaada dari segi saiz data ujian yang dijana juga menunjukkan bahawa kedua-dua strategi memberikan hasil yang cukup kompetitif.

Selain daripada implimentasi teragih untuk kedua-dua strategi penjanaan data ujian, satu perlaksanan data ujian teragih dan automatik dipanggil pelaksana data ujian (TSE) strategi telah berjaya dibangunkan. Satu kajian kes telah dibuat untuk melaksanakan dan mengukur liputan kod bagi data ujian yang telah dijana menggunakan kedua-dua penjana data ujian; TS_OP dan TS_OT dengan TSE untuk kekuatan interaksi, *t* yang berbeza dalam persekitaran mesin berbilang dan persekitaran mesin tunggal. Kecepatan juga diperolehi daripada segi masa ujian bagi kedua-dua strategi; TS_OP dengan TSE dan TS_OT dengan TSE semasa beroprasi dalam mesin pelbagai berbanding dengan persekitaran mesin tunggal.

DISTRIBUTED T-WAY TEST GENERATION STRATEGIES USING TUPLE SPACE APPROACH

ABSTRACT

When generating a *t*-way (where *t* indicates the interaction strength) test suite for large and complex software systems, the number of interaction between software components to be covered for higher order *t*-way is likely to be huge and potentially leads towards a combinatorial explosion problem. Apart from being an NP complete problem, the computational complexity for *t*-way test suite generation also grows rapidly as the value of *t* increases. The resultant test case number in test suite also increases exponentially as the value of interaction strength, *t* is increases. In this manner, *t*-way test suite generation with large input parameter and high interaction strength require significantly high computational power and memory spaces.

A myriad of useful *t*-way test suite generation strategies have been implemented recently using the sequential algorithm on standalone machines. Although helpful, the computational power and memory space of a standalone machine is arguably insufficient especially when dealing with large input parameters and high interaction strength. Furthermore, most of available strategies on *t*-way test suite generation cannot extend the computing work from standalone machines into a multiple machine environment.

In order to address above mentioned issues, this research has successfully develops two distributed *t*-way test suite generation strategies based on two approaches; Test Suite Generator One Parameter (TS_OP) strategy which is based on "one-parameter-at-a-time" approach and Test Suite Generator One Test (TS_OT) strategy based on "one-test-at-a-time" approach on distributed shared memory platform using tuple space technology. Both strategies are capable of distributing the

test suite generation computing workload among a network of participating PCs, thus solving the intense problem of computation and limited memory resources during test suite generation.

A complexity analysis for both strategies indicate that the test size growth follows closely the theoretical formulation in term of the number of parameter, *p*; the number of parameter value, *v* and the interaction strength, *t*. An encouraging result on speedup is obtained for both strategies thus indicated the effectiveness of using tuple space technology in distributing the *t*-way test suite generation computing work. A benchmarking comparison against existing strategies in terms of generated test size also indicated that both strategy give sufficiently competitive results.

Apart from the distributed implementation of both test suite generation strategies, a distributed and automated test suite execution called TSE strategy has been successfully developed. A case study is also successfully implemented to execute and measure code coverage of the generated *t*-way test suite using both test suite generators; TS_OP and TS_OT with TSE for different interaction strength, *t* on multiple machine environment and single machine environment. Speedup is also obtained in term of testing time for both strategies; TS_OP with TSE and TS_OT with TSE while running in multiple machines as compared to single machine environment.

CHAPTER ONE

INTRODUCTION

Software testing is an important activity in software development lifecycle where it is able to offer confidence and assurance on the quality and reliability of a particular software or system under test (Harrold, 2000). Potentially, it can be an endless process which requires a lot of testing process and effort before a quality software can be ready and marketed to the mass public. Although desirable, an exhaustive and thorough testing is not possible due to limited resources and time to market constraints (McMinn, 2004).

Software testing has many objectives of which three of the most significant ones are validation of software to meet its business requirement, verification of software to meet its technical specification and detection of defects within the software that must be fixed (Bentley, 2005). In order to fulfill the aforementioned objectives, several software testing activities would need to be carried out where all these activities are divided into three main testing process phases (see Figure 1-1).

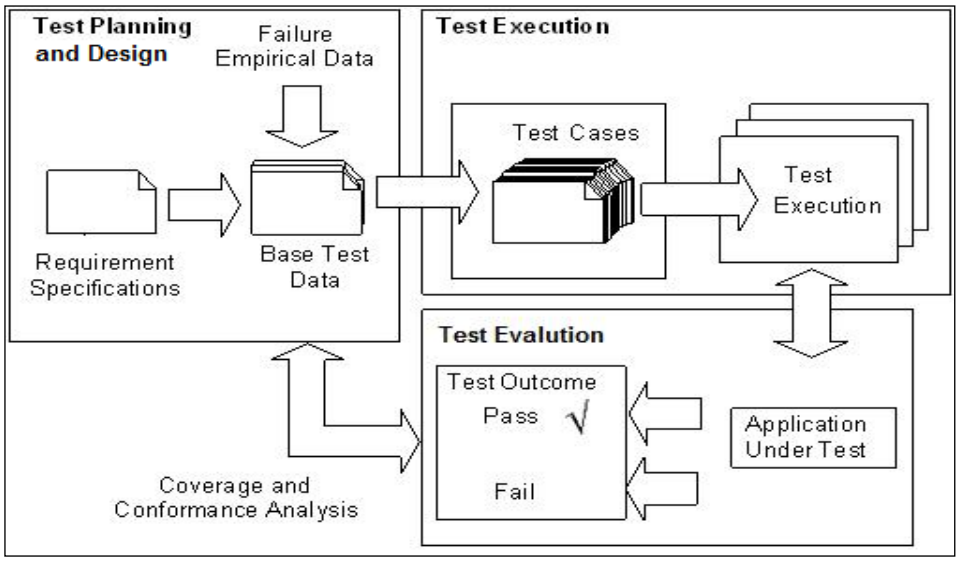

Figure 1-1 : Software Testing Process (Zamli, 2008)

The phases involved in the software testing process include the Test Planning and Design phase, the Test Execution phase and the Test Evaluation phase. Each of these phases would also include sub-activities of the testing process where in the Test Planning and Design phase, the sub-activities are test planning, test control, test analysis and design. The Test Execution phase involved activities such as test preparation and test execution while the Test Evaluation phase involves test evaluation, test reporting and test closure activities.

In the Test Planning and Design phase, the first activity is test planning where the main goal is to establish the plan and scope of testing work. Here, activities involved include identifying the test objective, the test strategy, the available resources and the planned schedule of test activities. To manage the ongoing activity throughout the whole testing process, the test control activity is carried out by comparing the actual progress against the planned schedule. A report on the testing work status including any deviations from the original plan is produced while necessary actions are also taken to ensure the mission and objectives of the testing work are met.

Apart from that, another important activity in the test planning and design phase is test analysis and design, which is concerned with analysis and design of software requirement and specification into actual test conditions and test cases. In this context, test analysis and design involves major tasks like designing, establishing and prioritizing of actual test cases; identifying and prioritizing of test conditions; identifying of necessary test data to support test conditions and test cases; and identifying of any required infrastructure and tools. For designing test case, test case design techniques such as boundary value analysis, equivalent partitioning and

decision table. In order to select, establish and prioritize the test cases, it could either be manually produced or automatically generated using test generation tools.

For the Test Execution phase, there are two important activities known as test preparation and test execution. The test preparation activities involve setting up of the test environment by preparing the test harnesses, writing an automated test scripts for defining test procedures and also verifying the correctness of the set up test environment.

Meanwhile, test execution activities involve running and executing the test cases as per the planned procedure and sequence either manually or automatically using test execution tools. This phase also involves recording the versions of the software under test, logging of the outcome of the test execution as well as comparing the expected outcomes with the actual outcomes. An incident report is then produced for each discrepancy result found.

For each discrepancy result, the test execution may be repeated for confirmation testing and regression testing. Regression testing involves ensuring that fixing an older defect does not lead to new defect in unchanged area or uncover other defects within the software under test. Confirmation testing involves confirming a fix by re-execution of a test which has failed.

In the Test Evaluation phase, test evaluation activities include evaluating the test adequacy, test exit criteria and test closure activities. Evaluating the exit criteria is an activity where the result of test execution is assessed against the defined test objectives and the expected results. The evaluation of exit criteria has the following major tasks that involve checking the overall test result against the test adequacy criteria specified in test planning and producing a test summary report for stakeholders.

The test closure activities involve gathering all data from completed test activities and checking the planned deliverables were delivered successfully. Apart from that, test closure activities also involve producing the acceptance documentation of the software system, finalizing and archiving the test environment, test infrastructure and the version of software under test. Finally, this will involve the handover of the successfully tested software system to the maintenance organization and also analysis of the lessons learned for future releases.

1.1 Overview of *T*-Way Testing

This section presents the definition of few terminologies used in *t*-way testing, the objectives of *t*-way testing and the interaction coverage analysis. It is important that the related terminologies are understood before going in-depth into *t*-way testing strategies likes input parameter, interaction strength, interaction element group, interaction element, interaction coverage, test case and test suite.

For any system under test, the input parameter consists of a number of parameters (or factors) and their respective associated parameter values (or levels). An input parameter is an array of parameters and its associated value. Interaction strength is the degree of interactions among parameters for a given input parameter. The interaction element group is a unique *t*-way interaction among the parameters for a given input parameter while the interaction element is a unique *t*-way combination of parameter values for a particular interaction element group. The interaction coverage is the number of distinct *t*-way interaction elements covered by one test case from the total number of uncovered *t*-way interaction elements where it is used for evaluating the fitness of each test case during the test suite generation. The test case with the highest interaction coverage is usually selected. Here, a complete test

case is an array of all parameters with one parameter's value for each. A test suite is a set of complete test cases that cover all possible interaction elements in a given input parameter.

As for term *t*-way testing, it can be defined as a software interaction testing method that guarantees interaction coverage of all *t*-way interactions among combinations of a given input parameter values and enable the detection of faults triggered by faulty *t*-way interaction inside a given software system. In order to ensure all faulty interaction among *t* parameters are detected for a given input parameters, every *t*-way combination of values of these parameters must be covered by at least one test case within the *t*-way test suite. Typically, the *t*-way test suite is a set of test cases that cover all possible *t*-way combination of values among *t* input parameters (i.e. automatically generated using a *t*-way test suite generator). Here, a typical *t*-way test suite generator is loosely called a *t*-way strategy.

After understanding the basic terminologies used in *t*-way testing, the objective of *t*-way testing will be elaborated. There are two main objectives for the test suite generation in *t*-way testing which are:

- To generate a test suite that has full interaction coverage that covers all possible *t*-way interaction elements at least once by the generated test cases.(i.e. fitness)
- To minimize the test suite size**,** i.e., to minimize the number of test cases that can cover all *t*-way interaction elements.

In order to achieve both objectives, a list of test case with certain criteria (i.e. highest coverage) of the interaction coverage is then selected. As one test case can only covers a number of *t*-way interaction elements, additional test cases are then

required to cover other uncovered *t*-way interaction elements. The interaction coverage analysis is then used to determine the additional test cases by evaluating the fitness of each test case during the test suite generation.

To illustrate this, an input parameter consists of four parameters, *p* and each parameter has two values *v* are used. The specified interaction strength, *t* is 3. The test suite *S* is a sub-array of *T* and consists of a number of test cases, where *T* is the array of all possible test cases. Each test case (in the test suite) is in the form of [v_{p0} v_{p1} v_{p2} v_{p3}] having one value for each parameter.

Assuming that test suite *S* is currently having just two test cases for four parameters: {[2 2 2 2], [2 1 2 2]} and each of the parameters can take two possible values, namely 1 or 2. Then, [2 2 2 2] is the test case, T1 and [2 1 2 2] is test case, T2. In this case, the interaction coverage for set of 3-way interaction elements covered by the test suite in hand will be calculated as follows.

N_1 = {[2 2 2 X], [2 2 X 2], [2 X 2 2], [X 2 2 2]}

N_2 = {[2 1 2 X], [2 1 X 2], [2 X 2 2], [X 1 2 2]}

where N_i is the distinct 3-way interaction elements covered by test case *i* and X is *don't care*. Accordingly, the total number of distinct 3-way interaction element covered by test case T1 is 4. As for test case T2, the number of covered interaction elements is only 3 now since the interaction element, [2 X 2 2] is already covered by T1. The overall number of covered interaction elements for test suite *S* = 4+4-1=7. The total number of all possible interaction elements can be calculated using formula as follows (Younis, 2010) :

$$\binom{p}{t} v^t = \frac{p!}{t!\,(p-t)!} v^t \ldots\ldots\ldots\ldots\ldots\ldots\ldots (1)$$

where *p* is the number of parameter, *v* is the number of parameter value and *t* is the interaction strength.

As an example, the total number of all possible 3-way interaction elements =

$$\binom{4}{3} 2^3 = \frac{4!}{3!(4-3)!}\ 2^3 = 32$$

The overall 3-way interaction coverage ratio for the test suite, *S* {[2 2 2 2], [2 1 2 2]} in which for this particular case is 7/32 = 0.22. Hence, if we have another test suite, {[l 2 2 2], [1 1 1 1]} would have an overall interaction coverage ratio of 8/32 = 0.25. This is due to each test case within the test suite having 4 number of distinct 3-way interaction elements which comparatively better than test suite *S* in term of interaction coverage.

In order to achieve full 3-way interaction coverage, all 32 interaction elements need to be covered by test cases within the test suite. For instance this test suite has a full 3-way interaction coverage, *iec* = 1, {[1 1 1 1], [1 1 2 2], [1 2 1 2], [1 1 2 1], [2 1 1 2], [2 1 2 1], [2 2 1 1], [2 2 2 2]}. However, full 3-way interaction coverage can be achieved with a different set of test cases depending on the strategy used.

An optimal size of test suite (minimum number of test cases within the test suite) is achieved when each *t*-way interaction element is covered only once by any of the test case within the test suite. If there are two or more test cases cover the same interaction element, then the test suite size may not potentially be the most optimal.

1.2 Example of *t*-Way Testing Approach

A simple customizable software system is used here to illustrate the idea of *t*-way testing for software interaction. Figure 1-2 represents the topology of an online mobile share trading system. This system enables the share trader to buy or sell stock online via a stock broking bank using a mobile device such as smart phones and tablets at anytime and anyplace. The system may use different components or

parameters. The modeling of test input data could be done using boundary value analysis, or equivalence partitioning of uniform and mixed parameter data.

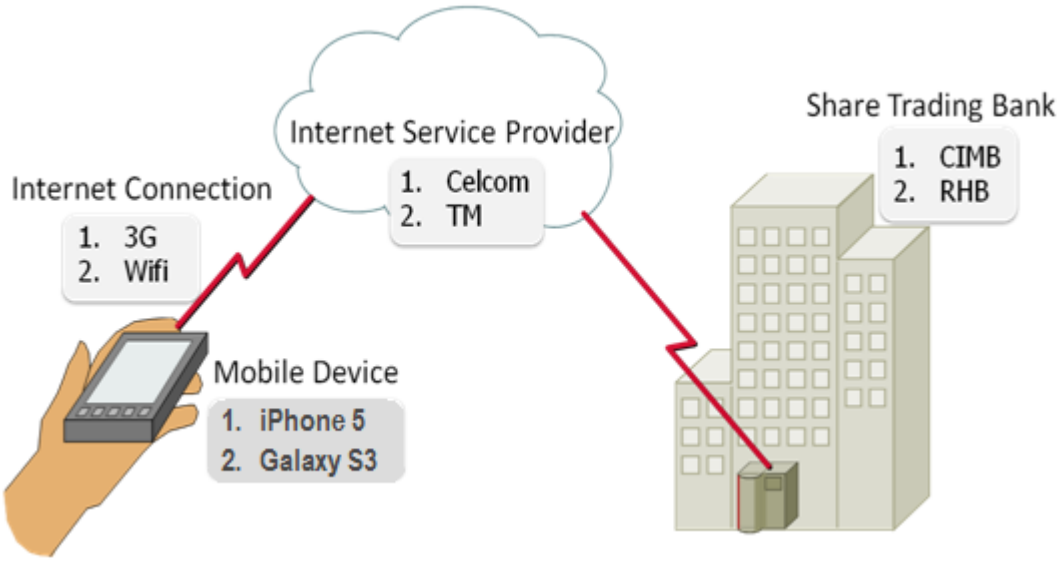

Figure 1-2 : Mobile Share Trading System

In this example, the system consists of four parameters. The mobile share trader user can use smart phones such as iPhone 5 and Galaxy S3 to carry out the functions. For any selected mobile device, the share trading platform at CIMB or RHB securities is assessable through internet connection via 3G or Wifi provided by mobile internet service provider such as Celcom and TM. Hence, the system in Figure 1-2 can be summarized as a four-parameter system with two values as described in Table 1-1.

Table 1-1 : Share Trading System components and configurations

Input Parameter	**Components or Parameters**			
	Internet Connection	Mobile Device	Internet Service Provider	Securities
Configurations or Parameter value	3G	iPhone 5	Celcom	CIMB
	Wifi	Galaxy S3	TM	RHB

The example demonstrates the test case reduction in *t*-way testing. In this case, the range of acceptable *t* is between two to four which is the maximum parameter number. Tables 1-2, 1-3 and 1-4 show examples of the generated test cases for *t*-way testing with varying *t* value of four, three and two respectively. The interaction strength, *t*=4 could also be referred to as exhaustive testing coverage as it involves explicit enumeration of all possible interaction element combinations of their input parameters where in this case the number is 16 test cases (i.e., 2^4).

Table 1-2 : Test suite for *t*=4 for Share Trading System.

Test Case No.	Internet Connection	Mobile Device	Internet Service Provider	Securities
1	3G	iPhone 5	Celcom	CIMB
2	3G	iPhone 5	Celcom	RHB
3	3G	iPhone 5	TM	CIMB
4	3G	iPhone 5	TM	RHB
5	3G	Galaxy S3	Celcom	CIMB
6	3G	Galaxy S3	Celcom	RHB
7	3G	Galaxy S3	TM	CIMB
8	3G	Galaxy S3	TM	RHB
9	Wifi	iPhone 5	Celcom	CIMB
10	Wifi	iPhone 5	Celcom	RHB
11	Wifi	iPhone 5	TM	CIMB
12	Wifi	iPhone 5	TM	RHB
13	Wifi	Galaxy S3	Celcom	CIMB
14	Wifi	Galaxy S3	Celcom	RHB
15	Wifi	Galaxy S3	TM	CIMB
16	Wifi	Galaxy S3	TM	RHB

Table 1-3 : Test suite for *t*=3 for Share Trading System

Test Case No.	Internet Connection	Mobile Device	Internet Service Provider	Securities
1	3G	iPhone 5	Celcom	CIMB
2	3G	iPhone 5	TM	RHB
3	3G	Galaxy S3	Celcom	RHB
4	3G	Galaxy S3	TM	CIMB
5	Wifi	iPhone 5	Celcom	RHB
6	Wifi	iPhone 5	TM	CIMB
7	Wifi	Galaxy S3	Celcom	CIMB
8	Wifi	Galaxy S3	TM	RHB

Table 1-4 : Test suite for *t*=2 for Share Trading System

Test Case No.	Internet Connection	Mobile Device	Internet Service Provider	Securities
1	3G	iPhone 5	Celcom	CIMB
2	3G	Galaxy S3	TM	RHB
3	Wifi	iPhone 5	TM	CIMB
4	Wifi	Galaxy S3	Celcom	RHB
5	Wifi	Galaxy S3	Celcom	CIMB
6	3G	iPhone 5	TM	RHB

Using an existing strategy, IPOG (refer to Chapter 2 for IPOG reviews), for *t*=3, the test cases can be systematically reduced to only 8 test cases and adheres to the 3-way interaction coverage. As for *t*=2, this can be further reduced to 6 test cases through the pairwise testing. However, the number of generated test cases varies depending on the *t*-way strategy used. The working example of the same input parameters with *t*=3 and *t*=2 can be found in (Rozmie et al., 2011) that produce 13 and 9 test case respectively. Therefore, using an efficient *t*-way testing strategy can systematically reduce the test case number while maintaining adequate interaction coverage.

To put the main work undertaken into perspective, the next sections will discuss the problem statement, define the aim and objectives of research works, scope of research as well as outlines the roadmap of the thesis.

1.3 Problem Statements

For a highly configurable and customizable software system like web applications (Wenhua et al., 2009, Wenhua et al., 2008, Sampath et al., 2008, Cohen et al., 2007), many interactions between software components need to be tested to ensure the quality and reliability of the system. Many recent evidences suggest that unwanted

interactions between software system parameters, operating system and the hardware environment can cause undesirable failures. The *t*-way testing strategy is often sought after to address this problem.

Myriad of available *t*-way testing strategies have been reviewed in literature in the past few years. However, most of the available work on *t*-way strategies (e.g. AETG (Cohen et al., 1997), TCG (Yu-Wen and Aldiwan, 2000), IPOG (Lei et al., 2007), MIPOG, Dens (Bryce and Colbourn, 2009), Jenny, ITCH, TVG, CTS (Hartman and Raskin, 2004), and TConfig (Williams, 2000)) employed sequential algorithm and were implemented on a standalone machine for generating the *t*-way test suite. Although useful, sequential algorithms and standalone machine can be counter-productive particularly when dealing with large input parameters and high interaction strength, *t*.

Most of the recent significant software system is commonly made of millions of line code with software size growing rapidly from megabytes to terabytes due to high demand from the mass public for new software applications, add-on features and more advanced software systems. Large size, complexity of software and rapid growth result in possibilities of new intertwines dependency among software input parameter or component involved thus justifying the need to support for high interaction strength. Furthermore, there are a few works (Kuhn et al., 2004, Kuhn et al., 2009, Kuhn et al., 2008) that indicate the needs for higher interaction strength to effectively detect faults in System under Test (SUT).

Moreover, for real, large and complex software systems, the sampling search space of higher order *t*-way interaction between parameter values is likely to be huge and hence, leading towards a combinatorial explosion problem. The resultant number of test case in the test suite also increases exponentially as the value of interaction

strength, *t* increases. Furthermore, producing a minimum *t*-way test suite for high interaction strength is a NP complete problem (Williams and Probert, 2001, Shiba et al., 2004, Kuo-Chung and Yu, 2002). Therefore, generating an optimal *t*-way test suite for higher order *t*-way with large input parameter requires significantly high computational power and memory resources.

Apart from generating an optimal *t*-way test suite for high interaction strength, an automated integration of test suite generation and test suite execution is also important to expedite the testing work. However, much of the existing works seems to focus only on test suite generation processes and an automated mapping of test data between test suite generation and test suite execution appears to be lacking (Zamli et al., 2011). Here, automatic mapping of data between test generation and test execution is highly desirable. Testers can focus more on the creative task of identifying and generating quality test cases rather than focusing on the repeatable task of manual data mapping process.

Recent advancement in parallel and distributed computing offers an alluring prospect to overcome the intense computation and limited memory resources problems during the test suite generation and execution where the distributed computing work in the network of participating workstations on different physical machines can harness more computing power and larger memory space. Currently, there are a number of strategies that proposed (Calvagna et al., 2009) and implemented using parallel and distributed processing such as G_MIPOG (Younis *et al.*, 2008) and MC-MIPOG (Younis and Zamli, 2010). Although useful, the existing parallel processing strategies have a few drawbacks. Both strategies implementations involve significant inter-process communication (IPC) for coordination, creation and deletion of combinatorial, horizontal extension and vertical extension threads. Both

strategies also use tightly coupled computation in generating the test suite; therefore sudden failure involving any of the connected threads could halt the computation of test suite generation work or produce wrong test suite results. Furthermore, both strategies are implemented using low level application programming interface (API) and required lots of knowledge and expertise on multithread programming and also require sound programming knowledge to implement on multiple operating system platforms.

In order to address the above issues, two distributed *t*-way test suite generation strategies, called the TS_OP strategy (i.e. based on the "one-parameter-at-a-time" approach) and the TS_OT strategy (i.e. based on the "one-test-at-a-time" approach) have been developed. Both test strategies are integrated with a distributed test suite execution called TSE. The distribution of the computing work is implemented using the Tuple Space middleware known as GigaSpaces (GigaSpaces, 2011). While running on multiple machines environments, the available memory resources of the network of workstations could be combined while more computational power could be garnered from the CPUs of the connected workstations to carry out the distributed computing's workload of the test suite generation and execution.

1.4 Thesis Aim and Objectives of the Study

The main aim of this research work is to develop and evaluate two distributed *t*-way test suite generation strategies based on the "one-parameter-at-a-time" approach called TS_OP and "one-test-at-a-time" approach called TS_OT on tuple space. Both strategies also address distributed test suite execution. The objectives of this research are as follow:

- To implement and investigate the performance of a distributed *t*-way test suite generation strategy called TS_OP.
- To implement and investigate the performance of a distributed *t*-way test suite generation strategy called TS_OT.
- To implement and investigate the performance of a distributed and automated test suite execution called TSE for both test suite generation strategies, TS_OP and TS_OT.

1.5 Scope of Research

The scope of this research works is limited to the implementation of two distributed test suite generation strategies using "one-parameter-at-a-time" approach and "one-test-at-a-time" approach and a distributed test suite execution strategy, both using a tuple space platform. The implementation is done on networked of Microsoft Window OS based PCs with GigaSpaces middleware installed in each PC.

For both TS_OP and TS_OT strategies, the implementation is carried out from one PC until maximum number of ten PCs used. Both strategies is tested with uniform and mixed input parameter data for their complexity analysis, scalability analysis and benchmark analysis with existing strategies. Scalability analysis is done in both single and multiple machine environments whereas the complexity and benchmark is only carried out for single machine environment.

Furthermore, the focus of this research work is the implementation of a distributed test suite execution, TSE. A case study is done on test suite execution using TSE with both test suite generation strategies, TS_OP and TS_OT for analysis of code coverage with varying interaction strength, *t* from 1 to 6 in one to six machine environment.

1.6 Thesis Outline

This thesis is structured as follows which is divided into six chapters. Chapter 1 begins with an explanation of software testing and the software testing process while the overview of *t*-way testing and example of *t*-way testing approach were also presented. The chapter then provides a brief explanation on the problem statement and laid out the main goal of this research and research objectives; scope of research and outlined the thesis content.

Chapter 2 provides a related works on *t*-way testing for test suite generation using two approaches which are "one-parameter-at-a-time" strategy and "one-test-at-a-time" strategies respectively. Apart from that, the related works on the test suite execution for generated *t*-way test suite and tuple space technology were also discussed.

Chapter 3 outlines the research methodology that has been carried out in term of literature review, feasibility study of available distributed shared memory, development of test generation strategies and test execution strategy. The experimental setup of each strategy is also presented in this chapter.

Chapter 4 provides detailed description of the design and implementation of two distributed *t*-way test suite generation strategies and a distributed test suite execution with a complete algorithm for each implemented strategy. This includes the distributed design consideration design which adopts the tuple space technology while the scaling of the processor to distribute the computing power is also discussed.

Chapter 5 discusses the detailed results from both single machine environment and multi machine environment for all strategies developed. The result in terms of test size of generated test suite and their generation times have been

recorded and discussed. The result was used to assess the optimality of test suite generated and the scalability analysis in term of the speedup gain on a multi machine environment. A case study was also undertaken to evaluate the distributed test suite execution performance in terms of code coverage and speedup gained in a single and multiple machine environments.

The conclusion of this research is explained in Chapter 6 where the achievements, contributions and problems are summarised. Conclusions are drawn from the experience gained from this work and the significance of findings along with a consideration for future works.

CHAPTER TWO

LITERATURE REVIEW

The software testing process phase was elaborated in the previous chapter highlighting a systematic interaction testing known as *t*-way testing. In a nut shell, *t*-way testing helps to reduce the number of test case from an exhaustive search space and ensure fault detection due to faulty software interaction.

Extending from the material in Chapter 1, this chapter will first describe the related work on *t*-way testing comprising of test suite generation and test suite execution. After that, a review on distributed processing approach and tuple space technology is presented. Finally, the last section summarizes the main points presented in this chapter.

2.1 Related Work on *t*-way Testing

There have been several research works on *t*-way testing especially on test suite generation strategies. However, useful and complete *t*-way testing strategies also need to comprise of both *t*-way test suite generation and test suite execution. In general, there were two adopted approaches for the generation of *t*-way test suites; either computational or algebraic approaches (Lei et al., 2007). A computational approach can be applied to any input parameter configurations such as uniform and mixed input parameter, but the computation of test suite generation can be intensive. On the other hand, algebraic approach usually involves only lightweight computations and in some cases, it can produce optimal test suites. However, this approach often imposes restrictions on input parameter configurations to which they can be applied.

In algebraic approach, the test suite is constructed using pre-defined rules using mathematical function such as Latin Square (Mandl, 1985, Stevens et al., 1998), Orthogonal Array (Burroughs et al., 1994, Chateauneuf et al., 1999, Hedayat, 1999), Covering Array (Martirosyan and Trung, 2004, Yin, 2003, Nurmela, 2004, Colbourn et al., 2006b, Colbourn et al., 2006a, Cohen et al., 2008, Sherwood, 2008, Danziger et al., 2009) and Graph Theory (Meagher and Stevens, 2005) to produce a *t*-way test suite. Other algebraic approaches are based on the idea of recursive construction based on orthogonal arrays, which allows larger test sets to be constructed from smaller ones such as TConfig (Williams, 2000) and (Aguirre et al., 2009).

As for the computational approach, there are a few search techniques that can be utilized for generation of *t*-way test suites such as the artificial intelligence approach and the pure computational approach. These approaches often rely on generating all possible interaction elements and search the uncovered interaction element combinations to generate the test suite until all interaction elements are covered. The computational approach can be further divided into:

2.1.1 Artificial Intelligence Search

The artificial intelligent technique usually starts from a pre-existing test suite and then apply a series of transformations using a fitness function to determine the test suite until a complete test suite is reached that covers all the combinations. Strategies that adopted artificial tracking techniques such as GAPTS (McCaffrey, 2009), Tabu Search (Nurmela, 2004, Walker Ii and Colbourn, 2009), Ant Colony Algorithm (ACA) (Shiba et al., 2004, Chen et al., 2009), Genetic Algorithm (GA) (Shiba et al., 2004, McCaffrey, 2009), Simulated Annealing (SA) (Cohen et al., 2003), Particle

Swarm Optimization (PSO) (Ahmed and Zamli, 2011) and augmented annealing (Cohen et al., 2008) as proposed in several literature.

Briefly, these strategies start from some known test suite. Then, a series of transformation were applied (starting from the known test suite) until an optimum test suite is reached that covers all the interaction elements. As such, these search techniques can produce a smaller test suite, but they typically take a longer time to complete. In addition to that, they can only support small parameters and values, with low interaction strength which was found by SA, Tabu Search, ACA and GA that reported results with interaction strength of only up to 3-way coverage.

2.1.2 Pure Computational Search

For the pure computational search technique, the *t*-way test suite generation can be further divided into two main categories known as "one-test-at-a-time" approach, which builds the test suite one test case at a time until all interaction elements are covered and the "one-parameter-at-a-time" approach which extends the test case by one parameter at a time until all parameter and interaction elements are covered.

Typical "one-test-at-a-time" is exemplified by the Automatic Efficient Test Generator (AETG) which iteratively builds a complete test case using the greedy search technique until all the interaction element combinations are covered (Cohen et al., 1997, Cohen et al., 1996), TCG (Yu-Wen and Aldiwan, 2000), Dens (Bryce and Colbourn, 2007). In AETG, the first pair of parameter and its value with the greatest occurrences in the uncover interaction element set is first selected. Then, the next parameter order is fixed using a random permutation of the remaining parameter. The next parameter value is selected based on the highest number of new pair covered by their combination with previous parameter value. The selection of parameter and it's

value is then carried out until a completed test case is generated. The resultant test case is then placed into the 50 candidate test case set and then one test case is selected from here and put in the final test suite set. After that, the test case generation is iteratively repeated until all interaction elements are covered. AETG is able to produce a non-deterministic test suite solution due to its random selection of test case.

Other examples of strategies that are in the same group as AETG include TCG, Dens, GTWay, PICT, CTS, ITCH, TVG and Jenny. In TCG, the parameter order is fixed according to the descending order of the parameter value where the number of candidate test case is equal to number of first parameter value. For the first candidate test case, the first value is the first parameter value and the next value is then determined by the number of new pair covered. If a tie occurred among the selected value, the least selected value is chosen to minimise the usage of that value. The individual value selection of each parameter is then continued until a complete test case is built. When all candidate test cases are generated, one test case is selected with the highest new pair covered and will be put into the final test suite. Contradictorily, both TCG will produce a deterministic test suite results due to the fixed rule in generating a test case that covers as many as possible of uncovered interaction elements in their greedy search of maximum interaction coverage.

Bryce et al. develop an enhanced DDA with support for higher interaction strength for *t*-way testing (Bryce and Colbourn, 2009). In Dens, Bryce et al a higher strength test suite generation using the greedy algorithm to select the parameter value based on their density value was developed. The selected value is inserted into the test case. The next value is selected based on their density until a complete test case is constructed. Due to the random insertion of first value into the test case, a higher

strength DDA is unable to produce a deterministic test suite results as its predecessor DDA.

Zamli et al. develop the GTWay (Zamli et al., 2011) by merging the interaction element based on their interaction element group to construct the test case with aims of higher interaction coverage. The GTWay also provides an execution support for automatic execution of the generated test suite.

Czerwonka develops a freeware tool named PICT where its core algorithm is based on the greedy algorithm and was similar to AETG with key differences that PICT is deterministic and does not produce any candidate test. On top of that, PICT also had rich features such as support variable strength generation, support constraint and seeding (Czerwonka, 2006).

Hartman et al. develop the Combinatorial Test Services (CTS) package that construct a *t*-way test suite using the direct and recursive construction algorithm. In solving the *t*-way test suite construction, the CTS package tries several alternatives and chooses the smallest array constructed. All *t*-way test suites with similar input configuration always produced similar sizes for each new construction because all the algorithms employed are deterministic (Hartman and Raskin, 2004).

An extension of CTS known as the IBM's Intelligent Test Case Handler (ITCH) is available in the Eclipse Plug-in tool(ITCH, 2010). ITCH is an IBM's Intelligent Test Case Handler that uses a combinatorial approach based on exhaustive search to generate the test cases. ITCH uses a sophisticated combinatorial algorithm based on combination of mathematical and greedy search to construct the test suites, thus, requiring substantial time to complete.

Both TVG and Jenny support *t*-way testing but there are only limited information regarding both strategy implementations found in written literature. Both

of the strategy implementation can be downloaded for free at their respective web site which are Test Vector Generator (TVGII, 2010) and Jenny (Jenny, 2010).

Another approach of pure computational strategy is categorized as the "one-parameter-at-a-time" approach. These are exemplified by IPOG (Lei et al., 2007). The IPOG strategy is generalized from IPO (Lei and Tai, 1998) in which a *t*-way test suite for the first *t* parameters is generated, and then in horizontal extension phase, each test case is added with a new parameter value at the *t*+1 parameter which covers maximum uncovered interaction elements. The newly extended test case is then selected and stored into the new *t*-way test suite. If all test cases are extended and there are still uncovered interaction elements then the vertical extension phase is required. In the vertical extension, a new test case is added into the test suite to cover for the uncovered interaction element combination. After the entire interaction elements are covered, the vertical extension is then completed. If the next parameter *t*+2 exist, then the horizontal phase and vertical phase are resumed for that parameter. The test suite generation will continue until all parameter are covered.

A number of variants has also been developed to improve the IPOG's performance which are: IPOG-D (Lei et al., 2008), IPOG-F (Forbes et al., 2008), IPOG-F2 (Forbes et al., 2008), MIPOG, G_MIPOG (Younis et al., 2008) and MC-MIPOG (Younis and Zamli, 2010). IPOG-D is a deterministic strategy that combines the IPOG strategy with a recursive D-construction to minimize the number of interaction element that needs to be enumerated during the generation of the test suites. The D-construction approach is a recursive procedure that can be used to double the number of parameters in a 3-way test suite. Although IPOG-D can generate the test suite faster than IPOG, the corresponding test size is usually bigger than IPOG.

Both IPOG-F and IPOG-F2 are non-deterministic strategies which implement randomization technique to break ties in the greedy selection during the horizontal growth. In general, the size of test suite generated by both strategies is competitive as compared to IPOG. As for their execution time, they are found to be faster than IPOG. Although both strategies are able to support uniform and mixed input parameter settings, the performance gain do not extend to the mixed input parameter value and IPOG seems likely to do better in such a situation. Unlike IPOG-F, IPOG-F2 is implemented with a heuristic search for horizontal growth algorithm, thus, allowing faster test generation time as compare to IPOG-F.

The MIPOG strategy is a deterministic strategy where each run will produce the same test suite size. Unlike IPOG in horizontal extension, the MIPOG strategy optimizes the extended test case by selecting a value that covers the maximum number of uncovered interaction element combinations. MIPOG also optimizes the *don't care* value by searching for uncovered interaction element that can be covered by the same test case. This is performed by means of exhaustive searching of uncovered interaction element that can be combined with this test case during the horizontal extension. Meanwhile, in the vertical extension, MIPOG creates a new test case by exhaustively search for a combination of interaction elements that covered the most uncovered *t*-way combinations. This process will improve the test suite size as well as increase the overall execution time of MIPOG.

Both G_MIPOG and MC-MIPOG are built based on the MIPOG strategy and can parallelize the test suite generation process. G_MIPOG is implemented on a grid network while MC-MIPOG is run on a standalone Intel multi core machine. Both strategies support higher order of *t* for test suite generation and can produce a smaller test suite as compare to others variant of IPOG.

Useful and complete *t*-way testing strategies should comprise of both the *t*-way test suite generation and test suite execution module. The generated *t*-way test suite can either be manually or automatically integrated for test execution. For manual integration, each generated test case is loaded and executed using an independent test execution tool whereas for automatically integration, each generated test case is parsed into an actual test case value and executed using an integrated test suite execution.

Currently, only GTWay has integrated an automatic test execution within its test generation strategy where the actual input parameter data is parsed into a symbolic data prior to test generation. This would help to improve the performance of the test suite generation. Before test generation begins, the interaction elements are pre-generated and stored in a file. Later, each interaction elements are merged together whenever possible with other interaction elements using backtracking algorithm for test suite generation. The test case is iteratively generated until all interaction elements are covered. The final test suite which consists of generated test case is then parsed into an actual test data and conveyed as the test data input for test suite execution module. The actual test data is loaded as a stub file and executed with software under test one at a time. After that, the test coverage result is obtained and compared for different interaction strength.

Superficially, both the MC-MIPOG and G_MIPOG strategies do address the distributed test suite generation. However, a more thorough investigation revealed a number of limitations. Firstly, both strategies implementation use multithreading that involved many inter-process communications (IPC) for coordination, creation and deletion of combinatorial, horizontal extension and vertical extension thread, which is prone to deadlock.