MIPOG: A PARALLEL T-WAY MINIMIZATION STRATEGY FOR COMBINATORIAL TESTING

BY

MOHAMMED ISSAM YOUNIS AL-KHIRO

**THESIS SUBMITTED TO UNIVERSITI SAINS MALAYSIA
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY**

SCHOOL OF ELECTRICAL AND ELECTRONIC ENGINEERING

October 2010

Acknowledgement

First of all, I would like to present my special thank and gratitude to my supervisor Assoc. Prof Dr. Kamal Zuhairi Zamli. His enthusiasm, constant advice, criticism, and arguments have helped me tremendously in the course of my PhD research. To Dr. Kamal, your great efforts will never be forgotten.

I would also like to thank the staffs in the School of Electrical and Electronic Engineering. To all members in the GRID computing cluster particularly Assoc. Prof Dr. Bahari, and Assoc. Prof Dr. Chan, thank you for providing a robust grid-computing environment for my research work. My sincere thank also goes to Keat for being prompt when fixing occasional network problems.

To Raghu Kacker, Rick Kuhn, Yu Lie, and Tony Opara, thank you for allowing the use of FireEye. To Jeff Offutt, thank you for granting the use of MuJava. To Myra Cohen, thank you for advising to use mutation testing as our case study. To Bob Jenkins, thank you for helping me understand how Jenny works. To Charles Colbourn, thank you for alerting me with the findings in your web site. Finally, no work here could be done without the generous grants from Universiti Sains Malaysia.

Table of Contents

List of Tables

List of Figures and Illustrations

List of Abbreviations and Nomenclature

Abbreviation	Meaning
ACA	Ant Colony Algorithm
ACTS	Advanced Combinatorial Testing Suite
AETG	Automatic Efficient Test Generator
AIDL	Auto IDentification Laboratory
CA	Covering Array
CIT	Combinatorial Interaction Testing
CPU	Central Processing Unit
CTS	Combinatorial Test Service
DDA	Deterministic Density Algorithm
EXACT	EXhaustive seArch of Combinatorial Test suites
G_MIPOG	Grid MIPOG
GA	Genetic Algorithm
GF	Galois Finite field
GUI	Graphical User Interface
IPO	In Parameter Order
IPOG	In Parameter Order General
ITCH	IBM's Intelligent Test Case Handler
JDK	Java Development Kit
MC_MIPOG	Multi_Core MIPOG
MCA	Mixed Covering Array
MIPOG	Modified IPOG
NIST	National Institute of Standards and Technology
OA	Orthogonal Array
OATS	Orthogonal Array Test System
OLS	Orthogonal Latin Square
RAM	Random Access Memory
RE	REcursive algorithm
RFID	Radio Frequency Identification
SA	Simulated Annealing
SDT	Software Development Technologies
SUT	System Under Test
TCAS	Traffic Collision Avoidance System
TCG	Test Case Generator
TConfig	Test Configuration
TOFU	Test Optimizer for Functional Usage
TS	USM RFID Tracking System
TVG	Test Vector Generator
USM	Universiti Sains Malaysia
V&V	Verification and Validation

MIPOG: STRATEGI PENGURANGAN T-HALA SELARI UNTUK PENGUJIAN BERGABUNGAN

Abstrak

Pengujian bergabungan merupakan bidang penyelidikan yang aktif dalam beberapa tahun kebelakangan ini. Salah satu cabaran adalah tertumpu kepada masalah letupan konfigurasi, yang biasanya memerlukan proses pengkomputeran yang sangat mahal bagi mecari sat set ujian berkesan yang merangkumi semua kekuatan interaksi yang dipilih (*t*). Perkomputeran selari boleh menjadi pendekatan yang berkesan untuk menguruskan kos tersebut, iaitu, dengan mengambil kira teknologi terkini arkitek berbilang teras dan GRID.

Bermotivasikan prospek dan cabaran yang digariskan diatas, tesis ini membincangkan tentang rekabentuk, implementasi dan penilaian strategi *t*-hala selari, dipanggil MIPOG (Modified IPOG), berdasarkan strategi yang sedia ada IPOG (In Parameter Order), untuk pengurangan data ujian t-hala secara sistematik. Tidak seperti IPOG, MIPOG menghapuskan kebergantungan antara nilai dan parameter. Hal ini membenarkan MIPOG mengekploitasi perkomputeran selari.

Dua variasi MIPOG diperkenalkan; satu untuk sistem multi-core (MC_MIPOG) dan satu lagi untuk sistem GRID (G_MIPOG). Keputusan daripada eksperimen yang dijalankan menunjukkan strategi ini mengatasi srategi sedia ada (IPOG, IPOD, IPOG, IPOF2, Jenny, TVG, TCOnfig, and ITCH) dalam aspek penghasilan saiz data ujian yang minima dan dalam masa yang berpatutan. Selain itu, strategi ini juga boleh digunapakai untuk kekuatan interaksi yang tinggi dan menyumbang kepada keputusan terbaik dalam literatur

untuk pembolehubah sekata dan tidak sekata. Merujuk kepada perkomputeran selari, MC_MIPOG dan G_MIPOG mengatasi MIPOG dengan pertambahan unit pemprosesan pusat sebagai nod perkomputeran.

MIPOG: A PARALLEL T-WAY MINIMIZATION STRATEGY FOR COMBINATORIAL TESTING

Abstract

Combinatorial testing has been an active research area in recent years. One challenge in this area is dealing with the combinatorial explosion problem, which typically requires a very expensive computational process to find a good test set that covers all the combinations for a given interaction strength (*t*). Parallelization can be an effective approach to manage this computational cost, that is, by taking the recent advancement of multicore and GRID architectures.

Motivated by such alluring prospects and challenges, this thesis discusses the design, implementation, and evaluation of an efficient parallelizable *t*-way strategy, called MIPOG (Modified IPOG) strategy based on its predecessor IPOG (In Parameter Order General) strategy, for systematic *t*-way test data minimization. Unlike earlier work, the MIPOG optimizes and removes the dependencies between parameter and values. In this manner, MIPOG permits the possibility for exploiting parallel computing.

In order to demonstrate the parallel implementation, two variants of MIPOG are introduced; one for the standalone multi-core system called MC_MIPOG (Multi_Core MIPOG) and the other for grid based multi-core environment called G_MIPOG (Grid MIPOG) respectively. Experimental results demonstrate that both of the proposed strategies outperform other existing strategies, in most cases, (e.g. IPOG, IPOD, IPOF, IPOF2, Jenny, TVG, TConfig, and ITCH) in terms of the generated test size with acceptable execution time. Furthermore, unlike other strategies, the proposed strategies

also support high degree of interaction and contribute to the best well-known results for both uniform and non-uniform distribution of variables. As far as parallelism is concerned, MC_MIPOG and G_MIPOG scale well against the sequential MIPOG with the increase of CPUs as computational node.

CHAPTER 1

INTRODUCTION

Nowadays, we are increasingly dependent on software to assist as well as facilitate our daily chores. In fact, whenever possible, most hardware implementations are currently being replaced by software. From the washing machine controllers, mobile phone applications to the sophisticated airplane control systems, our growing dependency on software can be attributed to a number of factors. Unlike hardware, software does not wear out. Thus, the use of software can also help to control maintenance costs. Additionally, software is also malleable and can be easily customized as the need arises.

The fact that software is malleable can be an issue as far as dependability and reliability are concerned. Here, software testing becomes immensely important especially if software is employed in harsh, critical, or life threatening applications. Covering as much as 40 to 50 percent of the total software development costs (Beizer, 1990), testing can be considered one of the most important activities for software validation and verification.

Surprisingly, most of the techniques used in software testing have not changed much. In fact, some techniques even dated back in early 80s. Back then, most significant program were less than 10,000 lines of codes. In those days, it may still be possible for trained testers to develop test suite by reading the whole program line-by-line and identifying all variables in order to trace the key paths of the program. Today, it is common to find

program with a few million lines of code. Due to its complexity, no individual can master the internals of the program implementation completely in order to permit effective source-code-level analysis and testing.

Additionally, to cater for customer demands for more functionality and innovation, the use of software components is also becoming popular trends, hence, subject to testing. Although useful as a way to avoid the need to reinvent the wheels, integrating components into a complete system can also be problematic. Here, there may be undesired interactions that can potentially introduce errors, hence, cause failures.

As one of the most important activities to ensure quality (and minimize if not avoid failure), software testing can be seen as an integral part of the software engineering lifecycle. Lack of testing can lead to disastrous consequences including loss of data, fortunes and even lives. For instance, consider the accident that occurred during the European Space Agency's launching of Ariane 5 in 1996. Investigation by independent researchers from Massachusetts Institute of Technology reveals that the disaster is caused by the mismatch of the hardware and software component faults (Lion, 2005). The component erroneously puts a 64 bit floating point number in to a 16 bit space, causing overflow error. This overflow error affected the rocket's alignment function, and hence, causing the rocket to veer off course and eventually exploded a mere 37 seconds after lift off. This incident wasted billion of Euros as far as satellite equipments and rocket's equipment on board.

Indeed, the aforementioned incident has highlighted the importance of software testing. In order to put the main work undertaken in this research work into perspective, the next section to come highlights an overview of software testing and discusses the problem statement as well as outlines the roadmap of the thesis.

1.1 Overview of Software Testing

Software testing refers to the process of finding errors/defects and/of ensuring that a particular software of interest meets its specification. The objectives for software testing include: find defects and, reduce risks for software failures, prove that the program is good or otherwise, and detect variation from specification in order to establish confidence that a program does what it is supposed to do (Myers, 2004).

In order to fulfill the aforementioned objectives, software-testing activities can be divided into three main stages. By dissecting software testing into its component parts, we can view the function of each and thus more clearly understand the whole testing activities. Termed test cycle, these stages are Test Planning stage, Test Execution stage, and Test Monitoring stage (see Figure 1-1).

As the name suggests, the Test Planning stage involves the planning on how and what kinds of test techniques to be employed as well as the resources involved (e.g. in terms of work force, costing, timing, and tools). Test planning stage also includes consideration on failure empirical data, which is, based on known problems with similar systems. Test

Execution stage involves the activities to define and execute the planned test cases. Finally, Test Monitoring stage involves analyzing coverage as well as monitoring whether or not the test result conforms to the specification.

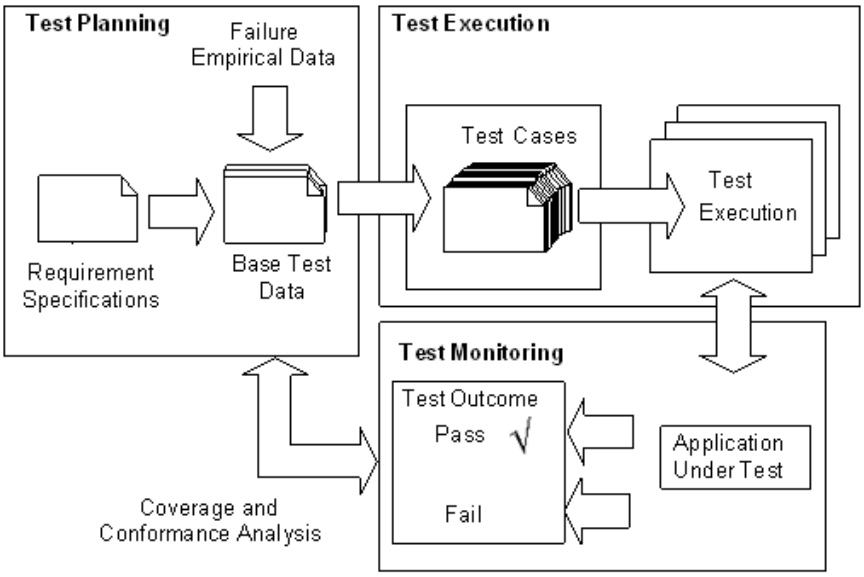

Figure 1-1 Test Cycle

Having described the stages within software testing, there is also a need to understand how software testing fits into the whole software engineering product development lifecycle. Referring to Figure 1-2, software development starts with the requirement elicitation phase. Here, the customers and stakeholders interact with the requirement engineers to produce the software specifications. Based on the specifications, software engineers and programmers collaborate to produce software design and implementations. This activity occurs in the implementation phase. Software testing falls under the

validation phase, which may occur in parallel with the requirement elicitation phase and implementation phase. The independent verification and validation (V&V) team needs to consult the requirement engineers for software specification. Based on the software specification, the V&V team produces the test cases to be executed against the software implementation. If the execution results satisfy the requirement specification, then the software is ready to be released, otherwise, some additional works need to be done to the design and implementation until conformance is achieved.

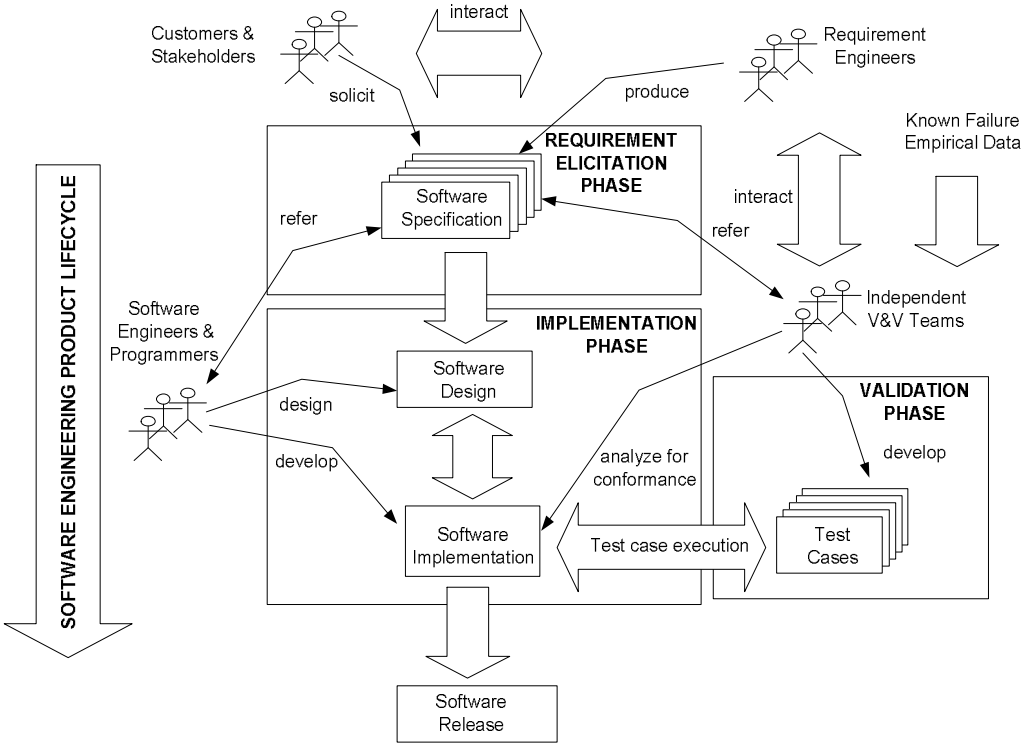

Figure 1-2 Software Engineering Product Lifecycle

Testing is potentially endless, that is, discovering all the defects is practically impossible. At some points, testing must be stopped for shipping the software. The question is when.

Realistically, testing is a trade-off between budget, time and quality. Profit models drive testing. The pessimistic and unfortunately most often used approach is to stop testing whenever some or any of the allocated resources, time, budget, or test cases - are exhausted. The optimistic stopping rule is to stop testing when either reliability meets the requirement, or the benefit from continuing testing cannot justify the testing cost (Yang and Chao, 1995).

Qualified software is designed to follow some degree of reliability and obey some coverage criteria (Mathur, 2008) Referring to Figure 1-3, software that does not follow any coverage criteria or have not been not checked against the specification is undesirable software whilst risky software is the one that has acceptable value of reliability, but lacks acceptable coverage. In the same manner, software that meets coverage goal without some confidence of its reliability is suspected software. Therefore, the desirable software meets both coverage and reliability needs.

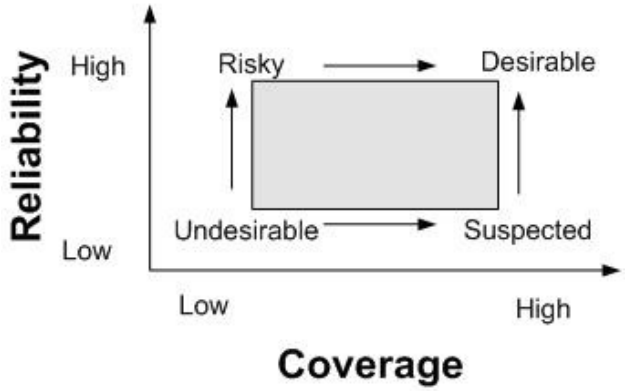

Figure 1-3 Software Reliability and Coverage (Mathur, 2008)

Given a compromise between coverage and reliability, the purpose of testing is not to prove anything, rather to reduce the perceived risk of not working to an acceptable value. Thus, the key challenges in software testing are not only dependent on the actual execution of the test cases but also the production of quality test cases.

In order to ensure acceptable software coverage and reliability, many combinations of possible input parameters, software and hardware environments as well as system configurations need to be tested and verified against for conformance. Although desirable, exhaustive software testing is prohibitively impossible due to resources as well as timing constraints.

As illustration, consider the option dialog Microsoft Internet Explorer software (see Figure 1-4). Even if only Advanced tab option is considered, there are already 54 possible configurations to be tested. With the exception of searching and under line links which take 4 and 3 possible values respectively, each configuration can take two values (i.e. checked or unchecked). Here, there are 2^{54}x4x3 combinations of test cases to evaluate. Assuming that it takes only one second for one test case, then it would require nearly $68x10^7$ years for a complete test of the Advanced tab option.

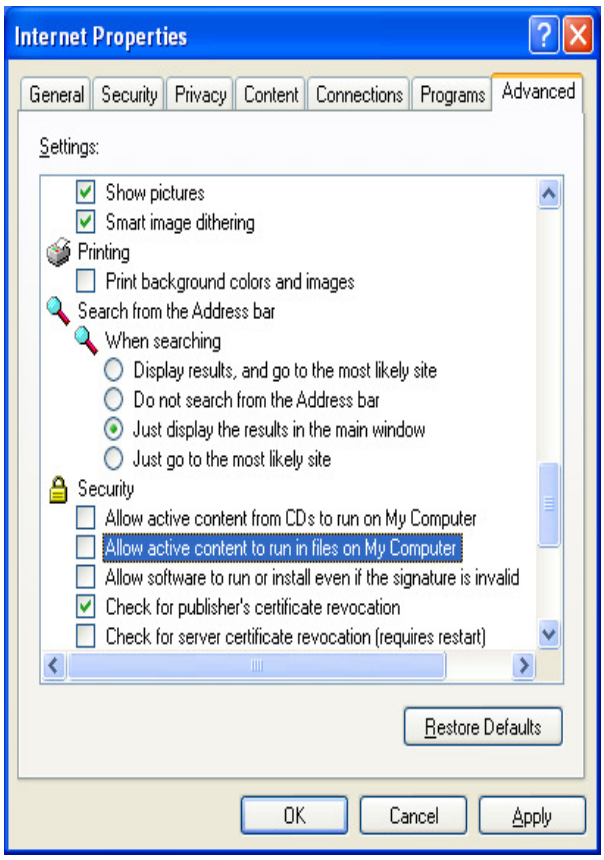

Figure 1-4 Advanced Option Dialog for Microsoft Internet Explorer

Similar situation can be observed when testing a typical hardware product. As a simple example, consider a hardware product with 30 on/off switches. To test all possible combination would require 2^{30} test cases. If the time required for one test case is one second, then it would take nearly 34 years for a complete test.

The two aforementioned examples highlight the combinatorial explosion problem involving software and hardware testing. The most recently systematic solution to this problem based on *t*-way strategy. Here, the *t*-way strategy (also termed Combinatorial

Interaction Testing) can systematically reduce the number of test data by selecting a subset from exhaustive testing combination based on the strength of interaction coverage (t).

Earlier studies (e.g. in (Cohen et al., 1994, Cohen et al., 2003) have suggested that pairwise testing (i.e. based on two-way interaction of variables) are effective in detecting most faults in a typical software system. While such conclusion may be true for some systems, it cannot be generalized to all faults found in a software system, especially when there are significant interactions between variables.

Now, considering more than two parameter interaction is not without difficulties. When the parameter interaction coverage t increases (i.e. as $t > 2$), the number of t-way tuples also increases exponentially. For example, consider a system with 10 parameters, where each parameter has 5 values (i.e. uniform distribution of parameter values). There are 1125 2-way tuples (or pairs), 15,000 3-way tuples, 131,250 4-way tuples, 787,500 5-way tuples, 3,281,250 6-way tuples, 9,375,000 7-way tuples, 17,578,125 8-way tuples, 19,531,250 9-way tuples, and 9,765,625 10-way tuples. The determination of the number of tuples will be discussed in Section 2.1.1.

As another example for non-uniform parameter values, consider the TCAS is an aircraft traffic collision avoidance system from the Federal Aviation Administration which has

been used as case study in other related works (Hutchins et al., 1994, Kuhn and Okun, 2006, Lei et al., 2007, Lei et al., 2008). Here, TCAS module has twelve parameters: seven parameters have 2 values, two parameters have three values, one parameter has four values, and two parameters have 10 values. Running exhaustive test requires 460800 tests (i.e. 10x10x4x3x3x2x2x2x2x2x2x2), or 12 way testing for this system (i.e. running such test may be impossible). Alternatively, 11-way testing requires 230400 test cases. 10-way requires 201601 test cases. 9-way requires 120361 test cases. 8-way requires 56742 test cases. 7-way requires 26061 test cases. 6-way requires 10851 test cases. 5-way requires 4196 test cases. 4-way requires 1265 test cases. 3-way requires 400 test cases. Finally, 2-way requires 100 test cases (see Section 4.1.2).

The two aforementioned examples highlights the difficulties when dealing with high interaction strength (t), that is, test data (and tuples) tend to increase tremendously leading toward combinatorial explosion problem. For this reason, there have been little results reported in the literature on the t-way testing for high t (i.e. t>6).

1.2 Problem Statements

Recent advancement in computing and hardware technologies dictates that software applications need to incorporate many new features and functionalities based on the consumer demands. As such, software applications grew tremendously in size from kilobytes to terabytes. Here, the net effect of software growth can often lead to

intertwined dependency between parameters involved, thus, justifying the need to support for high interaction strength.

Additionally, obtaining an optimal test suite for high interaction strength is an NP hard problem, that is, it is often unlikely that an efficient strategy exists that can always generate minimum number of test cases. Nevertheless, the support for qualified systems can not be a forgone demand. The nature of birthing more complex systems with higher degree of interactions, the requirement of non-sequential, more scalable, optimum minimization strategy that offers a high degree of interaction still dominates.

Achieving high degree of interaction with optimum results altogether requires significant computational costs. Here, parallelization can be useful to manage this computational cost, that is, by taking advantage of the recent advancement of GRID computing and multi-core architecture.

Motivated by these prospects and challenges, this research work is devoted to investigate an efficient parallelizable strategy, called MIPOG (Modified IPOG) strategy based on its predecessor IPOG strategy, for systematic *t*-way test data generation. Unlike earlier work, the MIPOG optimizes and removes the dependencies between parameter values. In this manner, MIPOG permits the possibility for exploiting parallelism. In order to demonstrate the parallel implementation, two variations of MIPOG are introduced; one

for the standalone multi-core system called MC_MIPOG (Multi_Core MIPOG) and the other for grid based multi-core environment called G_MIPOG (Grid MIPOG) respectively.

1.3 Thesis Aim and Objectives

The main aim of this research is to develop and evaluate a new parallelizable *t*-way strategy, called MIPOG, for *t*-way test data generation.

The main objectives of the work undertaken were:

i. To design and implement the MIPOG strategy as a prototype implementation tool. In order to demonstrate the parallel implementation, two variations of MIPOG are introduced; one for the standalone multi-core system (MC_MIPOG) and the other for the grid based multi-core environment (G_MIPOG) respectively.

ii. To investigate the speedup gain from parallelizing MIPOG in MC_ MIPOG and G_MIPOG.

iii. To investigate and evaluate the performance of MIPOG strategy in terms of the number of test data generation as well execution time against existing works.

1.4 Research Scope

As discussed in Section 1.1, test cycle is partitioned into three stages: Test Planning stage, Test Execution stage, and Test Monitoring stage (see Figure 1-1). This research

work is more focus in the intertwined phase between Test Planning and Test Execution, which is test data generation. Specifically, the research work adopting *t*-way minimization strategy for test data generation.

1.5 Research Methodology

With parallelism in mind, the research methodology is partitioned into three phases as follows.

i. Survey the state-of-the-art in t-way test data generation. Based on this survey, identify a suitable candidate strategy for benchmarking and possible parallelization support.

ii. Implement sequential and parallel versions of the modified strategy.

iii. Benchmark and evaluate the implementations.

1.6 A List of Research Contributions

The research contribution from this research work can be stated from different perspectives as follows.

i. MIPOG is the first strategy that permits parallelization through MC_MIPG and G_MIPOG whilst keeping the test suite identical.

ii. Unlike other strategies, MIPOG also permits high degree of interaction for $t>6$.

iii. MIPOG contributes to the well known best results published by Colburn for CAs (see Figure 1-5).

iv. MIPOG contributes to derive optimal test suite for MCAs as compared with other existing strategies.

v. During this research work, a number of international publications has been produced (see the List of the Selected Publications).

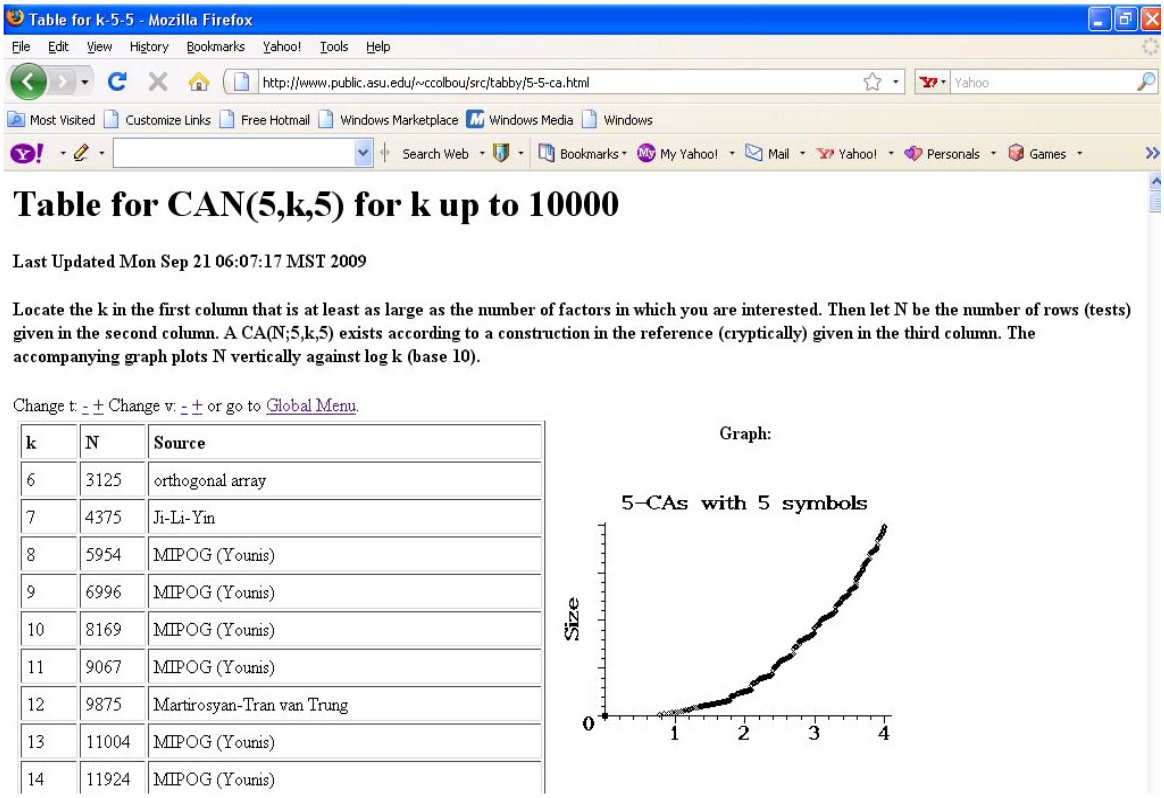

Figure 1-5 MIPOG Contribution to the benchmark results (Colbourn)

1.7 Thesis Outline

This thesis is organized into five chapters as follows.

Chapter 2 sets the scene on the state-of-the art in *t*-way test data generation. It begins with a description of the mathematical foundation on *t*-way testing as background materials. Additionally, the chapter provides a study of combinatorial growth of tuples as a function of increasing the strength of coverage, the number of parameters, and the values of each parameter. Next, a systematic example is considered. The test case generation strategies are discussed, analyzed, and compared. Finally, the chapter ends with the research problems as well as elaborated the proposed research methodology further in order to address such problems.

Chapter 3 explores the design criteria of the MIPOG strategy, and addresses the demands highlighted in Chapter 2. The detailed of the proposed strategies are discussed and implemented as a sequential version (i.e. MIPOG). Next, a Multi-Core version implementation is also highlighted (i.e. MC_MIPOG). Then, a distributed version implementation on the grid (i.e. G_MIPOG) is derived. Finally, the implementation details are sketched.

Chapter 4 involves three parts. Firstly, control experimentations are proposed to investigate the performance of the implemented strategy, in the form of test size and execution time, as the strength of coverage, number of parameters, and values are changed. Secondly, comparisons with other works are also highlighted. Finally, a number of experimentations are undertaken to demonstrate speedup gain against the sequential MIPOG with the increase of CPUs as computational node.

The conclusion of this work is given in Chapter 5, where the achievements, contributions and problems are summarized. Additionally, the main research aim is revisited and the usefulness of MIPOG and its variants are debated. Conclusions are drawn from the experience gained from this work and the significances of findings along with the consideration for future work.

CHAPTER 2

LITERATURE REVIEW

Chapter 1 has established the importance of software testing (i.e. as a way to measure quality). Additionally, Chapter 1 has also advocated the fact that testing for all combination of parameters, although desirable, is infeasible due to lack of resources as well as strict time-to-market constraints. Thus, systematic strategies are required to reduce the number of test cases by selecting a subset from the exhaustive space.

Building from the materials for Chapter 1, this chapter sets the scene on the fundamentals of *t*-way testing strategies. Firstly, this chapter gives some backgrounds and terminologies for the *t*-way testing. Then, this chapter discusses and surveys the state-of-the-art on *t*-way testing. Next, this chapter also includes the applications (art-of-the-practice) of the *t*-way testing in the literature. Finally, this chapter provides the industrial requirements; outlines the challenges; and proposed the methodology for the development of the MIPOG.

2.1 T-Way Testing Fundamentals

This section presents some important definitions and mathematical foundations for the *t*-way combinatorial explosion problem. Any system under test (termed SUT) consists of number of parameters (or factors) with their associated values (or levels). The test case

generator aims to generate test case suite according to the coverage interaction criteria. The strength of coverage (interactions criteria) is explained as follows.

The 1-way interaction (or 1-wise) criteria states: for each parameter, values to be included at least once in the test suite. This is the simplest coverage criteria. Similarly, a pairwise (or 2-way) criterion states that each pair of values for each pair of parameters to be included at least once. Here, it should be noted that the same test case could cover more than one pair of values. Such as, *t*-wise (or *t*-way) criteria is a general extension to the over mentioned criteria for pairwise. *t*-way requires every possible combination of values of *t* parameters be included in some test case in the test suite at least once (Williams and Probert, 1996). A special case of *t*-wise coverage is exhaustive coverage. Exhaustive coverage requires all possible combinations of values of all parameters be included in the test suite (Grindal et al., 2005).

Often, *t*-way testing can be abstracted to a covering array. Throughout out this thesis, the symbols: *p*, *v*, and *t* are used to refer to number of parameters (or factor), values (or levels) and interaction strength for the covering array respectively. Earlier works suggested two definitions for describing the covering array (Cohen, 2004). The first definition is based on whether or not the numbers of values for each parameter are equal. If the number of values is equal (i.e. uniformly distributed), then the test suite is called Coverage Array (CA). Now, if the number of values in non-uniform, then the test suite is called Mixed Coverage Array (MCA).

Normally, the CA takes parameters of N, *t*, *p*, and *v* respectively (i.e. CA(N,*t*,*p*,*v*)). For example, CA (9, 2, 4, 3) represents a test suite consisting of 9x4 arrays (i.e. the rows represent the size of test cases (N), and the column represents the parameter (*p*)). Here, the test suite also covers pairwise interaction for a system with 4 3 valued parameter. When the CA is the most optimal result, the covering array can be rewritten as CAN (N,*t*,*p*,*v*).

Alternatively, MCA takes parameters of N, *t*, and Configuration (C). In this case, C captures the parameters and values of each configuration in the following format: v_1^{p1} v_2^{p2}, ….. v_n^{pn} indicating that there are p1 parameters with v1 values, p2 parameters with v2 values, and so on. For example, MCA (1265, 4, $10^2 4^1 3^2 2^7$) indicates the test size of 1265 which covers 4-way interaction. Here, the configuration takes 12 parameters: 2 10 valued parameter, 1 4 valued parameter, 2 3 valued parameter and 7 2 valued parameter.

Having described the notations, the following subsections describe the determination of the number of tuples (i.e. parameter values interactions or *t*-way combinations), and highlight how the number of tuples is proportional to *p*, *v*, and *t* respectively both for CA and MCA. Additionally, the subsection also describes how the general *t*-way strategies work and why are there many strategies in the literature. Finally, a step-by-step example is given for demonstration and analysis purposes.

2.1.1 Determination for Number of Tuples in CAs

Cohen et al. reported that the number of tuples can be determined as follows(Cohen et al., 1996).

Number of tuples = $\binom{p}{t} v^t = \frac{p!}{t!(p-t)!} v^t$.

2.1.2 Determination for Number of Tuples in MCAs

It is worth noting here that unlike CA which has a static equation to determine the number of interaction elements, MCA requires explicit calculation based on the number of defined parameters and values in order to determine the number of tuples. This is done by considering the sum of products of each individual's interaction sets. For example, when MCA (N, 3, $3^1 2^3$) is considered, the total number of interaction elements = 3*2*2+ 3*2*2 + 3*2*2+ 2*2*2= 44. When MCA (N, 2, $3^1 2^3$) is considered, the total number of interaction elements = 3*2+ 3*2+3*2+2*2 + 2*2+ 2*2= 28. Finally, for MCA (N, 4, $3^1 2^3$), the total number of interaction elements =3*2*2*2= 24.

2.1.3 Objective of the T-Way Strategy

The main objectives of the *t*-way strategy are:

1. Generate a test suite that covers the tuples for a certain degree of interaction (*t*) at least once. This test suite is desired to be with minimal test size as possible.
2. Generate the test suite in acceptable time.

2.1.4 How does a T-Way Strategy Work?

Earlier studies demonstrate that there are two approaches are facilitating the generation of *t*-way test suite, that is, either adopting algebraic or computational strategies (Cohen, 2004). In short, algebraic strategies generate the test suite directly by means of mathematical transformations (Lei et al., 2007). In contrast, computational strategies generate all tuples first. Next, the computational engines try to cover each tuple in tuples space at least once.

2.1.5 Why do Many Strategies Exist?

Answering the question of whether a unique test suite exists that covers tuples at least once is NP-complete (Tai and Lei, 2002, Shiba et al., 2004, Colbourn et al., 2010). Furthermore, Seroussi and Bshouty suggest that the problem of finding the minimum size of test suite for an arbitrary set of tuples is at least as hard as this problem (i.e. NP-hard) (Seroussi and Bshouty, 1988). Therefore, it is often unlikely that an efficient strategy exists that can always generate optimal test set (i.e. each *t*-way interaction is covered by minimum number of test cases).

As such, there are many reasons to answer this question. Firstly, there is no unique solution to the problem of covering tuples at least once (i.e. NP-complete problem). Moreover, it is unlikely to find a unique strategy that always generates optimal number of test case (i.e. NP-hard problem). Furthermore, some strategies are designed to minimize the execution time for generating test suite. Therefore, different strategies exist to address the combinatorial explosion problem from different perspectives.

2.1.6 Systematic Example

In order to illustrate how a *t*-way test data generation strategy works, consider the following running example. Ideally, this running example serves as a model of a typical software system implementation (see Figure 2-1).

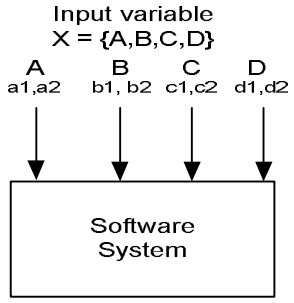

Figure 2-1 Model of a Typical Software System Implementation

Referring to Figure 2-1, let the input variable consists of a set X = [A, B, C, D]. For simplicity sake, let us assume that the starting test case for X, termed *base test case*, has

been identified in Table 2-1 (with 4 parameters and 2 values). Here, symbolic values (e.g. a1, a2, b1, b2, c1, c2) are used in place of real data values to facilitate discussion.

Table 2-1 Base Values

	Input Variables			
	A	**B**	**C**	**D**
Base Values	a1	b1	c1	d1
	a2	b2	c2	d2

In this case, at full strength t= 4 (i.e. exhaustive combinations), the number of test cases = (the number of values) $^{\text{the number of parameters}}$ = 2^4 = 16. These 16 exhaustive combinations can be generated based on a simple technique (see Table 2-2). Here, one can view each variable as a column matrix. For column A, one must repeat the input *a1* 8 times followed by *a2* (also 8 times) to reach 16. This is because there are 16 combinations with 2 specified inputs (i.e. 16/2 = 8 times). Now for column B, one must alternately repeat the input *b1* 4 times followed by *b2* (also 4 times) to reach 16. Similarly, for column C, one must repeat *c1* 2 times followed by *c2* (also 2 times) to reach 16. Finally, for column D, one can alternately repeat *d1* and *d2* to reach 16.

Table 2-2 Exhaustive Test Suite

A	**B**	**C**	**D**
a1	b1	c1	d1
a1	b1	c1	d2
a1	b1	c2	d1
a1	b1	c2	d2
a1	b2	c1	d1
a1	b2	c1	d2
a1	b2	c2	d1
a1	b2	c2	d2
a2	b1	c1	d1
a2	b1	c1	d2
a2	b1	c2	d1
a2	b1	c2	d2
a2	b2	c1	d1
a2	b2	c1	d2
a2	b2	c2	d1
a2	b2	c2	d2

Using the same example, Table 2-3 illustrates briefly, how a typical strategy (e.g. Jenny (Jenkins)) generates test cases. The description of Jenny can be seen later in Section. 2.2. Here, assume that the interaction coverage strength is 3-wise (i.e. t=3). The possible 3-way interaction tuples are between ABC, ABD, ACD, and BCD.

Table 2-3 The Generated Test Suite using Jenny

Test Case #	A	B	C	D
1	a1	b1	c1	d1
2	a1	b1	c2	d2
3	a1	b2	c1	d2
4	a1	b2	c2	d1
5	a2	b1	c1	d2
6	a2	b1	c1	d1
7	a2	b2	c2	d1
8	a2	b2	c2	d2

As seen in Table 2-3, applying the Jenny strategy yields a final test set of 8 test cases only. Using the notation described earlier, we can write the result as CA (8,3,4,2). Here, there is a reduction of 50% (i.e. from 16 in the exhaustive test to 8). Even though, for a small system configuration, reduction from 16 to merely 8 test cases is not that impressive, but consider a larger example: a manufacturing automation system that has 20 controls, each with 10 possible settings, a total of 10^{20} combinations, which is far more than a software tester would be able to test in a lifetime. Surprisingly, all pairs of these values can be checked with only 180 tests if they are carefully constructed (Kuhn et al., 2009).