CRASH RECOVERY SUPPORT FOR VARIABLE STRENGTH T-WAY TEST GENERATION STRATEGY

SYAHRUL AFZAL BIN CHE ABDULLAH

UNIVERSITI SAINS MALAYSIA

CRASH RECOVERY SUPPORT FOR VARIABLE STRENGTH T-WAY TEST GENERATION STRATEGY

by

SYAHRUL AFZAL BIN CHE ABDULLAH

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

ACKNOWLEDGEMENT

بسم الله الرحمن الرحيم

Firstly, I would like to express my sincere appreciation and heartfelt thanks to my supervisor Prof. Dr. Kamal Zuhairi Zamli, for his guidance throughout this research work. For the record, the work was partially funded by "Investigating *t*-way Test Data Reduction Strategy using Particle Swarm Optimization Technique" grant from Ministry of Higher Education (MOHE), Malaysia and also Research Intensive Faculty (RIF) grant – "Integrating Seamless Crash Recovery Support for *t*-way Test Generation Strategy" (Kod: 600-RMI/DANA 5/3/RIF (304/2012)) from Research Management Institute (RMI), Universiti Teknologi MARA (UiTM), Malaysia.

Secondly, I would like to thank all staff of Universiti Sains Malaysia (USM) especially staff of the School of Electrical & Electronic Engineering for their kind help and support.

Thirdly, I would like to thank all staff of Universiti Teknologi MARA (UiTM) especially staff of the Faculty of Electrical Engineering for their kind help and support.

Lastly, I would like to thank my wife, my family and my colleague, Dr. Zainal Hisham Che Soh for being there for me.

TABLE OF CONTENTS

ACI	KNOWL	LEDGEMENT	ii
TAI	BLE OF	CONTENTS	iii
LIS	Γ OF TA	ABLES	vi
LIS	r of fi	GURES	ix
LIS	Γ OF AE	BBREVIATIONS	xi
ABS	STRAK		xiii
ABS	STRACT		xvi
CHA	APTER (ONE: INTRODUCTION	
1.1	Overvie	ew of Software Testing	20
1.2	Probler	m Statements	27
1.3	Thesis	Aim and Objectives	31
1.4	Scope of	of Research	31
1.5	Thesis	Outline	32
CHA	APTER 7	TWO: LITERATURE REVIEW	
2.1	Test de	esign strategies	35
	2.1.1	Boundary value analysis	36
	2.1.2	Equivalence partitioning	37

	2.1.3	Cause-effect graphing	37
2.2	Orthogo	onal Arrays and Covering Arrays	39
2.3	Notatio	n	42
2.4	Exampl	le of <i>t</i> -way test generation strategy	43
2.5	Exampl	le of variable strength <i>t</i> -way test generation strategy	53
2.6	Analysi	is of empirical evidence in <i>t</i> -way test generation strategies	57
2.7	Related	work on t-way test generation strategies	61
2.8	Summa	ry	69
CHA	APTER 7	ΓHREE: DEVELOPMENT OF VARIABLE STRENGTH	
77. 33			
T-W	AY TES	ST GENERATION STRATEGY	
T-W 3.1		T GENERATION STRATEGY lology of variable strength <i>t</i> -way test generation strategy	70
	Method		70 72
3.1	Method	lology of variable strength <i>t</i> -way test generation strategy	
3.1	Method The TS	lology of variable strength <i>t</i> -way test generation strategy GCR strategy	72
3.1	Method The TS 3.2.1	lology of variable strength <i>t</i> -way test generation strategy GCR strategy Uncovered <i>t</i> -way sets (interaction elements) generator	72 74
3.1	Method The TS 3.2.1 3.2.2 3.2.3	lology of variable strength <i>t</i> -way test generation strategy GCR strategy Uncovered <i>t</i> -way sets (interaction elements) generator Test case selector	72 74 80
3.1	Method The TS 3.2.1 3.2.2 3.2.3	lology of variable strength <i>t</i> -way test generation strategy GCR strategy Uncovered <i>t</i> -way sets (interaction elements) generator Test case selector Crash recovery	72 74 80 87
3.1	Method The TS 3.2.1 3.2.2 3.2.3 Implem	dology of variable strength <i>t</i> -way test generation strategy GCR strategy Uncovered <i>t</i> -way sets (interaction elements) generator Test case selector Crash recovery	72 74 80 87 91
3.1	Method The TS 3.2.1 3.2.2 3.2.3 Implem 3.3.1	GCR strategy Uncovered <i>t</i> -way sets (interaction elements) generator Test case selector Crash recovery entation Issues Justification of using Java TM SE Development Kit 7	72 74 80 87 91 92

CHAPTER FOUR: RESULTS AND DISCUSSION

4.1	Evaluat	ion of TSGCR strategy	100
	4.1.1	Demonstration of the crash recovery support	100
	4.1.2	Performance overhead of the crash recovery support	104
	4.1.3	Comparison between fixed interval checkpoints with dynamic	107
		checkpoints	
	4.1.4	Comparison with existing variable strength <i>t</i> -way test generation	107
		strategies	
	4.1.5	Estimated lower bound	124
4.2	Discuss	ion	125
4.3	Summa	ry	128
CHA	APTER I	FIVE : CONCLUSION	
5.1	Overvie	·w	129
5.2	Conclus	sion and Research Contribution	131
5.3	Suggestions for Future Work		133
REFERENCES		134	
APP	PENDICE	ES	
LIST	Γ OF PU	BLICATIONS	
CLOSSARV			

LIST OF TABLES

		Page
Table 2.1	Experiments for four (4) input parameters with three (3) values using Orthogonal Array	40
Table 2.2	Input parameters of order page in an online shopping mall system	44
Table 2.3	Optimal total number of test cases (i.e. unique combinations) for the online shopping mall system (when <i>t</i> is varied)	47
Table 2.4	Interaction elements for the online shopping mall system at t = 2 constructed according to the group	48
Table 2.5	Interaction elements covered by the first test case and the second test case	50
Table 2.6	Example of one of optimal test suite for the online shopping mall system at $t = 2$	51
Table 2.7	Optimal total number of test cases (i.e. unique combinations) for the online shopping mall system (when t is varied and mixed t)	54
Table 2.8	Changing the input parameters of order page in the online shopping mall system	55
Table 2.9	Optimal total number of test cases (i.e. unique combinations) for the online shopping mall system (when t is varied and mixed t)	56

Table 2.10	Summary of empirical evidence to justify stronger interaction strength, <i>t</i> in <i>t</i> -way testing (arranged in chronological order)	59
Table 2.11	Summary of design techniques used by the strategies and their features	67
Table 3.1	The truth table contains lists of Boolean true and false for $t = 2$	74
Table 3.2	Data structure for uncovered interaction elements of TCAS module	77
Table 3.3	Subset of [T, T, T, T, F, F, F, F, F, F, F, F] of TCAS module at $t = 2$	79
Table 4.1	The average time taken using fixed interval checkpoints (after the specified time has elapsed)	105
Table 4.2	The average time taken using fixed interval checkpoints (after <i>N</i> th test case)	106
Table 4.3	The average time taken using dynamic checkpoints	107
Table 4.4	Estimated lower bound for TCAS module	111
Table 4.5	Sizes of VS interactional test suites for Set 1, $VSCA(N:2,3^{15},\{C\})$	113
Table 4.6	Sizes of VS interactional test suites for Set 2,	114

Table 4.7	Summary of result for Table 4.5 and Table 4.6	116
Table 4.8	Sizes of VS interactional test suites for Set 3, $VSCA(N: 2, 4^35^36^2, C)$	117
Table 4.9	Sizes of VS interactional test suites for Set 4, $VSCA(N: 2, 3^{20}10^2, C)$	119
Table 4.10	Sizes of VS interactional test suites for Set 5, $VSCA(N:3,4^13^72^2,C)$	120
Table 4.11	Sizes of VS interactional test suites for Set 6, $VSCA$ (N : 2, $10^19^18^17^16^15^14^13^12^1$, C)	121
Table 4.12	Summary of results for Table 4.8, 4.9, 4.10 and 4.11	122
Table 4.13	Summary of comparison of TSGCR with VS-PSTG	124

LIST OF FIGURES

		Page
Figure 1.1	A Deming Cycle (Moen and Norman, 2010)	21
Figure 1.2	Software Testing Cycle (Zamli et al., 2009)	22
Figure 1.3	Most common challenges faced by software tester (EUROSTAR, 2013)	24
Figure 2.1	The "Image Properties" option dialog in Microsoft Paint	36
Figure 2.2	Pop-up message in Microsoft Paint	38
Figure 2.3	Orthogonal Array, L9(34)	40
Figure 2.4	Code snippets to depict 2-way interaction failure detected only when the aforementioned conditions are met	46
Figure 2.5	Covering Array, CA (9: 2, 34)	52
Figure 3.1	Research methodology on developed variable strength <i>t</i> -way test generation strategy	71
Figure 3.2	Overview of the software architecture of TSGCR strategy	73
Figure 3.3	Matching using set interaction operation	76
Figure 3.4	The Greedy Algorithm template (Cohen, 2004)	81
Figure 3.5	Illustrative steps for creating a test configuration candidate	82
Figure 3.6	Illustrating steps in crash recovery	88

Figure 3.7	Pseudo code to depict fixed interval checkpoints (after <i>N</i> th test case)	90
Figure 3.8	Pseudo code snippets to depict fixed checkpoints (after the specified time has elapsed)	91
Figure 3.9	Code snippets to depict dynamics checkpoints	91
Figure 3.10	The number of different systems per category (Andlinger, 2013)	95
Figure 4.1	Snapshot of TSGCR application during a normal operation	101
Figure 4.2	Snapshot of TSGCR application after restarting from the crash	102
Figure 4.3	Snapshot of TSGCR application after gracefully recovered from the failure and resume its normal operation	103

LIST OF ABBREVIATIONS

Abbreviation Meaning

ACA Ant Colony Algorithm

ACID Atomicity, Consistency, Isolation and Durability

ACS Ant Colony System

AETG Automatic Efficient Test Generator

AI Artificial Intelligence

AOF Append-Only File

API Application Programming Interface

ASIC Application-Specific Integrated Circuit (ASIC)

AUT Application Under Test

CA / CAs Covering Array / Covering Arrays

CPU Central Processing Unit

DBMS Database Management Systems

DOE Design of Experiments

FPGA Field Programmable Gate Array

GA Genetic Algorithm

GUI Graphical User Interface

HDL Hardware Description Language

IDE Integrated Desktop Environment

IPO Input Parameter Order

IPOG Input Parameter Order Generalized

ISTQB International Software Testing Qualifications Board

ITCH IBM's Intelligent Test Case Handler

JDK Java Development Kit

Abbreviation Meaning

JVM Java Virtual Machine

MCA Mixed Covering Array

NIST National Institute of Standards and Technology

NP Nondeterministic Polynomial time

OA / OAs Orthogonal Array / Orthogonal Arrays

OPAAT One Parameter At a Time

OTAAT One Test At a Time

PDCA Plan-Do-Check-Act

POJO Plain Old Java Object

PSO Particle Swarm Optimization

PSTG Particle Swarm Test Generator

RAM Random Access Memory

RDBMS Relational Database Management Systems

SA Simulated Annealing

SDR Software-defined Radio

TCAS Traffic Collision Avoidance System

TCG Test Case Generator

TSGCR Test Suite Generator with Crash Recovery support

TVG Test Vector Generator

VS Variable Strength

VS-PSTG Variable-Strength Particle Swarm Test Generator

WHITCH IBM Intelligent Test Case Handler

SOKONGAN PEMULIHAN KEGAGALAN UNTUK STRATEGI PENJANAAN UJIAN HALA-T DENGAN KEKUATAN BOLEH UBAH

ABSTRAK

Selalunya, cabaran terbesar dalam pengujian perisian berkait dengan hakikat bahawa ia tidak boleh dilaksanakan untuk menguji kesemua parameter-parameter input kerana kekangan-kekangan seperti kos, sumber dan masa. Mempertimbangkan faktor-faktor ini, penguji-penguji perisian perlu melakukan pilihan kes-kes ujian yang sesuai supaya sumber-sumber yang sedia ada digunakan dengan cara yang terbaik. Dalam konteks Ujian Kombinasi, penguji-penguji sering mengambil pendekatan strategi penjanaan ujian hala-t (di mana t menunjukkan kekuatan interaksi). Bukti empirikal dalam literatur menunjukkan bahawa strategi penjanaan ujian hala-t telah berjaya mengurangkan kes-kes ujian dengan ketara sambil mengekalkan keupayaan pengesanan kecacatan daripada proses pengujian. Banyak kemajuan yang berguna sudah dicapai berkenaan dengan pembangunan strategi penjanaan ujian hala-t. Walau bagaimanapun, beberapa isu masih kekal terutamanya dalam konteks menangani kerumitan dan saiz perisian (iaitu baris-baris kod) yang semakin meningkat mengakibatkan jumlah interaksi antara parameter-parameter input yang tinggi. Pertamanya, penjanaan ujian boleh mengambil jangka masa yang panjang, sebarang gangguan adalah mahal kerana keseluruhan penjanaan perlu dimulakan semula dari awal. Masa dan usaha-usaha akan menjadi sia-sia. Keduanya, strategi sedia ada melakukan pemilihan terlalu awal pada nilai terbaik parameterparameter input apabila melakukan pensampelan kes-kes ujian. Atas sebab ini, strategi-strategi ini kurang mencukupi dari segi menjana saiz sut ujian yang optimum. Di sini, untuk meningkatkan keupayaan ujian hala-t, terdapat juga

keperluan untuk mempertimbangkan strategi kekuatan interaksi boleh ubah. Pendekatan ini sering disukai kerana kompromi dari segi saiz sut ujian kerana strategi ini memberi tumpuan pengujian di mana ia mempunyai nilai yang paling berpotensi yang biasanya dikaitkan dengan analisis risiko dan keutamaan. Untuk menangani isu-isu ini, kajian ini membangunkan strategi penjanaan data ujian dengan kekuatan interaksi boleh ubah, yang dikenali sebagai Penjana Sut Ujian dengan sokongan Pemulihan Kegagalan (TSGCR). Tidak seperti strategi-strategi yang sedia ada, TSGCR menggunakan algoritma tamak bertingkat, yang lewat memilih nilai yang terbaik sehingga ia memenuhi peraturan tertentu. Untuk menyediakan operasi yang boleh diharap, TSGCR juga membenarkan sokongan pemulihan kegagalan bersepadu sebagai sebahagian daripada strategi itu sendiri. Kerana potensi proses penjanaan data ujian yang akan mengambil jangka masa yang panjang (iaitu disebabkan oleh pemilihan parameter-parameter input dan nilai-nilai yang agak besar), TSGCR boleh menghadapi kegagalan transaksi secara paksaan (contohnya seperti kegagalan kuasa atau kesilapan sistem) atau penggantungan pelaksanaan penjanaan secara sukarela (contohnya untuk memberi ruang untuk pengiraan yang lain) membolehkan pemulihan status dan data ke status lepas yang konsisten. Untuk menilai kedayasaingan TSGCR, penjanaan ujian diuji dengan paramater-parameter input yang seragam dan campuran dan prestasi (dari segi saiz sut ujian yang dihasilkan) dibandingkan dengan strategi-strategi penjanaan ujian dengan kekuatan hala-t yang berubah-ubah yang sedia ada dengan menggunakan konfigurasi penanda aras piawaian yang terkenal (berdasarkan enam set eksperimen). Hasil kajian penanda arasan menunjukkan bahawa bagi konfigurasi interaksi VS untuk parameter input seragam, TSGCR mendapat tujuh Δ dengan nilai 0, iaitu sama nilai seperti penyelesaian terbaik yang diperolehi dengan strategi-strategi yang lain,

tujuh Δ dengan nilai + ve, iaitu mampu untuk mendapatkan penyelesaian yang terbaik; iaitu empat belas daripada empat puluh empat keputusan eksperimen. Manakala bagi konfigurasi interaksi VS untuk parameter input campuran, TSGCR mendapat dua puluh tujuh Δ dengan nilai 0, lapan Δ dengan nilai + ve; iaitu tiga puluh lima daripada empat puluh satu keputusan eksperimen. Oleh itu, hasil keputusan menunjukkan bahawa TSGCR menghasilkan keputusan yang kompetitif berbanding kebanyakan strategi-strategi yang sedia ada.

CRASH RECOVERY SUPPORT FOR VARIABLE STRENGTH T-WAY TEST GENERATION STRATEGY

ABSTRACT

Often, the biggest challenge in software testing relates to the fact that it is not feasible to test for all the input parameters exhaustively owing to constraints in costs, resources and time. Considering these factors, software testers must appropriately sample the test cases in order to best utilize the resources at hand. Within the context of Combinatorial Testing, testers often resort to t-way test generation strategy (where t indicates the strength of interaction). Empirical evidence in the literature indicated that t-way test generation strategy has managed to minimize the test cases significantly whilst maintaining the fault detection capability of the testing process. Much useful progress has been achieved as far as the development of t-way test generation strategy is concerned. Nevertheless, some issues remain especially in the context of addressing ever increasing complexity and size of software (i.e. lines of code) resulting into high number of interaction among input parameters. Firstly, the test generation can be painstakingly long, interruption is expensive as the whole generation process needs to be restarted from scratch. Time and efforts will also be wasted. Secondly, existing strategies commit too early on selection of the best value of input parameters when sampling of the test cases. For this reason, these strategies were less sufficient in terms of generating optimal test suite size. Here, to enhance the t-way testing capability, there is also a need to consider variable-strength strategy. This approach is often favored because of the compromise in terms of test suite size as the strategy focuses testing where it has the most potential value which usually is associated with a risk analysis and priority. In order to address these issues, this research develops a variable-strength (VS) interaction t-way test generation strategy, called Test Suite Generator with Crash Recovery support (TSGCR). Unlike existing strategies, TSGCR adopts Multilevel Greedy algorithm, which delays choosing the best value until it satisfies certain rules. To provide a reliable operation, TSGCR also permits crash recovery support integrated as part of the strategy itself. As the test generation can potentially be long lasting processes (i.e. due to large selection of input parameters and values), TSGCR tolerates involuntary transaction failures (e.g. such as power failure or system errors) or voluntary execution suspension (e.g. to give ways for other computations) enabling restoration of state and data to the last consistent state. To evaluate the competitiveness of TSGCR, the test generator is tested with uniform and mixed input parameters and the performance (in terms of the generated test suite size) is compared with existing variable strength t-way test generation strategies using well-known standard benchmark configurations (based on six sets of experiments). Benchmarking results showed that for VS interaction configurations for uniform input parameters, TSGCR is able to get seven Δ with 0 value, i.e. similar value to the best solution obtained by other strategies, seven Δ with +ve values, i.e. able to get the best solution; from fourteen out of forty four experimental results. While for VS interaction configurations for mixed input parameters, TSGCR is able to get twenty seven Δ with 0 value, eight Δ with +ve values; from thirty five out of forty one experimental results. Hence, the results demonstrated that TSGCR produces competitive results as far as the size of the test suite is concerned against most existing strategies.

CHAPTER ONE

INTRODUCTION

Nowadays, people have the tendency to increasingly rely on electronic devices to accommodate their daily life, e.g. automobile, gadget and home appliances. By combining various hardware and software technologies, these devices are providing options to make our life better, i.e. for increasing comfort and efficiency. The technologies inside these devices vary because of the differences in functionalities and innovations.

Typically, manufacturers or solution providers are inclined to replace most hardware implementations with software for cost savings. The reason is quite obvious; unlike hardware, software does not wear out. Moreover, software is malleable and can be easily customized as the need arises, e.g. adding new functionalities (Klaib et al., 2008) such as using a Field Programmable Gate Array (FPGA) or implementing a Software-defined Radio (SDR).

FPGA contains an array of programmable logic blocks and a hierarchy of reconfigurable interconnects that allow the blocks to be wired together using a software programming language called Hardware Description Language (HDL) (Sadrozinski and Wu, 2010). A soft-core processor, which is a HDL model of a specific processor (CPU) can also be customized for a given application and synthesized for FPGA (Tong et al., 2006). At the same time, FPGA can still be configured by a designer even after manufacturing process.

Likewise components in SDR, such as amplifiers, filters and mixers, in the radio communication system which have been commonly implemented in hardware but are instead implemented by means of software on a personal computer or embedded system (Dillinger et al., 2003).

Because of the inclination to choose software-based solutions (e.g. FPGA, SDR, etc.), the complexity and size of software (i.e. lines of code) is ever-increasing while at the same time, testing the software has evolved from a routine quality assurance activity into a sizeable and complex challenge in terms of manageability and effectiveness (Geetha Devasena and Valarmathi, 2012).

Software quality and reliability are the main criteria for success in the software industry. If software is faulty, it is prone to do unexpected behavior resulting into undesirable outcome. Considering that software is becoming more complicated; software testing is becoming immensely important because statistical data shows that it accounts as much as 50 percent of the total software development cost and even more for mission safety critical system (Ammann and Offutt, 2008). Besides, software testing adds considerably to the length of the development cycle.

Lack of testing can lead to disastrous consequence if software is deployed in mission safety critical life threatening application. For instance, in February 25, 1991, the Patriot battery at Dhahran, Saudi Arabia, failed to track and intercept an incoming Iraqi Scud missile that later struck an army barrack, killing 28 Americans. The software problem caused by an inaccurate tracking calculation that became worse when the longer the system is in operation. In this incident, i.e. after 100

hours, the inaccuracy was serious enough to cause the system to look in the wrong place for the incoming Scud (Blair et al., 1992).

As another illustration, consider a software that controls an airbag system of a car (Montoya, 2013). Failure to accommodate certain conditions in the software, such as while the owner of the car is in the middle of driving and attempting a drift, an unwanted self-deployment of the airbag (since the airbag sensors predicted a rollover is imminent) can prove fatal (Ireson, 2011).

All the aforementioned incidents have highlighted the importance of testing the software thoroughly especially in life threatening applications. As such, the next section will discuss the overview of how software testing is done.

1.1 Overview of Software Testing

Software testing relates to activities concerned with planning, preparation and evaluation of software products in order to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects (ISQTB, 2012).

To carry out the aforementioned activities, software testing communities have provided a number of useful techniques that can be used to identify possible defects and anomalies. If most of the defects and anomalies are detected, the risk for software failure can indeed be minimized while establishing confidence that the software is working like it was intended to do (Zamli et al., 2009, Bentley, 2005, Harrold, 2000).

Figure 1.1 depicts a Deming Cycle that is often used for the control and continuous improvement of processes and products. The circle contains an iterative four-step management method termed PDCA (Plan–Do–Check–Act) (Moen and Norman, 2010).

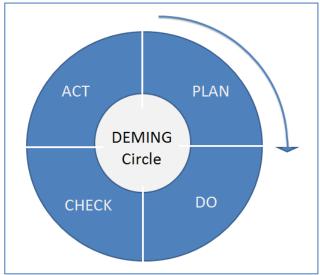


Figure 1.1 : A Deming Cycle (Moen and Norman, 2010)

The steps in each successive PDCA cycle are as follows:

- (a) Plan, where the objectives and processes necessary to deliver results in accordance with the expected output (the target or goals) are established,
- (b) Do, where the processes are executed according to the aforementioned plan,
- (c) Check, where the actual results are analyzed and compared against the expected output to ascertain any differences. Any deviation in implementation from the plan is studied for further actions, and
- (d) Act, where the corrective actions are requested to close the gap between the actual results versus the planned output. The differences are analyzed to determine their root causes.

Figure 1.2 depicts a Software Testing Cycle proposed by Zamli et al., (2009). The Software Testing Cycle has been designed based on the four steps adopted from the Deming Cycle. Here, as shown in Figure 1.2, the software testing activities can be categorized into three main stages.

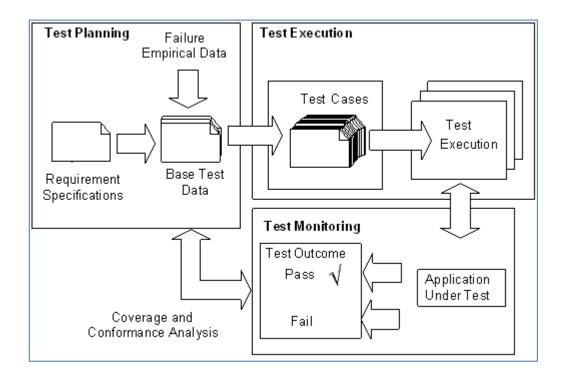


Figure 1.2 : Software Testing Cycle (Zamli et al., 2009)

By dissecting the activities into these stages, function of each stage can be described separately and thus the whole software testing activities can be comprehended clearly. Termed test cycle, these stages are Test Planning stage, Test Execution stage, and Test Monitoring stage (which consists of the check and act steps adopted from the Deming Cycle).

As the name suggests, Test Execution stage involves the activities to define and execute the planned test cases (and observing the results) with the intention to find bug/defect/error (e.g. a mistake in the specified requirement. Actually, the

bug/defect/error can also be found in every phases of software development process with different naming convention (terms), i.e. anomaly, crash, exception, failure, fault, incident or side effect). Usually, the procedure of running and executing the test cases are performed automatically using test scripts.

As depicted by Figure 1.2, software tester tests Application Under Test (AUT) using test cases. The test cases are generated from base test data using requirements specification and also include consideration on failure empirical data, which is based on known problems with similar systems (i.e. many products or systems, such as missiles and air bags, are considered one time/usage systems, hence the failure empirical data is a good candidate for test data in testing these systems). To test for a particular objective, such as to verify compliance with a specific requirement, software tester uses the requirements specification in order to determine what kind of input parameters and their values, execution pre-conditions, expected results and execution post conditions for the test cases (ISTQB, 2012).

Then, in the Test Monitoring stage, the results of the test execution will be checked whether they conform to the specification or not, as well as analyzing the test coverage (what gets tested, e.g. in Functional testing, a slice of functionality of the whole software is tested) against the stopping criteria to determine either the software testing process is done or not.

To ensure success, both aforementioned test cycle requires a good planning. As a result, we need to properly manage the testing by planning what we want to do before and after test execution. Therefore, the Test Planning stage involves the