Software Engineering Framework for Developing Software Testing Products

Mohammed Issam Younis
School of Electrical and Electronic Engineering
Universiti Sains Malaysia Engineering Campus
14300 Nibong Tebal, Penang, Malaysia
Email: younismi@gmail.com

Abstract

Software testing is a perfect candidate for the union of academic and industrial minds. The proposal gives the basic principals that must be taken into account to achieve high-productivity production-environment that provides linking between academic and industrial aspects to achieve that goal. The proposal gives an analysis of the available technology and finally gives the specifications and architectural of a framework for implementation testing aspects products.

.

1. Introduction

Many software systems today are built using various components. Often, system faults are caused by unexpected interactions among these components. One solution to remove any such faults from a system is software testing. Testing is a process that requires a great deal of time and resources. It is widely recognized in the computer science community that testing consumes approximately 50% of the total cost of developing new software. Furthermore, the cost of testing new hardware and safety critical systems is even higher. Inadequate testing can lead to catastrophic consequences.

Testing is an important but expensive part of the software and hardware development process. To thoroughly test a large software or hardware system, many combinations of possible inputs must be tried and the expected behavior of the system must be verified against the systems requirements. However, the size of a test suite required to test all possible interaction combinations could be prohibitive in even a moderately sized project. Therefore, it is necessary to decrease the set of test configurations by selectively testing only a subset of this test configuration into systematic manner [1].

One approach to software testing is pairwise testing. Pairwise testing helps in detecting faults caused by interactions among two parameters. Pairwise testing achieves higher block and decision coverage than traditional methods for a commercial email system [2].

However, it is not necessary that faults are only caused by the interaction between two parameters. There are chances that faults can be caused by the interaction of more than two parameters. For Example, by applying *t*-way testing to a telephone software system showing that several faults can only be detected under certain combinations of input parameters [3]. In fact, a study conducted by The National Institute of Standards and Technology (NIST) has shown that about 95% of actual faults involved up to 4-way interactions in the software studied. And using up to 6-way combinatorial software testing can detect almost all of the faults [4] [5]. Therefore, it is necessary to test interactions between more than two parameters.

A strategy that tests interactions among more than two parameters is t-way testing. T-way testing, where the value of t is usually small and is referred to as the degree of interaction requires that for any t parameters, every combination of their values should be covered by at least one test. T-way testing guarantees that all t-way combinations are tested together. The main principle behind it is that not every parameter is responsible for every fault in a system, and many faults can be exposed by interactions involving only a few parameters.

To illustrate the concept of t-way testing, consider a Traffic Collision Avoidance System (TCAS) module. TCAS is an aircraft collision avoidance system from the Federal Aviation Administration, and has been used in other studies of software testing [6] [7] [8]. 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 (i.e., 10*10*4*3*3*2*2*2*2*2*2*2), or 12 way testing for this system. Running such test may be impossible. Alternatively, 11-way testing requires 230400. 10-way requires 201601. 9-way requires 120361. 8-way requires 56742. 7-way requires 26061. 6-way requires 10851. 5-way requires 4196. 4-way requires 1265. 3-way requires 400. Finally, 2-way

requires only 100 test cases.

Earlier work suggests that t-way sampling strategy can be effective to systematically reduce the test data set to some manageable combinations. Also, it shows that the size of generated test set proportional logarithmically with the number of parameters. Finally, earlier work reports that finding minimal test size is NP-completeness problem (i.e., no unique solution to find minimal test size), and that is why different sampling strategies exists in the aims of minimizing the test set. Building and complementing earlier work, the paper proposes an efficient framework to standardize the available tools.

This paper is organized as follows. Section 2 discusses some related work. Section 3 gives analysis of available tools. Section 4 gives the planning and the architectural design to build the framework. Finally, section 5 gives the conclusion.

2. Related Work

A reasonable amount of work has been done on t-way testing in the past, but most of it focused on pairwise or 2-way testing. Various tools are available which implement these approaches. Some of classifications to these strategies exist. One of classifications focus on either the strategy deterministic or not [9]. Other classification is either the strategy computational or algebraic. Here we classify the strategies according to their supporting t-way testing. What follows is a brief overview on such work previously carried out or work which is still in progress.

For pairwise testing there are on the shelf commercial test tools. As examples: OATS (Orthogonal Array Test System) [10] [11], IRPS [12], AllPairs1 [13], AllPairs2 [14], IPO [15], TCG (Test Case Generator) [16], Pro-Test [17], CTS (Combinatorial Test Services) [18], ReduceArray2 [19], TestCover [20], DDA (Deterministic Density Algorithm) [21], OA1 [22], CTE-XL [23], CaseMaker [24], PICT [25],rdExpert [26], OATSGen [27], SmartTest [28], and EXACT(EXhaustive seArch of Combinatorial Test suites) [29]. Other tools that support 3-way testing (as well as 2-way) are: AETG [30] (Automatic Efficient Test Generator), employs a greedy algorithm to construct the test case, that is, each test covers as many uncovered combinations as possible. Because AETG uses random search algorithm, the generated test case is highly non-deterministic (i.e. the same input parameter model may lead to different test suites [9]). Other variants to AETG that use stochastic greedy algorithms are: GA (Genetic Algorithm) and ACA (Ant Colony Algorithm) [31]. In some cases, they give optimal solution than original AETG, although they share the common characteristic as far as being non-deterministic in nature. Some approaches opted to adopt heuristic search techniques such as hill climbing and simulated annealing (SA) [31]. Briefly, hill climbing and simulated annealing strategies start from some known test set. Then, a series of transformations were applied (starting from the known test set) until an optimum set is reached to cover all the pairwise combinations [29].Unlike strategy that builds a test set from scratch (like AETG), heuristic search techniques can predict the known test set in advance. As such, heuristic search techniques can produce smaller test sets than AETG and IPO, but they typically take longer time to complete [6].

The following existing tools that support up to 6-way testing and are either open source or free for academic use:

IBM's Intelligent Test Case Handler also known as ITCH tool [32], uses the sophisticated combinatorial algorithms to construct test suites. It enables the user to generate small test suites with strong coverage properties, choose regression suites and perform other useful operations for the creation of systematic software test plans. ITCH is a replacement of CTS [1]. TConfig [33], which is from University of Ottawa. Test Vector Generator (or TVG) [34], and Jenny [35]. Based on limited information available in the literature, ITCH implements a combination of several algebraic methods (the details of the combination are not known), and TConfig implements a recursive construction method. Both Jenny and TVG seem to implement a computational method, but the details of their algorithms are not clear. Also Jenny has the ability to generate more than 6-way testing. Finally, FireEye [6] tool which implements In-Parameter-Order-General strategy (or IPOG) which is generalized IPO strategy. Till the time of this writing the FireEye still under development.

3. Analysis

Through the study and observation the test case generation products, the following points are considered:

1. There is no standardized framework to build the test generator. So, there are different implementations supported by different vendors. Due to this point it is time consuming for both industrial developers and academic researchers to learn the use of each product.

2. There is no standardized formatting for input/output parameters variables.
3. The test generations products share the need of prove of correctness tools. This can be made by sharing standard tool that demonstrates correctness. Doing that save the efforts for each vendor.
4. There is a need to integrate the test generator with other testing tools. Switching from one generator to other. In this case, the overall work must be re-done.
5. Testing expected to run in different operating systems. So, the products must be functional in cross-platform environment.
6. The system is adaptable for future needs.

The next section gives the proposed framework to achieve our analyzing goals.

4. Planning and Architectural Design

In this section we explain how to build our framework. And give the architectural design for the framework.

To achieve cross-platform functionality Java is the most candidates programming language, furthermore java is pure object programming language with multiple platform look & feel and has enormous library that provide short cut development cycle.

The suggested framework based on IT-Design pyramid, see figure 1. We suggest OOP to implement this pyramid.

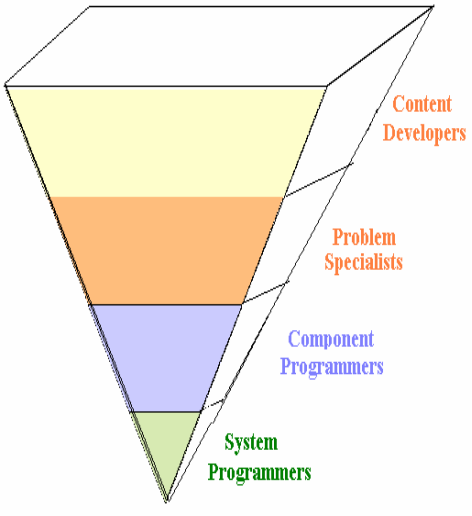

Figure 1. IT Design

In OOP community there are four level of programming according to the skill of the programmer:

1. Architect; the term architect is somewhat loosely defined in the industry. The architect is defined as someone who develops an object-oriented solution that will meet complex sets of goals and requirements. Whether the specification calls for scalability to millions of users, or a very small memory footprint, the design must perform. The architect has no prerequisites of programming, but has a broad scope of program design skills and object-oriented methodologies [8].
2. System-programmer; is a very high trend person of specific programming language (such as Assembly, C, C++, Java, etc) that implement (Coding) the design.
3. Component-programmer; is a very high trend person based on strong theory and has well expert of how to convert the theoretic-specification provided by the problem-specialists into ready-made component.
4. Developer; is a person who has an excellent expert of how to utilize and develop the system. Programmers normally supply the developers by a ready-made and customizable functionality to save their development time in terms of *Application Programming Interfaces* (APIs), and then the developer can use them in his applications or develops other classes (libraries) for other developers.

In this framework each party has its own works as follows:

a. The problem specialists give their theory to the components programmers and developer, in our case study the static components given to the components programmers (e.g., test case generator) while the dynamic components are developed by the developers (e.g., coverage demonstrator). Normally problem specialists are problem specific that may have no experts in computer. In other direction the problem specialists take the advantages of the products that developed by developers instead of building them from scratch.
b. The developers integrate the components provided by the components programmers with the ability to select the components derived by specific vendors. They treat the components as black-boxes. They do not know how the actual implementation is done.
c. The components programmers provide the concrete implementation of the components i.e., treat the components as white-boxes.

d. The system-programmers tie the components programmers to the developers. This accomplished by providing standard APIs to the developer and standard SPIs to the components programmers.

5. Conclusion and Discussions

In this paper we propose industrial standard framework for building testing products. The benefits for this framework can be derived as follows:

1. Twin academic-industrial benefits. The academics will take the advantages of the developed products and give their enhancement, researches, and components with short cut time, while the industry take these researches and integrate with system as developer libraries and/or components.
2. Solving the problem of multiple vendors and multiple versions while making both the developer and end-users independent.
3. The separation of concerns is inherently provided by the framework due to modularity design.
4. The standardization is provided by the system programmers that give industrial standard APIs to the developers and industrial standard SPIs to the components programmers.
5. The framework provides very high re-usability. The system is reusable due to the following factors:
   5.1. The engines and their correspondence SPIs classes provide the functionality of client-stub and server-skeleton respectively, which are reused during any interaction between the client and the server (provider).
   5.2. A service-provider can serve many clients at the same time.
   5.3. A good implementation of the clients-components (i.e., partitioning the developer layer into sub-layers) which yields heterogeneous ready-made libraries that can be re-used by another application. And that leading the way towards shortcut development cycle.
   5.4. The system obeys the standards strictly.
6. The system is scaleable that can be extended horizontally (by adding functionality) and vertically (by adding providers).
7. The adaptive maintenance is achieved by the framework that enables the developers to install alternatives implementation.
8. The system is upgradeable which means of replacement the current provider by another version (e.g., when fast or more reliable version is available), without effecting the applications.
9. The system is cross-platform because it uses only the basic Java-Classes (i.e., pure Java implementation).
10. The test code and test-data is centrally maintained, which results in cheaper maintenance. At the same time the clients remain independent.
11. The ability to derive standard metrics to compare the efficiency of each product.

References

[1] C. P. Jayaswal, "Automated Software Testing Using Covering Arrays." MSc thesis: University of Texas at Arlington December 2006.

[2] K. Burr and W. Young, "Combinatorial Test Techniques: Table-based Automation, Test Generation and Code Coverage," in *Intl. Conf. on Software Testing Analysis & Review*, 1998.

[3] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model based testing in practice," in *Proc. of the Intl. Conf. on Software Engineering (ICSE)*, 1999, pp. 285–294.

[4] D. R. Kuhn and M. J. Reilly, "An Investigation of the Applicability of Design of Experiments to Software Testing," in *Proceedings of the 27th NASA/IEEE Software Engineering Workshop*, NASA Goddard Space Flight Center, December 2002.

[5] D. R. Kuhn, D. Wallace, and A. Gallo, "Software Fault Interactions and Implications for Software Testing," *IEEE Transactions on Software Engineering* vol. 30, June 2004.

[6] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG: A General Strategy for TWay Software Testing," in *14th Annual IEEE Intl. Conf. and Workshops on the Engineering of Computer-Based Systems*, Tucson, AZ, March 2007, pp. 549-556.

[7] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proc. 16th Intl. Conf. On Software Engineering*, May 1994, pp. 191–200.

[8] D. R. Kuhn and V. Okun, "Pseudo-exhaustive Testing For Software," in *30th NASA/IEEE Software Engineering Workshop*, April 25-27, 2006.

[9] M. Grindal, J. Offutt, and S. F. Andler, "Combination Testing Strategies: A Survey," GMU Technical Report ISE-TR-04-05 July 2004.

[10] K. A. Bush, "Orthogonal Arrays of Index Unity" *Annals of Mathematical Statistics* vol. 23, pp. 426-434, 1952.

[11] R. Mandl, "Orthogonal Latin Squares: an Application of Experiment Design to Compiler Testing," *Communications of the ACM* vol. 28 (10), pp. 1054-1058, 1985.

[12] M. I. Younis, K. Z. Zamli, and N. A. M. ISA, " IRPS - An Efficient Test Data Generation Strategy for Pairwise Testing " *accepted for publication in KES 2008*.

[13] http://www.satisfice.com/testmethod.shtml.

[14] http://www.mcdowella.demon.co.uk/allPairs.html.

[15] Y. Lei and K. C. Tai, "In-Parameter-Order: A Test Generating Strategy for Pairwise Testing," *IEEE Transaction on Software Engineering* vol. 28 (1), pp. 1-3 2002.
[16] Y. Tung and W. S. Aldiwan, "Automating Test Case Generation for the New Generation Mission Software System " in *Aerospace Conference Proceedings*, Big Sky, MT, USA, 2000, pp. 431-437.
[17] http://www.sigmazone.com/protest.htm.
[18] http://www.alphaworks.ibm.com/tech/cts.
[19] G. T. Daich, "Testing Combinations of Parameters Made Easy," in *IEEE Systems Readiness Technology Conference (AUTOTESTCON 2003)*, Sept. 2003, pp. 379- 384.
[20] http://www.testcover.com.
[21] R. Bryce and C. J. Colbourn, "The Density Algorithm for Pairwise Interaction Testing " *Software Testing, Verification and Reliability* vol. 17 pp. 159 - 182 2007.
[22] http://www.software-metrics.org.
[23] E. Lehmann and J. Wegener, "Test Case Design by Means of the CTE XL " in *8th European International Conference on Software Testing, Analysis & Review (EuroSTAR 2000)* Kopenhagen, Denmark December 2000.
[24] Y. Tsubery, "Implementation Of CaseMaker In MIS BU – Comverse ": http://www.casemakerinternational.com, Sept. 2007
[25] http://download.microsoft.com.
[26] L. Copeland, *A Practitioner's Guide to Software Test Design*: STQE Publishing, 2004.
[27] R. Krishnan, S. M. Krishna, and P. S. Nandhan, "Combinatorial Testing: Learnings from our Experience," *ACM SIGSOFT Software Engineering Notes* vol. 32, pp. 1-8, May 2007.
[28] http://www.smartwaretechnologies.com.
[29] J. Yan and J. Zhang, "Backtracking Algorithms and Search Heuristics to Generate Test Suites for Combinatorial Testing" in *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, September 2006, pp. 385-394
[30] M. B. Cohen, "Designing Test Suites for Software Interaction Testing." PhD thesis: University of Auckland, 2004.
[31] T. Shiba, T. Tsuchiya, and T. Kikuno, "Using Artificial Life Techniques to Generate Test Cases for Combinatorial Testing," in *28th Annual Intl. Computer Software and Applications Conference (COMPSAC'04)*, Hong Kong, China, September 2004, pp. 72-77.
[32] http://www.alphaworks.ibm.com/tech/whitch.
[33] http://www.site.uottawa.ca/~awilliam.
[34] http://sourceforge.net/projects/tvg.
[35] http://www.burtleburtle.net/bob/math.