Advances in Computer Science and Information Technology (ACSIT)
Print ISSN: 2393-9907; Online ISSN: 2393-9915; Volume 1, Number 2; November, 2014 pp. 86-89

http://www.krishisanskriti.org/acsit.html

# An Anecdote to Automated Test Case Generation Techniques using GUI and Mutation Testing

**Kumar Gaurav**

*1Maharaja Surajamal Institute Janak Puri New Delhi*
*E-mail: kgsingh81@gmail.com*

**Abstract:** *Software testing has been defined by various eminent scholars in the past. A definition by Abran and Moore (2004) defines testing as "an activity performed for evaluating product quality and for improving it, by identifying defects and problems". Further, automation in software testing involves use of special software to control the execution of tests and the comparison of actual outcomes with predicted outcomes. Test automation can automate some repetitive but necessary tasks in a formalized testing process already in place, or add additional testing that would be difficult to perform manually. The broad areas of testing are code driven testing, GUI testing and API driven testing.*

*Automated GUI became prominent because of issues raised with Manual GUI testing. An Automated GUI Testing tool can playback all the recorded set of tasks, compare the results of execution with the expected behavior and report success or failure to the test engineers. Once the GUI tests are created they can easily be repeated for multiple numbers of times with different data sets and can be extended to cover additional features at a later time. Automated GUI Testing is a more accurate, efficient, reliable and cost effective replacement to manual testing.*

*Mutation testing is another method of software testing in which program or source code is deliberately manipulated, followed by suite of testing against the mutated code. The mutations introduced to source code are designed to imitate common programming errors. A good unit test suite typically detects the program mutations and fails automatically.This paper makes an attempt to review the extant literature and explore test case generation techniques based on GUI and mutation testing.*

**Keywords**: *Testing, GUI, Mutation testing*

## 1. INTRODUCTION

Some software testing tasks, such as extensive low-level interface regression testing, can be laborious and time consuming to do manually. In addition, a manual approach might not always be effective in finding certain classes of defects. Test automation offers a possibility to perform these types of testing effectively. Once automated tests have been developed, they can be run quickly and repeatedly. This paper makes an attempt to review the extant literature and explore test case generation techniques based on GUI and mutation testing.

## 2. SOFTWARE TESTING

According to the IEEE Software Engineering Body of Knowledge testing is "anactivity performed for evaluating product quality, and for improving it, by identifying defects and problems.

Software testing consists of the dynamic verification of the behavior of a program. Myers (1979)defines testing as "the process of executing a program with the intent of finding errors". According toAmmann and Offutt (2008) testing means "evaluating software by observing its execution". Utting andLegeard (2007) names testing "the activity of executing a system in order to detect failures". Whittaker(2000) says that "software testing is the process of executing a software system to determine whether itmatches its specification and executes in its indented environment".All these definitions show that the inherent nature of software testing is the execution of the implementationunder test.Further, the purpose of testing is to identifyfailures and problems when the software does not behave as expected.

### 2.1 Approaches to Test Automation

In software testing, test automation is the use of special software to control the execution of tests and the comparison of actual outcomes with predicted outcomes. Test automation can automate some repetitive but necessary tasks in a formalized testing process already in place, or add additional testing that would be difficult to perform manually. There are many approaches to test automation; however below are the general approaches used widely:

Code-driven testing: The public interfaces to classes, modules or libraries are tested with a variety of input arguments to validate that the results that are returned are correct.

GUI testing: A testing framework generates user interface events such as keystrokes and mouse clicks, and observes the changes that result in the user interface, to validate that the observable behavior of the program is correct.

API driven testing: A testing framework that uses programming interface of the application to validate, the behavior under test. Typically API driven testing bypasses application user interface altogether.

Code-driven testing: A growing trend in software development is the use of testing frameworks such as the xUnit frameworks (for example, JUnit and NUnit) that allow the execution of unit tests to determine whether various sections of the code are acting as expected under various circumstances. Code driven test automation is a key feature of agile software development, where it is known as test-driven development (TDD). Unit tests are written to define the functionality *before* the code is written. However, these unit tests evolve and are extended as coding progresses, issues are discovered and the code is subjected to refactoring. Only when all the tests for all the demanded features pass is the code considered complete.

**Graphical User Interface (GUI) testing**

Many test automation tools provide record and playback features that allow users to interactively record user actions and replay them back any number of times, comparing actual results to those expected. The advantage of this approach is that it requires little or no software development. This approach can be applied to any application that has a GUI. However, reliance on these features poses major reliability and maintainability problems. Relabelling a button or moving it to another part of the window may require the test to be re-recorded. Record and playback also often adds irrelevant activities or incorrectly records some activities.

A variation on this type of tool is for testing of websites. Here, the "interface" is the webpage. This type of tool also requires little or no software development. However such a framework utilizes entirely different techniques because it is reading HTML instead of observing window events.

Another variation is script-less test automation that does not use record and playback, but instead builds a model of the Application Under Test(AUT) and then enables the tester to create test cases by editing in test parameters and conditions.This requires no scripting skills, but has all the power and flexibility of a scripted approach.] Test-case maintenance seems to be easy, as there is no code to maintain and as the AUT changes the software object objects can simply be re-learned or added. It can be applied to any GUI-based software application.] Theproblem is the model of AUT is actually implemented using test scripts, which have to be constantly maintained whenever there is change to the AUT.

API driven testing: API driven testing is also being widely used by software testers as it's becoming tricky to create and maintain GUI-based automation testing. Programmers or testers write scripts using a programming or scripting language that calls interface exposed by the application under test. These interfaces are custom built or commonly available interfaces like COM, HTTP, and Command line interface. The test scripts created are executed using an automation framework or a programming language to compare test results with expected behaviour of the application.

**2.1.1 The Testing Process**

Software testing in various stages of the development lifecycle constitutes of three parts: selection or generation of specific test cases, execution of these test cases, and evaluation of not only the quality of the software under test but also of the test cases themselves. That is, the test effort also needs to be evaluated for its thoroughness.

Test case generation involves selecting a particular set of test cases (a test suite) within an often practically infinite domain of program execution. Various mechanisms for systematically generating test cases with different selection criteria have been proposed, but test case generation is still often left to the programmer. It is no surprise that test case execution, being the most amenable to automation, has the most sophisticated automation tools available.

When test cases are executed, it should achieve the twofold goal of finding defects, andincreasing confidence in the quality of the software under test. To detect defects, it mustbe possible to compare the state of the computation after a test case is run with a specified expected state. Often this comparison is done by consulting an oracle—a software artifact that decides whether a test case has passed or failed. Oracles themselves are often either manually constructed, or automatically derived from a software system's specifications.

Even if no defects were found during testing, however, no guarantee can be made that the software under test is defect-free. That testing can show the presence of bugs but not their absence [1]. However, we can have some metrics that give a sense of the defect revealing capabilities of our test suite. Various program coverage metrics have been traditionally used for this.

## 3. AUTOMATED TEST CASE GENERATION- A REVIEW

The main aim of this paper is to present various techniques available for test case generation.GUI based testing and mutation testing have been specifically chosen through extant literature review.

### 3.1 GUI Testing

Although the use of GUIs continues to grow, GUI testing has remained a neglected research area. GUI based testing is still in a nascent stage and little research has been done in this area. But there is potential to use techniques from general software

testing and tailor them for GUI testing. A number of research efforts have addressed the automation of the test case generation for GUIs. Several finite-state machine (FSM) models have been proposed to generate test cases [2, 3]. In this approach, the software's behaviour is modelled as a FSM where each input triggers a transition in the FSM. A path in the FSM represents a test case, and the FSM's states are used to verify the software's state during test case execution.This approach has been used extensively for test generation of hardware circuits [4].

Avritzer et al. [5] have proposed a technique for software load testing, which hascharacteristics that may be relevant to GUI testing. This technique assesses how the system performs under a given load.The goal of this technique is to generate test cases to test software's resource allocation strategies rather than its functionality. Load testing is done after the software has been thoroughly tested for correctness of functionality. The testcase generation process uses an operational profile that describes the expected workload of the software once it is operational. The operational profile consists of the number andtypes of inputs to the software, the probability distribution of each type of input, and the average input arrival rate. This type of testing is attractive for GUIs since it is possible to obtain similar profiles from user sessions recorded during usability testing. However, amajor limitation of this technique is that the software has to be represented by a Markovchain model. GUIs have a large number of states, and a state description that encodes asequence of states may be impractical.

### 3.2 Mutation Testing

Mutation Testing is a fault-based testing technique which provides a testing criterion called the "mutation adequacy score". Themutation adequacy score can be used to measure the effectiveness of a test set in terms of its ability to detect faults. The general principle underlying Mutation Testing work is that the faults used by Mutation Testing represent the mistakes that programmers often make. By carefully choosing the location andtype of mutant, we can also simulate any test adequacy criteria.Such faults are deliberately seeded into the original program, by simple syntactic changes, to create a set of faulty programs called mutants, each containing a different syntactic change. To assessthe quality of a given test set, these mutants are executed against the input test set. If the result of running a mutant is differentfrom the result of running the original program for any test casesin the input test set, the seeded fault denoted by the mutant isdetected. One outcome of the Mutation Testing process is mutation score, which indicates the quality of input test set. The One outcome of the Mutation Testing process is themutation score, which indicates the quality of the input test set.The mutation score is the ratio of the number of detected faultsover the total number of the seeded faults.

The history of Mutation Testing can be traced back to 1971 ina student paper by Richard Lipton [144]. The birth of the fieldcan also be identified in papers published in the late 1970s byDeMillo et al. [66] and Hamlet [107].Mutation Testing can be used for testing software at the unitlevel, the integration level and the specification level. It has beenapplied to many programming languages as a white box unit testtechnique, for example, C programs [6], [7],[8], [9], [10] Java programs [11], [12], [13], [14], C# programs [15]–[19], SQL code [20], [21],[23], [24] and AspectJ programs [25], [26], [27]. Besides using Mutation Testing at the software implementationlevel, it has also been applied at the design level to test thespecifications or models of a program. For example, at the designlevel Mutation Testing has been applied to Finite State Machines[28], [29], [30], Security Policies [31], [32], [33], and Web Services. Mutation Testing has been increasingly and widely studiedsince it was first proposed in the 1970s. There has been muchresearch work on the various kinds of techniques seeking toturn Mutation Testing into a practical testing approach. However,there is little survey work in the literature on Mutation Testing.

## 4. CONCLUSION

Both the GUI based testing and Mutation testing has not been extensively reviewed in the past. Both the techniques have numerous applications as researched upon the authors in the past. The paper outlines these applications which can be of help in future works.

## REFERENCES

[1] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "The Design of a Prototype Mutation System for Program Testing," in *Proceedings of the AFIPS National Computer Conference*, vol. 74. Anaheim, New Jersey: ACM, 5-8 June 1978, pp. 623–627.

[2] Chow, T. S., "Testing software design modeled by finite-state machines", IEEE trans. on Software Engineering SE-4, 3 (1978), pp.178-187.

[3] Clarke, J. M., "Automated test generation from a behavioral model", In Proceedings of Pacific Northwest Software Quality Conference (May 1998), IEEE Press.

[4] H. Cho, G.D. Hachtel and F. Somenzi, "Redundancy identification/removal and test generation for sequential circuits using implicit state enumeration", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 12, 7 (July 1993), pp. 935- 945.

[5] Avritzer, A., and Weyuker, E. J., "The automatic generation of load test suites and the assessment of the resulting software", IEEE Transactions on Software Engineering 21, 9 (Sept. 1995), pp. 705- 716.

[6] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford, "Design of Mutant Operators for the C Programming Language," Purdue University, West Lafayette, Indiana, Technique Report SERC-TR-41-P, March 1989

[7] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Interface Mutation: An Approach for Integration Testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 3, pp. 228–247, May 2001.

[8] A. K. Ghosh, T. O'Connor, and G. McGraw, "An Automated Approach for Identifying Potential Vulnerabilities in Software," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P'98)*, Oakland, California, 3-6 May 1998, pp. 104–114.

[9] H. Shahriar and M. Zulkernine, "Mutation-Based Testing of Buffer Overflow Vulnerabilities," in *Proceedings of the 2nd Annual IEEE International Workshop on Security in Software Engineering*, 28 July -1 August, Turku, Finland 2008, pp. 979–984.

[10] H. Shahriar and M. Zulkernine, "Mutation-Based Testing of Format String Bugs," in *Proceedings of the 11th IEEE High Assurance SystemsEngineering Symposium (HASE'08)*, Nanjing, China, 3-5 Dec 2008, pp. 229–238

[11] P. Chevalley, "Applying Mutation Analysis for Object-oriented Programs Using a Reflective Approach," in *Proceedings of the 8th Asia-Pacific Software Engineering Conference (APSEC 01)*, Macau, China, 4-7 December 2001, p. 267.

[12] P. Chevalley and P. Th´evenod-Fosse, "A Mutation Analysis Tool for Java Programs," *International Journal on Software Tools for Technology Transfer*, vol. 5, no. 1, pp. 90–103, November 2002.

[13] Y.-S. Ma, Y.-R. Kwon, and A. J. Offutt, "Inter-class Mutation Operators for Java," in *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*. Annapolis, Maryland: IEEE Computer Society, 12-15 November 2002, p. 352.

[14] Y.-S. Ma, A. J. Offutt, and Y.-R. Kwon, "MuJava: An Automated Class Mutation System," *Software Testing, Verification & Reliability*, vol. 15, no. 2, pp. 97–133, June 2005.

[15] A. Derezi´nska, "Object-oriented Mutation to Assess the Quality of Tests," in *Proceedings of the 29th Euromicro Conference*, Belek, Turkey, 1-6 September 2003, pp. 417– 420.

[16] A. Derezi´nska, "Advanced Mutation Operators Applicable in C# Programs," Warsaw University of Technology, Warszawa, Poland, Technique Report, 2005.

[17] A. Derezi´nska, "Quality Assessment of Mutation Operators Dedicated for C# Programs," in *Proceedings of the 6th International Conference on Quality Software (QSIC'06)*, Beijing, China, 27-28 October 2006.

[18] A. Derezi´nska and A. Szustek, "CREAM- A System for Object-Oriented Mutation of C# Programs," Warsaw University of Technology, Warszawa, Poland, Technique Report, 2007.

[19] A. Derezi´nska and A. Szustek, "Tool-Supported Advanced Mutation Approach for Verification of C# Programs," in *Proceedings of the 3th International Conference on Dependability of Computer Systems (DepCoS-RELCOMEX'08)*, Szklarska Porˆeba, Poland, 26-28 June 2008, pp. 261–268.

[20] W. K. Chan, S. C. Cheung, and T. H. Tse, "Fault-Based Testing of Database Application Programs with Conceptual Data Model," in *Proceedings of the 5th International Conference on Quality Software (QSIC'05)*, Melbourne, Australia, 19 -20 September 2005, pp. 187–196.

[21] H. Shahriar and M. Zulkernine, "MUSIC: Mutation-based SQL Injection Vulnerability Checking," in *Proceedings of the 8th International Conference on Quality Software (QSIC'08)*, Oxford, UK, 12-13 August 2008, pp. 77–86.

[22] J. Tuya, M. J. S. Cabal, and C. de la Riva, "SQLMutation: A Tool to Generate Mutants of SQL Database Queries," in *Proceedings of the 2nd Workshop on Mutation Analysis MUTATION'06)*. Raleigh, North Carolina: IEEE Computer Society, November 2006, p. 1.

[23] J. Tuya, M. J. S. Cabal, and C. de la Riva, "Mutating Database Queries," *Information and Software Technology*, vol. 49, no. 4, pp. 398–417, April 2007.

[24] S. Lee, X. Bai, and Y. Chen, "Automatic Mutation Testing and Simulation on OWL-S Specified Web Services," in *Proceedings of the 41st Annual Simulation Symposium (ANSS'08)*, Ottawa, Canada., 14-16 April 2008, pp. 149–156.

[25] S. D. Lee, "Weak vs. Strong: An Empirical Comparison of Mutation Variants," Masters Thesis, Clemson University, Clemson, SC, 1991.

[26] S. C. Lee and A. J. Offutt, "Generating Test Cases for XML-Based Web Component Interactions Using Mutation Analysis," in *Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE'01)*, Hong Kong, China, November 2001, pp. 200–209.

[27] J. B. Li and J. Miller, "Testing the Semantics of W3C XML Schema," in *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, Turku, Finland, 26-28 July 2005, pp. 443–448.

[28] S. S. Batth, E. R. Vieira, A. R. Cavalli, and M. U. Uyar, "Specificationof Timed EFSM Fault Models in SDL," in *Proceedings of the 27th IFIPWG 6.1 International Conference on Formal Techniques for Networkedand Distributed Systems (FORTE'07)*, ser. LNCS, vol. 4574. Tallinn,Estonia: Springer, 26-29 June 2007, pp. 50–65.

[29] N. Bombieri, F. Fummi, and G. Pravadelli, "A Mutation Model forthe SystemC TLM2.0 Communication Interfaces," in *Proceedings ofthe Conference on Design, Automation and Test in Europe (DATE'08)*,Munich, Germany, 10-14 March 2008, pp. 396–401.

[30] S. P. F. Fabbri, M. E. Delamaro, J. C. Maldonado, and P. Masiero, "MutationAnalysis Testing for Finite State Machines," in *Proceedings ofthe 5th International Symposium on Software Reliability Engineering*,Monterey, California, 6-9 November 1994, pp. 220–229.

[31] Y. Le Traon, T. Mouelhi, and B. Baudry, "Testing Security Policies:Going Beyond Functional Testing," in *The 18th IEEE InternationalSymposium on Software Reliability*. Trollh¨attan, Sweden: IEEEComputer Society, 5-9 November 2007, pp. 93–102.

[32] T. Mouelhi, F. Fleurey, and B. Baudry, "A Generic Metamodel ForSecurity Policies Mutation," in *Proceedings of the IEEE InternationalConference on Software Testing Verification and Validation Workshop*.IEEE TRANSACTIONS ON SOFTWARE ENGINEERING 29*(ICSTW'08)*. Lillehammer, Norway: IEEE Computer Society, 9-11April 2008, pp. 278–286.

[33] T. Mouelhi, Y. Le Traon, and B. Baudry, "Mutation Analysis forSecurity Tests Qualification," in *Proceedings of the 3rd Workshop onMutation Analysis (MUTATION'07)*, published with *Proceedings ofthe 2nd Testing: Academic and Industrial Conference Practice andResearch Techniques (TAIC PART'07)*. Windsor, UK: IEEE ComputerSociety, 10-14 September 2007, pp. 233–242.