# Accessing and Evaluating AspectJ based Mutation Testing Tools

Mayank Singh
Research Sholar,
Uttarakhnd Technical University,
CSED, JPIET, Meerut

Shailendra Mishra
CSED, Bipin Chandra Tripathi Kumaou Engineering College, Dwarahat,
Dstt. Almora, UK

Rajib Mall
Department of Computer Sciecne & Engineering,
IIT, Kharagpur, WB

## ABSTRACT
Software testing plays a crucial role in software development life cycle. Without testing, quality of software product is questionable. Mutation testing, widely accepted fault based testing technique. Aspect Oriented Programming is a new methodology that introduces the concept of modularization. AspectJ is an aspect oriented programming language that provides the concept of pointcut and advice. With new features, AOP introduces new faults that can be easily handled by mutation testing. In this paper, we evaluate the available AspectJ based mutation testing tools and identify the basic requirements that must be satisfied by any developed tool.

## Keyword
Mutation testing, Automated Mutation Testing tool, Fault based mutation testing tool, Mutation testing tool for AspectJ, AOP based Mutation testing tool.

## 1. INTRODUCTION
Software testing is the activity of establishing confidence that a system does what it is supposed to do and does not what it is not supposed to do. Since it is impossible to build an error-free system, testing a system is an essential process in software development. Thus, a great deal of research on software testing has been carried out for many years [2,4].

Traditionally, software systems have been developed in a procedural environment and then in object oriented environment. Recently, a new approach to system decomposition has become popular called aspect-oriented programming (AOP), which makes it possible to clearly express programs involving such aspects, including appropriate isolation, composition and reuse of the aspect code [5,12].

The advent of object-oriented methodologies pulled the state of the system into individual objects, where it could be made private and controlled through access methods and logic [31]. This leads to the current situation: Developers are still having difficulty fully expressing a problem into a completely modular and encapsulated model. Although breaking a problem into objects makes sense, some pieces of functionality must be made available across objects. Aspect-oriented programming (AOP) is one of the most promising solutions to the problem of creating clean, well-encapsulated objects without extraneous functionality.

The purpose of this work is to attempt to evaluate advantages and limitations of the current AspectJ based mutation testing tools. In this paper, we will find out the basic requirements to develop a mutation testing tool for AspectJ based systems. In this paper, we will compare the available AspectJ based mutation testing tools on the basis of requirements and identify the limitations. The structure of the paper is as follows. Section 2 describes the mutation testing process. Section 3 describes the current aspect oriented mutation testing tools. Section 4 describes the basic requirements to develop a mutation testing tool. Finally, section 5 makes a conclusion on the requirements on the basis of current aspect oriented mutation testing tools and proposes future research plans.

## 2. MUTATION TESTING
Fault-based testing strategies test software by generating test data that will find specific, common types of faults. Mutation testing is a fault-based testing technique, proposed by DeMillo, Lipton, and Sayward [1] in 1978. Mutation testing is a software analysis method in which faults are deliberately injected into a program, in order to determine whether or not a set of test inputs can distinguish between the original program and the programs with injected faults [3]. Mutation analysis is based on the adequacy criterion that seeks to measure the quality of test data used to exercise a given program (mutation adequacy) [4]. The quality of a test set is related to the ability of that test set to differentiate the program being tested from a set of marginally different, and presumably incorrect, alternate programs. Thus, the goal of the tester during mutation analysis is to create test cases that differentiate each mutant program from the original program by causing the mutant to produce different output [32].

The two basic assumptions underlying the mutation technique are the competent programmer hypothesis and the coupling effect [33, 34]. The competent programmer hypothesis states that the competent programmer will produce programs, which, if not actually correct, are close to being so. In other worlds, a program written by a competent programmer may be incorrect, but it will differ from a correct version by relatively simple faults. The coupling effect states that complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect most complex faults [35,36].

## 2.1 Mutation testing process

The processes of traditional mutation testing are as in the following [37,38].

i. Construct the mutants of a test program.
ii. Add test cases to the mutation system (generated manually or automatically).
iii. The test case is first executed against the original version of the test program, then checks the output of the program on each test case to see if it is correct.
iv. If the output is incorrect, a fault has been found and the program should be corrected and the mutation process restarted. If correct, that test case is executed against each live mutant.
v. The output of mutant program is compared to the expected output. If the output of a mutant differs from that of the original program on the same test case, the mutant is killed.
vi. After all of the test cases have been executed against all of the live mutants, each remaining mutant falls into one of these two categories:
   - Equivalent mutants: Once identified as an equivalent mutant, there is no need for the mutant to remain in the system for further consideration.
   - The mutant is killable, but the test set of test cases is insufficient to kill it. In this case, new test cases need to be created to kill the remaining live mutants.
vii. The process of adding test cases, examining expected output, and executing mutants continues until the tester is satisfied with the number of dead mutants.

This testing process is graphically shown in Figure 2. The solid boxes represent steps that are automated by traditionally, and the dashed boxes represent steps that are done manually [37].

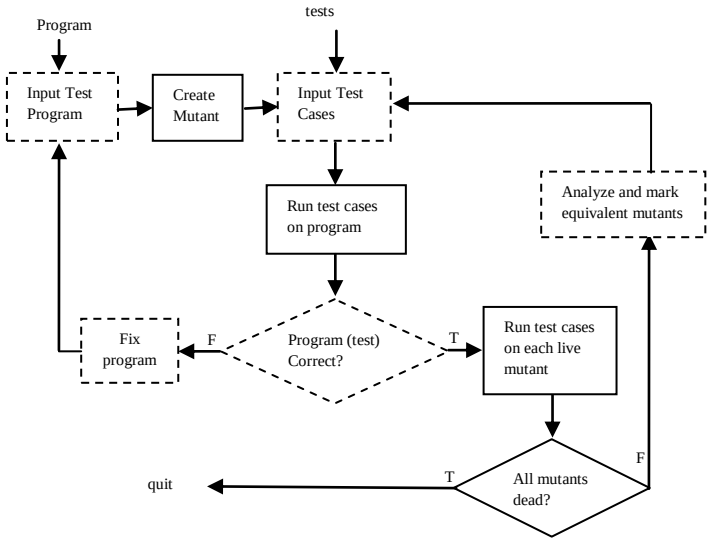


**Figure 1: Traditional Mutation Testing Process [30]**

# 3. CURRENT ASPECTJ BASED MUTATION TESTING TOOLS

## 3.1 MuAspectJ

Jckson and Clarke proposed a mutation testing tool for aspect oriented programming named, MuAspectJ [7] . MuAspectJ generates mutants for AspectJ programs based on aspect oriented and non-aspect oriented specific mutation operators. MuAspectJ evaluated in terms of the quality of generated mutants. To evaluate the quality of mutants benchmarking metrics is used against well known Java mutation testing tool, MuJava [8]. The quality is in terms of location coverage and mutation density. Location coverage is a measure of the proportion of locations for which mutants are generated. Mutation density is a measure of the number of mutants that are generated for a location [8,9].

Mutation analysis is the way to measure testability and can be used in testing experiments. Primary goal to develop MuAspectJ is to measure the testability of AspectJ programs through experiments. To generate and evaluate mutants, Health Watcher system is used [10,11]. MuAspectJ uses pointcut, advice and declarations locations to implement mutation operators for mutant generation [12].

MuAspectJ is implemented as an eclipse plug-in that operates on AspectJ projects. Tool implementation uses some components like Source File Finder component, which identifies all Java and AspectJ source files in an AspectJ project under analysis, Parser, that creates a Document Object Model (DOM) to represent the source, and AspectJ Compiler, that is used to compile each candidate mutant.

## 3.2 AJMutator

Delemare presents a mutation testing tool for mutation analysis of AspectJ Pointcut Descriptors named, AjMutator [13]. To generate a set of mutants, AjMutator implements several mutation operators that introduce faults in pointcut descriptors [14]. AjMutator classifies the mutants according to the set of joinpoints they match compared to the set of joinpoints matched by the initial PCD. An interesting result is that this automatic classification can identify equivalent mutants for a particular class of PCDs. AjMutator can also run a set of test cases on the mutants to give a mutation score.

AOP introduces new kinds of fault types that should be addressed by testing techniques. Faults can be located in the advice, in the PCD or can arise from the composition of the aspects. The PCD is the place that is the most fault-prone in an aspect, as observed by Ferrari et al. [12].

AjMutator automatically classifies the mutants by comparing the sets of joinpoints matched by the mutant and the initial PCD. They automate this classification at compile time by leveraging the static analysis performed by the compiler that computes the set of joinpoints matched by the PCDs. This classification is benefit to conclude the equivalent mutants if the mutants matches the same set of joinpoints. If the set of joinpoint is different, the advice is not correctly woven, and it can cause huge side effects.

AjMutator is separated in three distinct parts:

1. The generation of mutant source files from AspectJ source file
2. The compilation of the mutant source files
3. The execution of a test cases on the mutants to calculate the mutation score of this set of test cases

The component, parser builds an abstract-syntax tree (AST) for each PCD in the AspectJ source files. A pretty-printer then produces a mutant AspectJ source file for each mutant AST. The parser has been developed using SableCC [15], an open-source compiler generator. The mutation operators are implemented using the visitor pattern. After the mutants have been generated, they need to be compiled. It relies on the abc compiler [16], which is an alternative compiler for AspectJ. The information is then used by AjMutator to classify the

mutants. The accuracy of the classification process depends on whether the original PCD of the mutant is static or dynamic.

The goal of a mutation analysis is to evaluate a test suite with a mutation score. AjMutator relies on JUnit for the test cases . A mutant is killed if at least one test case has a different result on the mutant system. So if all the test cases pass on the original system, a mutant is killed if at least one test case fails. Two different systems are used to evaluate AjMutator. The first system is an Auction system, and the second is the Health-Watcher [17].

## 3.3 ProteumAj

The tool implements reference architecture for software testing tools named RefTEST [20], from which the main functional modules were derived. Proteum/AJ supports the four main steps of mutation testing, as originally described by DeMillo et al. [1]:

(i) The original program is executed on the current test set and test results are stored;
(ii) The mutants are created based on a mutation operator selection that may evolve in new test cycle iterations;
(iii) The mutants can be executed all at once or individually, as well as the test set can be augmented or reduced based on specific strategies; and
(iv) The test results are evaluated so that mutants may be set as dead or equivalent, or mutants may remain alive.

Main input of Proteum/AJ is the target application that must be a compressed file. This file contains all modules (classes, aspects and libraries) of the application under test. The Application Handler module then runs a pre-processing step, whose outputs are the decompressed original application and a list of target aspects. The decompressed application is sent to the Test Runner module together with the test case files. The Test Runner executes the application on the available test set by invoking the JUnit Ant task. The results are stored for further evaluation of mutants.

The Mutation Engine receives as input the list of target aspects identified by the Application Handler and the set of mutation operators selected by the tester. It produces the set of mutants that are passed to the Mutant Compiler. This module invokes the ajc compiler through the iajc Ant task provided with the AspectJ API [24]. The Mutant Compiler detects non-compilable mutants which are classified as anomalous. For compileable mutants, the weaving information produced by the ajc compiler is collected at this stage and further used by the Mutant Analyser module. Proteum/AJ runs JUnit test cases to evaluate the mutants.

## 3.4 Advice Tracer

Delamare proposes a test-driven approach for the development and validation of the PCD. They developed a tool, AdviceTracer [25], which enriches the JUnit API with new types of assertions that can be used to specify the expected joinpoints. AdviceTracer can determine at runtime which advice (defined in a particular aspect) is executed and at which place in the base program. This information can then be used to build oracles that specifically target the presence or absence of an advice, and do not just check if the advice executes correctly [12].

The AdviceTracer tool [25] allows a programmer to write test cases that focus on checking whether or not a joinpoint has been matched by the PCD [27]. More precisely, AdviceTracer is used to specify an oracle that expects the presence or absence of an advice at a particular point in the base program. Test cases can specify the PCD without executing the behavior of the advice [26].

## 3.5 Angalabagan & Xie' Tool

Angalabagan proposed a new framework that automatically identifies the strength of each pointcut and generates pointcut mutants with different strengths [6]. Developers can inspect the pointcut mutants and their join points for pointcut correctness or choose the mutants for conducting mutation testing. They conducted an empirical study on applying our framework on pointcuts from existing AspectJ programs [12]. The results show that the framework can provide valuable assistance in generating effective mutants that are close to the original pointcuts and are of appropriate strength.

The proposed framework serves the following purposes: generating relevant mutants and detecting equivalent mutants. Finally the framework reduces the total number of mutants from the large number of initial generated mutants. The framework also classifies the mutants and ranks them using a string similarity measure to help the developer choose a mutant that resembles closely the original one.

The input to the framework is AspectJ source code and Java bytecode of the base program. The output from framework is a ranked list of pointcut mutants for each original pointcut in the AspectJ source code and the differences of the join points matched by the original pointcut and the pointcut mutants.

The main components of framework are: pointcut parser, which identifies pointcuts in the given AspectJ source code, joint point candidate identifier, which identifies the join point candidates from the given Java bytecode for the base program, mutant generator, which forms mutants for the pointcuts identified by the pointcut parser, and pointcut tester, which verifies the join point candidates identified by the candidate identifier against a pointcut identified by the pointcut parser. In general, the pointcut tester, developed based on an AspectJ unit testing framework [28], can be used to verify pointcuts of an aspect class without weaving the aspect code to the base program.

# 4. REQUIREMENTS TO DEVELOP A MUTATION TESTING TOOL

From the evolution of these aspect oriented mutation testing tools, we identifies some requirements that should be provided by mutation based testing tool [18,28,29]. The identified requirements are as follows:

1. Mutant generation level: It includes the generation of mutants either byte code level or source code level.
2. Produce Non-Executable Mutants: It includes the generation of non executable mutants. If mutants are not executable then the mutants is anomalous that is not included in mutation analysis.
3. Mutants Format: It includes the format of generated mutants i.e. Separate Class File, Separate Source File, In-Memory or Grouped in Source Files.
4. JUnit Support: For test cases generation JUnit is used. This requirement includes the use of JUnit.
5. Handling of Test Cases: It includes the execution of test cases, activation or deactivation of test cases.
6. Handling of Mutants: It includes the generation of mutants, selection of mutants, execution and analysis of mutants.
7. Adequacy Analysis: It includes the calculation of mutation score on the basis of total used mutants, equivalent mutants and dead mutants.

8. Test Case Reduction: It includes the reduction of used test cases by eliminating the redundant test cases.
9. Unrestricted Program Size: This requirement includes the used size of program for testing.
10. Support for testing Strategies: It includes the order of mutation operators to apply on the target application.
11. Independent Test Configuration: Test input and output should not be restricted by the tool.
12. Test case generation: Automatically generation of test cases should be included by the tool.
13. Test case editing: It includes the modification of existing test cases or alteration of available test cases.
14. Interface: Which types of interface are including to test the target programs i.e. menu or command line or code browser.
15. Automatic program execution: It includes the execution of original programs as well as mutants to be executed or compiles automatically.
16. Evolution of Equivalent mutants: This requirement includes the generation of equivalent mutants and makes a procedure to record the equivalent mutants.
17. Test phase supported: It includes the supporting test phases i.e. unit, integration or system level.

The following table 1 shows the limitations and advantages of AspectJ based mutation testing tools on the basis of identified requirements.

**Table 1: Basic Requirements to Develop AspectJ Based Mutation Testing Tool**

| Req. | MuAspectJ | AjMutator | Proteum/ AJ | Advice Tracer | Anbalagan & Xie |
|---|---|---|---|---|---|
| Mutant Generation Level | Source Code | Source Code | Source Code | Source Code | Source Code |
| Produce Non-Executable Mutants | Partial | Partial | Partial | Partial | No |
| Mutant Format | Grouped in Source File | Separate Source File | Separate Source File | Grouped in Source File | Grouped in Source File |
| JUnit Support | Yes | Yes | Yes | Yes | No |
| Handling of Test Cases | Partial | Partial | Partial | Partial | No |
| Handling of Mutants | Partial | Partial | Yes | No | Partial |
| Adequacy Analysis | Partial | Partial | Partial | Partial | Partial |
| Unrestricted Program Size | No | No | Yes | No | No |
| Support for Testing Strategies | No | No | Partial | No | No |
| Independent Test Configuration | No | Yes | Yes | No | No |
| Test Case Generation | Yes | Automatic | Automatic | Yes | Yes |
| Test Case Editing | No | No | No | No | No |
| Interface | Browser Plug-in | Command Line | Command Line | Command Line | Command Line |
| Automatic Program Execution | Yes | Yes | Yes | No | Yes |
| Evolution of Equivalent Mutants | Yes | Yes | Yes | No | No |
| Test Phase Supported | Unit | Unit | Unit | Unit | Unit |

## 5. CONCLUSION AND FUTURE SCOPE

Despite the weakness of these mutation testing tools, we find them indispensable. These tools provide a different way of testing research. Major weaknesses of these tools are performance, complexity and user interfaces. Major disadvantages of available AspectJ based mutation testing tool is that they support only specific technique. We should develop a complete testing tool which includes at least important testing techniques. Another drawback is that different tools provide different interfaces which are difficult to remember as well as complicated to handle`. Even for the same testing technique, features of different tools are different which makes it complex to choose the best tool. Use of different external tools to develop different testing tool is another drawback of these tool. AspectJ based system level mutation testing is completely missing. Our future scope is to develop an AspectJ based system level mutation testing tool with use of only one external tool named, JUnit. We will try to develop a complete mutation testing tool for unit as well as system level to overcome the drawbacks of currently available mutation testing tools.

## 7. REFERENCES

[1] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," Computer, vol. 11, no. 4, pp. 34–41, April 1978.
[2] Beizer B., Software Testing Techniques, 1990.
[3] Michael C, Promoting Firm Mutation Testing to Strong Mutation Testing with Stochastic Methods, TR 95-003-07, Nov. 1995.
[4] R. T. Alexander, J. M. Bieman, S. Ghosh, and B. Ji, "Mutation of Java Objects," in Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE' 02).

Annapolis, Maryland: IEEE Computer Society, pp. 341–351, 12-15 November 2002.

[5] P. Anbalagan and T. Xie, "Efficient Mutant Generation for Mutation Testing of Pointcuts in Aspect-Oriented Programs," in Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION' 06). Raleigh, North Carolina: IEEE Computer Society, p. 3, November 2006.

[6] P. Anbalagan and T. Xie, "Automated Generation of Pointcut Mutants for Testing Pointcuts in AspectJ Programs," in Proceedings of the 19th International Symposium on Software Reliability Engineering (ISSRE' 08). Redmond, Washington: IEEE Computer Society, pp. 239–248, 11-14 November 2008.

[7] Andrew Jackson and Siobhán Clarke, "MuAspectJ: Mutant Generation to Support Measuring the Testability of AspectJ Programs", Technical report (TCD-CS-2009-38), ACM, September 2009.

[8] 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.

[9] Jeff Offutt, Yu-SeungMa and Yong-Rae Kwon, "The Class-LevelMutants of MuJava", Workshop on Automation of Software Test (AST 2006). pages 78-84, Shanghai, China, May 2006,

[10] T. Sugeta, J. C. Maldonado, and W. E. Wong, "Mutation Testing Applied to Validate SDL Specifications," in Proceedings of the 16th IFIP International Conference on Testing of Communicating Systems, ser. LNCS, vol. 2978, Oxford, UK, p. 2741, 17-19 March 2004.

[11] Uira Kulesza, Claudio Sant'Anna, Alessandro Garcia, Roberta Coelho, Arndt von Staa, and Carlos Lucena, "Quantifying the effects of aspect-oriented programming: A maintenance study", in ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance, IEEE Computer Society, pages 223-233, Washington, DC, USA, 2006.

[12] Fabiano Cutigi Ferrari, Jose Carlos Maldonado, and Awais Rashid, "Mutation testing for aspect-oriented programs" in ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation, IEEE Computer Society, pages 52-61, Washington, DC, USA, 2008.

[13] R. Delamare, B. Baudry, and Y. Le Traon, "AjMutator: A Tool For The Mutation Analysis Of AspectJ Pointcut Descriptors," in Proceedings of the 4th International Workshop on Mutation Analysis (MUTATION'09), published with Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation Workshops. Denver, Colorado: IEEE Computer Society, pp. 200–204, 1-4 April 2009.

[14] Roger T. Alexander, Stephan Herrmann, Dehla Sokenou, "Testing aspect-oriented programming Pointcut Descriptors", in Proceedings of International conference AOSD, ACM, page no 33-38, 2006.

[15] E. M. Cagnon and L. J. Hendren. Sablecc, "An object oriented compiler framework", in TOOLS'98: Proceedings of the Technology of Object-Oriented Languages and Systems, IEEE Computer Society, pages 140–154, August 1998.

[16] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhot´ak, O. Lhot´ak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, " abc: an extensible aspectj compiler", in AOSD'05: Proceedings of the 4th international conference on Aspect-oriented software development, ACM, pages 87–98, New York, NY, USA, 2005.

[17] Healthwatcher. http://www.comp.lancs.ac.uk/

[18] Fabiano Cutigi Ferrari, Elisa Yumi Nakagawa, José Carlos Maldonado, Awais Rashid, "Proteum/AJ: a mutation system for AspectJ programs", in Proceedings of the tenth international conference on Aspect-oriented software development companion (AOSD'11), ACM New York, NY, USA , 2010.

[19] José Carlos Maldonado, Márcio Eduardo Delamaro, Sandra C. P. F. Fabbri, Adenilso da Silva Simão, Tatiana Sugeta, Paulo Cesar Masiero, Auri Marcelo Rizzo Vincenzi, "Proteum: A family of tools to support specification and program testing based on mutation", in Mutation 2000 Symposium (Tool Session), pages 113–116. Kluwer, 2000.

[20] E. Y. Nakagawa, A. S. Sim˜ao, F. C. Ferrari, and J. C. Maldonado, "Towards a reference architecture for software testing tools", in SEKE'07, pages 157–162, 2007.

[21] AspectJ-front. http://strategoxt.org/Stratego/Aspe ctJFront - accessed on 01/09/2011.

[22] Ant. http://ant.apache.org/ - accessed on 01/09/2011.

[23] iBATIS data mapper. http://ibatis.apache.org/ - accessed on 01/09/2011

[24] AspectJ documentation, 2010. http://www.eclipse.org/ aspectj/docs.php - accessed on 01/09/2011.

[25] R. Delamare, B. Baudry, , S. Ghosh, and Y. Le Traon, "A Test-Driven Approach to Developing Pointcut Descriptors in AspectJ," in Proceedings of the 2nd International Conference on Software Testing Verification and Validation (ICST' 09), Davor Colorado, pp. 376–38, 01-04 April 2009.

[26] L. Ye and K. D. Volder, "Tool support for understanding and diagnosing pointcut expressions", in AOSD'08: Proceedings of the 7th international conference on Aspect oriented software development, New York, NY, USA, ACM, pages 144–155, 2008.

[27] C. Koppen and M. Storzer, "Pcdiff: Attacking the fragile pointcut problem", in European Interactive Workshop on Aspects in Software (EIWAS), September 2004.

[28] J. Horgan and A. Mathur, "Assessing testing tools in research and education", IEEE Software, 9(3):61–69, 1992.

[29] L. Madeyski N. Radyk, "Judy – a mutation testing tool for Java", Published in IET Softw., Vol. 4, Iss. 1, pp. 32–42, 2010.

[30] Scholivé, M., Beroulle, V., Robach, C., Flottes, M.L., Rouzeyre, B., "Mutation Sampling Technique for the Generation of Structural Test Data", published in proceedings of Design, Automation and Test in Europe (DATE'05), 2005.

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

[32] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, and M. E. Delamaro, "Proteum/FSM: A Tool to Support Finite State Machine Validation Based on Mutation Testing," in Proceedings of the 19th International Conference of the Chilean Computer Science Society (SCCC' 99), Talca, Chile, p. 96, 11-13 November 1999.

[33] R. J. Lipton and F. G. Sayward, "The Status of Research on Program Mutation," in Proceedings of the Workshop on Software Testing and Test Documentation, pp. 355–373, December 1978.

[34] J. Offutt, "The Coupling Effect: Fact or Fiction," ACM SIGSOFT Software Engineering Notes, vol. 14, no. 8, pp. 131–140, December 1989.

[35] J. Offutt, "Investigations of the Software Testing Coupling Effect", ACM Transactions on Software Engineering and Methodology, vol. 1, no. 1, pp. 5–20, January 1992.

[36] K. S. H. T. Wah, "An Analysis of the Coupling Effect I: Single Test Data," Science of Computer Programming, vol. 48, no. 2-3, pp. 119– 161, August-September 2003.

[37] Offutt A., A Practical System for Mutation Testing: Help for the Common Programmer, Twelfth International Conference on Testing Computer Software, 99-109, Washington D.C. June 1995.

[38] DeMillo R., Constraint-Based Automatic Test Data Generation, IEEE Transactions on Software Engineering, 17(9): 900-910, 1991.