Automatic Generation and Execution of Mutants

Madhuri Sharma
Center for Development of Advanced Computing, Noida

Neha Bajpai
Center for Development of Advanced Computing, Noida

ABSTRACT

To obtain the high quality software, there is a use of Mutation testing to measure the quality of our test suite. Fault insertion based techniques have been used for measuring test adequacy and testability of programs. Mutants are generated by introducing the faults in the original program. Tests Cases are adequate if they detect all the mutants. This paper describes a survey study to investigate the generation and execution of mutants.

General Terms

Software Testing

Keywords

Software Testing, Mutation Testing, Mutation Testing Process, Cost Reduction Mutants.

1. INTRODUCTION

Software testing is an important phase of software development life cycle. Since, exhaustive testing of software is not possible, so many techniques are invoked. In software testing, a *failure* is an external, incorrect behavior of a program - incorrect output, or runtime failure. A *fault* is the incorrect statement in the program that causes a failure.
Automating software testing activities can increase the quality and drastically decrease the cost of software development. Test automation is the use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, and test reporting functions.

Object-oriented design programming and languages offers many advantages to software developers and provide the solutions to old problems. However, the object-oriented language features introduce the new kind of problems.

2. MUTATION TESTING

DeMillo et al. had described about the approach of seeding the faults into the program through various mutation operators. Mutation testing is a fault-based testing technique that measures the effectiveness of test suites for fault localization, based on seeded faults [4]. Faults (Syntactic changes) are introduced into the program and thus generate an error in the program, called mutants. Mutants are generated by applying the mutation operator say a relational operator replaces '>' with '>=', '<', '<=','==','! ='. Test cases are then applied on both the original program and mutants and to check whether the original program & mutants give the same result or not. If original program output is different than that of output with mutant, then the mutant is said to be *killed or dead* otherwise said to be as *live.*

Then, there can be two possibilities in case of live mutant: - either the generated mutant is equivalent or cannot be killed. When the mutants are not killed and is not able to differentiate between original program and non-equivalent mutants which means our test case is inadequate. So to kill that mutant we need more new test cases. A test suite that can kill all non-equivalent mutants is said to be *adequate*. The main objective of mutation testing is to kill more live mutants as more we can.

The goal of mutation testing is to assess the quality of tests and use these assessments to help construct more adequate test and thus produce a suite of valid tests which can be used on real programs.

3. MUTANT PROGRAM

Program mutation is a powerful technique for the assessment of the goodness of tests. It provides a set of strong criteria for test assessment and enhancement. Program mutation is a technique to assess and enhance your tests; it is referred to *program mutation* as *mutation*. When testers use mutation to assess the adequacy of their tests, and possibly enhance them, is known as *mutation testing*. Sometimes the act of accessing test adequacy using mutation is also referred to known as *mutation analysis.*

Mutation testing is a powerful technique for use during unit, integration, and system testing. Mutation testing works by seeding faults in the software program. Various mutation operators are used to create these faulty programs. These programs are called *mutants.* The mutants depict software faults that may be caused by programmers while writing the software. Test cases are then executed on these mutants to determine if they have been *killed* or not. Test sets that kill all the mutants are considered to be good as they successfully detect all the possible program faults.

- By modifying a program to contain simple errors and demanding that test data be discovered to distinguish the erroneous versions of the program from the original program, those simple errors can be guaranteed as absent from the original.

4. MUTATION OPERATORS

Mutation Operators are applied on the original program to generate the mutants. There are huge numbers of operators for procedural and object oriented programming. A procedural language contains the simple syntactic changes in the program such as changing the arithmetic, relational, logical operators. The object-oriented contains some extensions of mutation operators such as encapsulation, inheritance, polymorphism, overloading, exceptional handling etc.

Mutation Operators are of two types: -

1) Traditional Mutation Operators
2) Class Mutation Operators

4.1 Traditional Mutation Operators

The traditional mutation operators are developed from procedural languages. However, running all these mutant operators generates a huge number of mutants and not all of them are effective because of overlaps [5]. However, these operators are language specific.

4.1.1 Arithmetic Operators

Mutants are generated by replacing, inserting, deleting the arithmetic operator with the other arithmetic operators.

4.1.2 Relational Operators

Mutants are generated by replacing, inserting, deleting the relational operator with the other relational operator.

4.1.3 Conditional Operators

Conditional Operators generate the mutants by replacing, inserting, deleting with the other operators.

4.1.4 Logical Operators

Logical Operators produce the mutants by applying the other conditional operator in original program.

4.1.5 Assignment Operators

The assignment operators create the mutants by applying the other assignment operators but one at a time.

4.1.6 Shift Operators

A shift operator creates the mutants by applying the other shift operator in the original program.

4.2 Class Mutation Operators

Class mutation operators were identified for testing object-oriented and integration issues [6].

4.2.1 Encapsulation

In encapsulation, mutants are created by applying operator which modifies deletes, insert the access level for instance variables and methods to other access levels, in order to check that accessibility is correct.

4.2.2 Inheritance

The Inheritance operator produce the mutants by deleting a hiding variable, and to check that variable defined and its accessibility in class and subclass will be properly correct or not.

4.2.3 Polymorphism

The Polymorphism operator creates the mutants in such a way that accessibility of the method having the same name and number of parameters accessible in a right manner or not.

5. MUTATION SCORE

Mutation score defines the relationship between the number of mutants killed by the test suite and the difference between the total number of mutants and the number of equivalent mutants to the original program. The objective of mutation score is to evaluate the test set adequacy against mutation testing. The higher the mutation score, the more adequate is the test set.

Mutation score is calculated as:-

$$\text{Mutation Score} = 100 * K / (N - E)$$

Where, K = Killed mutants

N = Number of mutants

E = Number of equivalent mutants

A set of test cases is *mutation adequate* if its mutation score is 100%.

6. EXAMPLE

AOR (Arithmetic Operator Replacement)

The Arithmetic Operator Replacement (AOR) operator is a traditional mutation operator. The AOR operator replaces each occurrence of an arithmetic operator by each of the other possible arithmetic operators.

Example:

```
public class try1  {
public static void main(String args[]){
        int m, p, c, s=0, a=0;
                m = 9;     p = 86;     c = 86;
                a = p + c;
                s = a - m;
        System.out.println("p & c: \n " + a);
        System.out.println(" s: \n " + s);
}  }
```

So mutants are generated by changing the arithmetic operator (a = p + c) to the other operators, but do the one change at a time.

Mutants are:

m1: a = p – c;

m2: a = p * c;

m3: a = p / c;

But when the test cases are applied on both the original program and mutants program also, and check the output of mutants program against the original program. If they produce the different output then they are kill mutants otherwise live mutant.

7. EQUIVALENT MUTANTS

Equivalent mutants are those mutants whose outputs are same as those of the original output. However, a number of equivalent mutants are generated by applying mutation operators.

Example: Mutant is generated by applying the relational mutation operator as which replaces the (>=) operator with the (>) operator as shown below:

```
if(a>=b)                    if(a>b)
return (a+b);               return (a+b);
else                        else
return (a-b);               return (a-b);
```

When the input values (a,b) are given as (10,6) they both gives the output.

However, when test cases are applied on it, then there will be no single test case that could be able to kill this equivalent mutant. This mutant is one of the obstacles for practical use of mutation testing. There may be requirement of tricky process to identify the equivalent mutants.

8. EXISTING MUTATION TESTING TOOL FOR JAVA PROGRAMS

8.1 JUMBLE

Jumble is a Java mutation testing tool and the purpose of mutation testing is to provide a measure of the effectiveness of test cases. A single mutation is performed on the code to be tested; the corresponding test cases are then executed. If the modified code fails the tests, then this increases confidence in the tests. Conversely, if the modified code passes the tests this indicates a testing deficiency. Jumble is a simple non-graphic tool that converts a text file to a version that facilitates studying the format of the file. Jumble randomly changes the order of letters in the text file leaving punctuation and capitalization intact [4].

- Jumble is an open-source tool that operates directly at a source code level to speed up mutation testing.
- The limited sets of mutation operators supported by Jumble are: Conditional, Binary Arithmetic Operations, Increments, Inline Constants, Class Pool Constants, Return Values, and Switch Statements. They are simplified in such a way that only one of the mutations defined by the mutation operators is, in fact, applied (e.g. '-' is replaced by '+', and not by each of the other operators, i.e.'*','/', and '%', as the AOR mutation operator assumes).
- *Disadvantage*: Jumble does not support the OO mutation operators. It also provides the fixed replacements with the other mutation operators.

Table 1.Mutation Operators Supported by Jumble [4]

Operator	Source	Mutant
Conditional	if (a >b)	if (!(a >b))
Binary Arithmetic	c = a + b;	c = a - b;
Increments	i++;	i--;
Inline Constants	long x = 01;	long x = 11;
Class Pool Constants	public String welcomeStr = "Welcome!";	public String welcomeStr = "__jumble__";
Return Values	return returnStr;	return null;
Switch Statements	case 0: i++; case 1: i = 4;	case 1: i++; case 0: i = 4;

8.2 JESTER

Jester (a mutation testing tool) that finds code that is not covered by tests. Jester does the some modifications in source code and runs the test cases, and if the test case passes Jester displays a message what has changed. Jester that indicates if the tests still pass when the expression (<=) is replaced by (<). This indicates that a test might be missing in which it makes a difference that it's "<=" rather than "<". With Jester, it matter whether the tests pass or not [7].

Jester modifies the java source code and recompiles the modified source code and then the test cases have to be run to check the output. For instance, it will change if (x > y) to if (false). If the test suite isn't paying close enough attention to notice the change, then a test is missing.

- Jester is an open-source tool and a very expensive way of mutation testing tools for Java mutation testing, owing to an oversimplified mechanism of mutants' generation.
- Actually, Jester offers a way to extend the *default set of mutation operations*, but problems concerning performance of the tool, as well as a limited range of possible mutation operators.
- Nevertheless, in comparison to a code coverage tool, Jester can spot untested code even if it is executed.
- *Disadvantage*: Jester's approach is to generate, compile and run unit tests against a mutant. The process repeats for every mutant of the source code and, thus, is inefficient. However, Jester takes a long time to run, and the results take some manual effort to interpret.

8.3 JUDY

Judy is an open source tool implementation with the features of automatic mutation testing process and mutant generation approach and a large number of supporting mutation operators.

Table2. Mutation Operators Supported by Judy [4]

Abbreviation	Description	Example mutation
ABS	Absolute Value Insertion	x = 2*a; →x = 2*abs(a);
AOR	Arithmetic Operator Replacement	x = a + b; -> x = a * b;
LCR	Logical Connector Replacement	x = a&&b -> x = a||b;
ROR	Relational Operator Replacement	if(a >b) -> if(a < b)
UOI	Unary Operator Insertion	X = 2 * a; -> x = 2*-a;
UOD	Unary Operator Deletion	if(a<-b) -> if(a < b)
SOR	Shift Operator Replacement	x =a << b; -> x= a>>b;
LOR	Logical Operator Replacement	x =a&b; -> x = a|b;
ASR	Assignment Operator Replacement	x+= 2; -> x-=2;
COR	Conditional Operator Replacement	if(a&&b) -> ! if(a&b)
EOA	Reference Assignment and Content Assignment Replacement	list l1,l2; l1 = new List(); l1 = l2; -> l1 = l2.clone();
JTD	this Keyword Deletion	this.r = r; -> r = r;
EOC	Reference Comparison and Content Comparison Replacement	Integer a = new Integer(1); Integer b = new Integer(1); boolean x = (a == b); -> boolean x =(a.equals(b));
JTI	this Keyword Insertion	this.r = r; -> this.r = this.r;
EAM	Accessor Method Change	circle.getX(); -> circle.getY();
EMM	Modifier Method Change	circle.setX(1); ->circle.setY(1);

8.4 Mu JAVA

MuJava is a tool written for java programs. It uses two sets of mutation operators: method-level and class-level. MuJava creates mutants using the various method-level and class-level operators. It then runs test cases on them and evaluates the mutation coverage for them. Test cases are written as separate classes that call methods in the classes that need to be tested [8].

- MuJava is not an open-source tool but offers a large set of both traditional and OO mutation operators for the Java language.
- Program which is under the test mutation operators do syntactic change in the program. However, this syntactical mistake represents the mistakes which are done by the programmer while writing code.
- MuJava, to save the compilation time it follows a 'do faster' approach in mutation testing process. This approach has been adopted primarily for object-oriented programs.
- Traditional Operators: AOR(Arithmetic Operator Replacement), AOI(Arithmetic Operator Insertion), AOD(Arithmetic Operator Deletion), ROR(Relational Operator Replacement), COR(Conditional Operator Replacement), COI(Conditional Operator Insertion), COD(Conditional Operator Deletion), SOR(Shift Operator Replacement), LOR(Logical Operator Replacement), LOI(Logical Operator Insertion), LOD(Logical Operator Deletion), ASR(Assignment Operator Replacement)
- Class Mutation Operators: AMC(Access Modifier Change), IHD (Hiding Variable Deletion), IHI(Hiding Variable Insertion), IOD(Overriding Method Deletion), IOP(Overriding Method Calling Position Change), IOR(Overriding Method Rename), ISK(*super K*eyword Deletion), IPC(Explicit Call of a Parent's Constructor Deletion), PNC(*new* Method Call with Child Class type), PMD(Member Variable Declaration with Parent Class type), PPD(Parameter Variable Declaration with Child Class type), PRV(Reference Assignment with other Comparable type), OMR(Overloading Method Contents change), OMD(Overloading Method Deletion), OAO(Argument Order Change), OAN(Argument Number Change), JTD(*this* Keyword Deletion), JSC(*static* Modifier Change), JID(Member Variable Initialization Deletion), JDC(Java-supported Default Constructor Create), EOA(Reference Assignment and Content Assignment Replacement), EOC(Reference Comparison and Content Comparison Replacement), EAM(Accessor Method Change), EMM(Modifier Method Change)
- *Disadvantage*: MuJava displaying of mutants are not shown in a convenient way and the problem of showing mutants in the context of the complete class is harder.

9. ARCHITECTURE OF MUTATION TESTING PROCESS

The mutation testing process starts with the generation of mutants of a program to be tested. So, first it takes the source code and generates the mutants and then compiles each mutant and performs the test cases on it and result is then to be analysed against the test case result with mutant and source code. This process is repeats for every mutant.

- **Mutation Generation Process**

The Mutant Generation Process includes the generation of different-different mutants on the basis of selective mutation operators.

- **Mutation Compilation Process**

The Mutant Compilation Process provides the compilation solution on the different-different mutants which will be generated from the mutation generation process.

- **Mutation Testing Process**

The Mutation Testing Process is the phase during which all the created and compiled mutants are tested.

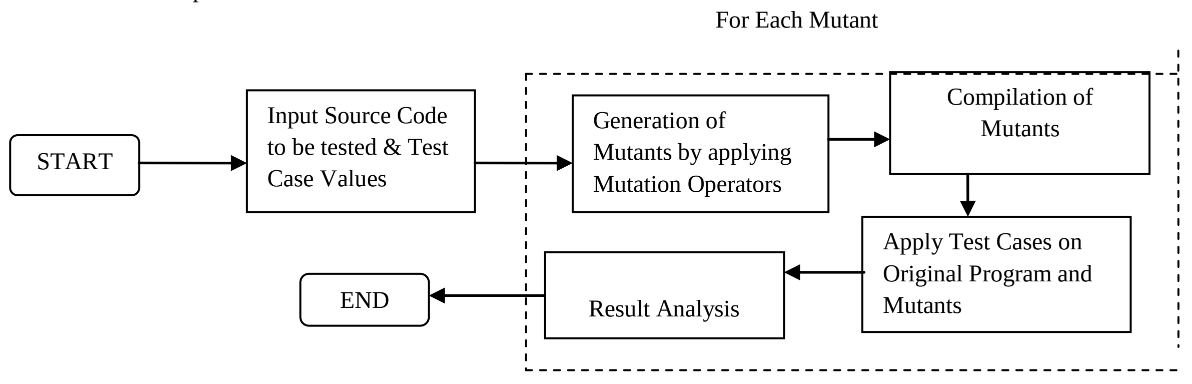
For Each Mutant
START
Input Source Code to be tested & Test Case Values
Generation of Mutants by applying Mutation Operators
Compilation of Mutants
Apply Test Cases on Original Program and Mutants
Result Analysis
END

Fig.1 Architecture of Mutation Testing Process

However, the mutation generation and testing activities mainly focuses on four kinds of activities:

9.1 Select Mutation Operators

This involves defining new mutation operators for different languages and types of testing.

9.2 Experiments in Mutation Generation Process

It deals with the experimentation in mutation generation process. Empirical studies have supported the effectiveness of mutation testing. Mutation testing has been found to be more effective in finding faults.

9.3 Designing Framework

This kind of activity in mutation testing research is designing mutation testing framework so that the testing process will be done automatically. Different types of mutation testing tools are developed: Jester, Jumble, MuJava, Judy. Main difference in between them is on the basis of mutation operators.

9.4 Reducing the cost of Mutation Testing

The major cost of mutation analysis depends on computational expense of generating and compiling large numbers of mutant programs. Mutation testing is an attractive but a time-consuming technique, which makes impractical to use without a reliable, fast and automated tool that generates mutants, runs the mutants against a suite of tests and reports the mutation score of the test suite. However the various approaches are defined to reduce this computational expense of generating and running large numbers of mutant programs.

10. COST REDUCTION TECHNIQUE

Mutation Testing Process is very computationally expensive testing technique, since it requires the execution of all the mutants which is very time- consuming process. However, there may be a requirement that all the mutants are necessary. So to give a practical approach to mutation testing process, many cost reduction techniques have been applied.

ı: Selection mutation Operator technique that takes a small subset of mutation operators which generate all the possible mutants, in order to achieve the test suite effectiveness. This idea was first suggested as ‘constrained mutation’ and then subsequently extended this idea calling it Selective Mutation. Mutation operators generate different numbers of mutants, and some mutation operators generate far more mutants than others, many of which may turn out to be redundant.

Higher Order Mutation: Higher order mutation is a new form of mutation testing. Main motivation is to find higher valuable mutants that denote the subtle faults.

Runtime Optimization Techniques: The interpreter based technique is one of the optimization techniques used in first generation of mutation testing tools. In this, the result of a mutant is interpreted from its source code directly.

Mutant Schema Generation: Mutants schema Generation approach designed to reduce the overall cost of traditional interpreter-based techniques. Instead of compiling the each mutant separately, the mutant schema technique generates a metaprogram. This metamutant can be used to represent all possible mutants. Therefore, to run each mutant against the test set, only this metaprogram need to be compiled. Thus the cost of this technique is composed of a one-time compilation cost and the overall run time cost. As this metaprogram is a compiled program its running speed is faster than the interpreter-based technique.

11. STEPS OF MUTATION TESTING

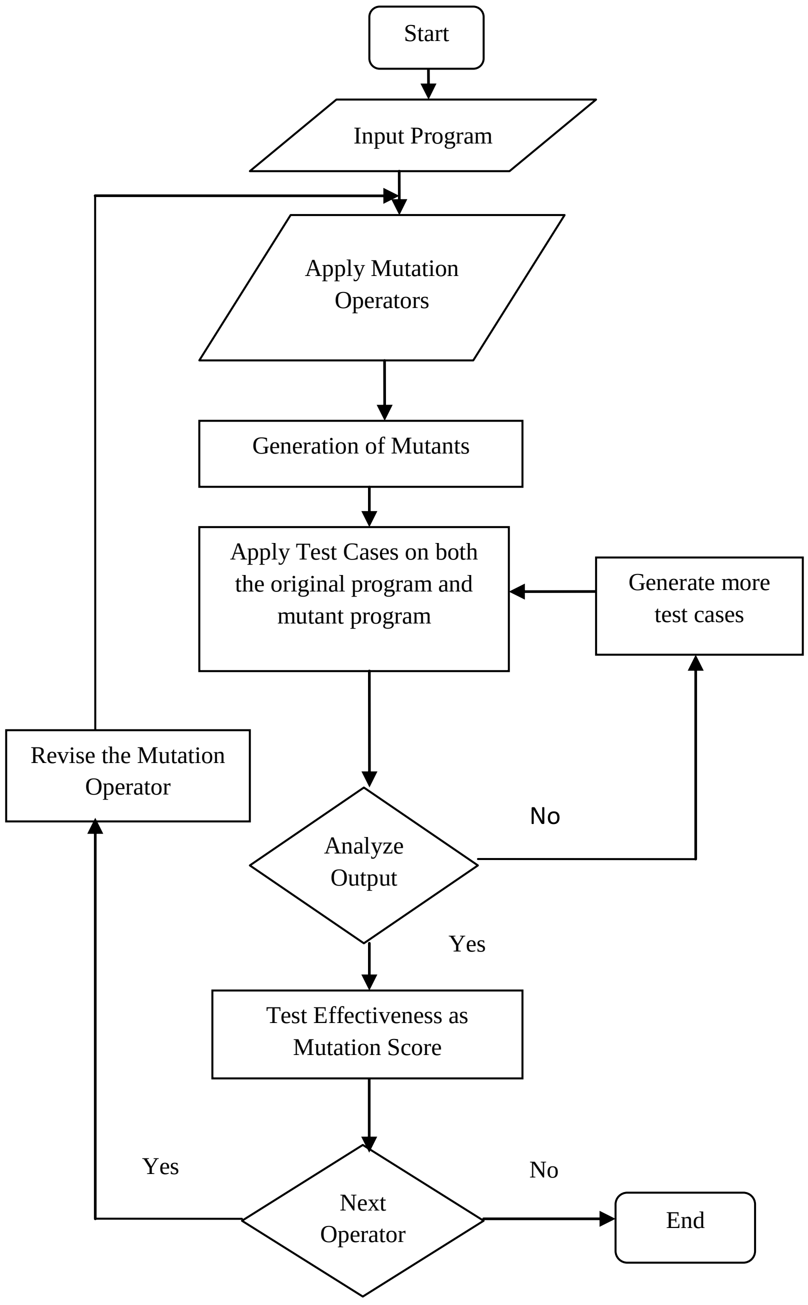
Start
Input Program
Apply Mutation Operators
Generation of Mutants
Apply Test Cases on both the original program and mutant program
Generate more test cases
Revise the Mutation Operator
Analyze Output
No
Yes
Test Effectiveness as Mutation Score
Yes
No
Next Operator
End

Fig. 2 Steps of Mutation Testing

12. CONCLUSION

Mutation is a powerful but complicated & computationally expensive testing method. Our aim to develop a framework which will automatically generate the mutants and then the test cases will be performed on the mutants. By modifying a program to contain simple errors and demanding that test data be discovered to distinguish the erroneous versions of the program from the original program, those simple errors can be guaranteed as absent from the original. Our target is to create a feasible mutation testing tool with minimal human involvement and significant performance improvement. Our complete system would provide almost complete automation to the tester.

Theoretical and experimental results have shown that mutation testing is an effective approach to measuring the adequacy of test cases. The cost of mutation testing has been a focus of concern. Developments in the mode of application of mutation testing can reduce the cost.

The OO mutants were derived from definitions of faults for subtype inheritance and polymorphism, so it is reasonable to expect tests from these mutants to find those kinds of faults. It is also possible that the mutant operators could be reduced by using a selective approach. However, more detailed investigation will be needed. In the future, we plan to extend the effectiveness study so that we can eventually determine a set of selective class mutation operators as was done with traditional mutation operators where prolific operators are eliminated. In future, this framework will work on a set of selective class mutation operators as was done with traditional mutation operators.

13. REFERENCES

[1] Mike Papadakis and Nicos Malevris “Automatic Test Case Generation Based on Mutation Testing” IEEE 21st International Symposium on Software Reliability Engineering , 2010 .

[2] M.RWoodward “MutationTesting – An EvolvingTechnique” , IEEE Conference.

[3] Aparajita Rao, Kavitha Elizabeth George, G.Logeshwari, S. Viveka Katherine, T.Mythili “A Model for the Development of a Mutation Testing Cum Test Case Generation Tool”, International Conference on Advances in Recent Technologies in Communication and Computing, 2009.

[4] Wyb.Wyspian´skiego 27, 50370 Wrocław, Poland “Judy - A mutation testing tool for Java”, Institute of Informatics, Wrocław University of Technology, 2008.

[5] Yu-Seung Ma, Korea “Description of Method-level Mutation Operators for Java”, Electronics and Telecommunications Research Institute, March 20, 2005.

[6] Yu-Seung Ma, Korea “Description of Class Mutation Mutation Operators for Java”, Electronics and Telecommunications Research Institute, November 7, 2005.

[7] Ivan Moore “Jester - a JUnit test tester”, Proc. Second Int. Conf. Extreme Programming and Flexible Processes in Software Engineering, May 2001, pp. 84-87.

[8] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon “MuJava : An Automated Class Mutation System”, Journal of Software Testing, Verification and Reliability, 15(2):97-133, June 2005.

[9] P. G. Frankl, S. N. Weiss, and C. Hu. “All-uses versus mutation testing: An experimental comparison of effectiveness”, Journal of Systems and Software, 38:235–253, 1997.

[10] Yue Jia and Mark “An Analysis and Survey of the Development of Mutation Testing” IEEE Transactions of Software Engineering, vol. To appear, 2010.

[11] A. J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf. "An Experimentational Determination of Sufficient Mutation Operators" ACM Transactions on Software Methodology, 5(2): 99-118, April 1996.

[12] R. A. DeMillo and A. J. Offutt. "Experimental Results from an Automatic Test Case Generator" ACM Transactions on Software Engineering Methodology, 2(2): 109-127, April 1993.

[13] W. E. Wong "On Mutation and Data Flow" PhD thesis, Purdue University, West Lafayette, December 1993.

[14] W. E. Wong and A. P. Mathur "Reducing the Cost of Mutation Testing: An Empirical Study" The Journal of Systems and Software, 31(3): 185-196, Dec. 1995.

[15] A. P. Mathur and W. E. Wong, "An Empirical Comparison of Mutation and Data Flow based Test Adequacy Criteria", Technical Report SERC-TR-135-P, Software Engineering Research Center, Purdue University, Feb. 1993.

[16] A. Jefferson Offutt and J. Huffman Hayes "A Semantic Model of Program Faults" ISSTA' 96, San Diego CA, USA.

[17] Vadim Okun "Specification Mutation for Test Generation and Analysis" 2004.

[18] Salas and Aichernig "An automatic Test Case Generation for OCL: a Mutation Approach" UNU IIST Report No. 321, May 2005.

[19] De Millo and Offutt "Experimental Results from an Automatic Test Case Generator" ACM Transaction on Software Engineering and Methodology, Vol 2, April 1993, Pages 109-127.

[20] A.J.Offutt and S. D.Lee "An Empirical Evaluation of Weak Mutation" IEEE Transactions on Software Engineering, 20(5): 337-344, May 1994.

[21] King K.N., Offutt A.J. "A Fortran Language System for Mutation-Based Software Testing" Software Prac. Exper., 1991, 21, (7), pp. 685-718.

[22] Offutt J., MA Y.S., Kwon, Y.R. "An Experimental Mutation System for Java" SIGSOFT Software Engineering Notes, 2004, 29, (5), pp. 1-4.

[23] Irvine S.A, TIN P., TRIGG L, Clearly J.G., Ingus S., Utting M.: "Jumble Java Byte Code to Measure the Effectiveness of Unit Tests" Proc. Testing: Academic and Industrial Conf. Practice and Research Techniques, September 2007, pp. 169-175.

[24] Andrews J.H., Briand L.C., Labiche Y.: "Is Mutation an Appropriate tool for Testing Experiments?" Proc. 27th Int. Conf. Software Engineering, May 2005, pp. 402-411.

[25] V. N. Fleyshgakker and S.N. Weiss, "Efficient Mutation Analysis: A New Approach" in Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 94), (Seattle, WA), pp. 185-195, ACM SIGSOFT, ACM Press, Aug. 17-19 1994.

[26] A. J. Offutt and S. D. Lee "How Strong is Weak Mutation?" in Proceedings of the Fourth Symposium on Software Testing, Analysis and Verification, (Victoria, British Columbia, Canada), pp. 200-213, IEEE Computer Society Press, October 1991.

[27] R. A. DeMillo, "Test Adequacy and Program Mutation" Proceedings of the 11th International Conference on Software engineering, p.355-356, May 1989.

[28] Hyunsook Do, Gregg Rothermal, "On the Use of Mutation Faults in Empirical Assessments of Test Case Priortization Techniques" IEEE Transactions on Software Engineering, v.32 n.9, p.733-752 September 2006.

[29] W. Eric Wong , Joseph R. Horgan , Saul London , Aditya P. Mathur "Effect of test set minimization on fault detection effectiveness" Proceedings of the 17th international conference on Software engineering, p.41-50, April 24-28, 1995.

[30] A. Jefferson Offutt, "A Practical System for Mutation Testing: Help for the Common Programmer" Proceedings of the 1994 international conference on Test, October 02-06, 1994, Washington, D.C.