MuJava: a mutation system for java.

Conference Paper · January 2006

DOI: 10.1145/1134425 · Source: DBLP

Jeff Offutt
George Mason University

MuJava: A Mutation System for Java

Yu-Seung Ma
Electronics and Telecommunication Research Institute, Korea
ysma@etri.re.kr

Jeff Offutt
George Mason University
U.S.A.
offutt@ise.gmu.edu

Yong-Rae Kwon
Korea Advanced Institute of Science and Technology
Korea
kwon@cs.kaist.ac.kr

ABSTRACT

Mutation testing is a valuable experimental research technique that has been used in many studies. It has been experimentally compared with other test criteria, and also used to support experimental comparisons of other test criteria, by using mutants as a method to create faults. In effect, mutation is often used as a "gold standard" for experimental evaluations of test methods. Although mutation testing is powerful, it is a complicated and computationally expensive testing method. Therefore, automated tool support is indispensable for conducting mutation testing. This demo presents a publicly available mutation system for Java that supports both method-level mutants and class-level mutants. MuJava can be freely downloaded and installed with relative ease under both Unix and Windows. MuJava is offered as a free service to the community and we hope that it will promote the use of mutation analysis for experimental research in software testing.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability

Keywords

Mutation system, Mutation testing, Java

1. INTRODUCTION

Mutation testing [9] is a fault-based technique that measures the effectiveness of test suites. Faults are introduced into the program by creating a set of faulty versions, called *mutants*. These mutants are created from the original program by applying *mutation operators*, which describe syntactic changes to the programming language. Tests are used to execute these mutants with the goal of causing each mutant to produce incorrect output.

ICSE'06, May 20–28, 2006, Shanghai, China.
ACM 1-59593-085-X/06/0005.

Empirical studies have supported the effectiveness of mutation testing. Andrews et al. [2] found that mutants, when using carefully selected mutation operators and after removing equivalent mutants, can provide a good indication of the fault detection ability of a test suite. Walsh [17] empirically found that mutation testing is more powerful than statement and branch coverage. Frankl et al. [7] and Offutt et al. [14] found that mutation testing was more effective at finding faults than data-flow.

Although mutation testing is powerful, it is complicated, time-consuming, and impractical without automated tools. Mutation tools have been developed and distributed for procedural programs. Mothra [5] and Proteum [4] are well-known and widely used mutation tools. Because mutation operators depend on the syntax of programming languages, they are usually developed for a specific programming language. For example, Mothra tests Fortran programs and Proteum tests C programs. However, there are few publicly available mutation tools for OO languages.

This demo presents a publicly available Java mutation system, MuJava. MuJava supports both generation and execution of mutants, is developed in Java to test Java programs, implements both method-level and class-level mutation operators, and includes a graphical user interface to help testers and researchers carry out mutation testing.

The contents of this short paper are as follows. Section 2 briefly describes the history of mutation systems. Section 3 summarizes the mutation operators for Java programming language. Our mutation system, MuJava, is described in Section 4, and final discussion is in Section 5.

2. PREVIOUS MUTATION SYSTEMS

The theory of mutation analysis began in 1971, when Richard Lipton proposed the initial concepts of mutation in a class term paper titled "Fault Diagnosis of Computer Programs." The first refereed publications appeared in the late 1970's; the DeMillo, Lipton, and Sayward paper [9] is generally cited as the seminal reference.

PIMS [1] was one of the first mutation testing tools. It pioneered the general process typically used in mutation testing of creating mutants (in its case, Fortran IV programs), accepting test cases from the users, and then executing the test cases on the mutants to decide how many mutants were killed. The most widely used tool among researchers was the 1987 Mothra mutation toolset [5], which provided an integrated set of tools, each of which performed an individual, separate task to support mutation analysis and testing. Because each Mothra tool was a separate program, it was easy to incorporate, and thus experiment with, additional types of processing.

Several variants of Mothra were created in the early 1990s, including one that implemented weak mutation [13], and several distributed versions. A compiler-integrated mutation tool for C was also developed [6], and a tool that was based on program schemata [16]. However, these tools were primarily used by the researchers who developed them, and the only widely used system besides the original version of Mothra has been the Proteum mutation system for C [4].

3. MUTATION OPERATORS FOR JAVA

MuJava uses two types of mutation operators, *method-level* mutation operators and *class-level* mutation operators.

Method-level mutation operators handle primitive features of programming languages. They modify expressions by replacing, deleting, and inserting primitive operators. Table 3 lists method-level mutation operators for Java.

Operator	Description
AOR	Arithmetic Operator Replacement
AOD	Arithmetic operator Deletion
AOI	Arithmetic operator Insertion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
COD	Conditional operator Deletion
COI	Conditional operator Insertion
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOD	Logical operator Deletion
LOI	Logical operator Insertion
ASR	Assignment Operator Replacement

Table 1: Method-level Mutation Operators for Java

Class-level mutation operators handle object-oriented specific features such as inheritance, polymorphism and dynamic binding. Table 3 lists method-level mutation operators for Java.

Detailed description of these operators can be found on the MuJava website [10].

4. MUJAVA

MuJava, (**Mu**tation System for **Java**), supports the entire mutation process for Java programs. It automatically generates mutants, runs the mutants against a suite of tests, and reports the mutation score of the test suite. This section briefly describes the tool and reflection, a key technique used for its implementation.

4.1 Description

Figure 1 describes the overall structure of the tool. MuJava consists of three main components: the mutant generator, the mutant viewer and the mutant executor. All three have graphical user interfaces.

The `mutant generator` generates both traditional mutants and class mutants. Figure 2 shows the interface for the `mutant generator`. Testers can select the files for which they want to create mutants and choose which mutation operators to apply. Pressing the "Generate" button prompts the tool to generate mutants. Mutants are generated in the form of source code, then they are compiled into byte code. After generation, the information for each mutant is shown by the `mutants viewer`.

Operator	Description
IHD	Hiding variable deletion
IHI	Hiding variable insertion
IOD	Overriding method deletion
IOP	Overridden method calling position change
IOR	Overridden method rename
ISI	*super* keyword insertion
ISD	*super* keyword deletion
IPC	Explicit call of a parent's constructor deletion
PNC	*new* method call with child class type
PMD	Instance variable declaration with parent class type
PPD	Parameter variable declaration with child class type
PCI	Type cast operator insertion
PCC	Cast type change
PCD	Type cast operator insertion
PRV	Reference assignment with other compatible type
OMR	Overloading method contents change
OMD	Overloading method deletion
OAC	Argument order change
JTI	*this* keyword insertion
JTD	*this* keyword deletion
JSI	*static* modifier insertion
JSD	*static* modifier deletion
JID	Member variable initialization deletion
JDC	Java-supported default constructor create
EOA	Reference and content assignment replacement
EOC	Reference and content assignment replacement
EAM	Accessor method change
EMM	Modifier method change

Table 2: Class-level Mutation Operators for Java

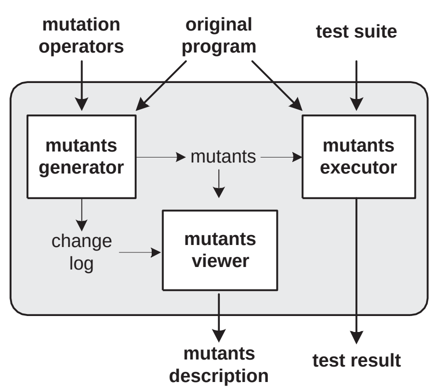

Figure 1: Structural architecture of MuJava

The `mutant viewer` shows how many and what types of mutants are generated. It also shows which part of the original source code was changed by each mutant. It helps testers perform two tasks: designing tests for mutants that are difficult to kill and identifying equivalent mutants. Because MuJava supports two types of mutation operators, it provides a separate mutant viewer for each of them. Figure 3 shows the interface for the `mutant viewer`, which lists generated mutants and displays the portions of the original source code that are changed by a mutant.

The `mutant executor` executes mutants against the test suite and shows the test result in the form of a mutation score of the test suite. Figure 3 shows the interface for the `mutant viewer`. Each mutant is executed by the appropriate class loaders. Test cases are supplied by the tester in a specific format, specifically, a Java class that contains one

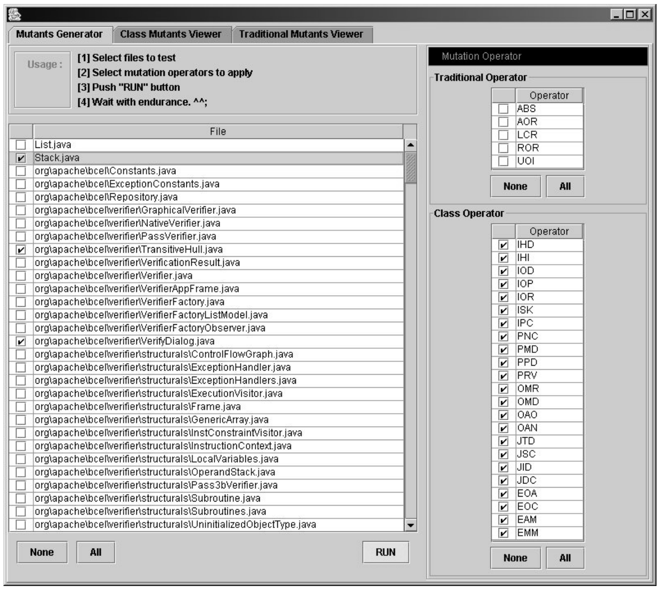

Figure 2: GUI for Generating Mutants

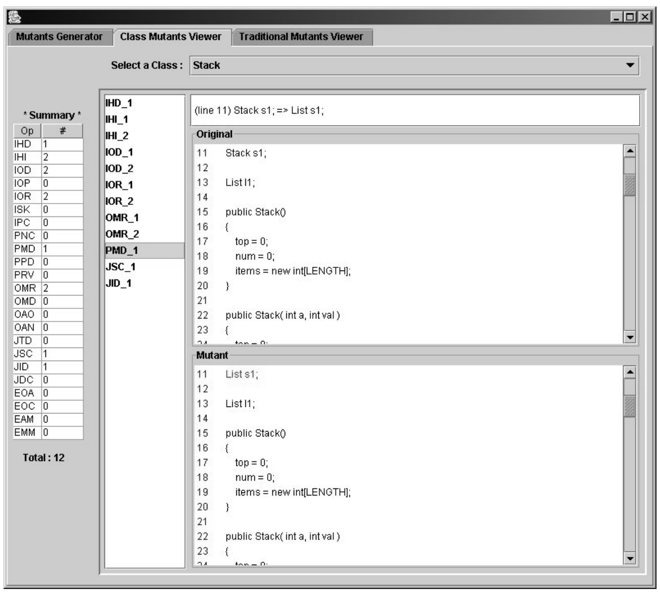

Figure 3: GUI for Analyzing Mutants

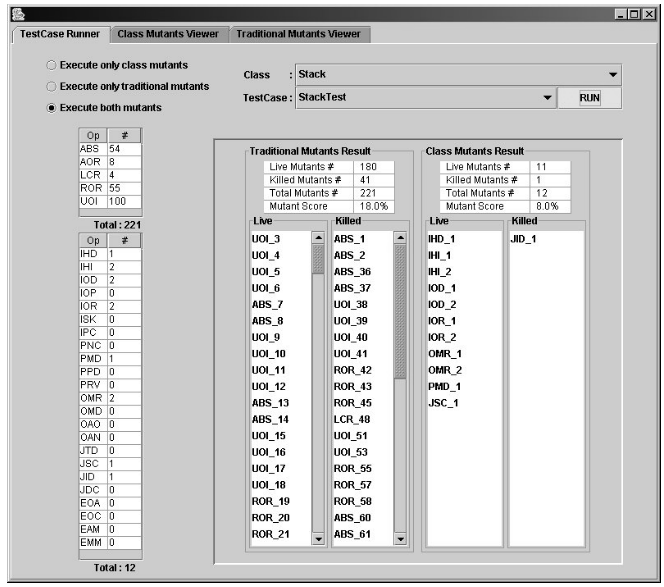

Figure 4: GUI for Running Mutants

```
public class StackTest {
    public String test1 () {
        String result;
        Stack obj = new Stack();
        obj.push (2);
        obj.push (4);
        result = obj.isFull () + obj.pop ();
        return result;
    }
    public String test2 () {
        String result;
        Stack obj = new Stack ();
        obj.push (5);
        obj.push (3);
        result = obj.pop () + obj.pop ();
        return result;
    }
}
```

Figure 5: An example test suite for class Stack

method per test. Each test method should have no parameters and return a string result that is used to compare outputs of mutants with outputs of the original class. Also, each test method should start with the string "test" and have public access. A small example test set for a `Stack` class is shown in Figure 5.

4.2 Implementation: a Reflection Approach

Mutation tools for OO languages must be designed and implemented differently from procedural languages. Mutation operators for conventional (non-OO) languages do not change type or data structure declarations. However, the mutation operators that have been designed for Java do, so an OO mutation system needs to be able to make those changes. It must also access information in a program from an OO standpoint. For instance, the IOD operator, which deletes an overriding method, needs to use inheritance relationships.

MuJava uses a *reflection* technique to satisfy those requirements, specifically, to generate and run mutants. *Reflection* [8, 11] is the ability of a program to observe and possibly modify its high level structure. *Reflection* is a natural way to implement mutation analysis for several reasons. First, it lets programmers extract OO-related information about a class by providing an object that represents a logical structure of the class definition. This helps solve the first problem in implementing a mutation analysis system, parsing the program. Second, it provides an API to easily change the behavior of a program during execution. This can be used to create mutated versions of the program. Third, it allows objects to be instantiated and methods to be invoked dynamically. Java provides a built-in reflection capability with a dedicated API [12]. This allows Java programs to perform functions such as asking for the class of a given object, finding the methods in that class, and invoking those methods. However, the Java language as defined does not provide full reflective capabilities. Specifically, Java only supports introspection, which is the ability to introspect data structures, but does not support alteration of the program behavior. Several reflection systems [3, 15] have been proposed to complement the Java reflection API [12]. Because of differences in these systems, which reflection system is selected can significantly affect the efficiency of mutation analysis and testing.

4.3 Availability

MuJava is the result of a collaboration between two universities, Korea Advanced Institute of Science and Technology (KAIST) in South Korea and George Mason University in the USA. The MuJava Web site is mirrored at both universities; `http://salmosa.kaist.ac.kr/LAB/mujava/` at KAIST and `http://www.ise.gmu.edu/~offutt/mujava/` at GMU. The Web sites have links to download the MuJava jar files, a description of the tool, and detailed instructions for how to install and use MuJava.

5. DISCUSSION

This paper has introduced a mutation tool that is available for software testing researchers and educators. It supports the entire mutation process and employs graphical user interfaces for each major step. We recently added an engine to detect equivalent mutants. This engine is embedded in the mutant generator and avoids generating mutants if the engine can determine that the mutant would be equivalent. Although the engine does not remove all mutants (a theoretically undecidable problem), our experience shows that it is definitely helpful.

Comparing with previous mutation systems for procedural programs, MuJava is very fast. However, it is relatively slow when it generates and runs lots of mutants.

This is still an ongoing project, and we expect to improve the tool and make new versions available. Nevertheless, the current version of MuJava is currently being used by researchers to apply mutation testing to Java classes, supporting a variety of software testing experiments. MuJava is also useful as teaching tool to teach students about mutation and about how to design effective tests.

6. REFERENCES

[1] D. M. S. Andre. Pilot mutation system (pims) user's manual. Technical report GIT-ICS-79/04, Georgia Institute of Technology, April 1979.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *27th International Conference on Software Engineering*, pages 402–411, May 2005.

[3] S. Chiba. Load-time structural reflection in Java. *LNCS*, 1850:313–336, 2000.

[4] M. E. Delamaro and J. C. Maldonado. Proteum – A tool for the assessment of test adequacy for C programs. *Proceedings of the Conference on Performability in Computing Systems*, pages 75–95, July 1996.

[5] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, July 1988.

[6] R. A. DeMillo, E. W. Krauser, and A. P. Mathur. Compiler-integrated program mutation. In *Proceedings of the Fifteenth Annual Computer Software and Applications Conference (COMPSAC' 92)*, Tokyo, Japan, September 1991. Kogakuin University, IEEE Computer Society Press.

[7] P. G. Frankl, S. N. Weiss, and C. Hu. All-use vs mutation testing: An experimental comparison of effectivenss. *Journal of Systems Software*, 28(3):235–253, September 1997.

[8] G. Kiczales, J. de Rivieres, and D. G. Bobrow. *The Art of the Metaobject protocol*. MIT Press, 1991.

[9] R. J. Lipton, R. A. DeMillo, and F. G. Sayward. Hints on test data selection: help for the practicing programmer. *IEEE Computer*, 11(4):34–41, November April.

[10] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. mujava home page. online, 2005. http://salmosa.kaist.ac.kr/LAB/MuJava/, http://ise.gmu.edu/ offutt/mujava/.

[11] P. Maes. Concepts and experiments in computational reflection. In *Preceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, volume 22 (12), pages 147–155, Orlando, FA, October 1987.

[12] S. Microsystems. Java reflection WWW page. Part of the Sun J2SE project, 2002. http://java.sun.com/j2se/1.4/docs/guide/reflection/ (accessed May 2004).

[13] A. J. Offutt and S. D. Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, May 1994.

[14] J. Offutt, J. Pan, K. Tewary, and T. Zhang. An experimental evaluation of data flow and mutation testing. *Software Practice and Experience*, 26(2):165–176, Feb 1996.

[15] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A class-based macro system for Java. *Reflection and Software Engineering*, LNCS 1826:117–133, June 2000. Heidelberg, Germany.

[16] R. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using program schemata. In *Proceedings of the 1993 International Symposium on Software Testing, and Analysis*, pages 139–148, Cambridge MA, June 1993.

[17] P. J. Walsh. *A measure of test case completeness*. PhD thesis, Univ. New York, 1985.