An Experimental Mutation System for Java

Jeff Offutt
Dept. of Information and Software Engineering
George Mason University
Fairfax VA, USA
ofut@ise.gmu.edu

Yu-Seung Ma and Yong-Rae Kwon
CS Div. EECS Dept.
Korea Advanced Institute of Science and Technology
Daejeon, South Korea
{ysma,kwon}@salmosa.kaist.ac.kr

ABSTRACT

Mutation is a powerful but complicated and computationally expensive testing method. Mutation is also a valuable experimental research technique that has been used in many studies. Mutation 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. This paper presents a publicly available mutation system for Java that supports both traditional statement-level mutants and newer inter-class 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.

1. INTRODUCTION

Mutation testing [8] 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.

Empirical studies have supported the effectiveness of mutation testing. Walsh [21] found empirically that mutation testing is more powerful than statement and branch coverage. Frankl et al. [10] and Offutt et al. [17] found that mutation testing was more effective at finding faults than data-flow.

Although mutation testing is powerful, it is complicated and time-consuming, and impractical to use without automated tools. Mutation tools have been developed and distributed for procedural programs. Mothra [6, 9] and Proteum [5] are well-known mutation tools that have been widely used. However, Mothra tests Fortran programs and Proteum tests C programs. As yet, there are no publicly available mutation tools for object-oriented languages. We know of one other mutation tool for Java, by Chevalley and Thévenod-Fosse [3]. However, it is not freely available for download, and perhaps more importantly for experimental researchers, it generates but does not support the execution of mutants.

This workshop paper introduces 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 traditional (statement-level) and object-oriented mutant operators, and includes a graphical user interface to help testers and researchers carry out mutation testing.

The contents of the paper are as follows. Section 2 briefly describes the history of mutation systems. Section 3 summarizes the object-oriented mutation operators. 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 [2, 11, 8]; the DeMillo, Lipton, and Sayward paper [8] is generally cited as the seminal reference.

PIMS [2, 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 [6, 9], 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 is 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 [16], and several distributed versions, including versions that ran on a vector processor [14], SIMD machines [12], hypercube (MIMD) machines [18, 4], and across a network (MIMD) of computers [23].

A compiler-integrated mutation tool for C was also developed [7], and a tool that was based on program schemata

[20]. However, these implementations have primarily been 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 [5].

3. MUTATION OPERATORS

MuJava uses two types of mutation operators. First are "traditional" statement-level mutation operators developed for procedural languages. Second are inter-class operators specifically developed to handle object-oriented specific features such as inheritance, polymorphism and dynamic binding [13]. The remainder of this paper refers to these as *traditional mutation operators* and *class mutation operators*.

MuJava uses the "selective mutation" operator set. Selective mutation was initiated by Wong and Mathur [22] and extended and experimentally validated by Offutt et al. [15]. The latter paper, based on Mothra, found that five of twenty-two mutation operators used by Mothra were able to provide almost the same effectiveness as non-selective mutation, with cost reduction of at least four times with small programs. The five operators from the Mothra selective set are shown in Table 3; MuJava applies the same set of operators to Java.

Operator	Description
ABS	Absolute value insertion
AOR	Arithmetic operator replacement
LCR	Logical connector replacement
ROR	Relational operator replacement
UOI	Unary operator insertion

Table 1: Five selective traditional mutation operators

The 24 class mutation operators were designed for Java classes by Ma, Kwon and Offutt [13] specifically to test object-oriented and integration issues. They were in turn designed from a categorization of object-oriented faults by Offutt, Alexander et al. [19]. No research has yet been carried out for selective mutation in the case of class mutation operators. The class mutation operators used by MuJava are listed in Table 3. Detailed description of the operators are in the previous paper [13].

4. MUJAVA

MuJava (**Mu**tation System for **Java**), is a mutation system that 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. We designed and built MuJava for our research goals, initially as a demonstration vehicle for inter-class mutation testing, and have recently updated it to be useful to other experimental researchers in software testing as well as educators. This section describes the tool and discusses its limitations.

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 provide graphical user interfaces.

The `mutant generator` generates both traditional mutants and class mutants. It takes two inputs, a collection

Operator	Description
IHD	Hiding variable deletion
IHI	Hiding variable insertion
IOD	Overriding method deletion
IOP	Overridden method calling position change
IOR	Overridden method rename
ISK	*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
PRV	Reference assignment with other compatible 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 and content assignment replacement
EOC	Reference and content comparison replacement
EAM	Accessor method change
EMM	Modifier method change

Table 2: Java inter-class mutation operators

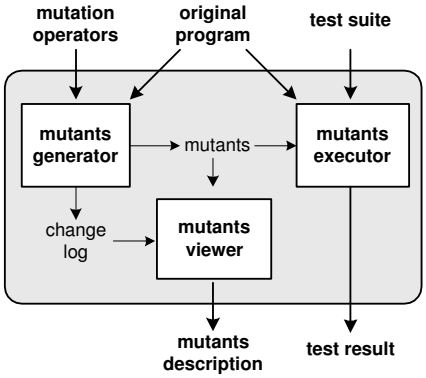

Figure 1: Structural architecture of MuJava

of Java files and the mutation operators, then produces mutants in the form of modified byte code. The graphical user interface for the mutant generator helps testers select the files to test and choose which mutation operators to apply.

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.

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. Each mutant is executed by the appropriate class loaders. MuJava tests Java classes, so a test case is in the form of a sequence of calls to methods in the class under test. The test suite should be created by the tester in a specific format, specifically, a Java class that contains one method per test. The test methods should have no parameters and return a string that represents the result of the test. This result is used to compare mutant output with original program output. A small example test suite

```
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 2: An example test suite for class Stack

for a `Stack` class is shown in Figure 2.

4.2 Limitations

It should be emphasized that MuJava is an experimental proof-of-concept tool that has some limitations, and is also still undergoing development. One current issue is with *equivalent mutants*. MuJava does not detect them, nor does it support manual elimination of equivalent mutants through the GUI.

MuJava also is relatively slow. We have implemented three versions of MuJava using different implementation techniques. The first relies on source code modification, by modifying the Java class file and recompiling each mutant. The second uses mutant schema [20] and the third generates mutants by modifying the byte code (bytecode translation). The schema and bytecode translation versions are both much faster than the source code modification version, but they also currently have some problems Thus the source code modification version is available now, and a future version will probably combine elements of mutant schema and bytecode translation.

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/~ofut/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 workshop 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. The MuJava Website includes links to the tools (Java jar files), a short description, and details for how to download, install, and run MuJava.

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 certainly useful enough to allow researchers to apply mutation testing to Java classes, supporting a variety of software testing experimental activities. MuJava is also useful as teaching tool to teach students about mutation and about how to design effective tests.

6. REFERENCES

[1] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta GA, September 1979.

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

[3] P. Chevalley and P. Thévenod-Fosse. A mutation analysis tool for Java programs. *Journal on Software Tools for Technology Transfer (STTT)*, pages 1–14, December 2002.

[4] B. Choi and A. P. Mathur. High-performance mutation testing. *The Journal of Systems and Software*, 20(2):135–152, February 1993.

[5] 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.

[6] 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.

[7] 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.

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[9] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[10] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. *The Journal of Systems and Software*, 38(3):235–253, 1997.

[11] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

[12] E. W. Krauser, A. P. Mathur, and V. Rego. High performance testing on SIMD machines. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 171–177, Banff, Alberta, July 1988. IEEE Computer Society Press.

[13] Y.-S. Ma, Y.-R. Kwon, and J. Offutt. Inter-class mutation operators for Java. In *Proceedings of the 13th International Symposium on Software Reliability*

Engineering, pages 352–363, Annapolis MD, November 2002. IEEE Computer Society Press.

[14] A. P. Mathur and E. W. Krauser. Mutant unification for improved vectorization. Technical report SERC-TR-14-P, Software Engineering Research Center, Purdue University, West Lafayette IN, April 1988.

[15] A. J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, April 1996.

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

[17] A. 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, February 1996.

[18] A. J. Offutt, R. Pargas, S. V. Fichter, and P. Khambekar. Mutation testing of software using a MIMD computer. In *1992 International Conference on Parallel Processing*, pages II–257–266, Chicago, Illinois, August 1992.

[19] J. Offutt, R. Alexander, Y. Wu, Q. Xiao, and C. Hutchinson. A fault model for subtype inheritance and polymorphism. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, pages 84–93, Hong Kong China, November 2001. IEEE Computer Society Press.

[20] 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.

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

[22] 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, December 1995.

[23] C. N. Zapf. Medusamothra – A distributed interpreter for the Mothra mutation testing system. M.S. thesis, Clemson University, Clemson, SC, August 1993.