Javalanche: Efficient Mutation Testing for Java

David Schuler · Andreas Zeller
Saarland University, Saarbrücken, Germany
{ds, zeller}@cs.uni-saarland.de

ABSTRACT

To assess the quality of a test suite, one can use *mutation testing*—seeding artificial defects (mutations) into the program and checking whether the test suite finds them. JAVALANCHE is an open source framework for mutation testing Java programs with a special focus on automation, efficiency, and effectiveness. In particular, JAVALANCHE assesses the *impact* of individual mutations to effectively weed out equivalent mutants; it has been demonstrated to work on programs with up to 100,000 lines of code.

Categories and Subject Descriptors

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

General Terms

Experimentation

Keywords

Mutation Testing

1. INTRODUCTION

Mutation testing assesses the quality of a test suite by applying small changes (mutations) to a program, and subsequently checking if the test suite detects them. Since mutation testing was first proposed by Richard Lipton in 1971 [4], several mutation testing frameworks have been developed, such as MOTHRA [1] for Fortran programs, Proteum for C, and Jester, μJava, and Jumble for Java. These frameworks differ in terms of mutation operators, efficiency and automation. In this paper, we present JAVALANCHE— an open source framework for mutation testing, which blends the best techniques from previous frameworks together with novel optimizations to allow efficient and fully automated mutation testing. A unique feature of JAVALANCHE is that it ranks mutations by their *impact* on the behavior of program functions. The greater the impact of an undetected mutation, the lower the likelihood of the mutation being equivalent (i.e., a false positive)—and the higher the likelihood of undetected serious defects.

ESEC-FSE'09, August 23–28, 2009, Amsterdam, The Netherlands.
ACM 978-1-60558-001-2/09/08.

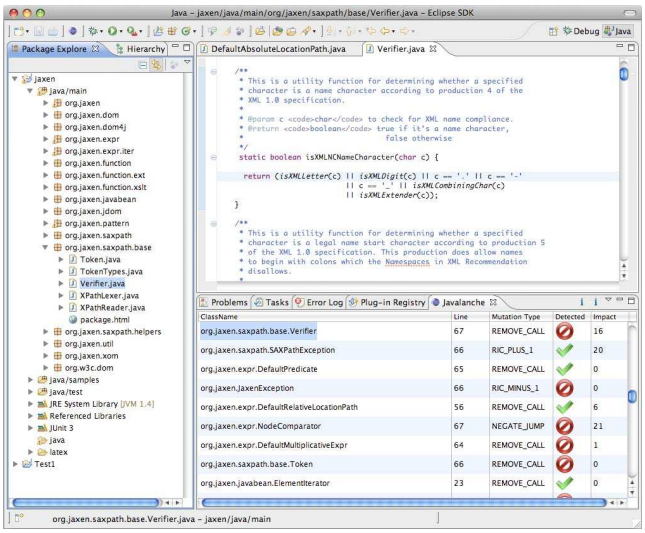

Figure 1: The JAVALANCHE Eclipse plug-in. JAVALANCHE lists all undetected mutants, ranked by their impact on the program behavior.

2. AUTOMATION AND EFFICIENCY

In order to asses the quality of test suites for realistically sized software projects, we developed JAVALANCHE with the focus on automation and efficiency. We thus implemented a large set of optimizations:

Selective mutation A small set of mutation operators may yield a sufficiently accurate approximation of the results obtained by using all possible operators [3]. JAVALANCHE uses the same small set: replace numerical constant, negate jump condition, replace arithmetic operator, and omit method calls.

Mutant schemata To reduce the number of generated versions, we use *mutant schemata* [6]: The program holds multiple mutations, each guarded by a runtime flag.

Coverage data Not all tests in the test suite execute every mutant. In order to avoid executing those tests, we collect *coverage information* for each test—and then only execute those tests that cover the mutated statement.

Manipulate bytecode We manipulate Java bytecode directly to avoid costly recompilation.

Parallel execution JAVALANCHE can execute several mutations in parallel, thus taking advantage of parallel and distributed computing.

Table 1: Description of subject programs.

Project Name	Program size (LOC)	Test code size (LOC)	Number of tests	Statement coverage (%)	Number of mutations	Mutation runtime (s)	Mutation score
ASPECTJ	94,902	14,736	339	33,73	14,357	347m 38s	29.75(70.18)
BARBECUE	4,837	3,160	137	50,45	18,358	9m 16s	4.59 (73.05)
COMMONS-LANG	18,817	32,756	1,662	84,09	19,566	79m 19s	22.45 (65.06)
JAXEN	12,449	8,371	680	66,77	9,972	117m 15s	47.15 (70.88)
JODA-TIME	25,879	48,130	3,496	87,72	23,833	272m 25s	63.29 (86.72)
JTOPAS	2,031	3,185	128	80.68	1,921	45m 27s	57.42 (72.95)
XSTREAM	14,388	15,618	1,005	77,23	10,607	115m 33s	60.41 (82.02)

Lines of Code (LOC) are non-comment, non-blank lines as reported by sloccount. For ASPECTJ, we only considered the core package tests.

Automation JAVALANCHE is fully automated, requiring only the name of a test suite, the base package name of the project, and the set of classes needed to run the test suite.

Some of these optimizations are also implemented in other mutation testing tools. For example, Jumble and μJava also manipulate bytecode directly, and μJava also uses mutant schemata. However, none of the other tools combines all these optimizations, uses coverage data, or allows parallel execution of mutations.

3. PERFORMANCE MEASUREMENTS

Table 1 shows the results for applying JAVALANCHE on seven programs, ranging from 2,031 lines of code (column 2) for JTOPAS to 94,902 for ASPECTJ. The test suites consist of 128 (column 4) test cases with 3,185 (column 3) lines of code for JTOPAS up to 3,496 with 48,130 lines for JODA-TIME. The number of mutations ranges from 1,921 (column 6) for JTOPAS to 23,833 for JODA-TIME. To our knowledge, this is the first time mutation testing has been applied to programs of such size.

Running JAVALANCHE (column 7) takes between 9 minutes for BARBECUE and 272 for JODA-TIME. [1] The low number for BARBECUE is due to the fact that JAVALANCHE only checks mutations that are covered by tests, and BARBECUE has a huge number of mutations in parts that are only exercised during class loading, which are not covered by tests explicitly. By analyzing the mutation testing results, we can calculate the mutation score for a project (column 8), which is the number of detected mutations divided by the total number of mutations. This score varies between 4.6 % for BARBECUE and 63.3 % for COMMONS-LANG. The relative low numbers are due to the fact that there are many mutations that are not covered, and thus cannot be detected. Therefore, we also give the mutation score for mutations that were covered, which ranges form 70.2 for ASPECTJ to 86.7 for JODA-TIME.

4. RANKING MUTATIONS BY IMPACT

Besides the time needed for mutation testing, another significant cost stems from *equivalent mutants*. Equivalent mutants are changes to the syntax of the program that do not change its semantics. Thus, it is impossible to write a test case that distinguishes the original program from the mutated one. When using mutants to improve a test suite (by writing additional tests for undetected mutants), the manual assessment of mutants place an additional burden on a developer. Therefore, we developed techniques that assess the *impact of mutations*.

An impact of a mutation is a measure how much the run of a mutant differs from a run of the normal program. By classifying the mutations according to their impact on *dynamic invariants* [5] and *code coverage* [2], we were able to significantly reduce the number of equivalent mutants. JAVALANCHE is designed in such a way that it supports these and further impact measurement techniques. Our studies [2, 5] have shown that focusing on mutations with impact keeps down the percentage of equivalent mutants. Intuitively, we also reason that if a mutation has lots of impact throughout the program's execution, and it is not detected by the test suite, it is also likely to create more severe errors than a mutation which has little impact.

[1]The runtime is measured in CPU time, e.g. 10 minutes on 5 cores would result in a runtime of 50 minutes.

5. CONCLUSION

If you plan to use mutation testing on Java programs, and are looking for an automated and efficient tool, you may wish to consider JAVALANCHE. Not only is it built for efficiency from the ground up, it also effectively addresses the problem of equivalent mutants. JAVALANCHE is publicly available at the JAVALANCHE web site

`http://www.javalanche.org/`

6. REFERENCES

[1] 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. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, Banff, Alberta, 1988. IEEE Computer Society Press.

[2] B. J. M. Grün, D. Schuler, and A. Zeller. The impact of equivalent mutants. In *Mutation 2009: International Workshop on Mutation Analysis*, Apr. 2009.

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

[4] A. J. Offutt and R. H. Untch. *Mutation 2000: Uniting the orthogonal*, pages 34–44. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[5] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *ISSTA '09: Proceedings of the 2009 International Symposium on Software Testing and Analysis*, July 2009. To appear.

[6] R. H. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using mutant schemata. In *ISSTA '93: Proceedings of the 1993 International Symposium on Software Testing and Analysis*, pages 139–148, New York, NY, USA, 1993. ACM.