Aspect-oriented Response Injection: an Alternative to Classical Mutation Testing

Bartosz Bogacki, Bartosz Walter

Institute of Computing Science, Poznań University of Technology, Poland
{Bartosz Bogacki, Bartosz.Walter}@cs.put.poznan.pl

Abstract. Due to increasing importance of test cases in software development, there is a need to verify and assure their quality. Mutation testing is an effective technique of checking if tests react properly to changes by introducing alterations to the original source code. A mutant which survives all test cases indicates insufficient or inappropriate testing assertions. The most onerous disadvantage of this technique is considerable time required to generate, compile mutants and then execute test cases against each of them. In the paper we propose an aspect-oriented approach to generation and execution of mutants, called response injection, which excludes the need for separate compilation of every mutant.

1 Introduction

Along with growing popularity of agile methodologies and open source movement, unit testing has become one of the core practices in modern software engineering. It is particularly important in eXtreme Programming [2], which explicitly diminishes the importance of other artifacts than source code and tests cases. In XP unit test cases not only verify if software meets functional requirements, but also enable refactoring, alleviate comprehension and provide guidance on how the production code should be used. Therefore, they contribute to many other important practices of XP.

Test-first coding [3] is an example of a practice which employs the test cases in an infrequently used way. It reverses the traditional order of activities at software development: the test cases get written prior to the production code and play the role of formally expressed requirements. System to be implemented is then treated as mere fulfillment of contracts imposed by tests. Poor quality tests effectively prevent such system from being successfully completed. Quality is here interpreted as the ability to discover possible flaws in the production code, which in turn requires the tests to cover every single piece of the code. The resulting measure, test coverage, is one of most important indicators assessing test quality. It reflects the percentage of source code covered by test cases. Low coverage indicates that tests are unlikely to discover changes or bugs introduced to the production code.

Mutation testing [4] is another technique introduced to verify the quality of the test suite. Unlike the coverage metrics, which only determine the constructs that are executed by tests, it figures out how test cases actually react to a faulty response from the source code. It is based on the assumption that high quality test cases discover any al-

Please use the following format when citing this chapter:

Bogacki, B., Walter, B., 2006, in IFIP International Federation for Information Processing, Volume 227, Software Engineering Techniques: Design for Quality, ed. K. Sacha, (Boston: Springer), pp. 273–282.

teration within the source code which makes the code to behave even slightly differently. The erroneous response is most often generated through simple source code modification. Hence, we use the term *mutation* and the faulty programs are called *mutants* of the original. Mutant is killed by test cases when it causes them to fail.

Mutation testing is considered an effective method of detecting code uncovered by test cases. Unfortunately, it has not been widely adopted by the software industry, mainly due to its high computational complexity and resulting low performance. Typically, every testing cycle includes multiple phases. First, the code needs to be analyzed and mutants get created, so that each mutant contains a single modification. Then every mutant is compiled and presented to all existing test cases, which themselves are not mutated. The time spent on processing a single mutant is a sum of all these factors, and then is multiplied by the number of mutants, which in total is quite complex. Therefore, mutation testing is still practically inapplicable for medium or large scale systems that comprise large number of tests.

In the paper we present *response injection* [15] – a novel approach to mutation testing, which employs aspect-oriented programming (AOP) [1, 8] to produce and execute mutants. It addresses mainly the complexity, which is the most onerous disadvantage of traditional mutation testing. Use of aspects removes the need for multiple compilations, which significantly reduces time required for testing.

In the section 2 of the paper we briefly summarize the status of research on mutation testing and present two existing frameworks: Jester and MuJava. Section 3 describes the concept of aspect-oriented mutations, presents its architecture and an example of use. Results of early evaluation are given in section 4. Finally, in section 5 we provide conclusions and directions for further research.

2 Overview of mutation techniques

Mutation testing, introduced in 1977 by Hamlet [4], has been developing for years as an academic research topic rather than an industry method of testing the tests. There are two main directions of works: one related to the scope and nature of changes, specifically the mutation operators and their variations, and the other one focused on performance improvement. The former one has been driven by the shift in the dominant paradigm of programming from structural to object-oriented. The efforts related to performance adhered to three basic rules: *do faster*, *do smarter* and *do fewer*. The first one targets at faster generating and executing mutants, the second one applies techniques of reusing the already acquired information in processing subsequent mutants, and the latter attempts to limit the number of mutants without loosing information. In order to preserve mutant's properties, Offutt [13] identified three conditions that it must satisfy:

1. The **reachability condition** is that the mutated statement must be reached by a call fro the test case;
2. The **necessity condition** is that once the mutated statement is executed, the test case must cause the mutant program to behave erroneously; the fault that is being modeled must result in a failure in the program's behavior;

3. The **sufficiency condition** states that the incorrect state must propagate to the calling test case and result in a failure.

A high quality mutant is expected to satisfy all these conditions. However, traditional mutation testing techniques often fail in achieving this goal.

2.1 Jester

Jester [6] is an open source, free mutation testing framework for Java developed and maintained by Ivan Moore. It became widely known in 2001, after the paper on Jester was presented on XP'2001 conference [12]. A testing cycle in Jester comprises thre phases: introducing a change to a source file, recompiling that file and running all tests. The mutation operators available in Jester are defined by user, but their capabilities are limited to plain text replacement. Examples include modifying literals, changing "true" to "false" and vice-versa, altering conditionals by replacing "if (" with "if (true ||" or "if(false &&", etc. The important disadvantage of Jester is that it performs no code analysis, which means it may easily produce equivalent or even invalid mutants. It results in lots of errors which require manual analysis and recovery. The critical issue concerning Jester is its poor performance, mainly due to necessity of compiling the source code after each mutation is created.

Although Jester may be a an acceptable opportunity for small programs, its applicability to larger projects is limited.

2.2 MuJava

MuJava is another mutation testing framework for Java. It has been developed by Ma, Offutt and Kwon [11] in response to Jester's basic deficiency: performance. MuJava utilizes two different methods to mutate programs: MSG for altering code behavior and bytecode instrumentation for changing program structure. It also employs a wide range of mutation operators, which allows for performing diverse mutations at different levels of code composition.

MSG method [14] is based on metamutants, derived from the program under test. They abstract the pieces of prospective code to be mutated, so that it can be instantiated with concrete values during execution. Every instance of metamutant is an ordinary mutant, which introduces a single fault. Because metamutants are compiled only once, they significantly improve the testing performance.

Bytecode manipulation is performed in MuJava with a BCEL, a specialized Java library which facilitates creation and instrumentation of the bytecode inside Java VM. It is employed to modify the structure of the tested bytecode, e.g. to add a field or a method to a class, to implement an interface in a class or to change inheritance hierarchy.

Both mutating techniques operate at low level, which removes the need for altering source code. The gain in performance of mutant generation and execution comes primarily from removal of the recurring compilation phase. Experiments determined the speedup of entire testing process to 5.1, while only in mutant generation phase it is even 9.3 times faster than with Jester [11]. However, we found no experiments comparing directly MuJava and Jester's performance.

3 Mutants generator

3.1 Concept of response injection

In traditional model of mutation testing, mutants are generated by small source code modifications, which preserve program's syntactic correctness. Modifications are introduced separately to ensure their effects do not compensate. A mutation can be recognized if it affects the method behavior verified by test cases. The behavior can be tested either directly, by examination of return value or exception thrown, or indirectly, if it changes the internal state of object. This leads to the conclusion that mutants are discovered in one of two ways: either by direct verification of method call result, or by examination of object attributes. To depict the above, let us consider the exemplary source code presented in Figure 1 and its test case in Figure 2.

```
public class Foo {
  public int bar(int a)
      throws IllegalArgumentException {
    if ((a > 5) || (a < 1)) {
      throw new IllegalArgumentException();
    }
    int c = a;
    for (int i = 0; i < a; i++) {
      c *= 10;
    }
    return c;
  }
}
```

Fig. 1. Exemplary source code under test

```
public void testBar () {
  assertEquals (3000, new Foo().bar(3));
  try {
    new Foo().bar(6);
    fail ("Exception not thrown for value: 6");
  } catch (IllegalArgumentException e) {}
  try {
    new Foo ().bar(0);
    fail ("Exception not thrown for value: 0");
  } catch (IllegalArgumentException e) {}
}
```

Fig. 2. Exemplary JUnit test method for method *bar()* in class *Foo*

For the above source code (Figure 1) the test (Figure 2) will fail (kill mutant) if the return value of the call to the method `Foo.bar()` with parameter *a* equal to 3 will

be different than 3000 or an unexpected exception will occur, or if parameter *a* equal to 0 or 6 will not make the method to throw an expected exception. However, no mutation will be found if it does not affect the method outcome, for example if the condition `if((a>5)||(a<1))` would be replaced with `if((a>5)||(a<1)||(a<10))`.

To create mutants sufficiently fast we need a method to non-invasively modify behavior of selected methods (one at a time), so that it poses a mutated effect on its callers without need for re-compilation at every change. This led us to selection of aspect-oriented programming (AOP) [8, 10].

AOP was originally invented as a response to an inability of object-orientated paradigm in providing encapsulation of features crosscutting unrelated parts of the developed system. Aspects allow for grouping such features and applying them to selected *joinpoints* – well defined points in program execution. Joinpoints with specifically defined criteria, called *pointcuts*, once captured, execute associated pieces of code (called *advices*) or change the program structure.

In the example (see Figure 2) all calls to `Foo.bar()` could be captured on the fly and their actual results (return value and/or exceptions) were mutated as if the modification had been introduced directly in the source code. We called this idea *response injection*, because the mock method response is injected instead of the actual object. Exemplary AspectJ implementation is shown in Figure 3.

```
public aspect FooMutant {
  int around():
      // capture a call to method bar()
      // defined outside this aspect
      call(public int bar(int))
         && !within (*..*Mutant) {
         // and return a mutated value instead
         return Integer.MAX_VALUE;
      }
}
```

Fig. 3. Exemplary aspect mutating behavior of method *bar()*

3.2 Architecture

The proposed system is composed of two collaborating aspects: *MutantGenerator* and *MutantExecutor*.

The first one captures the original flow of the code executed by a test case and is responsible for mutating the results of the tests method. It takes over the control at every method call and has a choice of replacing its execution with own code or proceeding with the existing one. In order to better mimic the normal program flow, the aspect executes each test case twice. During the first pass it captures the information from the original program flow and generates mutants. During the second pass, it runs the test once per each mutant and looks if the mutant is killed. Figure 4 depicts the original program flow with sequence diagram.

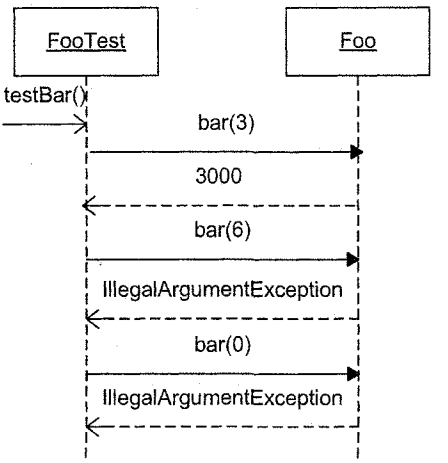

Fig. 4. Sequence diagram for original program flow

As a comparison to the original flow, Figure 5 presents the program flow with `MutantGenerator` aspect.

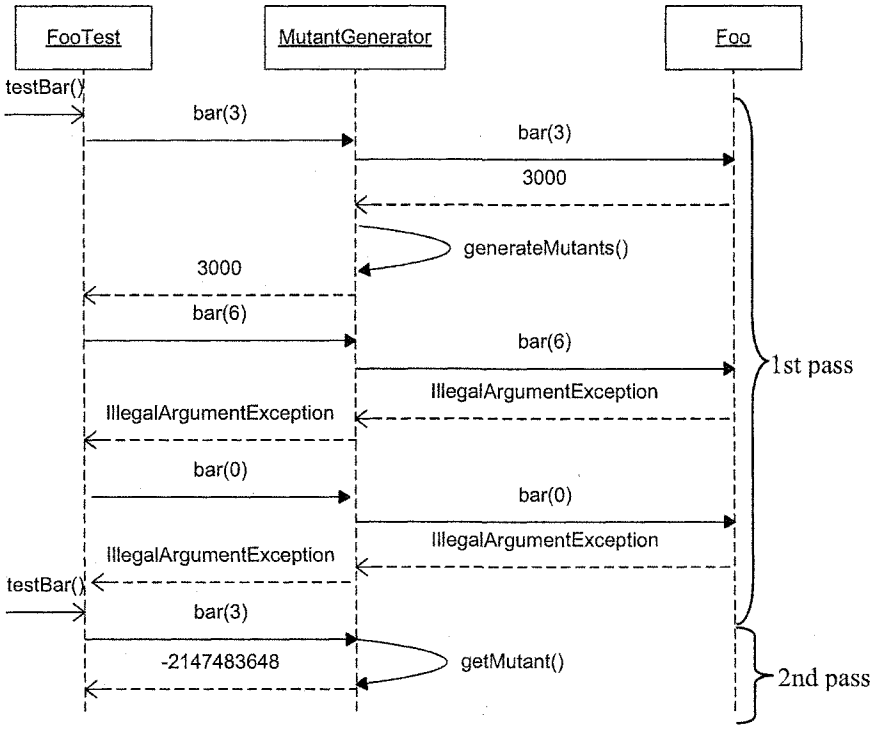

Fig. 5. Sequence diagram for modified program flow employing *MutantGenerator*

Each test case must be executed a number of times, once for each mutant. This leads to introduction of another aspect, `MutantExecutor`, that wraps the test code execution. Its responsibility is to handle each call to the testing method in test case and wrap it with subsequent executions of mutants generated by `MutantGenerator`. MutantExecutor plays the role of meta-mutant, which includes all mutants for a given method, but requires only a single compiling. It also intercepts any exceptions, assures that they do not propagate to the JUnit TestRunner and instead presents results of the test case execution. Figure 5 presents the sequence diagram for the testing routine with both `MutantGenerator` and `MutantExecutor`.

For the prototype implementation we used AspectJ [1, 8, 10] compiler to build code and tests, and JUnit [7] as a testing library.

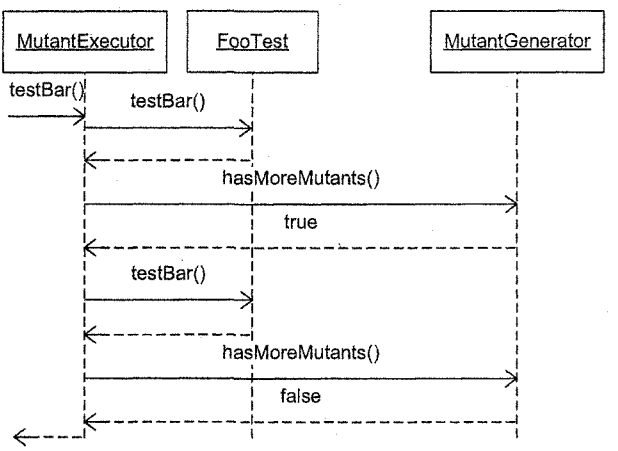

Fig. 6. Sequence diagram for modified program flow employing both *MutantGenerator and MutantExecutor*

3.3 Mutation example

Currently the prototype uses only simple mutation operators, dealing with changing primitive types and String objects, yet they seem sufficient to present the idea. For example for `int` variable of value `result` the mutations include: `-result`, `result+n`, `result-n`, `Integer.MIN_VALUE`, `Integer.MAX_VALUE`, and `0`, where n is a random integer. The only mutation we currently apply to objects is `null` value. In future, we plan to introduce more sophisticated mutants for objects (which could benefit from an on-fly object creation with dynamic proxy).

Considering our exemplary code, for `Foo.bar(3)` call, we end up with the following mutants: –3, 3 + n, 3 – n, –2147483647, 2147483647, 0. All such mutants get killed by the test case.

4 Early evaluation results

In order to evaluate the proposed solution, we conducted an experiment with the prototype tool. As an object of experiment we selected Commons Lang v2.1 [5] from the Apache Jakarta Project. Commons Lang features a very good code coverage: it includes over 1250 tests, with 90.9% of conditionals coverage and 91% of statements coverage. The size of the code measured in NCLOCs (non-commented lines of code) exceeded 13K.

To setup a context for our evaluation we decided to compare the results with Jester's. We selected Jester due to its popularity in eXtreme Programming community. However, Jester deficiencies prevented it from objective and unbiased evaluation. A mutation that violates the code syntactic correctness makes Jester hang, which requires manual fixing. To avoid that the code needs to be carefully tagged, which affects the measurement. Therefore, the experiment was meant to show a tendency, not exact results.

Experiment was performed on a PC with Intel Pentium 1.7GHz Centrino with 1GB of RAM, running Windows XP Professional and Java VM 1.4.2_08.

4.1 Performance

Execution time was measured for Commons Lang test cases. As we were unable to execute Jester for entire project due to the abovementioned facts, we decided to limit the experiment to a few selected test suites only. Figure 7 presents the results.

TestSuite	Jester	Response Injection	LOC	NCLOC	Tests #	Speedup
MathTestSuite	1532 sec.	50 sec.	4908	1988	163	30.6
BuilderTestSuite	782 sec.	49 sec.	6836	2310	247	16
EnumTestSuite (enum)	82 sec.	41 sec.	921	222	63	2
EnumTestSuite (enums)	85 sec.	45 sec.	916	225	64	1.9
ExceptionTestSuite	278 sec.	39 sec.	1912	765	62	7.1
MutableTestSuite	58 sec.	38 sec.	1376	378	49	1.5
TimeTestSuite	2250 sec.	69 sec.	3456	1652	40	32.6
AllLangTestSuite	**no data**	76 sec.	39175	13838	1245	*approx.1776*

Fig. 7. Summary of generation, compilation and execution times for selected test suites of Apache Jakarta Commons Lang project

Reducing the scope of the experiment does not significantly affect compilation time for our prototype, because it still requires compiling entire project with AspectJ. This introduces a constant timing factor, which is independent from size of the tested package, while Jester requires a repeated compilation of every mutant.

The results obtained from Jester and the aspect tool cannot be directly compared.; however, the results allow for drawing some conclusions. Despite of inaccuracies in measurement, the aspect-oriented response injection tool appeared considerably faster for all packages that could be compared with Jester. The gain appears higher for larger testing suites, which could suggest that it could be exploited in production environment.

4.2 Quality

Effective mutation testing benefits not only from performance gain. The other factor is mutants quality, interpreted as their ability to discover bugs with minimal effort.

Adherence to Offutt's conditions is one of quality measures. Noticeably, the response injection approach fulfills all of them. Reachability is ensured by the mutants generation process: mutated statement is always reachable for a test case, because it was injected in response to a call to the statement in test code. Similarly, the necessity condition is preserved as well: the mutated code actually behaves incorrectly, because its response is altered. Sufficiency condition, which requires that a fault is propagated up to the test case, is satisfied by mutating directly the actual method called by the test case.

To assess the quality of generated mutants we analyzed classes from *org.apache.jakarta.commons.math* package. Jester produced 1136 mutants for that package, and 189 of them survived the testing phase. We reviewed them manually in order to assess their applicability in test code improvement. In most cases they have not been killed because they did not meet some of the Offutt's conditions (reachability, necessity or sufficiency).

For the same code base the aspect-oriented tool generated 1978 mutated responses. Test cases indicated that only 3 injected responses did not make any test case to fail. All of them required more strict assertions to be introduced to the test cases, but did not violate any of the conditions.

5 Conclusions

The results of initial evaluation of the presented tool show that use of aspects in mutation testing appears a promising opportunity. The prototype we built generates the mutants much faster than popular Jester, while preserving three required properties: reachability, necessity and sufficiency. The main functional difference is that it traverses the existing test cases to learn the code usage, and then evaluates if the tests are exhaustive enough. Jester, on the other hand, mutates the code independently from test cases, which allows it for assessing the code coverage. That is the reason why the quality of mutants generated by the prototype cannot be directly compared to the Jester's. However, it appears to produce mutants of higher quality by avoiding the redundant equivalent mutants. It also, unlike Jester, performs mutation in strict accordance with the test coverage.

Use of aspects preserves the production source code intact and also allows for various mutation operators, changing both behavior and structure of the code under test.

Further directions of research and development include support for objects, implementation of other mutation operators and a larger scale evaluation.

Acknowledgements

The work has been supported by the Rector of Poznań University of Technology as a research grant BW/91-429.

References

1. AspectJ Project HomePage, http://www.eclipse.org/aspectj/ (visited in January 2006)
2. Beck K.: Extreme Programming Explained. Embrace change. Addison-Wesley, 2000.
3. Beck K.: Test-Driven Develoment. By Example. Addison-Wesley, 2003.
4. Hamlet R.G.: Testing programs with the aid of compiler. IEEE Transactions on Software Engineering, Vol. 3(4), July 1978, pp.279-290
5. Jakarta Commons Lang Project, http://jakarta.apache.org/commons/lang/
6. Jester HomePage, http://jester.sourceforge.net/ (visited in January 2006)
7. JUnit HomePage, http://www.junit.org (visited in January 2006)
8. Kiczales G., Lamping J. et al.: Aspect Oriented Programming. In: Proceedings of ECOOP 1997, Lecture Notes in Computer Science 1241, Springer Verlag, pp. 220-242.
9. Kim S., Clark J., McDermid J.: Assessing test set adequacy for object oriented programs using class mutation. In: Proceedings of Symposium on Software Technology (SoST'99), pages 72-83, Sept. 1999.
10. Laddad R.: AspectJ in Action. Manning Publications, 2003
11. Ma Y., Offutt J., Kwon Y. R.: MuJava. An automated Class Mutation System. In: Software Testing, Verification and Reliability. June 2005. Vol. 15(2), pp. 97-133.
12. Moore, I.: Jester a Junit test tester. In: Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering, XP2001. Springer 2001.
13. Offutt A. J.: A Practical System for Mutation Testing: Help for the Common Programmer. Test Conference, 1994. Proceedings., International.
14. Untch R., Offutt A. J., Harrold M. J.: Mutation analysis using program schemata. In: Proceedings of the 1993 International Symposium on Software Testing, and Analysis, pages 139-148, Cambridge MA, June 1993
15. Bogacki B., Walter B.: Evaluation of test code quality with aspect-oriented mutations. In: Abrahamsson P., Marchesi M., Succi G.: Proceedings of 7th International Conference in Extreme Programming and Agile Processes in Software Engineering, Oulu (Finland), June 2006, Lecture Notes in Computer Science 4044, Springer Verlag, pp.202-204.