Evaluation of Test Code Quality with Aspect-Oriented Mutations

Bartosz Bogacki and Bartosz Walter

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

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 [1], which explicitly diminish 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, which explicitly or implicitly rely on their ability to effectively discover bugs.

Mutation testing [2] is a technique used for verifying the quality of tests. It figures out how the test cases actually react to faulty response received from deliberately altered production code. High quality tests are expected to uncover any mutation of the source code which makes it to behave even slightly differently. Such modified code (called *mutant*) is killed when it causes at least one test case to fail.

Despite of its advantages, mutation testing has not been widely adopted by software industry. The main drawback its high complexity: it usually includes multiple phases of mutating source code, compilation and running the tests. Therefore, the technique is in practice inapplicable for medium or large size systems.

In the paper we present a prototype tool for mutation testing, which employs aspect-oriented programming (AOP) [3] to generate and execute mutants. It follows the control of existing test cases and examines how they deal with the altered production code, while significantly reducing time required to create and run mutants.

2 Architecture of Aspect-Oriented Mutants Generator

In traditional model of mutation testing, mutants are generated by arbitrary or directed production code modifications, e.g. operator replacement, redefinition of a method etc. The mutations are performed in separation in order to avoid possible cross-cutting side effects. Depending on the scope of changes, they are or not externally visible to test cases through altered results of method execution. To depict the above, let us consider an exemplary source code presented at Fig. 1 and its test case at Fig. 2. Te test will fail (kill mutant) if one of three conditions is met: (1) the return value of the method `Foo.bar()` called with parameter 3 is different than 3000, or (2) an

P. Abrahamsson, M. Marchesi, and G. Succi (Eds.): XP 2006, LNCS 4044, pp. 202 – 204, 2006.

unexpected exception occurs, or (3) the parameter values 0 or 6 do not make the method to throw an expected exception. However, the mutant cannot be discovered if it does not affect the method outcome.

```
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 ("Expected exception for value: 6");
  } catch (IllegalArgumentException e) {}
  try {
    new Foo ().bar(0);
    fail ("Expected exception for value: 0");
  } catch (IllegalArgumentException e) {}
}
```

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

Hence, it seems sufficient to observe the reaction of test cases to such properties, without tracking individual changes in the production code and expecting the change to reveal with tests cases failures. In order to dynamically and non-invasively access the method results, we employed the capabilities of Aspect-Oriented Programming. In the example (see Fig. 2) all calls to `Foo.bar()` could be captured on the fly by an aspect and their actual results (return value and/or exceptions) would be replaced with mutants, just as if the mutation had been introduced directly into the source code.

The proposed prototypic tool, which exploits this observation, is actually composed of two collaborating aspects: *MutantGenerator* and *MutantExecutor*. The first one follows the original flow of a test case and captures control at every method call. In order to better mimic the normal program behavior, the aspect executes each test case twice. First, it runs the original method and stores its results and context. Secondly, it generates mutants of the results, applying typical testing rules, e.g. an integer yields following mutants: *0*, *–value, value* ± *n, Integer.MIN_VALUE* and *Integer.MAX_VALUE*.

Subsequently, the other aspect, *MutantExecutor*, wraps test code execution and runs each of the generated mutants. Its responsibility is to capture each call to the tested method in a test case and replace it with subsequent executions of the mutants generated by *MutantGenerator*. It also intercepts any exceptions that may be thrown, preventing them from being propagated to the TestRunner, which could falsely classify them as assertion failures.

It is important to notice that both aspects are core parts of the tool and do not need to be created or compiled specifically for the production code to be mutated.

4 Conclusions

To evaluate this approach, we have built a prototype based on AspectJ [4] compiler to build code and tests and with JUnit [5] as the testing library. Early experiments show that it appears to generate and run the mutants a few orders of magnitude faster that the popular Jester [6]. The savings result mainly from the fact that the tool does not require multiple mutant compilations, reduces the number of equivalent and transparent mutants, and preserves the syntactic correctness of the mutated code. However, it differs from Jester in that it learns the code usage from existing test cases, and then mutates the code. Jester, on the other hand, mutates the code insight into test cases, which allows for assessing the code coverage, but also leads to redundant or transparent mutants.

Currently the prototype deals only with primitive Java types and `null` values for objects. In future, we plan to employ an on-fly object creation with dynamic proxies and implement other mutation operators as well as perform 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. Beck K.: Extreme Programming Explained. Embrace change. Addison-Wesley, 2000.
2. Hamlet R.G.: Testing programs with the aid of compiler. IEEE Transactions on Software Engineering, Vol. 3(4), July 1978, pp.279-290.
3. 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.
4. AspectJ Project HomePage, http://www.eclipse.org/aspectj/ (visited in January 2006).
5. JUnit homepage, http://www.junit.org (visited in January 2006).
6. Moore I.: Jester. A Junit test tester. In: Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering, XP2001.