Applying Mutation Analysis for Object-Oriented Programs Using a Reflective Approach

Philippe Chevalley*
LAAS-CNRS, 7 avenue du Colonel Roche
31077 Toulouse Cedex 4, France
chevalle@laas.fr

Abstract

This paper presents a mutation analysis tool based on a reflective macro system. Mutation analysis is a powerful and computationally expensive technique that measures the effectiveness of test cases for revealing faults. The principal expense of mutation analysis is that many faulty versions of the program under test, called mutants, must be repeatedly executed. This technique has originally been developed in the framework of procedural programs, and should be revisited to consider some types of faults likely to appear in object-oriented environments. The mutation system detailed in this paper deals with object-oriented concepts introduced in the JAVA *programming language, and is able to generate automatically mutant programs according to 20 types of object-oriented faults.*

1. Introduction

Software testing involves exercising a program on a set of test case input values and comparing the actual output results with expected ones [1]. Since exhaustive testing is usually not tractable, test strategies are faced with a problematic task that is: how to select a minimum set of test cases that is sufficiently effective for revealing potential faults in a program? An evaluation criterion for test strategies is to measure the effectiveness of generated test cases. Mutation analysis [5] is an evaluation technique that assesses the quality of test cases by examining whether they can reveal certain types of faults. It consists of creating a sample of faulty programs from the original program under test, by injecting faults, called mutations. Mutations are single-point syntactically correct changes introduced one by one in the original program. A mutant is a copy of the original program containing one mutation. If test cases cause a mutant to produce different outputs than the original program, the mutation is revealed and said killed by the test cases. If the mutant is not killed, it falls into one of the two categories: (i) the mutant is functionally equivalent to the original program, which means that no test cases can kill it since it always produces the same outputs than the original program; (ii) the mutant is not equivalent and the test cases failed to reveal the mutation. The percentage of non-equivalent mutants killed by a set of test cases represents the mutation score, which is an effectiveness measure for test cases.

The syntactic modifications responsible for mutant programs are determined by a set of mutation operators. This set generalizes typical programming errors and is determined according to the programming language used for developing the program under test. For procedural programming, a mutation operator is for instance to replace an arithmetic operator in the original program with other syntactically correct changes representing the programmer using a wrong arithmetic operator. Concepts introduced in object-oriented languages such as JAVA represent new causes of errors [2, 9]. Little research has been done on the definition of new mutation operators for targeting potential object-oriented faults. Kim *et al.* [6, 7] propose 15 mutation operators for JAVA programs. However, it seems not realistic to evaluate the effectiveness of test cases without the aid of an automated mutation system for reducing the workload of the tester.

We describe a mutation analysis tool based on a reflective macro system for implementing 20 mutation operators for JAVA programs. The paper is organized as follows. In Section 2, we first discuss on mutation operators for object-oriented programs. Section 3 gives an overview of the tool. Finally, conclusions and future work are given in Section 4.

2. Mutation Analysis for OO Programs

Mutation analysis is a fault-based technique that aims at measuring the effectiveness of test cases. It involves constructing a set of mutant programs, each of which is a copy

*The author is with Rockwell-Collins France as a CIFRE fellow.

of the program under test that differs by exactly one mutation. A mutation is a single, syntactically correct change that is made by applying a predefined transformation rule (called a *mutation operator*) to the original program. Since the efficiency of mutation analysis heavily relies on the adequacy of mutation operators for representing potential programming errors [4], we can wonder whether mutation operators developed in the context of procedural programming are still sufficient in an object-oriented environment. If traditional mutation operators are still valid for object-oriented programs, they must be completed with new operators specifically designed for representing faults related to the object-oriented features introduced by the programming language used for developing the program under test.

2.1. Class Mutation

Class mutation [6] is a mutation system for object-oriented programs developed for JAVA. It defines a set of 15 mutation operators that target plausible faults likely to occur due to object-oriented features introduced in the programming language. Class Mutation operators deal with a large variety of concepts such as class declarations, information hiding, inheritance, polymorphism, and method overloading.

These mutation operators only target possible flaws related to object-oriented concepts. They are obviously not sufficient to measure the effectiveness of test cases. Indeed, these Class Mutation operators must be integrated in a mixed approach considering both categories of faults: traditional and object-oriented faults. Kim *et al.* [7] investigated the effectiveness of object-oriented test strategies in combining two sets of mutation operators. They used Class Mutation operators in addition with 22 traditional operators. These latter operators were translated and adapted from the Mothra's Fortran mutation system to be applied to the JAVA programming language. The traditional operators implemented in the Mothra mutation system were derived from studies of programmer errors and correspond to simple errors that programmers typically make. After several years of refinement through several mutation systems, these 22 mutation operators have demonstrated their ability for measuring the effectiveness of test cases over more than 20 years. The complete set of mutation operators used by the Mothra mutation system is detailed in [8].

Table 1 summarizes the Class Mutation operators [6]. Each of the 15 mutation operators is represented by a three-letter acronym. For example, the "Access Modifier Change" (AMC) operator causes each access control modifier in a program to be replaced by each other modifier. JAVA defines 4 access control modes for attributes and methods: *public*, *package (by default)*, *protected*, and *private*. This operator targets the access control (information hiding) feature of object-oriented programming. We do not intend to comment on the role of each mutation operator presented in Table 1 (see [6, 7] for comments and examples).

Table 1. Class Mutation operators

Operator	Description
AMC	Access Modifier Change
AND	Argument Number Decrease
AOC	Argument Order Change
CRT	Compatible Reference Type replacement
EHC	Exception Handler Change
EHR	Exception Handler Removal
FLS	Field and Local variables Swap
HFA	Hiding Field variable Addition
HFR	Hiding Field variable Removal
HLR	Hiding Local variable Removal
ICE	Instance Creation Expression change
OMR	Overriding Method Removal
POC	Parameter Order Change
SMC	Static Modifier Change
VMR	oVerloading Method Removal

2.2. Extending Class Mutation

The term *Class Mutation* defines a set of recent object-oriented specific mutation operators. To go a step further in the plausible faults a programmer can make, we propose to extend Class Mutation with five new operators. These operators are the result of our own programming experience with object-oriented languages, and JAVA specifically. They target features different than those targeted with the 15 Class Mutation operators. As an example, three out of five [3] additional operators (MNC, RAC, and RCC) are described hereafter.

MNC (Method Name Change) Operator

This operator changes a method call with another compatible method call. It represents the programmer using a wrong method name instead of the specified one. Since a class may have several methods with different names but accepting the same signature (i.e., the same number and type of formal parameters, and returning the same type), the programmer can use a method name instead of the specified one. The MNC operator is justified by the fact that this type of faults cannot be detected at compile-time when methods are compatible (i.e., have the same signature). A class has usually several compatible methods. It is for instance the case with methods only used for returning the value of a private attribute. These methods are called accessor methods or "get" methods in the programmers' jargon.

The class Point presented in Figure 1 declares two methods (`getX` and `getY`) accepting the same signature. These methods can be called on an instance (i.e., an object) of the

```
public class Point {
    public int getX() { return x; }
    public int getY() { return y; }
    ... }
```

Figure 1. Declaration of class Point

Original Code:

```
Point point1, point2;
point1 = new Point();
point2 = point1;
```

RAC Mutant:

```
Point point1, point2;
point1 = new Point();
point2 = point1.clone();
```

Figure 2. RAC mutation operator example

class Point. If the original code is `point.getX()` where `point` references an object Point, this statement can be replaced by a call to the `getY` method, changing then the desired functional behavior. The mutated statement is then: `point.getY()`.

RAC (Reference Assignment and Content assignment replacement) Operator

The RAC operator aims at replacing a reference assignment with a content assignment by cloning the assigned object. It represents a fault the programmer can make when being confused with assignment by value and assignment by reference. The `clone` method is used for duplicating an object. It means creating a new object initialized with the current state of the copied object. The set of attributes of an object defines its current state. The example presented in Figure 2 replaces a reference assignment with a content assignment by creating a clone of the object referenced by the variable `point1`.

In JAVA, an object can be duplicated with the `clone` method when it implements the "cloneable" interface. A lot of standard classes of the JAVA Application Programming Interface (API) implement that interface, i.e., they define a `clone` method for duplicating an object. The RAC operator can then often be applied on a program.

RCC (Reference Comparison and Content comparison replacement) Operator

In the same way, the RCC operator changes a reference comparison with a content comparison, and vice versa. It also targets faults a programmer can easily make when being confused with the reference of an object and its state.

Original Code:

```
1  Point point1 = new Point(1, 2);
2  Point point2 = new Point(1, 2);
   ...
n  boolean b = (point1 == point2);
```

RCC Mutant:

```
1  Point point1 = new Point(1, 2);
2  Point point2 = new Point(1, 2);
   ...
n  boolean b = (point1.equals(point2));
```

Figure 3. RCC mutation operator example

Figure 3 presents an example with two objects Point initialized with the same coordinates.

In the original code, the Boolean variable b is *false* since the variables `point1` and `point2` reference two different Point objects. The RCC operator replaces the "==" logical operator with the method call "equals". The difference is that the logical operator deals with the reference of an object, whereas the "equals" method checks whether the states of the two objects are the same. In the mutated code, the Boolean variable b contains the value *true*.

3. Generating Object-Oriented Mutations

In a first step, we developed traditional mutations using a syntactic analyzer. This analyzer creates for each JAVA file an Abstract Syntax Tree (AST) that is examined for generating mutant files. One of the drawbacks of this approach is that ASTs are used for representing the source code but do not give immediate access to object-oriented features of the source code. Indeed, Class Mutation operators require examining object-oriented features for classes (such as inheritance relations, definition of attributes and methods), and this is not easily achieved using traditional ASTs. A key idea for Class Mutation operators is to use a macro system (developed for JAVA) that structures the information derived from the analysis of the source code and presents that information in an object-oriented viewpoint. The macro system must be sufficiently robust to handle an AST that may be huge. Indeed, implementing Class Mutation operators require a tool that must gather all the information present in a program. Unlike programs developed with other programming language, a JAVA program is usually made of an important number of files. If it was manageable to analyze each JAVA file separately to generate traditional mutant programs, Class Mutation operators require browsing through the information that can be found in several files. For instance, the CRT mutation operator needs to run through inheritance relationships to create mutations. Indeed, this operator replaces a reference type with compatible types, which are deduced from inheritance relationships.

Using a Reflective Macro System: OpenJava

A reflective macro system aims at changing the program behavior according to another program. It performs textual substitutions so that a particular aspect of a program is separated from the rest of that program. OPENJAVA [10] is a compile-time reflective system. A major idea of OPENJAVA is that macros (meta programs) deal with class metaobjects representing logical entities of a program instead of a sequence of tokens or ASTs. OPENJAVA distinguishes two levels of code. The program that performs reflective computation is called a *meta-level* program while the program to which the reflective computation is performed is called a *base-level* program. These two levels interact through a metaobject protocol (MOP), which is an object-oriented interface that gives programmers the ability to write meta-level programs in order to implement new extended language features. MOPs can be used at compile-time or at runtime. OPENJAVA uses the meta-information at compile-time, which provides better performance than runtime reflective systems.

The OPENJAVA compiler accepts source programs and generates *bytecode* for the JAVA Virtual Machine (JVM). The difference from regular JAVA compilers is that the OPENJAVA compiler refers to meta-level programs in addition to regular libraries. To perform textual substitutions, the system uses a translator module that takes a source program to generate an AST representing information from an object-oriented standpoint. Then, the module transforms the AST according to the meta-level program. Finally, it generates source code in the regular JAVA language from the AST nodes representing elements of the program: e.g., class declarations, variable declarations and method calls. Mutation operators are then implemented in using the notion of macro expansions.

According to this approach, a mutation analysis tool has been developed for implementing 26 mutation operators (6 traditional and 20 object-oriented operators) [3]. The tool provides a convenient graphical user interface to make mutation analysis faster and less painful. The tester can use the interface to generate mutants, to identify equivalent mutants, and to visualize statistics details.

4. Conclusions

The contribution of this paper is twofold. It proposes to extend Class Mutation with new mutation operators, which enforce the promising efficiency of that technique for considering (more) plausible faults related to the object-oriented features existing in the JAVA programming language. Class Mutation operators deal with complex object-oriented concepts, which harden the task of creating mutations. Thus, it is structurally impossible for tools supporting traditional mutation analysis to implement Class Mutation operators with slight modifications. Combining reflection and mutation analysis is an innovative approach that constitutes the foundations of our mutation analysis tool.

Since mutation analysis is a fault-based technique that requires large amounts of computation, our tool contributes to reducing the cost of mutation analysis by offering a solution to automatically creating mutants.

Acknowledgements

The author would like to thank very much Pascale Thévenod-Fosse for her invaluable suggestions, David E. Statezni of Rockwell-Collins, Inc. for his advice throughout this project, and Christophe Moine and Jérôme Chancel for their useful contribution throughout the development of the tool, within the framework of student projects. The author also wishes to thank Juan-Carlos Ruiz-Garcia for his interesting discussions on reflection.

References

[1] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, USA, Seconde Edition, 1990.

[2] R. Binder. *Testing Object-Oriented Systems – Models, Patterns, and Tools*. Addison-Wesley, 1999.

[3] P. Chevalley and P. Thévenod-Fosse. A Mutation Analysis Tool for Java Programs. LAAS Report N° 01356, Toulouse, France, Sept. 2001. (Submitted for Publication).

[4] M. Daran and P. Thévenod-Fosse. Software Error Analysis: A Real Case Study Involving Real Faults and Mutations. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA'96)*, pages 158–171, San Diego, USA, Jan. 1996.

[5] R. DeMillo, R. Lipton, and F. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, Apr. 1978.

[6] S. Kim, J. Clark, and J. McDermid. Class Mutation: Mutation Testing for Object-Oriented Programs. In *Proceedings of the Net.ObjectDays Conference on Object-Oriented Software Systems*, Erfurt, Germany, Oct. 2000.

[7] S. Kim, J. Clark, and J. McDermid. Investigating the Effectiveness of Object-Oriented Testing Strategies with the Mutation Method. In *Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, San Jose, USA, Oct. 2000.

[8] K. King and A. Offutt. A Fortran Language System for Mutation-based Software Testing. *Software – Practice and Experience*, 21(7):685–718, July 1991.

[9] D. Kung, P. Hsia, and J. Gao, editors. *Testing Object-Oriented Software*. IEEE Computer Society, 1998.

[10] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A Class-Based Macro System for Java. In *Reflection and Software Engineering*, volume 1826 of *LNCS*, pages 117–133, Heidelberg, Germany, June 2000. Springer.