Int J Softw Tools Technol Transfer (2003) 5: 90–103 / Digital Object Identifier (DOI) 10.1007/s10009-002-0099-9

Regular contribution

A mutation analysis tool for Java programs

P. Chevalley*, **P. Thévenod-Fosse**

LAAS-CNRS, 7, avenue du Colonel Roche, 31077 Toulouse Cedex 04, France; E-mail: thevenod@laas.fr

Published online: 17 December 2002 –

Abstract. Program mutation is a fault-based technique for measuring the effectiveness of test cases that, although powerful, is computationally expensive. The principal expense of mutation is that many faulty versions of the program under test, called mutants, must be created and repeatedly executed. This paper describes a tool, called JavaMut, that implements 26 traditional and object-oriented mutation operators for supporting mutation analysis of Java programs. The current version of that tool is based on syntactic analysis and reflection for implementing mutation operators. JavaMut is interactive; it provides a graphical user interface to make mutation analysis faster and less painful. Thanks to such automated tools, mutation analysis should be achieved within reasonable costs.

Keywords: Test case evaluation – Mutation analysis – Mutation operators – Object-oriented programming – Java – Reflective systems – OpenJava

1 Introduction

This paper uses the following definitions [13]. A *fault* can be created at any time in any phase of software development. During program execution, when the faulty instruction is triggered by an appropriate input pattern, the fault becomes active and may produce one or several errors (incorrect internal states). An *error* may propagate by creating other new error(s); if and when the erroneous data affect the output result(s), i.e., the program supplies wrong output value(s), a *failure* occurs, thus revealing the presence of the fault.

Software testing involves exercising a program on a set of test case input values and comparing the output results with expected ones [6]. Any discrepancy reveals a residual fault in the program under test. Since exhaustive testing on the whole input domain of the program is not tractable, test strategies are faced with the issue of selecting a minimum set of test cases that is assumed to be sufficiently effective for revealing potential residual faults a priori unknown.

Program mutation, originally proposed by DeMillo et al. [5], is a fault-based technique for measurement of test case adequacy. Test case adequacy refers to the ability of test case input values in revealing faults. Given a program P, the mutation technique consists of creating a (large) sample of faulty programs from P, by injecting faults called mutations. Mutations are single-point syntactically correct changes introduced one by one in the original program P. A mutant of P is a copy of P containing one mutation. Then, the adequacy of a set of test cases T is measured by examining to what extent T is able to reveal the injected mutations, that is, by evaluating the proportion of mutants of P that produce different outputs than P in response to at least one of the test case input values included in T. In other words, the mutation technique assesses the ability of a test set T to distinguish small syntactic modifications from the program under test P.

The syntactic modifications responsible for mutant programs are determined by a set of mutation operators, called hereafter a *mutation system*. These operators generalize 'typical' programming faults and are determined according to the programming language used for implementing the original program P. Previous work was mainly concentrated on the definition of mutation operators for procedural programs (e.g., Fortran [16], C [1]). As regards object-oriented (OO) programs, recent papers

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

Correspondence to: Pascale Thévenod-Fosse

of Kim et al. [10, 11] have extended program mutation to be applicable to the JAVA language. The authors define a mutation system – called *Class Mutation* – which includes 15 new operators that target plausible faults likely to occur due to OO specific features such as inheritance, or polymorphism.

A major drawback of the mutation technique is its high cost of application, first due to the high number of mutants created. This has motivated the definition of selective mutation systems, which aim to reduce the number of generated mutants through a reduction in the number of mutation operators [1, 16]. Nevertheless, whatever the mutation system, the technique needs the support of a tool to automatically create a sample of mutants.

This paper presents a prototype tool, called JAVAMUT, implementing a mutation system for JAVA programs. It offers a set of 26 operators, including six traditional operators (similar to those initially defined for procedural programs), the 15 operators of the Class Mutation system, and five new operators that target additional OO features. It also provides a graphical user interface (GUI) that facilitates the analysis of the results of large campaigns of mutation experiments. To the best of our knowledge, no other mutation tools that implement OO mutation operators are currently available. JAVAMUT runs on any platform supporting the JAVA technology. Implementing the current version of the tool required to develop approximately 25 000 LOC distributed among 158 JAVA classes.

The paper is organized as follows. In Sect. 2, we recall the basis of program mutation, and briefly present current mutation systems and tools. Section 3 gives an overview of our tool. Its implementation is further described in Sect. 4 (traditional mutation operators), and in Sect. 5 (OO mutation operators). Section 6 shows the graphical user interface currently available. Section 7 reports on first empirical results in using the tool on two mid-size case studies related to different domains: an avionics application (6500 LOC) and a banking application (6200 LOC). Conclusions and future work are given in Sect. 8.

2 Program mutation and related work

2.1 Scope of program mutation

Program mutation is a fault-based technique for measurement of test case adequacy [5, 6]. The mutation technique consists of creating a (large) sample of faulty programs from the original program P to be tested. Each faulty program – called *mutant* – is a copy of P containing one single-point syntactically correct change. This type of changes is called *mutation*. If test cases cause a mutant to produce different outputs than P, the injected mutation is revealed, and the mutant is said *killed* by the test cases. If a mutant is not killed, it falls into one of two categories: (i) the mutant is (functionally) equivalent to P – that is, it is indistinguishable from P under the whole set of test cases that may be defined from the input domain of P; or, (ii) the mutant is not equivalent to P but the test cases failed to reveal the mutation, thus indicating a weakness of the test cases.

For a program P, a set of mutants M^P and a set of test cases T, a mutation score – denoted $ms(P,T)$ – is defined as the proportion of non-equivalent mutants included in M^P that are killed by T. It gives a measure of the effectiveness of T in revealing the mutants of M^P that are distinguishable from P. It is a number in the interval $[0, 1]$. A high score indicates that T is very close to being adequate for P relative to the mutant set M^P. Indistinguishable mutants are not accounted for since no test cases can kill them.

Obviously, an acute question concerns the representativeness of $ms(P,T)$ with respect to the adequacy of T relative to real faults. It is directly related to the types of syntactic modifications responsible for mutant programs. These modifications are determined by a set of mutation operators, which forms a mutation system, and we will return to this issue in Sect. 2.2.

The pioneer paper on program mutation [5] provided a basis for a large number of further developments. They may be classified in two broad categories depending on the purpose of the use of program mutation:

- Mutations may be used as a "testing criterion", that is, as a guide for creating a set of test cases T for a given program P (see e.g., [1, 16]). In that case, the tester's task is to produce a set of test cases T that kill all the distinguishable mutants of M^P. T is said mutation adequate since, by construction, it reaches a perfect mutation score $ms(P,T) = 1$. When mutations are used for this purpose, we will speak of **mutation testing**.
- Mutations may be used to compare the scores reached by different sets of test cases on a given program (see e.g., [11, 21]). In that case, we will speak of **mutation analysis**. The aim of mutation analysis may be, for example: (i) to investigate the mutation adequacy of a given test strategy (other than mutation testing) by comparing the scores provided by different sets of test cases created according to the strategy; (ii) to compare the effectiveness of different test strategies by comparing the scores provided by several sets of test cases created according to each strategy, and analyzing the reasons why some mutants are killed by some but not all the test strategies.

Whatever the use of program mutation, the technique relies on the definition of a set of mutation operators specific to the language of the target program P. Section 2.2 gives a brief (non-exhaustive) overview of current mutation systems and associated tools.

2.2 Mutation systems and tools

Mutation systems have been firstly defined for several procedural languages (Assembler, FORTRAN, C, Pascal). The operators included in these systems aim to perform several types of change: replacement of operands, expression modification, and statement modification. Obviously, they are specialized according to the target programming language. For example, the mutation system for FORTRAN defined by King & Offutt, and implemented in the Mothra tool, consists of 22 operators [12]. For the C language, the Proteum (Program Testing Using Mutants) tool involves 71 operators [1]. Both tools are mainly dedicated to mutation testing. A third tool, named SESAME (Software Environment for Software Analysis by Mutation Effects), was designed for mutation analysis purposes [20]. A salient feature of this tool is that it is multilanguages: it implements mutation systems for several procedural languages (Assembler, Pascal, C), and also for a synchronous data flow language (LUSTRE). Here, we do not give more details on these traditional mutation operators (concrete examples will be given in Sect. 4, where they are adapted to the JAVA language).

All these 'traditional' mutation operators are derived from actual observations of programmer faults [6]. They conform with typical programming faults under the "Competent Programmer Hypothesis", which assumes that the most likely faults correspond to such simple faults. Moreover, the restriction of the mutants to single faults is justified by the 'coupling effect', which says that complex faults are coupled to simple faults in such a way that test cases that detect simple faults in a program will detect most complex faults. Note that the coupling effect has been supported theoretically and experimentally [9, 15]. This is in favor of a good (although not perfect) representativeness of mutation faults with respect to programming faults.

As regards other types of faults (e.g., design faults), a detailed experimental comparison of the internal errors (variations in the program internal state) produced by mutations and by real (design) faults was reported in [4]. This investigation was conducted on a C program of about 1000 LOC. It showed that although the studied mutations (created by the SESAME tool) were simple faults, they can create erroneous behaviors as complex as – and thus, as difficult to detect as – those identified for the real faults. A suitable consistency between errors produced by real faults and by mutation faults was observed. This lends support to the representativeness of the mutation score with respect to the adequacy of the test set relative to real faults (in spite of the fact that mutations obviously do not reproduce design faults).

Such traditional mutation operators can be applied to OO programs. However, they may not be sufficient to involve specific features in OO programming (e.g., inheritance, polymorphism). Recent work of Kim et al. addresses this issue [10]. It defines a set of 15 mutation operators, termed Class Mutation, which targets plausible faults that are likely to occur due to OO features in the JAVA language. As far as we are aware, these operators are not implemented in a tool. In the mutation analysis experiments reported in [11], a JAVA version of the Mothra tool, developed by the authors, was used to automatically generate traditional mutants. However, the OO mutants, as defined by the Class Mutation system, were manually created. As we will see in Sect. 5, our tool includes these operators (they will be presented in Sect. 5).

2.3 Practical limitation of program mutation

In practice, mutation systems generate a large number of mutants, some of them being potentially equivalent to the original program. As said in Sect. 2.1, equivalent mutants are not accounted for in the mutation score, since no test sets can reveal them. Hence, a major limitation of program mutation is its high cost of application for two reasons: (i) a large sample of mutants should be executed; and (ii) for every mutant alive it must be determined whether or not it is equivalent to the original program, and this remains essentially a manual, difficult and time-consuming task.

To make program mutation less expensive, an alternative consists of reducing the number of mutation operators to be used. This has motivated the definition of selective mutation systems in the context of FORTRAN [16] and C [1]. These papers are focused on mutation testing. They define subsets of mutation operators that may be used to create test cases preserving a high mutation score. These subsets form selective mutation systems that can drastically decrease the number of mutants generated (e.g., a reduction of 77.56% is obtained in [16]). As a result the number of potential equivalent mutants to be identified is also reduced.

As regards the issue of determining equivalent mutants, partial (automatic) solutions have been proposed [8, 17]. However, this problem is undecidable and thus, will never be fully automated. In the present state of the art, the time consumed identifying equivalent mutants depends on the tester's skill and familiarity with the program under test. This task is sometimes so complex that the tester cannot decide whether or not some live mutants are equivalent. These mutants, termed *stubborn mutants* [8], may be non-equivalent, but very difficult to kill.

To conclude on the practical limitation of program mutation, we think that its main actual drawback is the need for human intervention to identify equivalent mutants. In our mind, the limitation applies more to mutation testing than mutation analysis. Since mutation testing aims at creating a set of test cases that reaches a perfect mutation score (using or not selective mutation systems), equivalent mutants have to be determined (to be not accounted for). In cases of mutation analysis,

which aims to compare mutation scores reached by different test sets and identify which mutations are revealed by some but not all the test sets, the problem is less crucial. Assuming that the percentage of equivalent mutants remain low, if they are accounted for in the score, every test set is penalized in the same way. Hence, the tester could only eliminate equivalent mutants that were easy to identify, stubborn mutants being considered as non-equivalent to avoid complex (or even infeasible) manual analysis.

3 Overview of the JavaMut tool

The JavaMut tool was developed in the framework of our current research work on defining a test strategy for critical avionics systems implemented in Java. The aim was to automatically conduct large campaigns of mutation analysis experiments to analyze the strengths and weaknesses of test cases designed from UML state diagrams [3]. Our tool provides a graphical user interface that helps the tester to customize mutation analysis. The interface is organized around three tabbed panels, each one offering a high-level functionality. The tester can indeed use the interface to generate mutants, to identify equivalent mutants, and to visualize statistics details (textual presentation). As an example, Fig. 1 shows the part of the interface used for generating mutants (the two other parts will be shown in Sect. 6). The tester can select the files for which he or she wants to create mutants, and also mutation operators to apply. Shortcut buttons have been designed to simplify the selection task for the tester. One of the major interests of JavaMut is in automatically creating mutants according to possibly a broad range of faults. Mutants are designed by applying a mutation operator to the original program. The mutation operators currently supported by the tool can be divided into two main classes:

– *Traditional mutations*: six operators are responsible for generating mutations related to procedural programming. The general outline for these six mutation operators is that for each statement, they attempt to transpose variables, modify arithmetic or relational operators, change constant values, and change precedence of operation by deleting parentheses. These

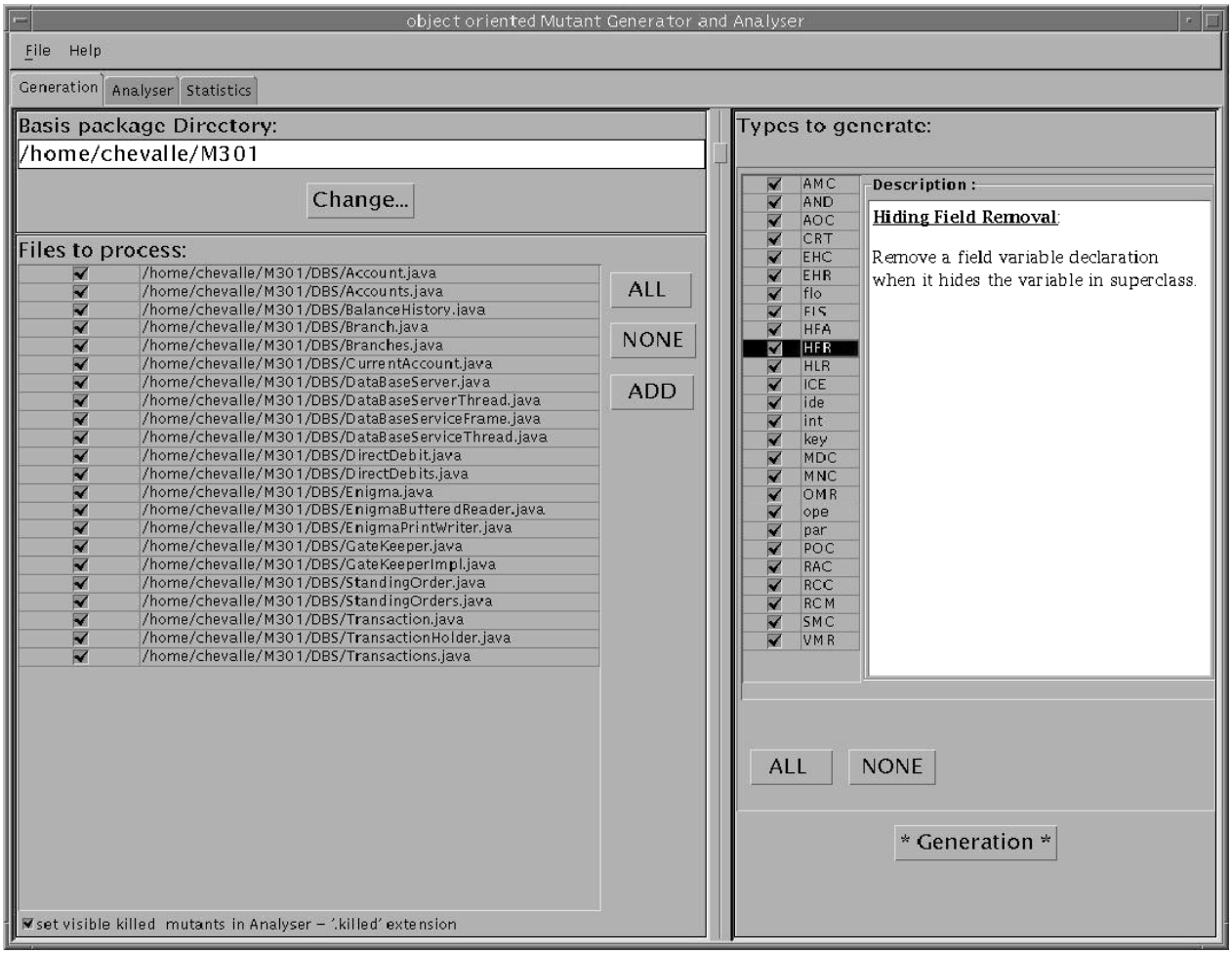

Fig. 1. Graphical interface for generating mutants

operators are JAVA adaptations of previous mutation operator definitions for other languages (see Sect. 4).
– *Object-oriented mutations*: twenty operators are responsible for generating mutations related to OO features introduced in JAVA. The twenty mutation operators have been issued from an extension of the *Class Mutation* system [10]. They deal with several OO features: e.g., inheritance, polymorphism, information hiding, and method overloading (see Sect. 5).

Mutation analysis systems generally create large numbers of mutant programs. The most obvious way to create a mutant program is to create a copy of the original program source code and change it by inserting a mutation. For many programming languages, this approach appears to be too computationally and spatially expensive, because the mutation analysis experiments would require storing and compiling thousands of mutated programs. This problem suggest the use of incremental compilation [12]. Despite its theoretical attractiveness, incremental compilation has not been used in the Mothra environment because the technology was undeveloped and there were few off-the-shelf incremental compilers available at the time. The JAVA development environment is recent and integrates advanced principles. It relies on two main tools: the JAVA compiler and the JAVA Virtual Machine (JVM). The JAVA technology behaves differently than procedural languages in two ways, which are important in the case of achieving program mutation. First, a JAVA program is usually distributed among several files containing (small) classes. The JAVA compiler is able to process each file separately. Second, the JAVA compiler does not produce a unique file containing a native binary code but several files containing the JAVA *bytecode* to be interpreted by the JVM. Although it can be argued that this approach is still expensive (even in the case of JAVA), we decided to create a file per mutation for the initial version of the tool. Another convenient solution would be to create a database containing the description of every mutation and then to insert the mutations, one by one, in the original program before executing them against the test cases. This solution has been integrated in SESAME [20] successfully for non-OO languages.

The mutants produced by JAVAMUT are stored in files the names of which are made of three elements: the name of the original file, a generation number related to the original file processed, and the type of mutation generated. These files are placed in a *buggyVersions* directory especially created in the current directory of the original file. For instance, the mutated file `Aircraft-1-ide.java` is a copy of the original file `Aircraft.java` containing an *identifier replacement (ide)* mutation. The number `1` means that it is the first mutation generated for the considered original file. According to this name convention, the tool automatically handles the mutated files to compile and execute.

4 Traditional mutation operators

Based on traditional mutation faults, a set of six mutation operators has been defined for the JAVA language. They are named using a three-letter acronym, written in lowercase to be easily differentiated from object-oriented mutation operators, which are written in uppercase. The six mutation operators are: *ide* for identifier replacement, *ope* for operator replacement, *key* for keyword replacement, *int* for integer constant value alteration, *flo* for floating-point constant value alteration, and *par* for precedence operation modification by deleting parentheses.

4.1 Description

The *ide* (identifier replacement) mutation operator generates faults representing an incorrect use of identifiers referring any structure: class, attribute, method, and variable. If `var1` and `var2` are identifiers in the original program, the operator defines the syntactic modification rule for replacing `var1` with `var2`, and reciprocally (Table 1). Such mutations are specific to the program under test. It is indeed required to customize the permutation rules to apply them according to identifiers present in the program.

The *ope* (operator replacement) mutation operator deals with arithmetic, binary, relational, and assignment operators that are present in the original program. It consists of replacing an operator with another compatible operator, or deleting an operator. Table 2 gives a list of compatible operators.

The *key* (keyword replacement) mutation operator deals with keywords of the programming language. It con-

Table 1. Mutation examples on identifiers

Operator	Description
`var1` and `var2`	permutation of identifiers
`on` and `off`	permutation of identifiers
`open` and `close`	permutation of identifiers

Table 2. Mutations on operators

Operator	Description
`*`, `/`, `+`, `-`, and `%`	permutation of arithmetic operators
`++` and `--`	permutation of increment and decrement operators
`^`, `&&`, `&`, `||`, and `|`	permutation of binary or Boolean operators
`==`, `!=`, `<=`, `>=`, `>`, `<`, and `=`	permutation of relational operators
`!` and `~`	permutation and deletion of negation operators
`>>` and `<<`	permutation of shift operators

Table 3. Mutation examples on keywords

Operator	Description
`this` and `super`	permutation of keywords
`true` and `false`	permutation of keywords
`transient` and `volatile`	permutation of keywords
`break`	deletion of keyword
`continue`	deletion of keyword
`synchronized`	deletion of keyword

Table 4. Mutations on integer and floating-point constant values

Operator	Description
`<<1`	one bit shifted to the left
`>>1`	one bit shifted to the right
`+1`	plus 1
`-1`	minus 1
`^(1<<n)`	bit n complemented
`~`	value replaced with its 1-complement

sists of replacing a keyword with another compatible keyword, or deleting a keyword (Table 3).

To change a constant value, the *int* (integer constant value alteration) and *flo* (floating-point constant value alteration) mutation operators apply an arithmetic operator to the constant value instead of replacing it with another value. Table 4 illustrates the type of operations to be applied on integer and floating-point constant values.

The *par* (precedence operation modification by deleting parentheses) mutation operator consists of changing the evaluation order of an expression by deleting parentheses. For instance, the expression $a-(b+c)$ can be altered by deleting parentheses to obtain the mutated expression $a-b+c$, which has a different meaning than the original one.

4.2 Implementation: a syntactic analysis approach

These six mutation operators have been implemented using syntactic analysis. For each Java file of the application, the tool creates an abstract syntax tree and walks through to generate mutant programs. JavaMut relies on the Java compiler to verify that the mutant program is syntactically and semantically correct. In considering compiler optimization, the tool also verifies that the bytecode of the mutant program is different from the original bytecode. The mutants that may be of interest are those that satisfy these two criteria: (i) the created mutant program must be compiled successfully; and then (ii) the generated bytecode must be different from the original bytecode. The tool stores only the mutants that fulfill both conditions.

5 Object-oriented mutation operators

Class Mutation [10, 11] defines a mutation system made of the 15 object-oriented mutation operators for Java listed in Table 5. For example, the AMC (Access Modifier Change) operator causes each access control modifier in a program to be replaced by each other modifier. Java defines four 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.

Table 5. 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

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. Section 5.1 presents the five new mutation operators. For details on the 15 Class Mutation operators, the reader can refer to [10, 11].

5.1 Description

Our five mutation operators are also named with the three-letter acronym convention, in uppercase: MDC for Method Definition Change, MNC for Method Name Change, RAC for Reference Assignment and Content assignment replacement, RCC for Reference Comparison and Content comparison replacement, and RCM for Remove Compatible Method. They are described hereafter.

5.1.1 MDC Mutation operator

The MDC (Method Definition Change) operator permutes the definition of two methods with the same signatures. It represents a fault due to a method definition

inversion in a class. This type of faults cannot be detected at compile-time when formal parameters and return types are compatible. A class generally contains several compatible methods. For instance, accessor methods designed to return the value of private attributes are often compatible methods when attributes are themselves compatibles. In the example below, the MDC operator permutes both compatible method definitions.

Original Code:

```
1  public class Point {
2      public int getX() { return x; }
3      public int getY() { return y; }
4      ... }
```

MDC Mutant:

```
1  public class Point {
2      public int getX() { return y; }
3      public int getY() { return x; }
4      ... }
```

5.1.2 MNC Mutation operator

The MNC (Method Name Change) mutation operator defines the rules to change a method call with another compatible method call. It represents the programmer using an incorrect method name instead of the specified one. Since a class may have several methods with different names but accepting the same signature (i.e., accepting the same number and type of formal parameters, and returning the same type), the programmer may use a given 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. The MNC operator deals with method calls whereas the MDC operator concerns method definitions.

Original Code:

```
1  Point point = new Point();
2  int x = point.getX();
```

MNC Mutant:

```
1  Point point = new Point();
2  int x = point.getY();
```

The class Point presented in the example above declares two methods (`getX` and `getY`) accepting the same signature. These two methods can be called on an instance (i.e., an object) of the class Point. If the original source 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 `point.getY()`.

5.1.3 RAC Mutation operator

The RAC (Reference Assignment and Content assignment replacement) mutation operator aims at modifying 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.

Original Code:

```
1  Point point1, point2;
2  point1 = new Point();
3  point2 = point1;
```

RAC Mutant:

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

In the example above, the RAC operator 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.

5.1.4 RCC Mutation operator

In the same way, the RCC (Reference Comparison and Content comparison replacement) mutation 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));
```

The example presented above shows two `Point` objects initialized with the same coordinates. In the original code, the Boolean variable `b` is *false* since the variables `point1` and `point2` reference two different objects. The RCC mutation 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 verifies whether objects are in identical states. In the mutated code, the Boolean variable `b` will contain the value *true*.

5.1.5 RCM Mutation operator

The RCM (Remove Compatible Method) mutation operator is based on the concept of implicit conversion between compatible primitive types. The JAVA language specification defines implicit conversion between the seven following primitive types: *char*, *byte*, *short*, *int*, *long*, *float*, and *double*. This list order gives the conversion order: for instance, it is possible to assign a *byte* value to a variable declared as *double*, but the reverse is not allowed. More specifically, JAVA also uses a kind of implicit conversion to consider a table as an object.

Original Code:

```
1  public class Printer {
2      public void print(int x) { ... }
3      public void print(float x) { ... }
4      ... }
```

RCM Mutant:

```
1  public class Printer {
2      // method deleted
3      public void print(float x) { ... }
4      ... }
```

In the example above, the method designed for printing an integer value may be deleted without causing any compiler error. The printing of an integer value is then achieved using the other method: the integer value is implicitly converted to a floating-point value before calling the appropriate `print` method.

5.2 Implementation: a reflective approach

Object-oriented mutation operators require to access information collected in a program from an object-oriented standpoint. For instance, the CRT operator, which replaces a reference type with compatible types, needs to run through inheritance relationships to create mutations. Using a classical abstract syntax tree is not convenient for accessing OO concepts present in a program. An original approach consists of combining reflection and syntactic analysis. The implementation of object-oriented specific mutation operators is based on a reflective macro system called OPENJAVA.

5.2.1 Using a reflective macro system

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 [19] is a compile-time reflective system. A major idea of OPENJAVA is that macros (meta programs) deal with class metaobject representing logical entities of a program instead of a sequence of tokens or abstract syntax trees (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 other reflective languages using the information at runtime (such as METAXA [7] and KAVA [22]).

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 Abstract Syntax Tree (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.

As an example, Figs. 2 and 3 show, respectively, a base-level program and a meta-level program. The programmer defines a class `Hello` instance of the metaclass `Verbose`. To indicate which metaclass is used for manag-

```
public class Hello instantiates Verbose {
    public static void main(String[] args) {
        hello();
    }
    static void hello() {
        System.out.println("Hello, world.");
    }
}
```

Fig. 2. Base-level program example

```
import openjava.mop.*;
import openjava.ptree.*;
public class Verbose extends OJClass {
   public void translateDefinition() {
      OJMethod[] m = getDeclaredMethods();
      for(int i=0; i<m.length; i++) {
          Statement s = makeStatement(
              "System.out.println(\"" +
              m[i] + " is called.\");");
          m[i].getBody().insertElementAt(s, 0);
      }
   }
}
```

Fig. 3. Meta-level program example

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("main is called.")
        hello();
    }
    static void hello() {
        System.out.println("hello is called.");
        System.out.println("Hello, world.");
    }
}
```

Fig. 4. Transformed base-level program

ing a metaobject, the clause `instantiates` must indeed be added in class declarations. The source code

```
public class Hello instantiates Verbose {...}
```

specifies that the metaobject for `Hello` is an instance of `Verbose`. In this case, `Verbose` represents the metaclass that is responsible for customizing the metaobject defining class `Hello`. Like every metaclass, `Verbose` inherits from the metaclass `OJClass`, which is a built-in class of OpenJava. The class `OJClass` contains macro expansions that can be redefined in `Verbose`. For instance, OpenJava invokes the macro expansion `translateDefinition` for each class declaration parsed in the source code. The metaclass defines transformations that consist of creating a trace. After the translation, the meta-information is merged with the `Hello` class definition (Fig. 4).

5.2.2 Implementation details

Since the purpose of a metaclass is to customize metaobjects representing class definitions and utilizations, the skeleton for generating object-oriented mutations is placed in a metaclass called `MG` (for Mutation Generator metaclass) in which methods redefine macro expansions for generating mutant programs related to class declarations.

Our tool, centered on the `MG` metaclass, was developed so that it can easily be extended with further mutation operators. The UML (Unified Modeling Language [2]) class diagram presented in Fig. 5 shows

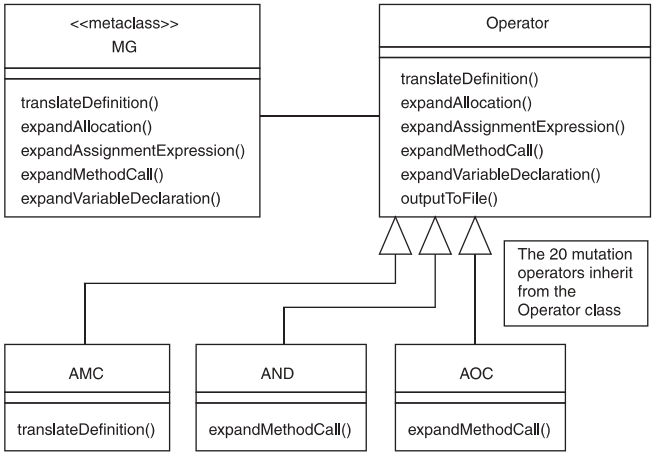

Fig. 5. Partial UML class diagram of the tool

that each mutation operator is implemented in an appropriate class, which redefines macro expansion according to its needs. For readability purpose, Fig. 5 presents only three classes (`AMC`, `AND`, and `AOC`) of the 20 classes implementing object-oriented mutation operators. `MG` extends the OpenJava built-in metaclass `OJClass`, which is not represented in Fig. 5, and redefines macro expansions that will be used for generating mutations.

The `MG` metaclass redefines every macro expansion declared in `OJClass`. When OpenJava calls any macro expansion (say, `expandVariableDeclaration`), `MG` distributes the call to the corresponding method in `Operator` (see Fig. 6). That method may be redefined in each subclass of `Operator`.

As an example, Fig. 7 presents the class `CRT`, which is a mutation operator that targets polymorphism. It changes the type of a (reference) variable with a compatible type coming from inheritance relationships (i.e., a superclass or an implemented interface). The `expandVariableDeclaration` method of class `CRT` is responsible for generating mutant programs according to that mutation operator. The principle is first to create a copy of the considered variable declaration. For each compatible type, a mutant is created: the `outputToFile` method writes the mutation by replacing the original declaration (referenced by `d`) with the mutated declaration (referenced by `mutant`).

```
1 public class MG extends OJClass {
2     Operator[] mutants = {new AMC(), ..., new VMR()};
3     public ClassDeclaration expandVariableDeclaration(Environment e, ClassDeclaration d) {
4         for(int i=0; i < mutants.length; i++)
5             mutants[i].expandVariableDeclaration(e, d);
6         return d;
7     }
8     ...
9 }
```

Fig. 6. MG metaclass sample for `expandVariableDeclaration` macro expansion

```
 1 public class CRT extends Operator {
 2     public Statement expandVariableDeclaration (Environment e, VariableDeclaration d) {
 3         OJClass clazz = OJClass.forName(d.getTypeSpecifier().getName());
 4         VariableDeclaration mutant = (VariableDeclaration) d.makeRecursiveCopy();
 5         while((clazz = clazz.getSuperclass()) != null) {
 6             mutant.setTypeSpecifier(TypeName.forOJClass(clazz));
 7             outputToFile(d, mutant);
 8         }
 9         return d;
10     }
11     ...
12 }
```

Fig. 7. Code extracted from the CRT mutation operator implementation

6 Tool's GUI for mutation analysis

To elaborate a mutation score, mutants equivalent to the original program must be identified and removed from the set of mutants M^P. Although Offutt et al. [17] have demonstrated that some equivalent mutants can be detected automatically, it remains an activity particularly difficult that requires a human intervention (see Sect. 2.3). If this step will never be fully automated, some aid can, however, be considered to make the tester's life easier. JavaMut provides to the tester a graphical interface that helps quickly to compare the mutated code with the original one (see Fig. 8). By selecting a mutant, the tester can immediately visualize the original code on the left side of the interface and the mutated code on the right side. The instruction lines affected by the mutation are displayed in red color. The mutation is moreover underlined. The graphical aid avoids to the tester the

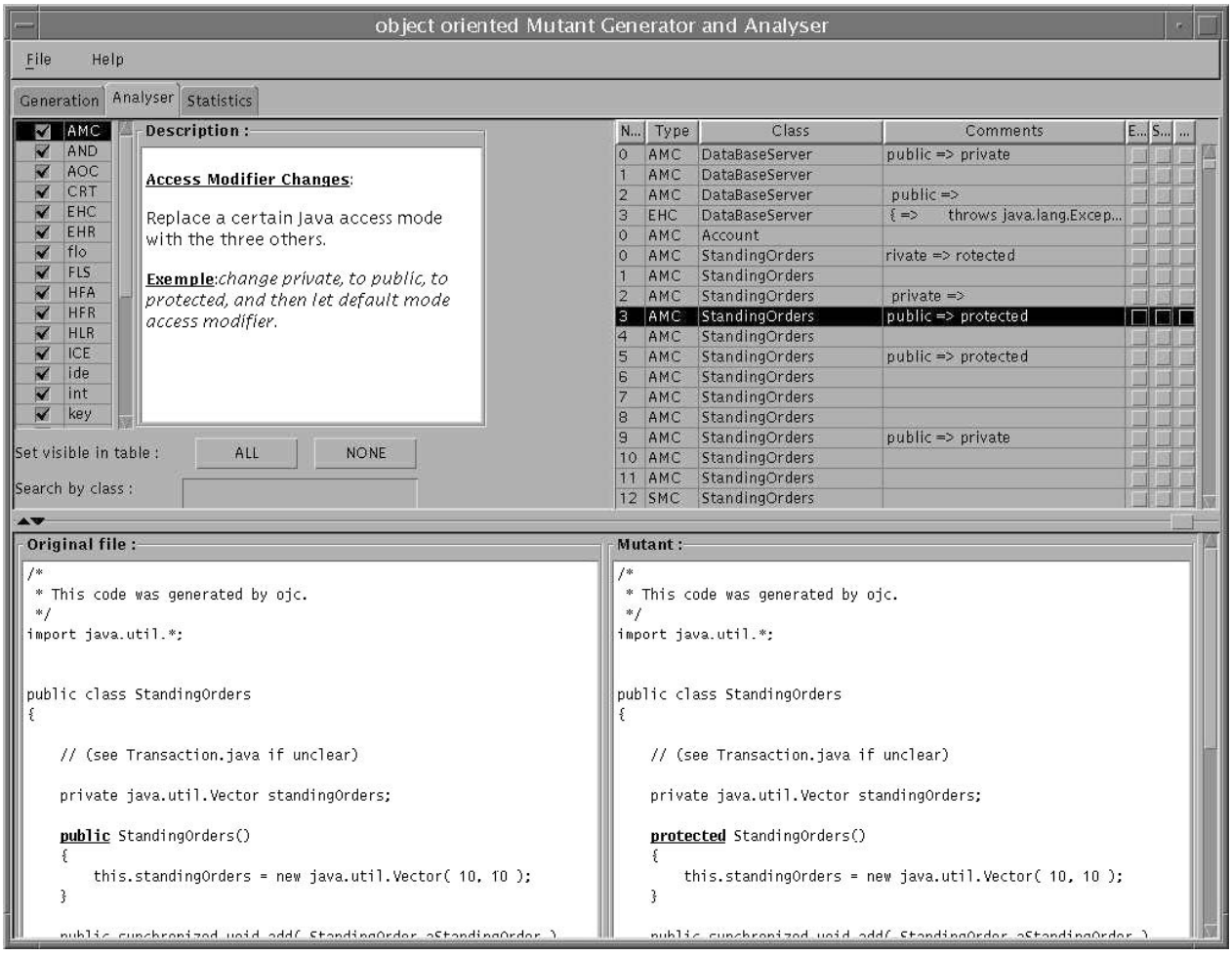

Fig. 8. Graphical interface for analyzing mutations

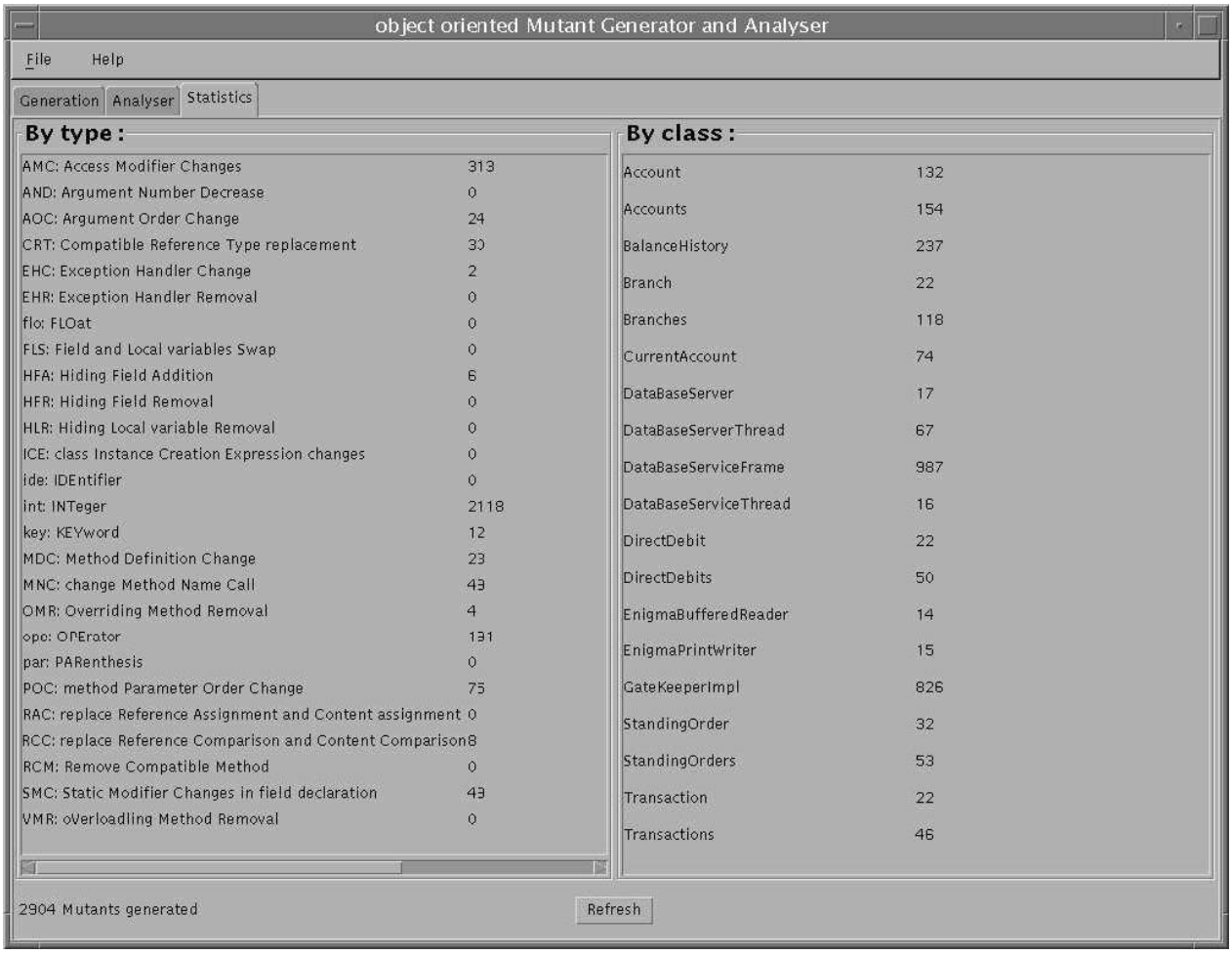

Fig. 9. Graphical interface for visualizing statistical details

cumbersome work of opening many files in an editor and searching in the code for the mutation. This interface is the result of our experience on achieving mutation analysis on a mid-size case study for evaluating a statistical testing strategy [3]. It helped us to reduce drastically the cost devoted to identifying equivalent mutants.

The third main part of the interface concerns the visualization of statistics data. It represents an aid for the tester to access statistics details automatically computed. Currently, the interface displays the number of generated mutants, distributed by class and mutation operator (Fig. 9). This part is still under development. In the short term, additional statistics data (such as the mutation score) will be automatically computed and provided at the tester's disposal via the graphical interface.

7 Empirical evaluation

JavaMut has been developed within the framework of our research dealing with software testing of object-oriented programs. We used it on UNIX workstations for the analysis of two mid-size case studies related to different domains: an avionics application and a banking application. Section 7.1 discusses the mutant generation profile for both applications. Section 7.2 gives details and results on the FGS case study for which mutation analysis has fully been achieved. Concerning the banking application, test cases are not available today.

7.1 Mutant generation profile

The first application is a research version of an avionics system developed at the Advanced Technology Center (ATC) of Rockwell-Collins, Inc. It focuses on the mode logic of a Flight Guidance System (FGS) for a General Aviation class aircraft. The Java implementation developed from the SCR (Software Cost Reduction) specification of this industrial example [14] is made of 115 classes of about 6500 LOC. The FGS case study has originally been developed to explore the feasibility of using formal notations to specify problem typical of those found in avionics. It has also been used as support of our theoretical work on automated generation of statistical test cases from UML state diagrams [3].

The second application concerns the client code of a (virtual) home-banking system (M301) [18]: it consists of 123 JAVA classes of about 6200 LOC. The M301 case study is limited to the application that is used by the home users of the home-banking system. The JAVA application contains a graphical interface to interact with the users and a communication mechanism to receive and send messages from/to the remote banking system.

The FGS and M301 case studies represent applications developed from different guidelines and integrating different object-oriented features. Their close size (of about 6500 LOC) makes easier the comparison of the mutant generation profile for both applications.

Figures 10 and 11 give details on the number of mutants created according to mutation operators for both case studies. Due to the architecture of the FGS application, several OO mutation operators did not produce mutants. It is indeed directly linked to the few object-oriented features used in the application. The JAVA program contains only a few inheritance relationships whereas many OO operators deal with that feature more or less. Moreover, there is no exception handler in the code and objects are not cloneable since classes of the application do not implement the JAVA `cloneable` interface.

Although applications are similar in size, the total number of mutants created is different: 6440 mutants (with 1795 traditional and 4645 object-oriented mutations) for the FGS case study, and 19 849 mutants (with 17 029 traditional and 2820 object-oriented mutations) for the M301 case study. Looking at the number of MNC mutants, we can argue that the FGS implementation contains a lot of methods that are compatible, which is not true for the M301 implementation. This latter implementation contains, however, a high number of integer constant values since the *int* mutation operator generated more than 16 000 mutants. In walking through the code, we found that constant values are used for two main reasons: (i) to set manually the layout of many components in the graphical interface; and (ii) to identify communication ports. Note that, through this couple of applications, we did not cover the whole set of object-oriented features targeted by the mutation operators implemented in JAVAMUT. Other case studies will be conducted to investigate the whole set of operators.

7.2 Applying mutation analysis to the FGS case study

The FGS case study was used as an experimental support to study the feasibility of a test case generation approach [3]. The first lessons learnt for the mutation analysis, now fully achieved on this case study, are given below.

Table 6 presents mutation analysis results involving a set of 3402 test cases executed against the 6440 mutants created from the FGS application using the tool. The analysis of equivalent mutants allowed us to get feedback on the relevance of some mutation operators, in

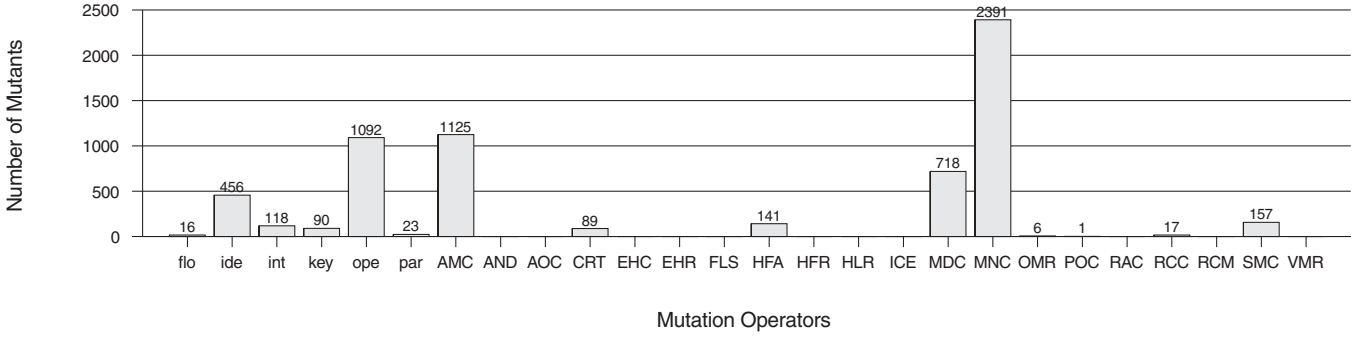

Fig. 10. Mutant generation profile for the FGS case study

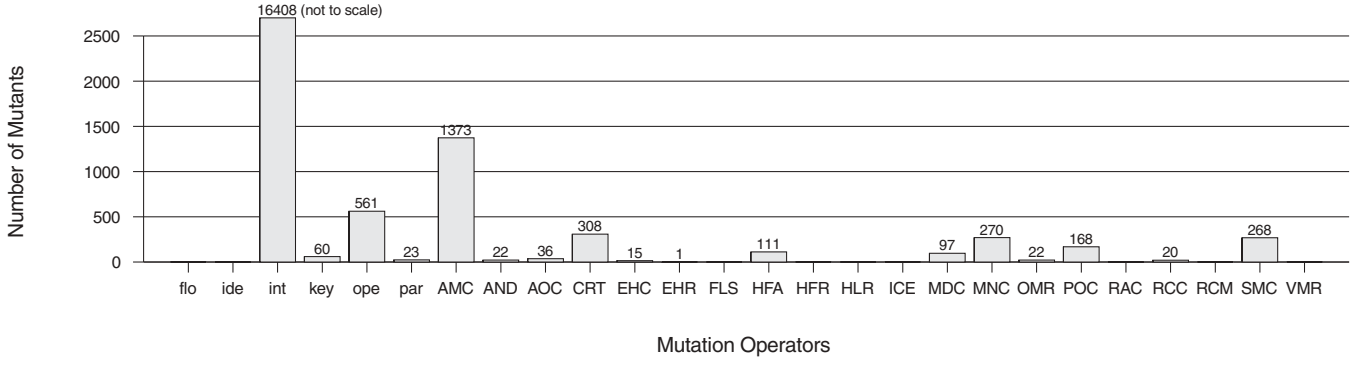

Fig. 11. Mutant generation profile for the M301 case study

Table 6. Mutation analysis results by operator

Mutation	Mutant			Killed
Operator	Created	Non-Equivalent	Killed	Rate
flo	16	16	16	100%
ide	456	393	343	87.28%
int	118	77	77	100%
key	90	90	77	85.56%
ope	1092	963	890	92.42%
par	23	20	20	100%
AMC	1125	0	–	–
CRT	89	87	81	93.10%
HFA	141	49	49	100%
MDC	718	478	449	93.93%
MNC	2391	1936	1569	81.04%
OMR	6	6	6	100%
POC	1	1	1	100%
RCC	17	17	17	100%
SMC	157	17	17	100%
Total	**6440**	**4150**	**3612**	**87.04%**

particular for AMC and SMC operators, which produced a lot of equivalent mutants. First, the FGS application does not contain any method overloading feature that could be involved in the production of non-equivalent AMC mutants. Second, the SMC mutation operator has also produced a lot of equivalent mutants because most of classes in the FGS application father a single object, which means that declaring an attribute *static* or not has no effect on the application's behavior. Without considering these two mutation operators for this application, we can then argue that the tool is efficient for producing (non-equivalent) mutants. This leads us to suggest avoiding the use of the AMC and SMC operators for applications presenting the same features as the FGS (i.e., no method overloading, and a single object created per class), in order to be more cost-effective. It is worth noting that during the test experiments no mutant failed to terminate. A timer is, however, used to detect that kind of problem.

Although the overall mutation score is high (87.04% in Table 6), 538 mutants remained alive after having completed the test experiments. The analysis of these mutants helped us to find out a weakness of our test strategy: since those mutants were grouped into few classes, it appears that the statistical test strategy does not allow us to sufficiently probe these classes [3]. Improvements are currently under study.

Other mutation analyses are in progress in order to compare the mutation adequacy of our statistical test cases with the one of other test cases developed at the Advanced Technology Center of Rockwell-Collins, Inc. Such experimental evaluations would not have been feasible without the help of an efficient mutation analysis tool for JAVA programs.

8 Conclusions and future work

The paper presents the first prototype GUI-based tool supporting mutation analysis of JAVA programs. The tool implements 26 mutation operators (including 20 object-oriented specific operators) targeting various types of plausible faults in a JAVA program. Two factors have motivated this tool: first, mutation analysis is a powerful and computationally expensive fault-based technique that cannot be considered without the aid of an efficient tool taking an active part in the automation of the technique; second, the JAVA language integrates object-oriented features that can be the basis for new types of faults non-existent in procedural languages.

If syntactic analysis can be used to implement traditional mutation operators, this approach is not convenient for implementing object-oriented operators, which are much more complex than traditional ones. A key and novel idea is to combine syntactic analysis and reflection for implementing both categories of mutation operators. The core of our tool is based on this approach. JAVAMUT has moreover been developed in such a way that it can easily be extended with further mutation operators.

First evaluation results allowed us to draw promising conclusions. Beyond its attractiveness for generating automatically a high number of mutants, the tool provides an easy and intuitive interface for simplifying the tester's (manual and complex) task in the identification of equivalent mutants. This interface helped us to reduce significantly the cost devoted to identifying equivalent mutants on the FGS application. Selective mutation is also a way to consider for reducing the cost of mutation analysis. JAVAMUT allows the tester to easily choose a subset of operators to define a selective mutation system. Further research will be performed through other case studies to classify mutation operators and provide practical hints on how to select sufficient operators.

Acknowledgements. We thank Christophe Moine and Jérôme Chancel for their useful contribution throughout the development of the tool, within the framework of student projects. We also thank very much David E. Statezni of the Advanced Technology Center (ATC) of Rockwell-Collins, Inc. for supporting the project. We would also like to thank the anonymous referees for their useful comments.

References

1. Barbosa, E.F., Maldonado, J.C., Rizzo Vincenzi, A.M.: Toward the determination of sufficient mutant operators for C. Software Test Verification Reliab 11: 113–136, 2001
2. Booch, G., Rumbaugh, J., Jacobson, I.: The unified modeling language user guide. Object Technology Series. Addison-Wesley, Reading, Mass., USA, 1999
3. Chevalley, P., Thévenod-Fosse, P.: An empirical evaluation of statistical testing designed from UML state diagrams: the flight guidance system case study. In: Proc. 12th International Symposium on Software Reliability Engineering (ISSRE 2001), Hong Kong, China, November 2001, pp. 254–263
4. Daran, M., Thévenod-Fosse, P.: Software error analysis: a real case study involving real faults and mutations. In: Proc.

1996 International Symposium on Software Testing and Analysis (ISSTA'96), San Diego, Calif., USA, January 1996, pp. 158–171
5. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: help for the practicing programmer. Computer 11:34–41, April 1978
6. DeMillo, R.A., Martin, R.J., McCracken, W.M., Passafiume, J.S.: Software testing and evaluation. Benjamin-Cummings, Menlo Park, Calif., USA, 1987
7. Golm, M., Kleinöder, J.: Jumping to the meta level: behavioral reflection can be fast and flexible. In: Proc. 2nd International Conference on Metalevel Architectures and Reflection (Reflection'99), Lecture Notes in Computer Science, vol. 1616. Springer, Berlin Heidelberg New York, 1996, pp. 22–39
8. Hierons, R.M., Harman, M., Danicic, S.: Using program slicing to assist in the detection of equivalent mutants. Software Test Verification Reliab 9: 233–262, 1999
9. How Tai Wah, K.S.: A theoretical study of fault coupling. Software Test Verification Reliab 10: 3–45, 2000
10. Kim, S., Clark, J.A., McDermid, J.A.: Class mutation: mutation testing for object-oriented programs. In: Proc. Net.ObjectDays Conference on Object-Oriented Software Systems, Erfurt, Germany, 2000
11. Kim, S., Clark, J.A., McDermid, J.A.: Investigating the effectiveness of object-oriented testing strategies with the mutation method. In: Proc. Mutation 2000: Mutation Testing in the Twentieth and the Twenty-First Centuries, San Jose, Calif., USA, 2000
12. King, K.N., Offutt, A.J.: A Fortran language system for mutation-based software testing. Software Pract Exper 21: 685–718, 1991
13. Laprie , J.-C. (ed): Dependability: basic concepts and terminology, vol. 5. Springer, Berlin Heidelberg New York, 1992
14. Miller , S.P.: Specifying the mode logic of a flight guidance system in CoRE and SCR. In: Proc. 2nd Workshop on Formal Methods in Software Practice (FMSP'98), Clearwater Beach, Fla., USA, 1998
15. Offutt, A.J.: Investigations of the software testing coupling effect. ACM Trans Software Eng Methodol 1: 3–18, 1992
16. Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H., Zapf , C.: An experimental determination of sufficient mutant operators. ACM Trans Software Eng Methodol 5: 99–118, 1996
17. Offutt, A.J., Pan, J.: Automatically detecting equivalent mutants and infeasible paths. Software Test Verification Reliab 7: 165–192, 1997
18. The Open University, Milton Keynes, UK. M301 Case Study. `http://m301.open.ac.uk/html/case_study/`, 2002
19. Tatsubori, M., Chiba, S., Killijian, M.-O., Itano, K.: OpenJava: a class-based macro system for Java. In: Reflection and Software Engineering, Lecture Notes in Computer Science, vol. 1826. Springer, Berlin Heidelberg New York, June 2000, pp. 117–133
20. Thévenod-Fosse, P., Waeselynck, H.: Software statistical testing based on structural and functional criteria. In: Proc. 11th International Software Quality Week (QW'98), San Francisco, Calif., USA, May 1998
21. Thévenod-Fosse, P., Waeselynck, H., Crouzet, Y.: Software statistical testing. In: Randell, B., Laprie, J.-C., Kopetz, H., Littlewood , B.(eds) Predictably dependable computing systems. Springer, Berlin Heidelberg New York, 1995, pp. 253–272
22. Welch, I., Stroud, R.: From Dalang to Kava – the evolution of a reflective Java extension. In: Proc. 2nd International Conference on Metalevel Architectures and Reflection (Reflection'99), Lecture Notes in Computer Science, vol. 1616. Springer, Berlin Heidelberg New York, 1999, pp. 2–21