Mutation Testing: Object-Oriented Mutation and Testing Tools

Z. Ivanković, B. Markoski, D. Radosav
University of Novi Sad, Technical Faculty "Mihajlo Pupin", Zrenjanin, Serbia
zdravko@tfzr.uns.ac.rs

Contribution to the State of the Art

DOI: **10.7251/JIT1102105I** UDC **004.7:006**

Abstract: Software testing represents activity in detecting software failures. Mutation testing represents a way to test a test. The basic idea of mutation testing is to seed lots of artificial defects into the program, test all defects individually, focus on those mutations that are not detected, and, finally, improve the test suite until it finds all mutations. Mutants can be created by mutating the grammar and then generating strings, or by mutating values during a production. Object-oriented (OO) programming features changed the requirements for mutation testing. Non object-oriented mutation systems make mutations of expressions, variables and statements, but do not mutate type and component declarations. OO programs are composed of user-defined data types (classes) and references to the user-defined types. It is very likely that user-defined components contain many defects such as mutual dependency between members/classes, inconsistencies or conflicts between the components developed by different programmers. Class Mutation is a mutation technique for OO programs which particularly targets plausible faults that are likely to occur due to features in OO programming. Mutation testing requires automated testing tools, which is not a trivial tool to make. Automated mutation tools must be able to parse the program and know its language. When the program is run, mutant can be killed by one of two possible scenarios: if a mutant crashes, or if the mutant goes into an infinite loop.

Key words: Mutation testing, Object-oriented mutation, schema-based mutation, reflection

Introduction

Software testing represents activity in detecting software failures. The scope of software testing often includes examination of code as well as execution of that code in various environments and conditions as well as examining the aspects of code: does it do what it is supposed to do and do what it needs to do. Software testing is faced with several problems. Bugs are not distributed uniformly across a program: "20% of modules contain 80% of the defects". Second problem represents risk, which is unevenly distributed. In every project there are some modules in which defects have serious consequences because they are frequently used, or because entire functionality depends on them. Tester would want his test suite to be focused on the defect-prone modules, and to make his testing efforts based on the risk, rather than achieving a specific coverage.

Many systems are tested according to principle: "if it does not crash, it is probably fine". In this way, test does not check the result, and a tester cannot determine how well a test does its job. This is an instance of Plato's old problem: "Who watches the watchmen". Mutation testing represents a way to test a test. A common way to test the quality of quality assurance is to simulate a situation in which quality assurance should trigger an alarm. In 1971, Richard Lipton adapted this concept to testing. His idea, presented in a paper called "Fault diagnosis of computer programs," was to seed artificial defects, called mutations, into the software under test, and to check whether the test suite would find them. If the test suite fails to detect the mutation, it would likely miss real defects, too, and thus must be improved.

MUTATION TESTING

The basic idea of mutation testing is to:

- seed lots of artificial defects into the program,
- test all defects individually,
- focus on those mutations that are not detected, and,
- improve the test suite until it finds all mutations [1].

This approach has a few benefits. First benefit is that tester can truly assess the quality of tests, not just to measure features of test execution. When modules with high risk are mutated, they can exhibit serious consequences. Third benefit comes with choice of good mutants. The more similar mutations are to real defects, the more likely you are to replicate the defect distribution in your program. Mutation is widely considered the strongest test criterion in terms of finding the most faults, but is also the most expensive.

Mutation testing is very time-consuming and as the number of mutations can easily go into the thousands, there can be several thousand build processes and test suite executions. Because of this, mutation testing requires a fully automated test [7]. There are some techniques which can improve efficiency of seeding mutants. First technique is to directly manipulate binary code. By mutating binary code rather than source code, you can eliminate the costly rebuild process after a mutation. The drawback is that binary code can be harder to analyze, in particular for complicated mutation operators. Second technique is to use mutant schemata. A mutation-testing framework produces a new mutated program version for every single mutation. However, one can also create a single version in which individual mutants are guarded by runtime conditions. Third technique is to ignore no covered code. A mutant can impact the program behavior only if it is actually executed. Therefore, programmer should mutate only statements that are covered by the test suite and run only those tests that exercise the mutation.

MUTATION TESTING GRAMMAR

Mutants can be created by mutating the grammar and then generating strings, or by mutating values during a production. Mutation can be applied to various artifacts, but it is primarily used as a program-based testing method. An input is valid if it is in the language specified by grammar, otherwise it is invalid. Any program should reject malformed inputs, which is a property that should be verified by tests. Testing could be accomplished by producing invalid strings from grammar, or by producing strings that are valid but that follow different derivation from preexisting strings. Both of these strings are called mutants.

Mutation is always based on set of mutation operators which are applied to a “ground” string. The ground string is the sequence of program statements in the program under test, and the mutants are slight syntactic variations of those statements. During program execution, the ground strings are valid inputs, and variations (mutants) are invalid inputs. For example, a valid input might be a request from a correctly logged in user in some application. The invalid version might be the same request from a user that is not logged in.

Mutation operator represents a rule that specifies syntactic variations of strings generated from grammar. Mutant represents result of one application of mutation operator over ground string. There are two issues in applying mutation operators. First is, should more than one mutation operator be applied at the same time to create one mutant? Strong empirical and theoretical evidence point that only one element should be mutated at a time. Second issue is, should every possible application of a mutation operation to a ground string be considered? Reason for this is that mutation subsumes a number of other test criteria, and if some operators are not applied, then that subsumption is lost.

When derivation is mutated to produce valid strings, the testing goal is to “kill” the mutants by causing the mutant to produce different output. Thus, mutation coverage (MC) equates to killing the mutants. The amount of coverage is usually written as percent of mutants killed and is called mutation score.

Definition for MC is: For each mutant m ∈ M, test requirement contains exactly one requirement, to kill m.

When a grammar is mutated to produce invalid strings, the testing goal is to run the mutants to see if the behavior is correct. The coverage criterion is therefore simpler, as the mutation operators are the test requirements. In this manner we have mutation operator coverage (MOC) and mutation production coverage (MPC). Definition for MOC is: For each mutation operator, test requirement contains exactly one requirement, to create a mutated string m that is derived using the mutation operator. Definition for MPC is: For each mutation operator, and each production that the operator can be applied to, test requirement contains the requirement to create a mutated string from that production

The number of test requirements for mutation depends on the syntax as well as the mutation operators. In most situations, mutation yields more test requirements than any other test criterion.

Object-oriented mutation

Object-oriented (OO) programming features changed the requirements for mutation testing. Non object-oriented mutation systems make mutations of expressions, variables and statements, but do not mutate type and component declarations. Traditional programming simply makes use of the built-in types and entities of a language which are unlikely to contain many errors. OO programs are composed of user-defined data types (classes) and references to the user-defined types. It is very likely that user-defined components contain many defects such as mutual dependency between members/classes, inconsistencies or conflicts between the components developed by different programmers.

The effectiveness of mutation testing heavily depends on the types of faults the mutation system is intended to represent. Class Mutation is a mutation technique for OO programs which particularly targets plausible faults that are likely to occur due to features in OO programming [4][8]:

- polymorphism
- method overloading
- inheritance
- information hiding
- static/dynamic states of objects
- exception handling

A. Polymorphism

In object-oriented systems, it is common for a variable to have polymorphic types. Polymorphism represents a property that a variable at runtime may refer to an object of a different type. This raises the possibility that not all objects that become attached to the same variable correctly support the same set of features. This may also cause runtime type errors which cannot always be detected at compile time [3][5]. Two mutation operators, CRT (Compatible Reference Type replacement) and ICE (class Instance Creation Expression changes), were designed to address this feature

CRT operator replaces a reference type with compatible types. For instance, the class type S can be replaced with the class type T provided that S is a subclass of T, or S can be replaced with the interface type K provided that S implements K.

The original code:

```
    S s = new S();
```

CRT mutants:

```
    T s = new S();
    K s = new S();
```

ICE operator is designed to change the runtime type of an object. This results in calling the constructors of compatible types, which will create the objects of the replaced types.

The original code:

```
    S s = new S();
```

ICE mutant:

```
    S s = new T();
```

B. Method Overloading

A class type may have more than one method with the same name as long as they have different signatures. When several versions of the same name method are available, there is a great possibility to an unintended method be called [6]. Method overloading feature can be handled by manipulating parameters in method declarations and arguments in method invocation expressions. Four mutation operators, POC (method Parameter Order Change), VMR (oVerloading Method declaration Removal), AOC (Argument Order Change), and AND (Argument Number Decrease), were designed to address this feature.

The POC operator changes the order of parameters in method declarations if the method has more than one parameter.

The original code:

```
public LogMsg(int level,
          String logKey,
          Object [] inserts) {…}
public LogMsg(int level,
          String logKey,
          Object insert)  {…}
```

POC mutant:

```
public LogMsg(String logKey,
          int level,
          Object []inserts) {…}
```

In the example, the POC operator creates a mutant of the first constructor by swapping the first and second parameters of the constructor. This mutant program is executed without compilation errors in spite of the fact that the types of the swapped parameters are totally different. The reason is that the instance creation expressions that call the first constructor in the original code are directed to the second constructor when the mutant is executed, because the second constructor is now better fitted. It is possible because arrays can be assigned to variables of type Object in Java. This example shows that there is a possibility of invoking a wrong constructor/method among the overloaded constructors/methods due to an unintended parameter type conversion.

VMR operator removes a whole method declaration of overloading/overloaded methods. In this way, tester can check whether the right method is invoked for the right method invocation expressions. The VMR operator can also provide coverage for the method overloading feature, i.e., checking if all the overloading/overloaded methods are invoked at least once – because test data must reference the method in order to notice that that method has been deleted.

AOC operator changes the order of arguments in method invocation expressions, if there is more than one argument. For example, the following mutant produced by the AOC operator represents the error of a wrong argument order and as both arguments have the same type (Java String), the order change in the mutant did not cause compilation problems.

The original code:

```
Trace.entry(“Logger”, “addLogCatalogue”);
```

AOC mutant:

```
Trace.entry (“addLogCatalogue”, “Logger”);
```

AND operator reduces the number of arguments one by one, if there is more than one argument in method invocation expressions. The original code has two different trace methods:

```
public void trace(int level, Object obj, String text)
{…}
public void trace(Object obj, String text) {…}
```

The original code:

```
Trace.trace(Trace.Event,this,sccsid);
```

AND mutant:

```
Trace.trace(this,sccsid);
```

Although the mutant has two arguments instead of three, it is successfully compiled because class Trace has two different trace methods. The original code calls the first method while the mutant calls the second method.

C. Inheritance

A class type may contain a method with the same name and the same signatures as the method declared in superclasses or superinterfaces. In this case, the method in a subclass overrides the method of a superclass (method overriding/hiding). When there is more than one method of the same name, it is important for testers to ensure that a method invocation expression actually invokes the intended method [2]. OMR (Overriding Method Removal) operator is designed to check that overriding/overridden methods are invoked appropriately.

OMR operator removes a declaration of an overriding/hiding method in a subclass so that a reference to the overriding method goes to the overridden/hidden method instead. If a test set fails to see any difference whether the overriding method is called or the overridden method is called, it implies that the current test set is inadequate. The OMR operator also checks coverage for the method overriding feature – i.e., overriding and overridden methods are invoked at least once.

A Java class may have two or more fields with the same simple name if they are declared in different interfaces and/or in classes. In this case, the field variables defined in a class hide the fields of the same name declared in superclasses or superinterfaces. While this feature is powerful and convenient, it might cause an unintended field being accessed, especially in a long and complex class hierarchy. The intent of the HFR (Hiding Field variable Removal) and HFA (Hiding Field variable Addition) operators is to check that hiding and hidden fields are accessed appropriately

HFR operator removes a declaration of a hiding field variable so the references to that field actually access the field in a superclass or a superinterface. This operator ensures that a test set is able to distinguish referencing a hidden field from referencing a hiding field. If a test set produces the same output even if a hiding field is removed, it indicates the test set is inadequate.

HFA operator adds field variables that appear in superclasses/ superinterfaces into the class under mutation so that the added fields hide those in superclasses/ superinterfaces.

Both the HFR and HFA operators check the coverage of field variables in the presence of inheritance because test data must accesses the hiding/hidden and inherited fields at least once. The difference is that the HFA operator checks that inherited fields are accessed at least once whereas the HFR operator checks that hiding/hidden fields are accessed.

D. Information Hiding

Object-oriented languages provide an access control mechanism that restricts the accessibility/visibility of attribute variables and methods. It is an important testing role to make sure that a certain access mode provides and restricts its intended accessibility/visibility at all times. The intended access control can also be broken in connection with other OO features such as inheritance. Java provides four possible access modes: public, private, protected, and default.

AMC (Access Modifier Changes) operator manipulates Java access specifiers to address the information hiding feature. This operator replaces a certain Java access mode with three other alternatives. The role of the AMC operator is to guide testers to generate enough test cases for testing accessibility/visibility. For example, a field declaration with a protected access mode will have three mutants.

```
The original code:
    protected Address address;

AMC Mutants:
    public Address address;
    private Address address;
    Address address; //default
```

E. Static/Dynamic States of Objects

Java has two kinds of variables – class and instance variables. The Java runtime system creates one copy of each instance variable whenever an instance of a class is created (dynamic). Class variables are allocated once per class, the first time it encounters the class (static).

SMC (Static Modifier Changes) operator is used to examine possible flaws in static/dynamic states. The SMC operator removes the "static" modifier to change a class variable to an instance variable or adds the modifier to change an instance variable to a class variable.

The original code:

```
public static int VALUE = 100;
private String s;
```

SMC Mutants:

```
public int VALUE = 100; //static is removed
private static String s; //static is added
```

F. Exception Handling

The most obvious mistake in exception handling is not specifying appropriate exception handlers in the required place. In Java, programmer either handles an exception (i.e., catches the exception by declaring a try catches block) or propagates it (i.e., declares it to throw in a throws statement of a method declaration). EHR (Exception Handler Removal) and EHC (Exception Handling Change) operators are declared for the feature of exception handling.

EHR operator modifies the declared exception handling statement (try-catch-finally) in two different ways.

- it removes exception handlers (catch clause) one by one when there is more than one handler
- it removes the exception handler and finally clause in turn when there exist one handler and the finally clause

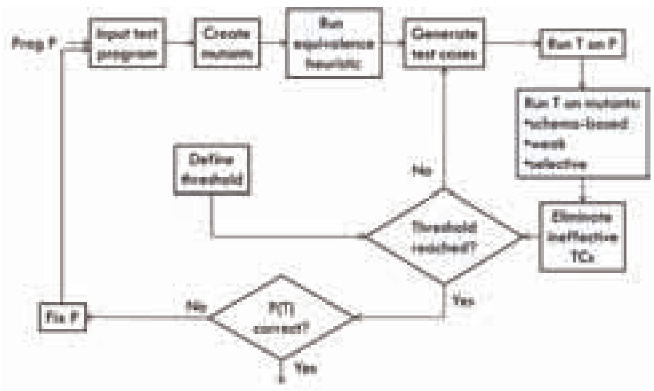
The EHR operator is not applied when there is only one handler without finally clause because it simply causes compilation errors. This operator gives coverage of catch and finally clauses.

EHC operator swaps the way of handling an exception. It catches the exception that is supposed to be propagated by changing a throws declaration to a try-catch statement, or propagates the exception that is supposed be caught within the method by changing a try-catch statement to a throws declaration.

TESTING PROGRAMS WITH MUTATION

Procedure of testing programs with mutation is shown in figure 1.

FIGURE 1 - TESTING PROGRAMS WITH MUTATION



The tester submits the program which should be tested. Automated system starts changing original statements by creating mutants. Optionally, those mutants are then analyzed by a heuristic that detects and eliminates as many equivalent mutants as possible. A set of test cases is then generated automatically and executed against the original program, and then against the program that contains mutants [9]. If the output of a mutant program differs from the original (correct) output, the mutant is marked as being dead and is considered to have been strongly killed by that test case. Dead mutants are not executed against subsequent test cases. Test cases that do not strongly kill at least one mutant are considered to be "ineffective" and eliminated. Once all test cases have been executed, coverage is computed as a mutation score (ratio of dead mutants over the total number of non-equivalent mutants). Mutation score of 1.00 means that all mutants have been detected.

A mutation score of 1.00 is usually impractical, so the tester defines a "threshold" value, which is a minimum acceptable mutation score. If the threshold has not been reached, then the process is repeated, each time generating test cases to target live mutants, until the threshold mutation score is reached. Up to this point, the process has been entirely automatic. To finish testing, the tester will examine expected output of the effective test cases, and fix the program if any faults are found. This leads to the fundamental premise of mutation testing: In practice, if the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects that fault.

Mutation Testing Tools

Mutation testing requires automated testing tools, which is not a trivial tool to make. Automated mutation tools must be able to parse the program and know its language. When the program is run, mutant can be killed by one of two possible scenarios:

- if a mutant crashes
- if the mutant goes into an infinite loop

The runtime system must handle both of these situations.

There are four ways to build mutation tools:

- interpretation approach
- separate compilation approach
- schema-based mutation
- reflection

A. Interpretation approach

A program under test is first parsed into an intermediate form. This is usually a special-purpose language designed specifically to support mutation. This language can easily handle the bookkeeping when mutants are killed as well as program failure. The usual way to handle infinite loops is first to run a test case on the original program, count the number of intermediate instructions executed, then run the test case on a mutant. If the mutant uses X times more intermediate instructions (X has usually been set at 10), then the mutant is assumed to be in an infinite loop and marked dead. The mutation testing tool directly modifies this intermediate form which represents a special purpose language.

This approach has several advantages:

- can easily handle the bookkeeping when mutants are killed
- can respond to program failure and infinite loops
- full control of the execution
- parsing the program and creating mutants is efficient
- creating mutants by making small changes to the intermediate form is simple
- only the rules for changing the intermediate form need to be stored on disk

Disadvantages of this approach are:

- mutation system must be a complete language system: parser, interpreter, and run-time execution engine
- complicated to implement and represents a significant investment
- slow execution (10 times slower than a compiled program). Researchers have found that it can take up to 30 minutes to run all mutants on a 30 line program.

B. Separate compilation approach

In this approach each mutant is created as a complete program by modifying the source of the original program which is under test. Then each mutant is compiled, linked and run.

Main advantage of this approach is fast execution.

However, there are several disadvantages:

- difficulties with bookkeeping when mutants are killed
- difficulties with handling run-time failures and infinite loops
- compilation bottleneck, particularly with large programs, but also with small programs that run very quickly, because the time to compile and link can be much greater than the time to execute
- difficulties with applying weak mutation

C. Schema-based approach

Schema-based approach consists of following steps:

- MSG (Mutant Schema Generation) encodes all mutations into one source-level program, called a metamutant
- metamutant is compiled and executed in the same environment at compiled program speed

These mutation systems are less complex and easier to build then interpretive systems because they do not need to provide the entire run-time semantics and environment. A mutant schema has two components, a metamutant and a metamethod set, both of which are represented by syntactically valid constructs.

In MSG, a program schema represents a template. A partially interpreted program schema syntactically resembles a program, but contains free identifiers that are called abstract entities. The abstract entities appear in place of some program variables, data type identifiers, constants, and program statements. A schema can be instantiated to form a complete program by providing appropriate substitutions for the abstract entities.

D. Reflection

Reflection represents an approach that combines the interpretive and compiler-based approach. Reflection allows a program to access its internal structure and behavior, and manipulate that structure, thereby modifying its behavior based on rules supplied by another program.

Reflection is possible only in languages that support it (Java and C#). Both support reflection by allowing access to the intermediate form (Java bytecode). Reflection can be achieved in three ways:

- Compile-time reflection allows changes to be made when the program is compiled
- Load-time reflection allows changes to be made when the program is loaded into the execution system (JVM)
- Run-time reflection allows changes to be made when the program is executed

Reflection has several advantages:

- it allows programmers extract information about a class
- it provides an API to modify the behavior of a program during execution
- it allows objects to be instantiated and methods to be invoked dynamically
- some of the OO operators cannot be implemented via MSG

Conclusion

First paper about mutation testing was published 30 years ago. Only now mutation testing becomes widely implemented. The reason for this is that automated testing is much more widespread than it was 10 years ago, and there is no mutation testing without it. Computing power keeps on increasing, and we can begin to afford the huge computing requirements imposed by mutation testing. Modern test case generators make it fairly easy to obtain a high coverage automatically but still, the test cases are not good enough. There is a variety of dynamic and static optimizations that make mutation testing reasonably efficient and also highly effective when it comes to improving test suites. All this implies that mutation testing will become much more commonplace in the future.

References

[1] Ammann P. and Offutt J., (2008) "Introduction to Software Testing", Cambridge University Press

[2] Brahma S. Punganti A., Pattanaik P.K., Prasad S. and Mall R., (2010) "Model-Based Mutation Testing of Object-Oriented Programs", Proceedings of 2nd international Conference on IT & Business Intelligence, India

[3] Finkbine R., (2003) "Usage of Mutation Testing as a Measure of Test Suite Robustness", Digital Avionics Systems Conference

[4] Ma Y.S., Harrold M.J. and Kwon Y.R., (2006) "Evaluation of Mutation Testing for Object-Oriented Programs", 28th International Conference on Software Engineering, China

[5] Ma Y.S., Offutt J., (2005) "Description of Class Mutation Operators for Java"

[6] Ma Y.S., Offutt J., (2005) "Description of Method-level Mutation Operators for Java"

[7] Riley T. and Goucher A., (2009) "Beautiful Testing – Leading Professionals Reveal How They Improve Software", O'Reilly

[8] Sunwoo K., Clark J. , McDermid J., (2000) "Class Mutation: Mutation Testing for Object-Oriented Programs", OOSS: Object-Oriented Software Systems

[9] Umar M., (2006) "An Evaluation of Mutation Operators for Equivalent Mutants", Department of Computer Science King's College, London

Submitted: November 07, 2011
Accepted: December 31, 2011