ESTIMATION OF MUTATION TESTING ROBUST IN DATA MINING

Naveen Kumar\(^1\), Sanjay Kumar Yadav\(^2\)

\(^{1,2}\)Department of Computer Science and Information Technology
Sam Higginbottom Institute of Agriculture, Technology & Science Allahabad, U.P., India

ABSTRACT

Testing refers to the inspection of software whether it fulfills the desire objectives or not by means of examines the presence of any error/bugs in a program. This paper presents results from experimental studies of object-oriented, class level mutation operators. Class mutation operators modify object oriented programming language features such as inheritance, polymorphism, dynamic binding and encapsulation. Several new class-level mutation operators are defined in this paper and an analysis of the number of mutants generated is provided. Techniques for eliminating some equivalent mutants are described and data from an automated tool are provided. One important result is that class-level mutation operators yield far more equivalent mutants than traditional, statement-level, and operators.

Keywords: Testing, Mutant, Mutation Testing, Mutation Operators.

1. INTRODUCTION

Mutation testing assumes that a program will be well tested if most simple faults are detected and removed. Simple faults are introduced by creating a set of faulty Versions, called mutants. Mutation operators describe syntactic changes to the programming language and are applied to the original program to create mutants. Testing can be manual or automatic testing where manual testing/static testing are very slow process because all work is paper work which is time consuming also. Automatic testing works dynamically in different parts iteratively. Software testing is of two types that are white box testing and black box testing. White Box Testing deals with inspecting interior parts of the program and Black Box Testing deals with input and respective output of the program, no need to deal with inner coding [1, 2].
Mutation Testing

One of the white box testing is mutation testing, mutation testing adopts “change and check” strategy for checking the adequacy of the test suites. In mutation testing original program are slightly modified and executed and then compared with the original output. By comparing outputs of original and modified version are compared if both gives the different output it means the software is highly efficient otherwise the software is less adequate. Let S is any object program and it produces output F with respect to the input I.

\[ I \rightarrow S \rightarrow F \]
\[ I \rightarrow S \rightarrow F' \]

If \( F \neq F' \), then it shows that it is highly efficient or called as dead mutant.
If \( F = F' \), then it shows that it is less efficient or less adequate test case.

If the mutant code gives the same result as original code then that test case is not adequate enough and that mutant is called the live or equivalent mutant. For large size software, a large number of mutants generation are required and automated mutant testing may efficiently generates large number of mutant code (NMC) keeping the fact in mind that [3, 4].

1.1 Related Work

DeMillo et al. [5], described about the approach of seeding the faults into the program through various mutation operators. Mutation testing basically is fault based testing which generate effective suites of test cases [6]. There is huge number of operators for procedural and object oriented programming. A procedural language contains the simple syntactic changes in the program such as changing the arithmetic, relational, logical operators. The object-oriented contains some extensions of mutation operators such as encapsulation, inheritance, polymorphism, overloading, exceptional handling etc. [7]. In C language there was a paper named as “Design of Mutant Operator for the C programming Language” shows the different mutation operators in mutation testing. This paper describes about the different mutant operators in ANSI C programming languages. Proteum is the tool used in C language for the automated generation of the different test cases. Error can be introduced in a program in many ways such as syntactic error, error in the expressions, error in the type of the data variable, incorrect position of the variables [3]. Mutation testing shown about the method level operators which can be useful in procedural as well as object oriented languages. Two new kinds of class level mutants for object oriented programs described the four different levels such as algorithm, class, cluster and system level of mutation for procedural and object oriented languages, also described about attribute and method mutants. Mutation testing for object-oriented programs described the additional features such as polymorphism, overloading, inheritance, overriding, information hiding and exceptional handling.

2. MUTATION OPERATORS

There are two types of mutation operators for object oriented languages: (1) those adapted from procedural languages and (2) those developed to handle object oriented-specific features. We refer to these mutation operators as traditional mutation operators and class mutation operators, respectively. We implemented both types in our mutation tool and used them for our study.

2.1 Traditional Mutation Operators

Due to the considerably high execution costs of mutation testing, researchers have proposed a selective mutation technique, which uses a subset of the mutation operators instead all mutation operators. For traditional mutation operators, five selective mutation operators, listed in Table 1, have been shown empirically to provide almost the same effectiveness as using the entire set of
mutation operators, with cost reduction, for the programs used in the study, of at least four times that of using the entire set [8]. Thus, we used this set of mutation operators for our studies.

Table 1. Selective Traditional Mutation Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>Absolute value insertion</td>
</tr>
<tr>
<td>AOR</td>
<td>Arithmetic operator replace</td>
</tr>
<tr>
<td>LCR</td>
<td>Logical connector replace</td>
</tr>
<tr>
<td>ROR</td>
<td>Relational operator replace</td>
</tr>
<tr>
<td>UOI</td>
<td>Unary operator insertion</td>
</tr>
</tbody>
</table>

2.2 Class Mutation Operators

No research has been done to develop a selective mutation technique for class mutation operators. Thus, we implemented the entire set of class mutation operators, which are listed in Table 2, presents a detailed description of these operators are given in [6].

Table 2. Class Mutation Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMC</td>
<td>Access modifier change</td>
</tr>
<tr>
<td>IHD</td>
<td>Hiding variable deletion</td>
</tr>
<tr>
<td>IHI</td>
<td>Hiding variable Insertion</td>
</tr>
<tr>
<td>IOD</td>
<td>Overriding method deletion</td>
</tr>
<tr>
<td>IOR</td>
<td>Overriding method rename</td>
</tr>
<tr>
<td>PMD</td>
<td>Member variable declaration with parent class type</td>
</tr>
<tr>
<td>PPD</td>
<td>Parameter variable declaration with child class type</td>
</tr>
<tr>
<td>PCC</td>
<td>Cast type change</td>
</tr>
<tr>
<td>OMD</td>
<td>Overloading method deletion</td>
</tr>
</tbody>
</table>

3. ESTIMATION OF EFFECTIVENESS OF CLASS MUTATION OPERATORS

Although object-oriented features introduce new kinds of faults, to be detected, these faults may not require a special testing method; some faults may be detected easily with any testing method or some faults may detected with a testing method adapted from traditional testing. In the former case, mutation operators generate easily killed mutants. In the latter case, class mutation operators generate mutants that can be killed by a test suite generated by traditional mutation method. Mutation operators that model these kinds of faults are useless because they raise execution cost without increasing test quality. In this empirical study, we examine whether class mutation operators generate mutants that are killed by the traditional mutation method.

3.1 Experimental Setup

For each class, we generated a test suite that kills all traditional mutants. We then run that test suite against class mutants, and determined the number of class mutants that were killed by the test suite.

Jester

Test-first programming is the least controversial and most widely adopted part of Extreme Programming (XP). By now the majority of professional Java programmers have probably caught
the testing bug. JUnit is the Java community's de facto standard test framework, and a system without a comprehensive JUnit test suite is incomplete. If projects have comprehensive test suites, than producing good-quality software that has some hope of working. But most code bases are quite complex.

**Code Coverage**

The next step beyond testing code is measuring the tests with a *code coverage* tool. Code coverage is a way of seeing how much code is covered by a set of tests. Confidence requires knowing not only that the program as a whole is tested but that each method is tested under all possible conditions. Traditionally such measurements have been performed by monitoring the tests as they execute, perhaps through the Java Virtual Machine Debugging Interface (JVMDI) or the Java Virtual Machine Tool Interface (JVMTI), or by directly instrumenting the bytecode. Any statements that are not executed at least once are not being tested [10]. This approach, taken by tools like Clover and EMMA is valuable for finding untested statements, but it's not enough. Knowing that a statement isn't executed by the test suite proves that it isn't being tested. However, the inverse is not true. If a line of code is executed, it doesn't necessarily follow that it's tested. It's entirely possible that the test doesn't check whether the line of code produces the correct result.

**Jester Performance**

Because Jester recompiles the code base and reruns the test suite for each change it makes, it runs orders of magnitude more slowly than more traditional tools like Clover. It's therefore important to pay some attention to performance. You can use a number of techniques to speed up Jester runs. First, if compiling takes a significant fraction of Jester's execution time, try a faster compiler. Many users have reported noticeable speed-ups by using Jikes instead of javac. It can change the compile command Jester uses in the jester.cfg file in Jester's main directory. Second, profile and optimize test suite. Normally don't worry too much about how fast the unit tests take to run, but any savings can be significant when multiplied by the thousands of times Jester executes the test suite. In particular, look for issues in the test suite that don't arise in normal code. JUnit reinitializes all fields for each and every method executed, so pulling test data out of fields and into local variables can speed things up significantly when the fields aren't used by every method in the test class. If the resulting code duplication offends sense of style, try splitting the test suite into smaller, more modular classes, in each of which all initial data is shared among all test methods [6,7].

**Insure++**

C and C++ developers have a unique problem: many errors in their code don't manifest themselves during testing. Code with subtle problems such as memory corruption may run flawlessly on one machine, but crash on another. To find and fix these problems prior to release, In this need a tool can expose the hidden defects in code. This is the value of Insure++. First introduced in 1993, Insure++ is Parasoft® flagship product. It was specifically designed to help developers and QA personnel find and fix the difficult runtime errors that traditional testing techniques fail to uncover. Using Insure++, can be detect these errors automatically. Insure++ reads and instruments C or C++ source code. Next, it creates an equivalent code by inserting test and analysis functions around every line, and runs the equivalent code to flush out hidden bugs. In addition, Insure++ performs coverage analysis, clearly indicating which sections of the code were tested. By integrating Insure++ into development environment, it can save weeks of debugging time and prevent costly crashes from affecting customers.
Insure++ Features

- Detection of memory corruption on heap and stack
- Detection of uninitialized variables, pointers, and objects
- Detection of memory leaks and other memory allocation/free errors
- STL checking for proper usage of STL containers and related memory errors
- Compile-time checks for type- and size-related errors
- Runtime tracing of function calls
- GUI and command line interface

3.2 Result and Analysis

Over 50% of class mutants are killed by the test suite. IPC, PNC, OMD, EAM and EMM operators show a killed rate greater than 95%. This high killed rate means that these mutant operators may not be useful in mutation testing of object oriented programs because they model the easily detectable faults. Conversely, some mutation operators such as EOA and EOC shows 0% killed rate. They can be considered to model object oriented faults that are difficult to detect. Therefore, they can be thought as good mutation operators for object oriented programs. Although the study uses a small number of sample classes and does not consider the entire set of class mutation operators, it shows that some class mutation operators model faults that can be detected by traditional mutation testing.

4. CONCLUSION

This paper presented the results from two empirical studies of mutation testing with a real object-oriented system. The first study shows that the number of class mutants is relatively small compared to traditional mutation. This small number can promote the application of mutation testing to object-oriented programs. The second study shows that some class mutation operators model faults that are detected easily by tradition mutation testing. Thus, when creating selective mutation operators for object-oriented programs, these mutation operators can be omitted. In this paper, a few areas stood out as especially important that a mutation testing tool that is fast, easy to use and suitable for today’s software development environment is much wanted for data mining approach. Efficient mutation operators are the key to quality mutation testing. Are the operators currently used for object oriented language and integration testing good enough?

5. REFERENCES


